CS3200: Programming Languages
Homework 8: data types and functors in Haskell

Required Problems

1. For this problem, we're doing to define a new type of list, called a Stream. Like our Set class on the last homework, this will duplicate a lot that you can do with lists, but will enforce that any Stream must be infinite. (The usual list type represents lists that may be infinite but may also have some finite length.)

   In particular, in our implementation, streams will behave much like lists, but will only have a cons constructor whereas the list type has two constructors, [] (the empty list) and (:) (cons), in our setup there is no such thing as an empty stream. So a stream is simply defined as an element followed by a stream:

   data Stream a = Cons a (Stream a)

   (a) Write a function to convert a Stream to an infinite list:

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

   (b) To test your Stream functions in the succeeding exercises, it will be useful to have an instance of Show for Stream s. 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, 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).

   (c) Write a function:

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

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

   (d) Make your Stream be an instance of Functor (as we did with binary trees in class), so that it applies an input function to every element in the Stream.

   (e) Write a function

   ```
   streamIterate :: (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 that is a function taking an element of type a to another element of type a which specifies how to transform the seed into a new seed, to be used for generating the rest of the stream.

   Example:

   ```
   streamIterate ('x' :) "o" == ["o", "xo", "xxo", "xxxo", "xxxxo", ...
   ```

   (f) Write a function

   ```
   streamInterleave :: Stream a -> Stream a -> Stream a
   ```

which interleaves the elements from 2 Streams. You will want streamInterleave to be lazy in its second parameter. This means that you should not deconstruct the second Stream in the function.

Example:

```
streamInterleave (streamRepeat 0) (streamRepeat 1) ==
    [0, 1, 0, 1, 0, 1, ...
```

(g) Now that we have some tools for working with streams, lets create a few:

Define the stream

```
nats :: Stream Integer
```

which contains the infinite list of natural numbers 00, 11, 22,...

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,...
```

where the n^{th} element in the stream (assuming the first element corresponds to n=1n=1) is the largest power of 22 which evenly divides n.

Hint: Try to find the a pattern that ruler follows. Use streamInterleave to implement ruler in a clever way that does not have to do any divisibility testing. Do you see why you had to make streamInterleave lazy in its second parameter?

2. Extra credit: Modify your Stream class so it is also of type Applicative, and define ¡*¿ appropriately.