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

*Key Idea:* modify our approach for dealing with lists so that we can deal with natural numbers.

First, we’ll review how we derived the list template.

;;; A (listof X) is one of:

;;; ∀ empty

;;; ∀ (cons X (listof X))

**Natural number template** Suppose we have a list `lst`.

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

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

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.

**Key Idea:** We must consider (1) a base case (2) a single natural number and (3) the rest of the natural numbers.
Natural numbers

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

Here \texttt{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 \( (\text{zero?} \ n) \) 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.

\[
\text{(define (nat-template n)}
\begin{array}{l}
\text{(cond [(zero? n) … ])}
\text{[else (… n …}
\text{ … (nat-template (sub1 n)) … ]])})
\end{array}
\]
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 \( n \).

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

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

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

*Key Point:* with a slight modification we extend our approach to talk about ranges of integers.

The symbol $\mathbb{Z}$ is often used *to denote the integers*.

We can add subscripts to define subsets (a.k.a. ranges) of the integers.

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

Other examples: $\mathbb{Z}_{> 4}$, $\mathbb{Z}_{<-8}$, $\mathbb{Z}_{\leq 1}$.
Non-zero base case: E.g. $\mathbb{Z}_{\geq 7}$

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:

`;; $\star 7$

`;; $\star (\text{add1 } \mathbb{Z}_{\geq 7})$

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.
(define (countdown-to-7 n)
  (cond [(= n 7) (cons 7 empty)]
        [else (cons n (countdown-to-7 (sub1 n)))]))

Note: in the Data Definition we add1 up from our base case but in the template we sub1 down to our base.
Generalizing **countdown and countdown-to-7**

*Key Point:* 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: basic idea

*Key Point:* 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.
Counting up template

Notice the *additional requires section* capturing the restriction on \( n \).

```scheme
;; nonpos-template: Int → Any
;; requires: \( n \leq 0 \)
(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)))).
```
Counting up example

;; (countup n) produces an increasing list from n to 0
;; countup: Int → (listof Int)
;; requires: n ≤ 0
;; Example:
(underbrace{(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)))]))
## Counting up to \texttt{b}

As before, we can generalize this to counting up to \texttt{b}, by introducing \texttt{b} as a second parameter in a template.

\texttt{;; (countup-to \ n \ b) produces an increasing list from \ n \ to \ b}
\texttt{;; countup-to: Int Int \rightarrow (listof Int)}
\texttt{;; requires: \ n \ \leq \ b}

\texttt{;; Example:}
\texttt{(check-expect (countup-to 6 8) (cons 6 (cons 7 (cons 8 empty)))))}

\texttt{(define (countup-to \ n \ b)}
\texttt{(cond [(= \ n \ b) (cons \ b \ empty)]}
\texttt{[else (cons \ n \ (countup-to (add1 \ n \) \ b)))]))}
Comparisons with imperative programming

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

```plaintext
def 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.
Data definitions vs. templates

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.

*Recall* that in the Data Definition, recursion moves *away from the base case*. In the template, recursion moves *towards the base case*.

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.
Module 07 Summary

Natural Numbers and Integers

1. There is a recursive definition for the set of natural numbers, Nat (which is similar to the one for lists). [5]
2. The functions add1, sub1 and zero? are used with Nat’s. [5–6]
3. This is also a recursive definition for ranges of integers with a finite end point. [11]
4. Use \( \mathbb{Z}_{\geq i} \) to specify integers greater than or equal to \( i \). [11]
5. We can count down to an integer [11–17] or count up. [18–21]
6. In the template, recursion moves towards the base. In the data definition, recursion moves away from the base. [23]