Chapter 8 - Recursion

*) Recursion is defining a function in terms of itself via self-referential expressions, i.e. the function continues to call itself and repeat its behaviour until some condition is met to return a result.

*) Being able to write recursive functions is essential for Turing completeness. Lambda calculus does not seem on the surface to have any means of recursion, because of the anonymity of expressions. But, we use a combinator - Y combinator or fixed-point combinator to write recursive functions in the lambda calculus. Haskell has native recursion ability based on the same principle as the Y combinator.

→ Factorial

```
factorial :: Integer → Integer
factorial x
     | x <= 0 = 1
     | otherwise = (*) x $ factorial $ x-1
```

→ Recursion can be seen as a special case of function composition, where rather than passing the result of the first fn. to a different fn., we pass it to the same fn, till the base case is hit. Recursion is self-referential composition. We apply a fn. to an argument, then pass that result as as argument to a second application of the same function & so on.

→ Bottom : term used in Haskell to refer to computations that donot successfully result in a value. The two main varieties of bottom are computations that failed with an error or those that failed to terminate.

*) maybe datatype (making a partial fn into a total one)

data maybe a = Nothing | Just a → allows us to take an argument and allows us to return the data we're wanting

Nothing → one way of saying that there is no result or data from the function without hitting bottom

•) makes all uses of nil values and most uses of bottom unnecessary.

•) for example

```
f :: Bool → Maybe Int      → type defn change
f False = Just 0           → needs to be wrapped in Just data constructor
f  _  = Nothing
```

→ Fibonacci Numbers (returning the xth member of the fibonacci series)

```
fibonacci :: Integer → Integer
        or
fibonacci :: Integral a ⇒ a → a
fibonacci 0 = 0
fibonacci 1 = 1
fibonacci x = fibonacci (x-1) + fibonacci (x-2)
```

→ Integral division from scratch

type synonym or alias.

```
type Numerator = Integer
type Denominator = Integer
type Quotient = Integer

dividedBy :: Numerator -> Denominator -> Quotient
```

something that can be done to make the code more readable

```
dividedBy :: Integral a => a -> a -> (a,a)
dividedBy num denom = go num denom 0
  where go n d count
          | n < d = (count, n)
          | otherwise =
              go (n-d) d (count+1)
```

quotient

remainder

changed the type signature to make it more polymorphic, and also to return a tuple

allows us to define a function via a where-clause that can accept more arguments than the top-level function dividedBy does.

↓

Here the top-level fn. takes two arguments, num & denom, but we need a third argument in order to keep track of how many times we do subtraction. That argument is count & is defined by a starting value of zero & is incremented by 1 everytime the otherwise case is invoked.

↓

It is not significant that we changed the argument names from num & denom to n and d. The go function has already been applied to them in the definition of dividedBy so that the num, denom & 0 are bound to n, d & count in the where clause.

```
dividedBy :: Num a => a -> a                    x y

dividedBy x y
                                                2 / 3
     | x < y = 0
     | x == y = 1
     | x > y = 1 + dividedBy (x-y) y
```

["one", "two", "three", "four"]

1 2 3 4 5 6 → ["one", "two", "three"

---

intersperse :: a → [a] → [a] : takes an element and a list & puts that element between the elements of the list