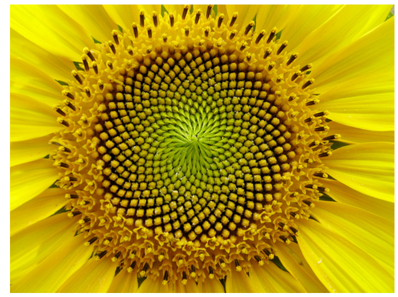
CIS 194: Homework 6

Due Monday, February 25

- Files you should submit: `Fibonacci.hs`

This week we learned about Haskell's *lazy evaluation*. This homework assignment will focus on one particular consequence of lazy evaluation, namely, the ability to work with infinite data structures.



Fibonacci numbers

The *Fibonacci numbers* F_n are defined as the sequence of integers, beginning with 0 and 1, where every integer in the sequence is the sum of the previous two. That is,

$$\begin{aligned} F_0 &= 0 \\ F_1 &= 1 \\ F_n &= F_{n-1} + F_{n-2} \qquad (n \geq 2) \end{aligned}$$

For example, the first fifteen Fibonacci numbers are

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, \ldots$$

It's quite likely that you've heard of the Fibonacci numbers before. The reason they're so famous probably has something to do with the simplicity of their definition combined with the astounding variety of ways that they show up in various areas of mathematics as well as art and nature.[1]

[1] Note that you may have seen a definition where $F_0 = F_1 = 1$. This definition is wrong. There are several reasons; here are two of the most compelling:

- If we extend the Fibonacci sequence *backwards* (using the appropriate subtraction), we find

  $$\ldots, -8, 5, -3, 2, -1, 1, 0, 1, 1, 2, 3, 5, 8 \ldots$$

  0 is the obvious center of this pattern, so if we let $F_0 = 0$ then F_n and F_{-n} are either equal or of opposite signs, depending on the parity of n. If $F_0 = 1$ then everything is off by two.
- If $F_0 = 0$ then we can prove the lovely theorem "If m evenly divides n if and only if F_m evenly divides F_n." If $F_0 = 1$ then we have to state this as "If m evenly divides n if and only if F_{m-1} evenly divides F_{n-1}." Ugh.

Exercise 1

Translate the above definition of Fibonacci numbers directly into a recursive function definition of type

```
fib :: Integer -> Integer
```

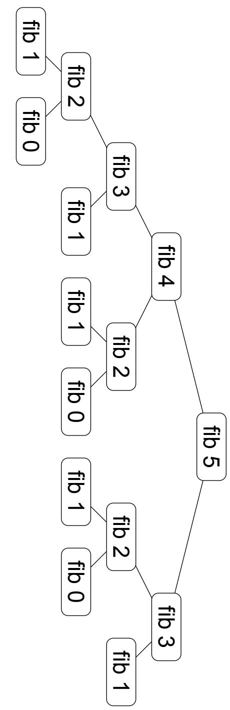
so that `fib n` computes the nth Fibonacci number F_n.

Now use `fib` to define the *infinite list* of all Fibonacci numbers,

```
fibs1 :: [Integer]
```

(*Hint*: You can write the list of all positive integers as `[0..]`.)

Try evaluating `fibs1` at the `ghci` prompt. You will probably get bored watching it after the first 30 or so Fibonacci numbers, because `fib` is ridiculously slow. Although it is a good way to *define* the Fibonacci numbers, it is not a very good way to *compute* them—in order to compute F_n it essentially ends up adding 1 to itself F_n times! For example, shown at right is the tree of recursive calls made by evaluating `fib 5`.



As you can see, it does a lot of repeated work. In the end, `fib` has running time $O(F_n)$, which (it turns out) is equivalent to $O(\varphi^n)$, where $\varphi = \frac{1+\sqrt{5}}{2}$ is the "golden ratio". That's right, the running time is *exponential* in n. What's more, all this work is also repeated from each element of the list `fibs1` to the next. Surely we can do better.

Exercise 2

When I said "we" in the previous sentence I actually meant "you". Your task for this exercise is to come up with more efficient implementation. Specifically, define the infinite list

```
fibs2 :: [Integer]
```

so that it has the same elements as `fibs1`, but computing the first n elements of `fibs2` requires only $O(n)$ addition operations. Be sure to use standard recursion pattern(s) from the `Prelude` as appropriate.

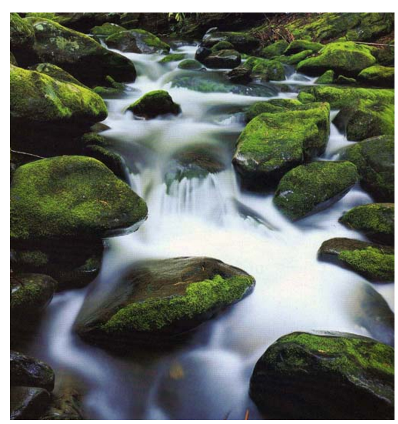
Of course there are several billion Haskell implementations of the Fibonacci numbers on the web, and I have no way to prevent you from looking at them; but you'll probably learn a lot more if you try to come up with something yourself first.

Streams

We can be more explicit about infinite lists by defining a type `Stream` representing lists that *must be* infinite. (The usual list type represents lists that *may* be infinite but may also have some finite length.)

In particular, streams are like lists but with *only* a "cons" constructor—whereas the list type has two constructors, `[]` (the empty list) and `(:)` (cons), there is no such thing as an *empty stream*. So a stream is simply defined as an element followed by a stream.

Exercise 3



- Define a data type of polymorphic streams, `Stream`.
- Write a function to convert a `Stream` to an infinite list,

  ```
  streamToList :: Stream a -> [a]
  ```

- To test your `Stream` functions in the succeeding exercises, it will be useful to have an instance of `Show` for `Streams`. However, if you put `deriving Show` after your definition of `Stream`, as one usually does, the resulting instance will try to print an *entire* `Stream`—which, of course, will never finish. Instead, you should make your own instance of `Show` for `Stream`,

```
instance Show a => Show (Stream a) where
  show ...
```

which works by showing only some prefix of a stream (say, the first 20 elements).

Hint: you may find your `streamToList` function useful.

Exercise 4

Let's create some simple tools for working with `Streams`.

- Write a function

```
streamRepeat :: a -> Stream a
```

which generates a stream containing infinitely many copies of the given element.

- Write a function

```
streamMap :: (a -> b) -> Stream a -> Stream b
```

which applies a function to every element of a `Stream`.

- Write a function

```
streamFromSeed :: (a -> a) -> a -> Stream a
```

which generates a `Stream` from a "seed" of type `a`, which is the first element of the stream, and an "unfolding rule" of type `a -> a` which specifies how to transform the seed into a new seed, to be used for generating the rest of the stream.

Exercise 5

Now that we have some tools for working with streams, let's create a few:

- Define the stream

```
nats :: Stream Integer
```

which contains the infinite list of natural numbers 0, 1, 2, . . .

- Define the stream

```
ruler :: Stream Integer
```

which corresponds to the *ruler function*

$$0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, 4, \ldots$$

where the nth element in the stream (assuming the first element corresponds to $n = 1$) is the largest power of 2 which evenly divides n.

Hint: define a function `interleaveStreams` which alternates the elements from two streams. Can you use this function to implement `ruler` in a clever way that does not have to do any divisibility testing?

Fibonacci numbers via generating functions (extra credit)

This section is optional but *very cool*, so if you have time I hope you will try it. We will use streams of `Integers` to compute the Fibonacci numbers in an astounding way.

The essential idea is to work with *generating functions* of the form

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n + \ldots$$

where x is just a "formal parameter" (that is, we will never actually substitute any values for x; we just use it as a placeholder) and all the coefficients a_i are integers. We will store the coefficients $a_0, a_1, a_2, \ldots$ in a `Stream Integer`.

Exercise 6 (Optional)

- First, define

  ```
  x :: Stream Integer
  ```

  by noting that $x = 0 + 1x + 0x^2 + 0x^3 + \ldots$.

- Define an instance of the `Num` type class for `Stream Integer`. Here's what should go in your `Num` instance:

  Note that you will have to add `{-# LANGUAGE FlexibleInstances #-}` to the top of your `.hs` file in order for this instance to be allowed.

  - You should implement the `fromInteger` function. Note that $n = n + 0x + 0x^2 + 0x^3 + \ldots$.
  - You should implement `negate`: to negate a generating function, negate all its coefficients.
  - You should implement `(+)`, which works like you would expect: $(a_0 + a_1 x + a_2 x^2 + \ldots) + (b_0 + b_1 x + b_2 x^2 + \ldots) = (a_0 + b_0) + (a_1 + b_1)x + (a_2 + b_2)x^2 + \ldots$
  - Multiplication is a bit trickier. Suppose $A = a_0 + xA'$ and $B = b_0 + xB'$ are two generating functions we wish to multiply. We reason as follows:

    $$\begin{aligned} AB &= (a_0 + xA')B \\ &= a_0 B + xA'B \\ &= a_0(b_0 + xB') + xA'B \\ &= a_0 b_0 + x(a_0 B' + A'B) \end{aligned}$$

    That is, the first element of the product AB is the product of the first elements, $a_0 b_0$; the remainder of the coefficient stream (the part after the x) is formed by multiplying every element in B' (that is, the tail of B) by a_0, and to this adding the result of multiplying A' (the tail of A) by B.

Note that there are a few methods of the `Num` class I have not told you to implement, such as `abs` and `signum`. `ghc` will complain that you haven't defined them, but don't worry about it. We won't need those methods. (To turn off these warnings you can add

```
{-# OPTIONS_GHC -fno-warn-missing-methods #-}
```

to the top of your file.)

If you have implemented the above correctly, you should be able to evaluate things at the `ghci` prompt such as

```
*Main> x^4
*Main> (1 + x)^5
*Main> (x^2 + x + 3) * (x - 5)
```

- The penultimate step is to implement an instance of the `Fractional` class for `Stream Integer`. Here the important method to define is division, `(/)`. I won't bother deriving it (though it isn't hard), but it turns out that if $A = a_0 + xA'$ and $B = b_0 + xB'$, then $A/B = Q$, where Q is defined as

$$Q = (a_0/b_0) + x((1/b_0)(A' - QB')).$$

That is, the first element of the result is a_0/b_0; the remainder is formed by computing $A' - QB'$ and dividing each of its elements by b_0.

Of course, in general, this operation might not result in a stream of `Integer`s. However, we will only be using this instance in cases where it does, so just use the `div` operation where appropriate.

- Consider representing the Fibonacci numbers using a generating function,

$$F(x) = F_0 + F_1 x + F_2 x^2 + F_3 x^3 + \dots$$

Notice that $x + xF(x) + x^2 F(x) = F(x)$:

$$\begin{array}{ccccccccccc} & & x & & & & & & & & \\ & & F_0 x & + & F_1 x^2 & + & F_2 x^3 & + & F_3 x^4 & + & \dots \\ & & & & F_0 x^2 & + & F_1 x^3 & + & F_2 x^4 & + & \dots \\ \hline 0 & + & x & + & F_2 x^2 & + & F_3 x^3 & + & F_4 x^4 & + & \dots \end{array}$$

Thus $x = F(x) - xF(x) - x^2 F(x)$, and solving for $F(x)$ we find that

$$F(x) = \frac{x}{1 - x - x^2}.$$

Translate this into an (amazing, totally sweet) definition

```
fibs3 :: Stream Integer
```

Fibonacci numbers via matrices (extra credit)

It turns out that it is possible to compute the nth Fibonacci number with only $O(\log n)$ (arbitrary-precision) arithmetic operations. This section explains one way to do it.

Consider the 2×2 matrix $\mathbf{F}$ defined by

$$\mathbf{F} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}.$$

Notice what happens when we take successive powers of $\mathbf{F}$ (see `http://en.wikipedia.org/wiki/Matrix_multiplication` if you forget how matrix multiplication works):

$$\mathbf{F}^2 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 1 \cdot 1 + 1 \cdot 1 & 1 \cdot 1 + 1 \cdot 0 \\ 1 \cdot 1 + 0 \cdot 1 & 1 \cdot 1 + 0 \cdot 0 \end{bmatrix} = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix}$$

$$\mathbf{F}^3 = \begin{bmatrix} 2 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix}$$

$$\mathbf{F}^4 = \begin{bmatrix} 3 & 2 \\ 2 & 1 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix}$$

$$\mathbf{F}^5 = \begin{bmatrix} 5 & 3 \\ 3 & 2 \end{bmatrix} \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} 8 & 5 \\ 5 & 3 \end{bmatrix}$$

Curious! At this point we might well conjecture that Fibonacci numbers are involved, namely, that

$$\mathbf{F}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

for all $n \geq 1$. Indeed, this is not hard to prove by induction on n.

The point is that exponentiation can be implemented in logarithmic time using a *binary exponentiation* algorithm. The idea is that to compute x^n, instead of iteratively doing n multiplications of x, we compute

$$x^n = \begin{cases} (x^{n/2})^2 & n \text{ even} \\ x \cdot (x^{(n-1)/2})^2 & n \text{ odd} \end{cases}$$

where $x^{n/2}$ and $x^{(n-1)/2}$ are recursively computed by the same method. Since we approximately divide n in half at every iteration, this method requires only $O(\log n)$ multiplications.

The punchline is that Haskell's exponentiation operator `(^)` *already uses* this algorithm, so we don't even have to code it ourselves!

Exercise 7 (Optional)

- Create a type `Matrix` which represents 2×2 matrices of `Integer`s.
- Make an instance of the `Num` type class for `Matrix`. In fact, you only have to implement the `(*)` method, since that is the only one we will use. (If you want to play around with matrix operations a bit more, you can implement `fromInteger`, `negate`, and `(+)` as well.)
- We now get fast (logarithmic time) matrix exponentiation for free, since `(^)` is implemented using a binary exponentiation algorithm in terms of `(*)`. Write a function

```
fib4 :: Integer -> Integer
```

  which computes the nth Fibonacci number by raising $\mathbf{F}$ to the nth power and projecting out F_n (you will also need a special case for zero). Try computing the one millionth or even ten millionth Fibonacci number.

Don't worry about the warnings telling you that you have not implemented the other methods. (If you want to disable the warnings you can add

```
{-# OPTIONS_GHC -fno-warn-missing-methods #-}
```

to the top of your file.)

On my computer the millionth Fibonacci number takes only 0.32 seconds to compute but more than four seconds to print on the screen—after all, it has just over two hundred thousand digits.