Chapter 5, "Types"

Background

- Haskell's basically the *Lambda Calculus* with *types*.
- *Types* layer over **LC** to make sure evaluation doesn't *get stuck*

Summary (1/4)

- `:type 't'` - query a type in the REPL
- `(->) a b` - the *arrow* or *function* type (functions from `a` to `b`). `a` and `b` are *type variables*.
- `(+) :: Num a => a -> a -> a` - *typeclass constrained* type variables.
  (Note: also valid are multiple constraints: `(Num a, Num b) => a -> b -> a`)

Summary (2/4)

- Arguments to functions are *curried*, or passed one-at-a-time.

  ```
  (+) :: Num a => a -> a -> a
  ```

  is really

  ```
  (+) :: Num a => a -> (a -> a)
  ```

- Partial application

  ```
  addStuff :: Integer -> Integer -> Integer
  addStuff a b = a + b + 5
  let addTen = addStuff 5
  ```

Summary (3/4)

- Explicit `curry` and `uncurry` functions

```
curry   :: ((a, b) -> c) -> (a -> b -> c)
uncurry :: (a -> b -> c) -> ((a, b) -> c)
```

- Operator section

```
(+1) :: Num a => a -> a
```

argument order matters!

`(2^)` means "2 to the power of *arg*"

`(^2)` means "*arg* squared"

Summary (4/4)

- "parametric polymorphism" - a type variable in a signature means it can be *any type*.
- `id :: a -> a` - identity function, because of *parametricity*, we know that all this function can do is return its argument (it can't have any type-dependent behavior)
- "type inference" - Haskell determines the type of expressions automatically (when possible)

Next time

Chapter 6, "Typeclasses"
Exercise template

Make this source file compile and make the tests pass.

https://gist.github.com/twopoint718/1c46ef58ee2dbc41ea186035938e97d2