07: Natural Numbers

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)`.
  - `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-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.

A Formal Definition of Natural Numbers (1/3)

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.

1 can be represented 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.)

A Formal Definition of Natural Numbers (2/3)

The successor function $S(n)$ produces the "next" natural number. We will use the Racket function `add1` as the successor function:

```
(add1 0) ⇒ 1
(add1 1) ⇒ 2
(add1 2) ⇒ 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.

$\longrightarrow$

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

A Formal Definition of Natural Numbers (3/3)


```
;; 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.

Developing `nat-template`

Suppose we have a natural number, *n*. Then it must conform to our data definition:

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

The test `(zero? n)` tells us which of these cases applies, yielding:

```
;; nat-template: Nat -> Any
(define (nat-template n)
  (cond [(zero? n) ...]  ;; n is 0
        [else ...]))     ;; n is (add1 k), for some k
```

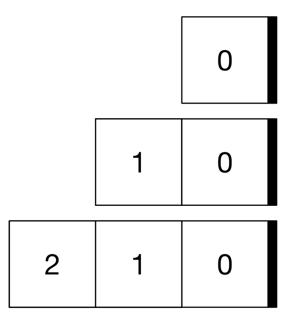
We can compute *k* with `(- n 1)` or `(sub1 n)`.

Because *k* is a natural number, we recursively apply the function we are constructing to it.

```
;; nat-template: Nat -> Any
(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 1)` ⇒ `(cons 1 (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
(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)))]))
```

Useful questions:

1. What do we produce in the base case?
2. In the recursive case, what (if anything) do we do to transform `n`?
3. What is the result of processing `(f (sub1 n))` recursively?
4. How do we combine steps 2 and 3 to obtain the result for `(f n)`?


```
;; (countdown n) produces a decreasing list of Nats from n to 0
;; Examples:
(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)))]))
```

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

Ex. 1

Write a recursive function `(sum-to n)` that consumes a `Nat` and produces the sum of all `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:

```
;; An integer in ℤ≥7 is one of:
;; ⋆ 7
;; ⋆ (add1 ℤ≥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`


```
;; (countdown-to-7 n) produces a decreasing list from n to 7
```

Tracing `countdown-to-7`:

```
(countdown-to-7 9)
⇒ (cons 9 (countdown-to-7 8))
⇒ (cons 9 (cons 8 (countdown-to-7 7)))
⇒ (cons 9 (cons 8 (cons 7 empty)))
```

> 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 `base`.

Here, the stopping condition will depend on `base`.

The parameter `base` has to **"go along for the ride"** (be passed unchanged) in the recursion.

> `countdown-to`

```
;; (countdown-to n base) produces a decreasing list from n to base
;; Examples:
(check-expect (countdown-to 4 2) (cons 4 (cons 3 (cons 2 empty))))
(check-expect (countdown-to 7 7) (cons 7 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))]))
```

```
(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))))
```

Ex. 2

Write a recursive function `(sum-between n b)` than consumes two `Nat`, with `n` $\geq$ `b`, and returns the sum of all `Nat` between `b` and `n`.

`(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 ℤ≤0 is one of:
;; ⋆ 0
;; ⋆ (sub1 ℤ≤0)
```

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

Since `(add1 (sub1 n)) => n` for all integers `n`, the inverse function we need is `add1`.

This suggests the following template.

> nonpos-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

```
;; (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)))]))
```

> countup-to

As before, we can generalize this to counting up to b, by introducing `base` 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))]))
```

Example: power

The countdown/countup pattern is not only applicable to building lists. Consider calculating n^e where e, the exponent, is an integer.

The key insight is that $n^e = n * n^{e-1}$ and that n^0 is 1.

```
(check-expect (power 2 0) 1)
(check-expect (power 2 1) 2)
(check-expect (power 3 3) 27)
```

```
;; nat-template: Nat -> Any
(define (nat-template n)
  (cond [(zero? n) ...]
        [else (... n
                   (nat-template (sub1 n)))]))
```

Example: power

With renaming, documentation, and adding parameters:

```
;; (power n e) computes n^e
;; power: Int Nat -> Int
(define (power n e)
  (cond [(zero? e) ...]
        [else (... n e
                   (power n (sub1 e)))]))
```

```
;; (power n e) computes n^e
;; power: Int Nat -> Int
(define (power n e)
  (cond [(zero? e) 1]
        [else (* n (power n (sub1 e)))]))
```

> Repetition in other languages

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.

> `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. You may not implement your own version, either.

Ex. 3

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

`(countdown-by 12 3)` ⇒ `(cons 12 (cons 9 (cons 6 (cons 3 empty))))`

`(countdown-by 11 3)` ⇒ `(cons 11 (cons 8 (cons 5 (cons 2 empty))))`

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

Ex. 4

This exercise recurses on a list and a `Nat` at the same time.

Complete `n-th-item`.

```
;; (n-th-item lst n) Produce the n-th item in lst, where (first lst) 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
;; Requires: n < (length lst)
(define (n-th-item lst n) ...)
```

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.

Summary: built-in functions

The following functions and special forms have been introduced in this module:

`add1 sub1`

You should complete all exercises and assignments using only these and the functions and special forms introduced in earlier modules. The complete list is:

```
* + - ... / < <= = > >= abs add1 and boolean? ceiling char-alphabetic? char-downcase
    char-lower-case? char-numeric? char-upcase char-upper-case? char-whitespace? char<=?
    char<? char=? char>=? char>? char? check-error check-expect check-within cond cons
    cons? cos define define-struct define/trace e else empty? error even? exp expt first
    floor integer? length list->string list? log max min modulo negative? not
    number->string number? odd? or pi positive? quotient remainder rest round sgn sin sqr
    sqrt string->list string-append string-downcase string-length string-lower-case?
    string-numeric? string-upcase string-upper-case? string<=? string<? string=? string>=?
    string>? string? sub1 substring symbol=? symbol? tan zero?
```