1 (*). Exercises from chapter 9-10 of The Craft of Functional Programming

- **(*)** Define the `length` function using `map` and `sum`.
- **(*)** What does `map (+1) (map (+1) xs)` do? Can you conclude anything in general about properties of `map f (map g xs)`, where `f` and `g` are arbitrary functions?
- Give the type of, and define the function `iter` so that
- `iter n f x = f (f (... (f x)))`

  where `f` occurs `n` times on the right-hand side of the equation. For instance, we should have

  ```
  iter 3 f x = f (f (f x))
  ```

  and `iter 0 f x` should return `x`.

- What is the type and effect of the following function?
- `\n -> iter n succ`

  `succ` is the successor function, which increases a value by one:

  ```
  Prelude> succ 33
  34
  ```

- **(*)** How would you define the sum of the squares of the natural numbers 1 to `n` using `map` and `foldr`?
- How does the function
- `mystery xs = foldr (++) [] (map sing xs)`
- `  where`
- `    sing x = [x]`

  behave?

- **(*)** If `id` is the polymorphic identity function, defined by `id x = x`, explain the behavior of the expressions
- `(id . f)  (f . id)  (id f)`

  If `f` is of type `Int -> Bool`, at what instance of its most general type `a -> a` is `id` used in each case?

- Define a function `composeList` which composes a list of functions into a single function. You should give the type of `composeList`, and explain why the function has this type. What is the effect of your function on an empty list of functions?
- **(*)** Define the function
- `flip :: (a -> b -> c) -> (b -> a -> c)`

  which reverses the order in which its function argument takes its arguments.

  The following example shows the effect of `flip`:

  ```
  Prelude> flip div 3 100
  33
  ```

2 (*). List Comprehensions and Higher-Order Functions

Can you rewrite the following list comprehensions using the higher-order functions `map` and `filter`? You might need the function concat too.

1. `[ x+1 | x <- xs ]`
2. `[ x+y | x <- xs, y <-ys ]`
3. `[ x+2 | x <- xs, x > 3 ]`
4. `[ x+3 | (x,_) <- xys ]`
5. `[ x+4 | (x,y) <- xys, x+y < 5 ]`
6. `[ x+5 | Just x <- mxs ]`

Can you it the other way around? I.e. rewrite the following expressions as list comprehensions.

1. `map (+3) xs`
2. `filter (>7) xs`
3. `concat (map (\x -> map (\y -> (x,y)) ys) xs)`
4. `filter (>3) (map (\(x,y) -> x+y) xys)`

3 (*). Generating Lists

Sometimes we want to generate lists of a certain length.

A. Write a generator

listOfLength :: Integer -> Gen a -> Gen [a]

such that listOf n g generates a list of n elements, where each element is generated by g. What property would you write to test that your generator behaves as it should?

B. Now use listOf to write a generator that generates pairs of lists of the same, random, length.

C.Take a look at the standard Haskell functions zip and unzip:

```
zip :: [a] -> [b] -> [(a,b)]
unzip :: [(a,b)] -> ([a],[b])
```

Write down two properties for these; one that says that zip is the inverse of unzip, and one that unzip is the inverse of zip. Note that unzip is not always the inverse of zip, so you need a condition! Could you make use of the generator you just defined?

Hint: make a datatype

```
data TwoSameLengthLists a = SameLength [a] [a]
```

and use your generator for part B. to make a generator for the above type. Then, make the above type an instance of Arbitrary. Finally, you can use it to write your property.