Day 1: Types and functions

A Quick Primer on Reading Type Signatures

All data has a type. All of it.

```
"julie" :: [Char]
```

```
5       :: Num a => a
```

It's not always what you expect.

A Quick Primer on Reading Type Signatures

```
id    :: a -> a

const :: a -> b -> a
```

Variables with the same name must have the same type. Variables with different names might be different types, or they might not. We leave open the possibility that they will be different.

A Quick Primer on Reading Type Signatures

When an argument *must* be a function, then that function's type has to have parentheses around it.

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

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

We'll talk about why in more detail in a bit.

A Quick Primer on Reading Type Signatures

Constraints are important. Constraints give us life.

```
(+)  :: Num    a => a -> a -> a
(<>) :: Monoid m => m -> m -> m
```

It would be just as well to write

```
(<>)  :: Monoid a => a -> a -> a
```

But variable names *sometimes* serve as a mnemonic.

A Quick Primer on Reading Type Signatures

Some things get complicated, but it's ok. It's like learning to read a new language: there are symbols that take getting used to, but you practice and soon you can.

```
sequenceA :: (Applicative f, Traversable t) => t (f a) -> f
foldMap   :: (Foldable t, Monoid m)         => (a -> m) ->
```

A Quick Primer on Reading Type Signatures

Importantly, we can compare type signatures and see similarities that help us learn new things.

```
fmap  :: Functor f     =>   (a -> b)   -> f a -> f b
(<*>) :: Applicative f => f (a -> b)   -> f a -> f b
(=<<) :: Monad m       =>   (a -> m b) -> m a -> m b
```

even when they seem quite different at first

```
(.)   :: (b -> c) -> (a -> b) -> a     -> c
dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
```

There's lots more of that here:
https://chris-martin.github.io/haskell-aligned/

Polymorphism

Two kinds of polymorphism: parametric and constrained.

Parametric:

```
id :: a -> a
```

Constrained:

```
abs :: Num a => a -> a
```

Parametricity

```
id :: a -> a
```

An unconstrained type variable can represent any type in the universe.

Parametricity

And the function must have the exact same behavior for any possible type that ends up in that slot.

Parametricity

What can you do to an `Int` and a `String` and also a function that are exactly the same?

There are lots of things we can do with a value of a type once we know the type; there are very few things we can do that will work equally correctly for *any* type.

Constraints

Most operations are associated with specific types or a constrained set of types.

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

Could all these as be the same type?

Constraints

```
(+)  :: Num    a => a -> a -> a
(<>) :: Monoid m => m -> m -> m
```

Could the `a` and the `m` be the same type?

Polymorphism

Compare:

```
a -> a
Num a => a -> a
[a] -> [a]
Maybe a -> Maybe a
Bool -> Bool
```

The Type System

Emphasis on *system*.

Type system

Types and typeclasses form structures analogous to *algebraic structures*.

Algebras are sets plus operations over those sets that satisfy some laws.

Algebras

Let's talk about a set – or type – that we're mostly familiar with: integers.

They have a couple of operations over them that behave similarly: addition and multiplication.

Addition and multiplication are:

- binary
- associative
- have identity elements

Algebras

We can ignore the differences between addition and multiplication, abstracting away from the implementation, and say that

Integers are a monoid.

They form a monoid under addition and multiplication.

Aside

(so they're kinda two monoids)

(but we're abstracting so that doesn't matter!)

Algebras

We could call that operation something like `addMult` perhaps. It would have a type like:

```
addMult :: Integral a => a -> a -> a
```

That gives us a binary operation, but we need an identity!

Aside

(the associativity law that should hold for that operation can – and should! –be tested for but is not enforced by the compiler ¯_( )_/¯)

Algebras

So `addMult` isn't enough by itself, to make the algebra with integers. It is polymorphic enough to define either operation, but it needs an identity.

Now we have a pair of things:

```
addMult :: Integral a => a -> a -> a
addId   :: Integral a => a
```

Or, really, three things, because multiplication also needs an identity value.

```
multId  :: Integral a => a
```

Now we have a typeclass, a set of operations that we can define for any type that is *like integers*.

Algebras

We would write an *instance* for different types: `Integer`, `Int`, etc., that defined implementations for addition and multiplication and their identity values, but that creates a problem.

Algebras

Addition and multiplication don't *have* the same implementation; they have the same *type* but not at all the same implementation.

Algebras

We'll go about this another way.

We'll give our operation an even more general name and only one identity method and but give each implementation its own type name!

Algebras

```
class Monoid a where
  mempty  :: a
  mappend :: a -> a -> a
```

Please note the infix operator for `mappend` is <>.

Also note what this tells us, the things this *can't* do.