Higher order functions : take fns. as parameters and return functions as return values.

↓

They are very important to the way computations are done in Haskell, i.e. by defining things rather than changing states of objects.

Curried functions → Every function officially takes one parameter. All functions that accepted several parameters have been Curried functions.

↓

max 4 5 : creates a fn. that takes a parameter and returns either 4 or that parameter, depending on which is bigger. Then 5 is applied to this function, and the desired result is produced

↓

(max 4) 5 ≡ max 4 5

Space between two things is function application.

↓

more like an operator with the highest precedence.

:t max

max :: (Ord a) ⇒ a → a → a

max :: (Ord a) ⇒ a → (a → a)

↓

max takes an a and returns a function that takes an a and returns an a.

↓

If we call a function with too few parameters, we get back a partially applied function, i.e. a fn. that takes addn. parameters as many we left out to run.

This is a neat way to create functions on the fly, so we can pass them to another fn. or to seed them with some data.

↓

```
multThree :: (Num a) => a -> a -> a -> a
multThree x y z = x * y * z
```

↓

((multThree 3) 5) 9 : first 3 is applied to multThree, returns a partially applied fn, which now accepts two parameters., gets 5, returns a partially applied fn, which now accepts one parameter, and multiplies the parameter with 15, and then applies it with 9. The function is now completely applied and returns 135.

↓

multThree :: (Num a) => (a) → (a → (a → a))

(a) → parameter

(a → (a → a)) → returns a fn. that takes a parameter and returns a fn. that takes a parameter and returns a value

(a → a) → a → a

↓

this type of type declaration is necessary for higher order functions.

function parameter

applyTwice :: (a → a) → a → (a) → return type

applyTwice f x = f (f x)

↳ returns a normal parameter which is used by the function again.

↓

i.e. we use the parameter f as a function, applying x to it be separating them with a space.

↓

applyTwice (3 :) [1] → [3, 3, 1]

↙ function ↘ argument

- Implementing zipWith (part of Haskell standard library)

↓

takes a fn. and two lists as parameters and then joins the lists by applying the fn. between corresponding elements.

```
zipWith' :: (a → b → c) → [a] → [b] → [c]
zipWith' _ [] _ = []
zipWith' _ _ [] = []
zipWith' f (x:xs) (y:ys) = (f x y) : zipWith' f xs ys
```

- Implement flip (takes a fn., and returns a fn. that is like our original fn, only the first two arguments are flipped)

```
flip' :: (a → b → c) → (b → a → c)
flip' f = g
    where g x y = f y x
```

```
flip' f x y = f y x
```

→ **Maps and filters** : map takes a fn. and a list and applies that function to every element in the list, producing a new list

```
map :: (a → b) → [a] → [b]
map _ [] = []
map f (x:xs) = f x : map f xs
```

filter is a fn. that takes a predicate and a list, and then returns a list that satisfy that predicate

```
filter :: (a → Bool) → [a] → [a]
filter _ [] = []
filter f (x:xs)
  | f x = x : filter f xs
  | otherwise = filter f xs
```

Mapping & filtering is the bread and butter of every functional programmer's toolbox

↓

doesn't matter how it is accomplished.

⇒ quicksort using filter

```
quicksort :: (Ord a) ⇒ [a] → [a]
quicksort [] = []
quicksort (x:xs) =
    let smallsorted = quicksort $ filter (<=x) xs
        largesorted = quicksort $ filter (>x) xs
    in smallsorted ++ [x] ++ largesorted.
```

(•) find the largest no. under 100,000 that is divisble by 3829

```
largestDivisble :: (Integral a) => a
largestDivisble = head (filter p [100000, 99999, ..])
    where p x = x `mod` 3829 == 0
```

head → largest number

Laziness acts here, evaluation stops once the suitable head is found.

(•) takeWhile function : takes a predicate and a list and then goes from the beginning of the list and returns the elements while the predicate holds true.

↓

once an element is found, for which the predicate doesn't hold, it stops.

↓

```
takeWhile (/= ' ') "elephants know how to party"
```

(•) sum of all odd squares that are smaller than 10000.

```
sum $ takeWhile (<10000) $ filter odd $ map (^2) [1..]
```

↓ can also be accomplished using list comprehension

```
sum $ takeWhile (<10000) [n^2 | n <- [1..], odd (n^2)]
```

Lambdas : basically anonymous functions that are used because we need some functions only once.

↓

Normally made with the sole purpose of passing it to a higher order function

↓

\ followed by parameters. After that comes a → and the function body. This is surrounded by parantheses

↓

```
numLongChains :: Int
numLongChains = length (filter (\xs -> length xs > 15)
                               (map chain [1..100]))
```

```
zipWith (\a b -> (a * 30 + 3) / b) [1] [2]
```

↓

another lambda fn. with multiple parameters.

→ function composition → can be done with the . function, which is defined like so:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
f . g = \x -> f (g x)
```

↓

example `map (\xs -> negate (sum (tail xs)))`

`[[1..5],[3..6],[1..7]]`

`map (negate . sum . tail) [[1..5],[3..6],[1..7]]`