1 What are Monoids?

I feel it in my folds
I feel it in my toes
Monoids are all around me
And so the feeling grows

Multiplying anything by one does effectively nothing: the number remains unchanged. Likewise $+0$ is also equivalent to the identity. You could also $+\!\!+$ the empty list onto something to no effect, your time would be better spent making yourself a cup of tea. Monoids encapsulate this pattern of behaviour that can be found almost everywhere across most data types and sets. You should already be accustomed to coming up with such pairs of operation (e.g. $+$) and identity (e.g. 0) for a data types through folds. Recall $sum = foldr\ (+)\ 0$.

A monoid is made up of three parts: a set, an operation ($\oplus$), and an identity value (ø). For the more mathematical among you this may remind you of groups and yes well done **gold star**! Monoids are just groups without an inverse.

See if you can match the following jumbled monoids!

Sets: Integers, Integers, Booleans, Sets, $[a]$, Booleans, functions
Operations: $\circ$, $\wedge$, $+\!\!+$, $\vee$, $\cup$, $+$, x
id: True, 0, [], id, 1, ø, False

Answer 1 *Here are all the matched monoids:*

Set	$\oplus$	ø
Integers	x	*1*
Integers	$+$	*0*
Booleans	$\wedge$	*False*
Booleans	$\vee$	*True*
Sets	$\cup$	ø
functions	$\circ$	*id*
$[a]$	$+\!\!+$	[]

2 Monoid Laws

As with what seems like all things in this functional land, monoids have some laws. More precisely there are three monoid laws. For something to be a monoid these laws must also be passed. It is a not enough to just provide a set, an operation and an identity at random.

Enter official mathsy table of laws:

1. Left identity $\quad ø \oplus x = x$
2. Right identity $\quad x \oplus ø = x$
3. Associativity $\quad (x \oplus y) \oplus z = x \oplus (y \oplus z)$

To put it simply these laws ensure that the identity acts as, well, an identity (performing the operation with any input and the identity should not change the input) and that the operation is associative (it doesn't matter which order the parameters are). I mean it wouldn't make sense to have $\langle Integer, (+), 2\rangle$ as a monoid since adding 2 changes your input.

3 The Monoid Type Class

In Haskell there is a monoid type class!

```
class Monoid m where
  mempty :: m
```

$mappend :: m \to m \to m$
$mconcat \ :: [m] \to m$
-- *defining* $mconcat$ *is optional, since it has the following* **default**:
$mconcat = foldr \ mappend \ mempty$

So if you want to tell Haskell about a monoid you pick a type class (this will be your set) end then tell Haskell the exciting news of what the operation (mappend) and identity (mempty) are. The cool thing about this type class is the $mconcat$ default which takes advantage of how you can fold all monoids.

Unfortunately Haskell has no way of demanding that functions you define as part of a type class instance fulfil the associated laws so it is possible for evil villains that want to see the world burn to define monoid instances that don't uphold the laws.

4 One Instance to Rule Them All

As with all type classes you can only define one instance per data type. This sounds like this is not going to work out because data types can have more than one monoid. Take the integers: they have $\langle Integer, (+), 0\rangle$ AND $\langle Integer, (*), 1\rangle$! Don't worry though there is no need to pick favourites because there is a sneaky way around this problem.

Origin Story:

A long time ago, in the early days of Haskell an ***evil wizard*** *broke into the Glorious Glasgow Haskell Compiler (a dangerous endeavour considering that it is stored in the heart of the most dangerous city in the world: Glasgae), and dastardly as he was cast a powerful curse on the type system. His name was* Zenzike *and he poured all his malice into the curse, stripping all data types of their ability to choose what monoid they wanted to be an instance of.*

For a time the data types merely resigned themselves to their fate until one day an $Integer$ *named William Wallace decided he had had enough. He decided that it was time for* ***FREEDOOMMMMM!!!*** *He rallied the data types around him and together they brought down the Great and Terrible* Zenzike*! Unfortunately the curse could not simply be undone. However they added the* **newtype** *keyword to the GHC. This liberated all the types, offering them a way around the curse. If they desired they could don* **newtype** *freeing them to choose their own instances. Not only for monoids, but for all type classes.*

And thus, all was well in the Glorious land of Haskell again.
THE END

Ooooookay now that I have set the scene (beautifully if I do say so myself) let's actually tell you how to do the thing. Let's say we define the monoid instance for $Integer$s thusly:

instance $Monoid \ Integer$ **where**
$mempty \ = 0$
$mappend = (+)$

But now I want to also be able to product $Integer$s, instead of just sum them. Well an easy thing to do is disguise an $Integer$ as another new data type, that we will call $Pint$ because puns (credit to the great and powerful *Zenzike*) . To make this new data type I will use the prophesied **newtype** keyword to create something that to all intensive purposes is an $Integer$.

newtype $Pint = Pint \ Integer$ **deriving** $Show$

Literally look at it! It is an $Integer$, just an $Integer$ with a little flag in the form of a constructor that says "Hello please can you not sum me, I'd like to be producted thanks!!!". Now you are completely free of the curse and can happily make a new monoid instance for $Pint$s!

instance $Monoid \ Pint$ **where**
$mempty \qquad\qquad\quad = Pint \ 1$
$mappend \ (Pint \ x) \ (Pint \ y) = Pint \ (x * y)$

Obviously, there is a little bit of faffing about since you can't directly use the $(*)$ function, unless of course you defined the Num instance for $Pint$. Additionally if it's $Integer$s you actually want to be dealing with (because too many $Pint$s is bad for you) then you do have to define a function to turn a $Pint$ back into an $Integer$:

$$fromPint :: Pint \to Integer$$
$$fromPint\ (Pint\ x) = x$$

You might be wondering why I haven't said that we need to define a function that turns $Integer$s into $Pint$s well that is because we already have one: $Pint$!!! The constructor literally has the type $Integer \to Pint$.

Now putting this all together we can product a list of $Integer$s using a monoid instance!

$$magic = (fromPint \circ mconcat \circ (map\ Pint))$$

Take that *Zenzike*!!! Now we possess a non specific version of $foldr$ that can dynamically decide based on the type of things in the list that it is given how to fold it. That is POWERFUL!!!