Function Composition

In function composition you have two functions, and you build a new function from them by applying one function first, then applying the second function to the result of the first. This is so common that it has a simple notation, namely that of a single dot:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = \x -> f (g x)
-- Can also write as
f . g x = f (g x)
```

With this definition in mind, we could have defined append earlier as simply append=(.).

As a trivial example of function composition, consider a linear function:

```
f x = 3 * x + 1
```

We can think of this function as a composition of two functions: We first multiply by 3, then we add 1. We can think of these two as sections. So a rather convoluted way of writing the above would have been:

```
f = (+ 1) . (3 *)
```

This doesn't really show the significance of this function, but it will hopefully help show a bit of the mechanics.

We will follow the book convention and also define an operator >.> which performs function composition but in the opposite order:

```
infixl 9 >.>     -- Specifies it to be left-associative. 9 is its priority.
(>.>) :: (a -> b) -> (b -> c) -> (a -> c)
f . g = \x -> g (f x)
```

For operator precedence consult this report[1].

As another example, suppose we wanted to do the following: Write a function that given a list of numbers squares them then adds the results. We could write this in a number of different ways:

```
sumSquares :: Num t => [t] -> t
-- List comprehension
sumSquares xs = sum [x^2 | x <- xs]
-- Using map
sumSquares xs = sum (map (^2) xs)
-- "Point-free" using function composition
sumSquares = sum . map (^2)
```

This type of programming has a certain elegance to it: We define a function via composition of known functions, without ever having to mention the function's actual parameter. This is often called **point-free style**, some times also called **tacit programming**.

It is also at times hard to read, so use it with caution.

[1] https://www.haskell.org/onlinereport/decls.html#fixity

Practice

1. Give a point-free definition of a function that is given a list of numbers finds the maximum of the first 3 elements.

2. Give a point-free definition of a function that given a list of lists returns a list containing the heads of those lists (you do not need function composition here, this is simply a curried function with only some parameters provided).

3. Consider the following "string scrambling" process: For each character, convert it to an integer via the method `ord :: Char -> Int`, then double that integer, finally convert the corresponding integer back to a character via `chr :: Int -> Char`. Do this for each character in the string. Write this function of type `String -> String` that performs this combined task. You can do this as a partially applied `map`, where the provided function is point-free and composes the functions `ord, chr` together with a section for the multiplication.