# 7: Binary

## Back to natural numbers

Let’s return to the simulation of natural numbers by Racket structures that we developed in lecture module 04.

Our goal is to improve the efficiency of this simulation, and to derive the method used to actually represent natural numbers and integers in a modern digital computer.

Here we have a difference in the perspectives of mathematics and computer science.

Mathematically, our simulation is correct, and its simplicity facilitates proofs.

Computationally, our simulation is not efficient enough, and we will trade some simplicity for better efficiency.

We’ll start with a review of our earlier work.

## Review of our simulation of natural numbers

```
(define-struct Z ())
(define-struct S (pred))
```

**Data definition:** a Nat is either `(make-Z)` or it is `(make-S p)`, where `p` is a Nat.

`(make-Z)` represents $0$; `(make-S p)` represents $n + 1$, where `p` is a Nat representing $n$.

We will call this definition the **unary definition of Nat**, because we will shortly introduce another one.

## Deficiencies of the unary definition of Nat

The unary representation of the natural number 3 is
`(make-S (make-S (make-S (make-Z))))`.

Imagine the unary representation of 1000. It wouldn't fit on this slide. Intuitively speaking, this means it takes up a lot of space in the memory of a computer.

Recall our implementation of addition.

```
(define (plus nat1 nat2)
  (cond
    [(Z? nat1) nat2]
    [(S? nat1) (make-S (plus (S-pred nat1) nat2))]))
```

Imagine how long the trace would be if we added the representations of 1000000 and 2000000. This corresponds to a long running time on a computer.

## Towards a better representation

We could try to interpret `make-S` as something other than “successor”. So `(make-S (make-Z))` could be a number different from one.

$$\begin{tikzcd}
0 \arrow[r, "S"] & 1 \arrow[r, "S"] & 2 \arrow[r, "S"] & {}
\end{tikzcd}$$

But consider the representation of the first million natural numbers. One of those numbers has to have a long representation, regardless of how we interpret `make-S`.

The bottleneck is that we have only one data constructor, `make-S`, that can be applied to an existing Nat to create a new one.

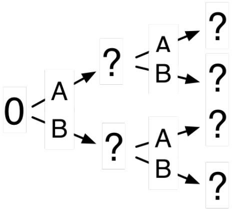
The simplest change we can make is to use two data constructors.

```
(define-struct Z ())
(define-struct A (way))
(define-struct B (gone))
```

## The binary definition of Nat, started

```
(define-struct Z ())
(define-struct A (way))
(define-struct B (gone))
```

Unlike with the unary definition, there is no significance to the names `A` and `B`, and the names of the fields are chosen to indicate “removal”.



## Desired properties of our definition

We get to choose the interpretation of the use of `make-A` and `make-B`. What properties do we want of this interpretation?

- Every natural number should be representable.
  None should be left out.
- The representation should be unique.
  Each natural number should have exactly one representation.
- Computation using the representation should be relatively straightforward and efficient.
- It would be nice if there were a “natural” interpretation of the representation, and if small numbers had a small representation.

## The binary definition of Nat, completed

```
(define-struct Z ())
(define-struct A (way))
(define-struct B (gone))
```

A Nat is either `(make-Z)`, or it is `(make-A p)`, or it is `(make-B p)`, where `p` is a Nat.

`(make-Z)` represents $0$.

`(make-A p)` represents $2n$, where `p` represents $n$.

`(make-B p)` represents $2n + 1$, where `p` represents $n$.

This is the **binary definition** of Nat.

**Immediate problem:** `(make-A (make-Z))` also represents $0$.

We add the rule: `make-A` cannot be applied to `(make-Z)`.

## Effect of our interpretation

The representation of $5$ is
`(make-B (make-A (make-B (make-Z))))`.

$$\begin{tikzcd}
 & & 2 \arrow[r, "\boxed{A}"] \arrow[dr, "\boxed{B}"'] & 4 \\
0 \arrow[r, "\boxed{B}"] & 1 \arrow[ur, "\boxed{A}"] \arrow[dr, "\boxed{B}"'] & & 5 \\
 & & 3 \arrow[r, "\boxed{A}"] \arrow[dr, "\boxed{B}"'] & 6 \\
 & & & 7
\end{tikzcd}$$

## A template for recursion on binary Nats

A Nat is either `(make-Z)`, or it is `(make-A p)`, or it is `(make-B p)`, where `p` is a Nat.

```
(define (my-nat-fn nat)
  (cond
    [(Z? nat) ...]
    [(A? nat) ...]
    [(B? nat) ...]))


(define (my-nat-fn nat)
  (cond
    [(Z? nat) ...]
    [(A? nat) ... (my-nat-fn (A-way nat)) ...]
    [(B? nat) ... (my-nat-fn (B-gone nat)) ...]))
```

## Converting binary Nats to built-in natural numbers

```
(define (my-nat-fn nat)
  (cond
    [(Z? nat) ...]
    [(A? nat) ... (my-nat-fn (A-way nat)) ...]
    [(B? nat) ... (my-nat-fn (B-gone nat)) ...]))

(define (from-Nat nat)
  (cond
    [(Z? nat) 0]
    [(A? nat) (* 2 (from-Nat (A-way nat)))]
    [(B? nat) (add1 (* 2 (from-Nat (B-gone nat))))]))
```

## Converting built-in natural numbers to binary Nats

`(make-Z)` represents $0$.

`(make-A p)` represents $2n$, where `p` represents $n$.

`(make-B p)` represents $2n + 1$, where `p` represents $n$.

```
(define (to-Nat k)
  (cond
    [(zero? k) (make-Z)]
    [(even? k) (make-A (to-Nat (/ k 2)))]
    [(odd? k)  (make-B (to-Nat (/ (sub1 k) 2)))]))
```

```
(define (to-Nat k)
  (cond
    [(zero? k) (make-Z)]
    [(even? k) (make-A (to-Nat (quotient k 2)))]
    [(odd? k)  (make-B (to-Nat (quotient k 2)))]))
```

## Implementing addition for binary Nats

```
(define (my-nat-fn nat)
  (cond
    [(Z? nat) ...]
    [(A? nat) ... (my-nat-fn (A-way nat)) ...]
    [(B? nat) ... (my-nat-fn (B-gone nat)) ...]))
```

```
(define (plus nat1 nat2)
  (cond
    [(Z? nat1) nat2]
    [(A? nat1) ... (plus (A-way nat1) nat2) ...]
    [(B? nat1) ... (plus (B-gone nat1) nat2) ...]))
```

If `nat1` represents $2x$ and `nat2` represents $y$,
then `(A-way nat1)` represents $x$.

What does $x + y$ tell us about $2x + y$?

This is too simplistic an approach.

## Using structural recursion on two binary Nats

```
(define (my-nat-fn nat1 nat2)
  (cond
    [(Z? nat1) ...]
    [(Z? nat2) ...]
    [(and (A? nat1) (A? nat2))
      ... (my-nat-fn (A-way nat1) (A-way nat2))]
    [(and (A? nat1) (B? nat2))
      ... (my-nat-fn (A-way nat1) (B-gone nat2))]
    [(and (B? nat1) (A? nat2))
      ... (my-nat-fn (B-gone nat1) (A-way nat2))]
    [(and (B? nat1) (B? nat2))
      ... (my-nat-fn (B-gone nat1) (B-gone nat2))]))


(define (plus nat1 nat2)
  (cond
    [(Z? nat1) nat2]
    [(Z? nat2) nat1]
    [(and (A? nat1) (A? nat2)) ... (plus (A-way nat1) (A-way nat2))]
    [(and (A? nat1) (B? nat2)) ... (plus (A-way nat1) (B-gone nat2))]
    [(and (B? nat1) (A? nat2)) ... (plus (B-gone nat1) (A-way nat2))]
    [(and (B? nat1) (B? nat2)) ... (plus (B-gone nat1) (B-gone nat2))]))
```

## More details of addition

```
(define (plus nat1 nat2)
  (cond
    [(Z? nat1) nat2]
    [(Z? nat2) nat1]
    [(and (A? nat1) (A? nat2)) ... (plus (A-way nat1) (A-way nat2))]
    [(and (A? nat1) (B? nat2)) ... (plus (A-way nat1) (B-gone nat2))]
    [(and (B? nat1) (A? nat2)) ... (plus (B-gone nat1) (A-way nat2))]
    [(and (B? nat1) (B? nat2)) ... (plus (B-gone nat1) (B-gone nat2))]))
```

```
(define (plus nat1 nat2)
  (cond
    [(Z? nat1) nat2]
    [(Z? nat2) nat1]
    [(and (A? nat1) (A? nat2))
       (make-A (plus (A-way nat1) (A-way nat2)))]
    [(and (A? nat1) (B? nat2))
       (make-B (plus (A-way nat1) (B-gone nat2)))]
    [(and (B? nat1) (A? nat2))
       (make-B (plus (B-gone nat1) (A-way nat2)))]
    [(and (B? nat1) (B? nat2))
       ... (plus (B-gone nat1) (B-gone nat2)) ...]))
```

## Addition for binary Nats, completed

```
(define (plus nat1 nat2)
  (cond
    [(Z? nat1) nat2]
    [(Z? nat2) nat1]
    [(and (A? nat1) (A? nat2))
       (make-A (plus (A-way nat1) (A-way nat2)))]
    [(and (A? nat1) (B? nat2))
       (make-B (plus (A-way nat1) (B-gone nat2)))]
    [(and (B? nat1) (A? nat2))
       (make-B (plus (B-gone nat1) (A-way nat2)))]
    [(and (B? nat1) (B? nat2))
       (make-A (plus-one (plus (B-gone nat1) (B-gone nat2))))]))

(define (plus-one nat)
  (cond
    [(Z? nat) (make-B (make-Z))]
    [(A? nat) (make-B (A-way nat))]
    [(B? nat) (make-A (plus-one (B-gone nat)))]))
```

## A natural interpretation of Nat

4 is represented by `(make-A (make-A (make-B (make-Z))))`.

To see the correspondence with a more natural interpretation, erase the parentheses and `make-Z`, replace each `make-A` with 0 and each `make-B` with 1, and reverse the resulting sequence of digits.

The result is 100. But this is not one hundred.

$4 = 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0$

$4 = 100_2$ (“one hundred in base two”).

This is known as **binary notation**.

A binary digit (0 or 1) is called a **bit.**

The rule that we don’t apply `make-A` to `(make-Z)` corresponds to the convention of not writing leading zeroes on numbers.

## Use of binary in computers

Binary notation is used to represent natural numbers in computers.

The two different bits, 0 and 1, can be represented by
two different voltage levels.

Eight bits are typically grouped into a **byte**, and four bytes
into a 32-bit **word**.

64-bit words are also used by an increasing number of computers.

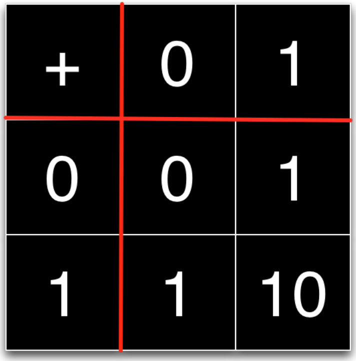
A 32-bit word representing the natural number 4 would be filled with
zeroes on the left:
`00000000000000000000000000000100`

## Reasons for the use of binary in computers

Why is binary used in computers and not decimal?

The code we have developed for binary addition is implemented in hardware.

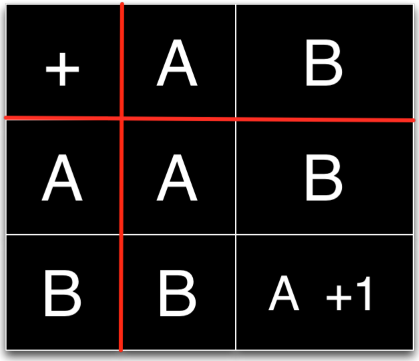
Our code essentially uses the addition table for binary digits.



| + | 0 | 1 |
|---|---|---|
| 0 | 0 | 1 |
| 1 | 1 | 10 |

## Reasons for the use of binary in computers

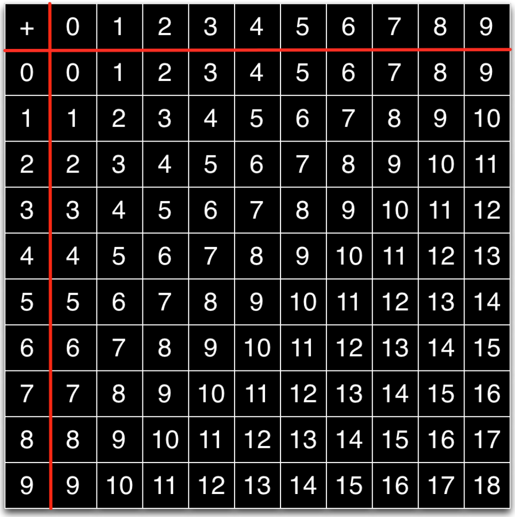
The correspondence between the binary addition table and our algorithm becomes clearer if we rewrite the table slightly.



| + | A | B |
|---|---|---|
| A | A | B |
| B | B | A +1 |

## Reasons for the use of binary in computers

In contrast, the addition table for decimal digits is much larger, and would lead to more complex hardware.



| + | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
| 2 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
| 3 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
| 4 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 |
| 5 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| 6 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 |
| 7 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 |
| 8 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 |
| 9 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 |

## Space analysis of binary Nats

We do not have a proper model of space or memory use.

For the time being, we'll take the number of constructors used in a Nat as a measure of memory use.

This is easy to compute.

```
(define (size nat)
  (cond
    [(Z? nat) 1]
    [(A? nat) (add1 (size (A-way nat)))]
    [(B? nat) (add1 (size (B-gone nat)))]))
```

## Space analysis of binary Nats

Let's define $S(n)$ as `(size nat)`, where `nat` is the binary representation of $n$.

We're interested in how quickly $S(n)$ grows.

| $n$ | `nat` | $S(n)$ |
|---|---|---|
| 0 | `(make-Z)` | 1 |
| 1 | `(make-B (make-Z))` | 2 |
| 2 | `(make-A (make-B (make-Z)))` | 3 |
| 3 | `(make-B (make-B (make-Z)))` | 3 |
| 4 | `(make-A (make-A (make-B (make-Z))))` | 4 |
| 5 | `(make-B (make-A (make-B (make-Z))))` | 4 |
| 6 | `(make-A (make-B (make-B (make-Z))))` | 4 |
| 7 | `(make-B (make-B (make-B (make-Z))))` | 4 |

**Conjecture:** For $2^k \leqslant n < 2^{k+1}$, $S(n) = k + 2$.

Or: For $k \leqslant \log_2 n < k + 1$, $S(n) = k + 2$.

Or: For $n > 0$, $S(n) = \lfloor \log_2 n \rfloor + 2$.

## Time analysis of plus-one

```
(define (plus-one nat)
  (cond
    [(Z? nat) (make-B (make-Z))]
    [(A? nat) (make-B (A-way nat))]
    [(B? nat) (make-A (plus-one (B-gone nat)))]))
```

The recursive application of `plus-one` is on an argument whose size is one smaller than `nat`.

Thus, the number of applications of `plus-one` starting with an argument `nat` is bounded above by `(size nat)`.

## Problem with analysis of plus

Our implementation of `plus` also reduces the size of its arguments in a similar fashion, but the last case is problematic:

```
[(and (B? nat1) (B? nat2))
   (make-A (plus-one (plus (B-gone nat1) (B-gone nat2))))]
```

The problem is accounting for the additional work done by `plus-one`.

The code we developed is actually efficient, but it is difficult to see why.

We will rewrite the code to make it clearer.

## Rewriting plus

```
[(and (B? nat1) (B? nat2))
      (make-A (plus-one (plus (B-gone nat1) (B-gone nat2))))]
```

Instead of using `plus-one`, we will develop a helper function `plus-s`, which computes the successor of the sum of its two arguments.

```
[(and (B? nat1) (B? nat2))
      (make-A (plus-s (B-gone nat1) (B-gone nat2)))]
```

`plus` has recursive applications of itself, and an application of `plus-s`.

It will turn out that `plus-s` has recursive applications of itself, and applications of `plus`.

`plus` and `plus-s` are **mutually recursive**.

## The development of plus-s

```
(define (plus-s nat1 nat2)
  (cond
    [(Z? nat1) ...]
    [(Z? nat2) ...]
    [(and (A? nat1) (A? nat2))
       ... (plus??? (A-way nat1) (A-way nat2)) ...]
    [(and (A? nat1) (B? nat2))
       ... (plus??? (A-way nat1) (B-gone nat2)) ...]
    [(and (B? nat1) (A? nat2))
       ... (plus??? (B-gone nat1) (A-way nat2)) ...]
    [(and (B? nat1) (B? nat2))
       ... (plus??? (B-gone nat1) (B-gone nat2)) ...]))
```

## plus-s completed

```
(define (plus-s nat1 nat2)
  (cond
    [(Z? nat1) (plus-one nat2)]
    [(Z? nat2) (plus-one nat1)]
    [(and (A? nat1) (A? nat2))
      (make-B (plus (A-way nat1) (A-way nat2)))]
    [(and (A? nat1) (B? nat2))
      (make-A (plus-s (A-way nat1) (B-gone nat2)))]
    [(and (B? nat1) (A? nat2))
      (make-A (plus-s (B-gone nat1) (A-way nat2)))]
    [(and (B? nat1) (B? nat2))
      (make-B (plus-s (B-gone nat1) (B-gone nat2)))]))
```

## Analysis of the new version of plus

The mutually-recursive `plus` and `plus-s` always reduce the size of their arguments in recursive applications.

When one argument becomes `(make-Z)` (size one), `plus-one` may be used, and this reduces the size of its argument in recursive applications.

The total number of applications of `plus`, `plus-s`, and `plus-one` starting with arguments `nat1` and `nat2` is bounded above by the maximum of the sizes of `nat1` and `nat2`.

Something similar is true of the number of applications of `plus` and `plus-one` in our original code, but the reasoning involved is more complicated.

This matches our intuition that the work done in adding two numbers should be proportional to the number of digits, not the value of the numbers (as it was for the unary definition of Nat).

## Simulating integers

Let's try to extend our efficient representation of natural numbers to an efficient representation of integers.

Our first attempt will not work so well, and we'll have to move to a different idea.

First, we will try to augment with a sign, as we do with decimal numbers. We can view $-8$ as a negative sign attached to the natural number $8$.

```
(define-struct Pos (nat))
(define-struct Neg (nat))
```

**Data definition:** an Int is either a `(make-Pos n)` or a `(make-Neg n)`, where `n` is a Nat.

## Problems with augmenting with a sign

Our first problem is that `(make-Pos (make-Z))` and `(make-Neg (make-Z))` both represent the integer $0$.

We solve this problem for decimal numbers by not writing $-0$.
We can similarly forbid `(make-Neg (make-Z))`.

A bigger problem arises when we try to implement addition.
Some cases are easy.

```
(define (iplus int1 int2)
  (cond
    [(and (Pos? int1) (Pos? int2))
       (make-Pos (plus (Pos-nat int1) (Pos-nat int2)))]
    [(and (Pos? int1) (Neg? int2))
        ???]
    ...))
```

Think about how we add a positive integer to a negative integer on paper.

This is too complicated.

## A better idea: augmenting with negative one

Instead of augmenting with a sign, we will augment the structures we had for Nat with a new structure to represent the quantity $-1$.

```
(define-struct N ())
```

**Data definition:** an Int is either `(make-Z)`, or `(make-N)`, or `(make-A p)` or `(make-B p)`, where `p` is an Int.

We keep the same interpretation of `make-A` and `make-B`.

`(make-Z)` represents $0$.

`(make-N)` represents $-1$.

`(make-A p)` represents $2i$, where `p` represents $i$.

`(make-B p)` represents $2i + 1$, where `p` represents $i$.

An immediate advantage is that our representation of non-negative integers (natural numbers) is the same in Int as it is in Nat.

All our code for `plus` is still useful. We only have to add cases that involve `(make-N)`.

## Ensuring unique representation

We still have the rule that we can't apply `make-A` to `(make-Z)`.

But now we have the problem that `(make-B (make-N))` and `(make-N)` both represent $-1$.

So we must add the rule that we can't apply `make-B` to `(make-N)`.

One way to enforce these rules is to create **smart constructors**, which are replacements for `make-A` and `make-B` with the rules built in.

```
(define (sA int)
  (cond
    [(Z? int) int]
    [else (make-A int)]))

(define (sB int)
  (cond
    [(N? int) int]
    [else (make-B int)]))
```

## Representations of some small negative integers

As before, we define $S(i)$ to be the number of constructors in the representation of $i$.

| $i$ | `nat` | $S(i)$ |
|---|---|---|
| $0$ | `(make-Z)` | 1 |
| $-1$ | `(make-N)` | 1 |
| $-2$ | `(make-A (make-N))` | 2 |
| $-3$ | `(make-B (make-A (make-N)))` | 3 |
| $-4$ | `(make-A (make-A (make-N)))` | 3 |
| $-5$ | `(make-B (make-B (make-A (make-N))))` | 4 |
| $-6$ | `(make-A (make-B (make-A (make-N))))` | 4 |
| $-7$ | `(make-B (make-A (make-A (make-N))))` | 4 |
| $-8$ | `(make-A (make-A (make-A (make-N))))` | 4 |

## Addition for Ints

```
(define (plus int1 int2)
  (cond
    [(Z? int1) int2]
    [(Z? int2) int1]
    ; add N cases here
    [(and (A? int1) (A? int2))
       (sA (plus (A-way int1) (A-way int2)))]
    [(and (A? int1) (B? int2))
       (sB (plus (A-way int1) (B-gone int2)))]
    [(and (B? int1) (A? int2))
       (sB (plus (B-gone int1) (A-way int2)))]
    [(and (B? int1) (B? int2))
       (sA (plus-one (plus (B-gone int1) (B-gone int2))))]))

(define (plus-one int)
  (cond
    [(Z? int) (sB (make-Z))]
    ; add N case here
    [(A? int) (sB (A-way int))]
    [(B? int) (sA (plus-one (B-gone int)))]))
```

## The missing cases for plus and plus-one with Ints

```
; missing cases for plus

[(and (N? int1) (N? int2)) (sA (make-N))]
[(and (B? int1) (N? int2)) (sA (B-gone int1))]
[(and (N? int1) (B? int2)) (sA (B-gone int2))]
[(and (A? int1) (N? int2)) (sB (plus (A-way int1) (make-N)))]
[(and (N? int1) (A? int2)) (sB (plus (make-N) (A-way int2)))]

; missing case for plus-one

[(N? int) (make-Z)]
```

The resulting code preserves the nice features of `plus` for Nats that made it efficient.

## Subtraction for Ints

Having a representation for negative numbers allows us to think about implementing subtraction.

We will implement subtraction in terms of negation, and negation in terms of another helper function, `flip`, which maps the representation of $i$ to the representation of $-(i+1)$.

```
(define (subt int1 int2) (plus int1 (negate int2)))
(define (negate int) (plus-one (flip int)))

(define (flip int)
  (cond
    [(N? int) (make-Z)]
    [(Z? int) (make-N)]
    [(A? int) (make-B (flip (A-way int)))]
    [(B? int) (make-A (flip (B-gone int)))]))
```

This also makes it clear that $S(i)$ grows like the logarithm to the base 2 of the absolute value of $i$.

## Two's complement notation

To create a more readable version of Int, as before, we erase parentheses, replace `make-A` by $0$ and `make-B` by $1$, and read right-to-left.

But we treat `make-Z` and `make-N` differently.

We replace `make-Z` by the left-infinite sequence of zeroes, which we'll write $\ldots 0$.

Similarly, we replace `make-N` by the left-infinite sequence of ones, which we'll write $\ldots 1$.

This makes every integer into a left-infinite sequence of 0's and 1's.

We saw that $-5$ was represented by
`(make-B (make-B (make-A (make-N))))`.

Its “readable” representation is $\ldots 1011$.

## Two's complement notation

```
 4 = ...0100
 3 = ...011
 2 = ...010
 1 = ...01
 0 = ...0
-1 = ...11
-2 = ...10
-3 = ...101
-4 = ...100
-5 = ...1011
```

The representation of integers in modern computers
(called **two's complement notation**) consists of these bit sequences truncated to the rightmost 32 or 64 bits.

For example, the representation of $-5$ would be
`11111111111111111111111111111011`.

Interpreted as a natural number instead, this 32-bit pattern represents 4294967291.

## Why is two's complement notation used?

In creating the code for `plus` operating on Ints, we reused all the code for `plus` operating on Nats, and added some new cases.

In hardware, this corresponds to being able to share components that implement functionality for both natural numbers (often called **unsigned integers**) and integers.

The short code for negation and subtraction corresponds to a small amount of additional hardware to support those features.

## Goals of this module

You should be able to write space- and time-efficient code that works with the unary and binary representations of natural numbers and integers.

You should be able to read and work with binary notation for natural numbers and two’s complement notation for integers.