4 Week of 04/19: 2-adic numbers

One of the standard ways of defining the natural numbers in Haskell, in terms of their Peano representations, is using the following algebraic data type:

```
data Nat = Zero | Succ Nat
```

which expresses that every natural number is either zero or the successor of some other natural number, and that we can build up natural numbers by starting with `Zero` and repeatedly applying `Succ`. (Although, because of Haskell's laziness, it's possible to define the "fake" natural number `x = Succ x`, which is somewhat relevant to what we'll see momentarily.) We've already seen how lots of arithmetic functions can be built up from this very simple interface for the natural numbers while studying Scheme, but in this recitation, we'll look at a way of representing natural numbers *directly in terms of their binary representations*. That is, we'll use sequences of zeros and ones to represent natural numbers, and implement some arithmetic operations directly using these representations (without converting to a more standard representation).

A possible way of representing natural numbers as binary strings would be to use a custom type which is essentially the same as lists of ones and zeroes. For instance, something like this:

```
data BinInt = Empty | Zero BinInt | One BinInt
```

where the constructors `Zero` and `One` refer to the least significant bit of the binary representation of the number. You could think of a binary representation as a stack with the topmost element being the least significant bit, the constructor `Zero` as a function that pushes a zero digit onto the stack, and the constructor `One` as a function that pushes a one digit onto the stack. For instance, the number 6, with the binary representation 110_2 could be represented as `Zero (One (One Empty))`. But there's a flaw in this representation - it's very redundant. For instance, 6 could be constructed in any of the following ways:

- `Zero (One (One Empty))`
- `Zero (One (One (Zero Empty)))`
- `Zero (One (One (Zero (Zero Empty))))`
- ...

which correspond roughly to writing 6 with different numbers of "leading zeros" as either 110_2 or 0110_2 or 00110_2 and so on.

We can eliminate this redundancy by modifying the definition of the `BinInt` type as follows:

```
data BinInt = Zero BinInt | One BinInt
```

What does this accomplish? By getting rid of the `Empty` list, we effectively force *every list* to be a lazy infinite list. (In fact, this kind of construction has its own name - it's called a **stream**.) This means that there's only one way to represent, say, the number 6 - it's the binary representation of 6 in which *all the leading zeros* are present, which we could denote $\cdots 000110_2$. In these new binary representations, there's no ambiguity when it comes to how many leading zeros to include, because every `BinInt` contains "all of its digits" for every place value. The representation of the natural number 0 would be constructed as a lazily infinite list of zeros, like this:

```
zero = Zero zero
```

and then we could represent 6 like this:

```
six = Zero (One (One six))
```

Before we go on to implementing arithmetic operations with this new type, let's instance it into `Show` so that we have an easy way of seeing the numbers we're working with. The following function takes an `Int` (assumed to be nonnegative) *n* and a `BinInt` and shows the least significant *n* digits of that `BinInt`, followed by ellipses:

```
showFirstDigs :: Int -> BinInt -> String
showFirstDigs 0 _ = "..."
showFirstDigs n (Zero b) = (showFirstDigs (n - 1) b) ++ "0"
showFirstDigs n (One b) = (showFirstDigs (n - 1) b) ++ "1"
```

Of course it's not possible to print out *all* of the digits of a given `BinInt`, nor is it possible *a priori* to know how many digits it will have before its infinite train of leading zeros begins. So to instance `BinInt` into `Show`, let's just print the first 20 digits of any number, as this will probably be enough for our purposes.

```
instance Show BinInt where
  show = showFirstDigs 20
```

We can also quickly write a function to convert standard natural numbers to our special representations by extracting their coefficients using integer division:

```
natToBin :: Int -> BinInt
natToBin 0 = zero
natToBin n = if (even n)
             then Zero (natToBin (div n 2))
             else One (natToBin (div n 2))
```

We can test these functions in `gchi` like this:

```
ghci> natToBin 21
...00000000000000010101
```

Now let's try implementing some simple arithmetic operations on natural numbers, using this peculiar way of representing them. Starting simple, we'll just write a function `plusone` that takes a `BinInt` and gives back its successor. (In `Nat`, this was trivial - the successor function was one of the constructors!) We can write this function as follows, using pattern matching:

```
plusone :: BinInt -> BinInt
plusone (Zero b) = One b
plusone (One b) = Zero (plusone b)
```

The first case uses the fact that adding 1 to a binary number whose least significant bit is a 0 can be achieved by simply flipping that bit to a 1. For a binary number with a least significant bit of 1, on the other hand, we will need to change this bit to a 0 *and carry a one* (to explain this in terms of long addition). You could also think of the second case in this definition as reflecting the following algebraic identity:

$$(2x + 1) + 1 = 2(x + 1)$$

keeping in mind that applying `Zero` to a natural number effectively performs a "left shift" that multiplies its value by two, whereas applying `One` multiplies its value by two and adds one.

We can also implement natural number addition using the following function of two `BinInt` arguments, which pattern matches on both arguments for a total of four cases:

```
plus :: BinInt -> BinInt -> BinInt
plus (Zero b1) (Zero b2) = Zero (plus b1 b2)
plus (Zero b1) (One b2) = One (plus b1 b2)
plus (One b1) (Zero b2) = One (plus b1 b2)
plus (One b1) (One b2) = Zero (plusone (plus b1 b2))
```

You can think of these four cases as reflecting the following four algebraic identities:

$$\begin{aligned} 2x + 2y &= 2(x + y) \\ 2x + (2y + 1) &= 2(x + y) + 1 \\ (2x + 1) + 2y &= 2(x + y) + 1 \\ (2x + 1) + (2y + 1) &= 2(x + y + 1) \end{aligned}$$

Alternatively, you can think of the four cases as if you were doing long addition of binary numbers, and had to decide the next step based on the least significant bits of the two numbers being added. Only in the fourth case, when both of the least significant bits equal one, do we need to *carry a one* to the subsequent bits, reflected in the call to `plusone` used in that case.

One of the two recitation exercises was to implement multiplication as well. We can do this by taking advantage of the following two identities:

$$\begin{aligned} 2x \cdot y &= 2(x \cdot y) \\ (2x + 1) \cdot y &= y + 2(x \cdot y) \end{aligned}$$

In our definition of `times`, we *could* pattern match over both arguments for a total of four cases again, but using these two identities we can define `times` easily by pattern matching over just one argument, like this:

```
times :: BinInt -> BinInt -> BinInt
times (Zero b1) b2 = Zero (times b1 b2)
times (One b1) b2 = plus b2 (Zero (times b1 b2))
```

we can test it like this:

```
ghci> times six six
...00000000000000100100
```

...which is in fact the correct binary representation of 36.

The really interesting thing about this way of representing the natural numbers is that we've inadvertently created a number system that contains a lot more numbers than we intented. Consider the following definition, which defines a perfectly valid element of `BinInt`:

```
wtf = One wtf
```

As a list of digits, this would be an infinite list of ones. But of course, there's no actual natural number which has infinitely many binary digits equal to one! So we've just built an element of our custom type that doesn't correspond to any natural number. And yet, we can still print the digits of `wtf`:

```
ghci> wtf
...11111111111111111111
```

Not only that, but we can *use the arithmetic function that we intended for natural numbers* to perform "arithmetic" with this "number-like object". After all, we defined `plusone`, `plus` and `times` for arbitrary elements of `BinInt`, with no stipulation that they correspond to actual natural numbers. Notice what happens if we try taking the successor of `wtf`:

```
ghci> plusone wtf
...00000000000000000000
```

It appears that the successor of `wtf` is 0! This makes sense in terms of how we defined `plusone`, actually: if you imagine adding 1 to `wtf` as if it were the binary representation of a real number using long addition, you can see that at each bit, we'd have to carry a 1 over to the next bit, and this carrying would propagate "all the way down" the number's digit sequence, leaving us only an infinite list of zeros. Having noticed that this number's successor is `zero`, we might want to give it a more meaningful name - perhaps `negone`?

The other lab exercise was to find additive inverses of various different natural numbers in `BinInt`, which we can do by examining how carrying occurs when we perform binary long addition with numbers other than 1. For instance, we have the following additive inverses of 5 and 7:

```
negfive = One (One (Zero negone))
negseven = One (Zero (Zero negone))
```

and we can verify that these are additive inverses of 5 and 7 by running `plus negfive (natToBin 5)` and `plus negfive (natToBin 7)`. As a matter of fact, the function `negative` can be implemented as follows:

```
negative :: BinInt -> BinInt
negative b = plus b (Zero (negative b))
```

Can you figure out why this works? (Here's a hint: if y is the additive inverse of x, then $y = x + 2y$.)

If you're looking for an extra challenge, you might consider the following questions:

- Can you find a *multiplicative* inverse of 3? That is, can you find a `b :: BinInt` such that `times b (natToBin 3)` gives 1?
- Can you find a multiplicative inverse of 5? How about 2?
- Does every `BinInt` have a multiplicative inverse? Can you write a `reciprocal :: BinInt -> BinInt` function? If not, which `BinInt`s *do* have multiplicative inverses?
- Can you solve the quadratic equation $2x^2 + x + 1 = 0$ for some `x :: BinInt`?