Chapter 4 - Defining Functions

→ Conditional expressions as in other languages

Ex:
```
abs :: Int → Int
abs n = if n ≥ 0 then n else -n
```
→ No need of return

→ Can also be nested

Ex:
```
signum :: Int → Int
signum n = if n < 0 then -1 else
             if n == 0 then 0 else 1
```

→ Must always have an else branch in Haskell

→ Guarded Equations

Ex:
```
abs n | n ≥ 0      = n
      | otherwise  = -n
```
→ Similar to how it's written in math

→ Can make multiple conditions easier to read

→ Catch all condition otherwise is defined to be true.

→ Pattern matching on it's arguments

Ex:
```
not :: Bool → Bool
not False = True
not True  = False
```

→ The underscore(_) is a wildcard pattern that matches any argument value

→ Patterns cannot repeat variables, order of the equations matters

→ Lambda expressions these functions can be constructed without naming

Ex:
```
λx → x + x
```
→ Nameless functions

↳ Backslash on keyboard

→ Lambda calculus which is what Haskell is based on

→ Can used to give an alternate way to understand currying functions

Ex:
```
add :: Int → Int → Int
add x y = x + y
```

↓ The complier will transform into this

means
```
add :: Int → (Int → Int)
add = λx → (λy → x + y)
```

→ Helps better to understand curried functions

→ Operator between two parameters can be converted into a curried function written before

Ex:
```
1 + 2
(+) 1 2
```
→ Both give 3

Allows you to do this:
```
(+1) 2 = 3
(+2) 1 = 3
```
↳ Can put arguments inside it or leave it empty

→ Helps with writing concise programs.

Note exercises done in definingFunction Exercise.hs