Function Chaining

It is very common in a program to apply multiple functions one after another. For example, to apply `f`, `g` and `h` to a value `x`, one could write:

```
myFunc x = h (g (f x))
```

These parentheses become cumbersome very quickly. The solution to this, is to introduce a higher-order function '`.`' that "chains" two functions after each other:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Using this operator, we can now write `myFunc` much more elegantly:

```
myFunc x = (h . g . f) x
```

The '`.`' function is read as "after", which means that the right-hand side is read as "`h` after `g` after `f` applied to `x`". This means that we apply `f` to `x`, apply `g` to the result, and finally apply `h` to this result. Note that the parentheses here are necessary, as `h . g . f x` is actually parsed as `h . g . (f x)`, which means something completely different.

Intermediate Haskell programmers don't like to write parentheses, so they have come up with a way to omit these parentheses. The solution is the `$`-function:

```
($) :: (a -> b) -> a -> b
```

This function seems completely pointless as it just represents function application. However, due to how it is parsed, this operator can separate the function and argument without the need for parentheses. We can now rewrite `myFunc` to:

```
myFunc x = h . g . f $ x
```

Notice that the function can be defined by just chaining `f`, `g` and `h` together. The argument `x` is now redundant. Thus, the function `myFunc` can be defined as:

```
myFunc = h . g . f
```

That is, by evaluating `h` after `g` after `f`.

- Write a function `applyAll :: [a -> a] -> a -> a` which applies a list of functions to a value, one after the other.

  ```
  Main> applyAll [(+ 2),(* 2)] 5
  12


  Main> applyAll [(: []). sum, filter odd] [1..8]
  [16]
  ```

- Write a function `applyTimes :: Int -> (a -> a) -> a -> a`, which applies a function a given number of times to a value. You should use only two explicit arguments in your code.

```
Main> applyTimes 5 (+ 1) 0
5

Main> applyTimes 4 (++ "i") "W"
"Wiiii"

Main> applyTimes 0 (error "Error!") 3.14
3.14
```

- As a variation on this theme, write a function `applyMultipleFuncs :: a -> [a -> b] -> [b]`, which takes an argument and a list of functions, and applies these functions to the given argument.

```
Main > applyMultipleFuncs 2 [(*2), (*3), (+6)]
[4,6,8]
```