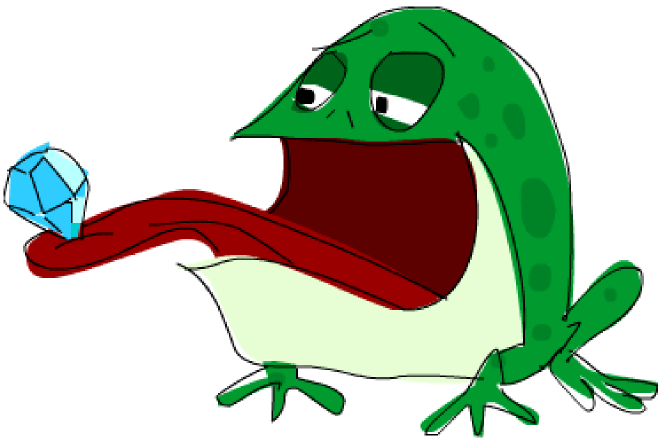
Resolvendo Problemas Funcionalmente | Índice | Um punhado de Monads

Functors, Applicative Functors e Monoids

A combinação de pureza em Haskell, funções de ordem superior, tipos de dados algébricos parametrizados e typeclasses nos permite implementar polimorfismo em um nível muito mais alto em relação as outras linguagens. Não precisamos pensar sobre a dependência dos tipos em uma grande hierarquia de tipos. Ao invés disso, pensamos a respeito de como os tipos podem se comportar e se conectar com as typeclasses apropriadas. Um **`Int`** pode se comportar como um monte de coisas. Ele pode se comportar como algo que compara a igualdade de coisas, como algo que ordena, como algo que enumera as coisas, etc.

As typeclasses são abertas, o que nos permite definir os nossos próprios tipos de dados, pense sobre como algo deve se comportar e conecte isso com as typeclasses que definem esse comportamento. Por causa disso e por causa do belo sistema de tipos de Haskell, que nos permite saber bastante sobre uma função apenas olhando para a sua declaração de tipo, podemos definir typeclasses que definem um comportamento bem amplo e abstrato. Nós já fomos apresentados a typeclasses que definem as operações que inspecionam duas coisas quaisquer e nos dizem se elas são iguais ou que comparam a ordem delas. Esses são comportamentos bastante abstratos e elegantes, mas nós já não pensamos neles como algo super ultra especial porque geralmente lidamos com eles ao longo de boa parte das nossas vidas. Recentemente nós descobrimos os *functors*, que basicamente são coisas que podem ser mapeadas. Isso é um exemplo de uma propriedade bastante útil e abstrata que as typeclasses podem descrever. Neste capítulo vamos dar uma olhada bem de perto em *functors*, juntamente com uma versão mais forte e útil de *functors* chamada de *applicative functors*. Vamos também dar uma boa olhada em *monoids*, que são uma espécie de isolante.

Functors redux



Já falamos sobre functors no próprio capítulo deles. Se você ainda não leu ele, provavelmente deveria dar uma olhada agora, ou talvez depois quando você tiver um pouco mais de tempo. Ou você pode simplemente fingir que já leu ele.

Ainda assim, aqui vai uma rápida revisão: Functors são coisas mapeaveis, assim como listas, **`Maybe`**s, árvores, e tal. Em Haskell, eles são descritos pela typeclass **`Functor`**, que tem apenas um método de typeclass, chamado **`fmap`**, que tem o tipo **`fmap :: (a -> b) -> f a -> f b`**. Ele diz: me de uma função que recebe um **`a`** e que retorna um **`b`** e uma caixa com um **`a`** (ou um monte deles) dentro dela e eu te darei uma caixa com ou **`b`** (ou um monte deles) dentro dela.

Um conselho de amigo. Muitas vezes a analogia das caixas é usada para nos ajudar a ter alguma intuição sobre como functors funcionam, e depois, provavelmente vamos usar a mesma analogia para applicative functors e monads. É uma analogia bacana que ajuda as pessoas a entenderem functors em um primeiro momento, apenas não leve isso tão ao pé da letra, porque para algumas functors essa analogia da caixa não se aplica muito bem. Um termo mais correto para definir o que as functors realmente são seria *contexto computacional*. O contexto pode ser que a computação pode ter um valor ou pode ter uma falha (**`Maybe`** e **`Either a`**) ou ele pode ter mais valores (listas), coisas desse tipo.

Se quisermos fazer um tipo construtor uma instância de **`Functor`**, ele deverá ter um tipo de **`* -> *`**, o que significa que ele recebe exatamente um tipo concreto como um tipo de parâmetro. Por exemplo, de um **`Maybe`** pode ser feita uma instância porque ele recebe um tipo como parâmetro para produzir um tipo concreto, como **`Maybe Int`** ou **`Maybe String`**. Se um tipo construtor tem dois parâmetros, como **`Either`**, temos de aplicar parcialmente o tipo construtor até que ele só tenha um parâmetro do tipo. Portanto, não podemos escrever **`instance Functor Either where`**, mas podemos escrever **`instance Functor (Either a) where`** e, em seguida, se imaginarmos que **`fmap`** é só para **`Either a`**, ele teria uma declaração de tipo de **`fmap :: (b -> c) -> Either a b -> Either a c`**. Como você pode ver, a parte **`Either a`** é fixa, porque **`Either a`** recebe apenas um tipo como parâmetro, ao passo que se **`Either`** recebesse dois então **`fmap :: (b -> c) -> Either b -> Either c`** não iria fazer sentido.

Aprendemos por enquanto como que um monte de tipos (bem, tipos construtores na verdade) são instâncias de **`Functor`**, como **`[]`**, **`Maybe`**, **`Either a`** e o tipo **`Tree`** que nós mesmo fizemos. Dizemos como queremos mapear funções para o nosso próprio bem. Nesse capítulo, vamos dar uma olhada em mais duas instâncias de functor, chamadas de **`IO`** e **`(->) r`**.

Se algum valor tiver o tipo de, digamos, **`IO String`**, isso irá significar que uma ação I/O, quando executada, irá até o mundo real e trazer alguma string para nós, que será o nosso resultado. Podemos usar **`<-`** na sintaxe *do* para atrelar esse resultado a um nome. Já mencionamos que ações I/O são como pequenas caixas com perninhas que vão até o mundo real e pegam algum valor para nós. Nós podemos inspecionar o que elas pegaram, mas depois de inspecionar, nós temos que devolver o valor de volta ao **`IO`**. Ao pensar nessa analogia da caixa com pequenas perninhas, nós conseguimos ver como **`IO`** age como um functor.

Vamos ver como **`IO`** é uma instância de **`Functor`**. Quando nós usamos uma função **`fmap`** sob uma ação I/O, nós esperamos receber de volta uma ação I/O que faz a mesma coisa, mas que tem a nossa função aplicada sobre seus valores resultantes.

```
instance Functor IO where
    fmap f action = do
        result <- action
        return (f result)
```

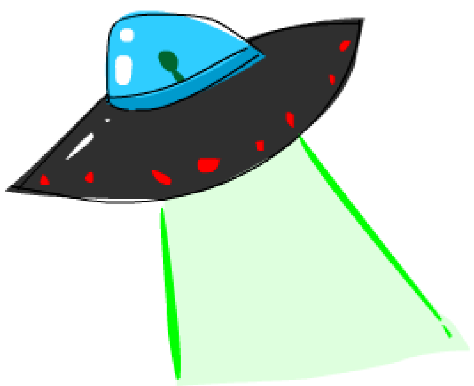
O resultado de mapear alguma coisa sobre uma ação I/O será uma ação I/O, por isso logo de cara nós usamos a sintaxe *do* pra colar duas ações e fazer uma nova. Na implementação do **`fmap`**, nós fizemos uma nova ação I/O que primeiro executa a ação I/O original e chama o seu resultado **`result`**. Em seguida, fazemos **`return (f result)`**. **`return`** é, como você já sabe, uma função que ira criar uma ação I/O que nada fará além de apenas apresentar isso como resultado. A ação que o bloco *do* irá produzir sempre tera o valor resultante da sua última ação. Por isso que nós criamos uma ação I/O que faz absolutamente nada, que irá apenas apresentar **`f result`** como o resultado da nova ação I/O.

Podemos brincar um pouco nisso para ganhar alguma intuição. É realmente muito simples. Da só uma olhada nesse pedaço de código:

```
main = do line <- getLine
          let line' = reverse line
          putStrLn $ "You said " ++ line' ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line' ++ " backwards!"
```

O usuário nos envia um valor (line) e nós devolvemos ele para o usuario ao contrário (com a função *reverse*). Veja como podemos re-escrever isso usando **`fmap`**:

```
main = do line <- fmap reverse getLine
          putStrLn $ "You said " ++ line ++ " backwards!"
          putStrLn $ "Yes, you really said" ++ line ++ " backwards!"
```



Da mesma forma que nós mapeamos com **`fmap`** o **`reverse`** em cima do **`Just "blah"`** para obtermos **`Just "halb"`**, nós podemos também fazer **`fmap reverse`** sobre o **`getLine`**. Isso porque o **`getLine`** é uma ação I/O do tipo **`IO String`** e mapeando **`reverse`** sobre ele iremos ter de volta uma ação I/O que irá lá fora no mundo real, pegará uma linha e irá aplicar o **`reverse`** em cima do resultado. Da mesma forma que podemos aplicar uma função para algo dentro de uma caixa **`Maybe`**, nós podemos também aplicar uma função para o que esta dentro de uma caixa **`IO`**, que apenas irá no mundo real para obter alguma coisa. Portanto quando nós associamos isso a um nome usando **`<-`**, esse nome irá refletir um resultado que já tem o **`reverse`** aplicado.

A ação I/O **`fmap (++"!") getLine`** se comporta como **`getLine`**, simplesmente apresentando sempre junto ao seu resultado um **`"!"`**.

Se olharmos em como o tipo **`fmap`** deve ser caso esteja atrelado ao **`IO`**, ele deverá ser algo como **`fmap :: (a -> b) -> IO a -> IO b`**. Aqui o **`fmap`** pega uma função, uma ação I/O e retorna uma nova ação I/O da mesma forma que a anterior com a exceção de que a função será aplicada apenas no seu próprio resultado.

Caso você notar que esta sempre associando um resultado de uma ação I/O a um nome, só para aplicar uma função nela e chamar alguma outra coisa, considere então usar o **`fmap`**, simplesmente porque assim fica mais bonitinho. Se você quiser aplicar diversas transformações em um dado dentro de uma *functor*, você pode declarar sua própria função em um nível de abstração mais alto, criando uma função lambda ou idealmente, usando composição de função:

```
import Data.Char
import Data.List

main = do line <- fmap (intersperse '-' . reverse . map toUpper) getLine
          putStrLn line
```

```
$ runhaskell fmapping_io.hs
hello there
E-R-E-H-T- -O-L-L-E-H
```

Provavelmente você já esta sabendo que `intersperse '-' . reverse . map toUpper` é uma função que recebe uma string, mapeia um `toUpper` sobre ela, aplica um `reverse` no resultado e por último aplica sobre esse novo resultado o `intersperse '-'`. É o mesmo que escrever `(\xs -> intersperse '-' (reverse (map toUpper xs)))`, porém mais bonitinho.

Uma outra forma de `Functor` que nós estivemos sempre lidando com ela mas que ainda não sabiamos é que `(->) r` é uma `Functor`. Provavelmente você esta um pouco confuso agora, pensando "que diabos esse `(->) r` significa"? O tipo de função `r -> a` pode ser reescrito como `(->) r a`, da mesma forma que podemos re-escrever `2 + 3` como `(+) 2 3`. Agora quando olhamos o `(->) r a`, nós podemos enchergar o `(->)` sobre uma ótica diferente, porque nós vemos ele apenas como um tipo construtor que pega dois parâmetros, assim como o `Either`. Mas lembre-se, nós vimos que um tipo construtor deve pegar exatamente um tipo de parametro para que ele possa ser uma instância de `Functor`. Esse é o porque não podemos fazer com que o `(->)` seja uma instância de `Functor`, porém se nós aplicarmos parcialmente isso ao `(->) r`, não nos trará problemas. Se a sintaxe permitir que tipos contrutores sejam parcialmente aplicados em partes (da mesma forma que podemos aplicar `+` ao fazer `(2+)`, que é o mesmo que `(+) 2`), você poderá escrever então `(->) r` como `(r ->)`. O que são funções *functors*? Bem, vamos dar uma olhada na implementação que esta em `Control.Monad.Instances`

> Usualmente marcamos funções que contém qualquer coisa e que retornam qualquer coisa como `a -> b`. `r -> a` é o mesmo esquema, apenas usamos letras diferentes para o tipo da variável.

```haskell
instance Functor ((->) r) where
    fmap f g = (\x -> f (g x))
```

Se a sintaxe permitisse, poderia ser escrito como

```haskell
instance Functor (r ->) where
    fmap f g = (\x -> f (g x))
```

Porém não permite, então temos que escrever da forma anterior.

Antes de mais nada, vamos pensar a respeito do tipo `fmap`. Ele é `fmap :: (a -> b) -> f a -> f b`. O que vamos fazer agora é substituir mentalmente todos aqueles `f`, que é a regra onde a instância do nosso functor trabalha, por `(->) r`. Vamos fazer isso para entender como o `fmap` se comporta nessa instância em particular. Substituindo temos `fmap :: (a -> b) -> ((->) r a) -> ((->) r b)`. Agora o que nós fazemos é escrever os tipos `(->) r a` e `(-> r b)` de forma infixa `r -> a` e `r -> b`, como normalmente fazemos com funções. O que obtemos agora é `fmap :: (a -> b) -> (r -> a) -> (r -> b)`.

Hmmm beleza. Mapear uma função sobre outra função vai produzir uma função, da mesma forma que mapear uma função sobre um `Maybe` vai produzir um `Maybe` e mapear uma função sobre uma lista irá produzir uma lista. O que, por exemplo, o tipo `fmap :: (a -> b) -> (r -> a) -> (r -> b)` nos diz? Então, perceba que isso recebe uma função a partir de `a` para `b` e uma função de `r` para `a` e retorna uma função a partir de `r` para `b`. Será que isso te lembra alguma coisa? Sim! Composição de funções! Nós empilhamos a saída de **`r -> a`** sobre a entrada de **`a -> b`** para ter a função **`r -> b`**, que é exatamente o que composição de funções é. Se você observar como a instância é definida acima, vai ver que aquilo é apenas uma composição de função. Outra forma de escrever essa instância pode ser:

```
instance Functor ((->) r) where
    fmap = (.)
```

Isso nos revela que usar **`fmap`** sobre funções é apenas composição de uma maneira óbvia. Digite em seu terminal **`:m + Control.Monad.Instances`**, já que ai é onde a instância é definida e então tente brincar com mapeamentos sobre funções.

```
ghci> :t fmap (*3) (+100)
fmap (*3) (+100) :: (Num a) => a -> a
ghci> fmap (*3) (+100) 1
303
ghci> (*3) `fmap` (+100) $ 1
303
ghci> (*3) . (+100) $ 1
303
ghci> fmap (show . (*3)) (*100) 1
"300"
```

Podemos chamar **`fmap`** como uma função infixa e então a semelhança com o **`.`** fica bem clara. Na segunda linha de entrada, estamos mapeando **`(*3)`** sobre **`(+100)`**, que resulta em uma função que vai pegar uma entrada, chamar **`(+100)`** nela e então chamar **`(*3)`** sobre o resultado. Nós realizamos a chamada dessa função com o **`1`**.

E como será que a analogia da caixa se encaixa aqui? Bem, se você forçar bastante a barra, ela se encaixa. Quando usamos o **`fmap (+3)`** sobre **`Just 3`**, é tranquilo de imaginar que o **`Maybe`** seria tipo uma caixa com algumas coisas dentro sobre as quais iremos aplicar a função **`(+3)`**. Mas e quando nós estamos fazendo um **`fmap (*3) (+100)`**? Nesse caso você pode imaginar a função **`(+100)`** como uma caixa que eventualmente irá conter o resultado. Da mesma forma que uma ação I/O pode ser uma espécia de caixa que se vai ao mundo real e volta pra gente com o resultado. Usando **`fmap (*3)`** no **`(+100)`** vai criar outra função que se comporta como **`(+100)`**, apenas antes de produzir um resultado, **`(*3)`** será aplicado no resultado. Agora a gente pode ver como o **`fmap`** age como **`.`** para as funções.

O fato de que **`fmap`** é uma composição de funções quando usada em funções não é tão absurdamente útil no momento, mas ao menos é bem interessante. Isso abre um pouco a nossa mente e nos permite entender que as coisas se comportam mais como cálculos do que caixas (**`IO`** e **`(->) r`** podem ser functors). A função começa a ser mapeada sobre o resultado do cálculo no próprio cálculo, porém o resultado desse cálculo é modificado com a função.

Antes de entramos nas regras que o **`fmap`** deve seguir, vamos pensar sobre o tipo desse **`fmap`** mais uma vez. O tipo dele é **`fmap :: (a -> b) -> f a -> f b`**. Estamos esquecendo que tem a restrição da classe **`(Functor f) =>`**, deixando isso de fora pra simplificar as coisas aqui, como estamos falando sobre functors sabemos o porque deixamos o **`f`** por aqui. Quando aprendemos pela primeira vez sobre [funções curried](#) de alta ordem, aprendemos que todas funções Haskell na verdade pega apenas um parâmetro. A função **`a -> b -> c`** na verdade só recebe uma único parâmetro do tipo **`a`** e então retorna uma função **`b -> c`**, que recebe só um parâmetro e retorna um **`c`**. É assim se a gente chamar uma função com alguns poucos parâmetros (parcialmente aplicamos ela), a gente recebe de volta uma função que pega o número dos parâmetros que restaram (se a gente pensar novamente sobre funções recebendo vários parâmetros). Portanto **`a -> b -> c`** pode ser escrito como **`a -> (b -> c)`**, para tornar o rearranjo (*currying*) mais aparente.

Seguindo a mesma linha de pensamento, se a gente escrever **`fmap :: (a -> b) -> (f a -> f b)`**, podemos então pensar no **`fmap`** não como uma função que pega uma função e um functor que retorna um functor, mas como uma função que pega uma função e retorna uma nova função da mesma forma que a anterior, ela apenas pega um functor como um parâmetro e retorna um functor como resultado. Ela pega uma função **`a -> b`** e retorna uma função **`f a -> f b`**. Isso é chamado de *levantar* uma função. Vamos brincar em torno dessa idea usando o nosso comando do GHCI **`:t`**:

```
ghci> :t fmap (*2)
fmap (*2) :: (Num a, Functor f) => f a -> f a
ghci> :t fmap (replicate 3)
fmap (replicate 3) :: (Functor f) => f a -> f [a]
```

A expressão **`fmap (*2)`** é uma função que pega um functor **`f`** sob números e retorna um functor sob números. Esse functor pode ser uma lista, um **`Maybe`** , uma **`Either String`**, tanto faz. A expressão **`fmap (replicate 3)`** vai pegar um functor sob qualquer tipo e retornar uma functor sob a lista de elementos daquele tipo.

> Quando digo *um functor sob números*, você deve entender isso como *um functor que tem números dentro dele*. O anterior é um pouco mais correto técnicamente, porém o último é mais fácil de entender.

Isso é ainda mais aparente se nós aplicarmos parcialmente, por exemplo, **`fmap (++"!")`** e associar isso a um nome no GHCI.

Você pode pensar no **`fmap`** tanto como uma função que recebe uma função e um functor e então mapeia essa função sobre o functor, como você pode pensar nele como uma função que recebe uma função e levanta essa função para que ela opere no functor. Ambas visões estão corretas em Haskell e se equivalem.

O tipo **`fmap (replicate 3) :: (Functor f) => f a -> f [a]`** significa que essa função irá trabalhar em qualquer functor. O que exatamente isso irá fazer, dependerá de qual functor iremos usar nela. Se a gente usar **`fmap (replicate 3)`** numa lista, a implementação do **`fmap`** para listas será usada, que é basicamente um **`map`**. Se usarmos isso em um **`Maybe a`**, ele irá aplicar o **`replicate 3`** ao valor dentro de **`Just`**, ou se ele for um **`Nothing`**, então permanecerá um **`Nothing`**.

```
ghci> fmap (replicate 3) [1,2,3,4]
[[1,1,1],[2,2,2],[3,3,3],[4,4,4]]
ghci> fmap (replicate 3) (Just 4)
Just [4,4,4]
ghci> fmap (replicate 3) (Right "blah")
Right ["blah","blah","blah"]
ghci> fmap (replicate 3) Nothing
Nothing
ghci> fmap (replicate 3) (Left "foo")
Left "foo"
```

A seguir, vamos dar uma olhada nas **leis dos functors**. Para que algo possa ser um functor, ela deve satisfazer algumas leis. É esperado que todas functors demonstrem cetos tipos de propriedades e comportamentos que são típicos de functors. Eles devem se comportar seguramente como coisas que podem ser mapeadas. Chamar o `fmap` em uma functor deve somente mapear a função sob o functor, apenas isso. Esse comportamento é descrito nas leis das functors. Existem duas dessas que todas instâncias de `Functor` devem cumprir. Elas não são forçadas automaticamente pelo Haskell, portanto você deve testa-las por sua conta própria.

A primeira lei dos functors decreta que se nós mapearmos o `id` de uma função sobre um functor, o functor que nós obtermos como retorno deverá ser o mesmo que o functor original. Se escrevermos isso um pouco mais formalmente, significará que `fmap id = id`. Basicamente então isso diz que se nós fizermos um `fmap id` sobre um functor, isso será o mesmo que apenas chamar `id` no functor. Lembre-se, `id` é a identidade da função, que só retorna seu parâmetro inalterado. Também podemos escrever isso como `\x -> x`. Se visualizarmos o functor como algo que pode ser mapeado em cima de alguma outra coisa, a lei `fmap id = id` irá parecer meio que trivial e óbvia.

Vamos ver se essa lei se sustenta sobre alguns valores de functors.

```
ghci> fmap id (Just 3)
Just 3
ghci> id (Just 3)
Just 3
ghci> fmap id [1..5]
[1,2,3,4,5]
ghci> id [1..5]
[1,2,3,4,5]
ghci> fmap id []
[]
ghci> fmap id Nothing
Nothing
```

Se nós olharmos na implementação de `fmap` por, digamos, `Maybe`, nós vamos descobrir porque a primeira lei dos functors se sustenta.

```haskell
instance Functor Maybe where
   fmap f (Just x) = Just (f x)
   fmap f Nothing = Nothing
```

Imaginemos que o `id` age como o parâmetro `f` na implementação. Nós percebemos que se usarmos `fmap id` sobre `Just x`, o resultado será `Just (id x)`, e como `id` só retorna isso como parâmetro, podemos deduzir então que `Just (id x)` é igual a `Just x`. Portanto agora sabemos que se nós mapearmos um `id` sobre um valor `Maybe` com um valor construtor `Just`, nós obteremos o mesmo resultado de volta.

Perceber que ao mapear um **`id`** sobre um valor **`Nothing`** irá retornar o mesmo valor é trivial. Então a partir dessas duas equações na implementação para **`fmap`**, nós vimos que a lei **`fmap id = id`** é verdadeira.

A segunda lei diz que compor duas funções e então mapear a função resultante sobre um functor deve ser o mesmo que primeiro mapear uma função sobre o functor e depois mapear a outra função. Formalmente escrevendo, isso significa que `fmap (f . g) = fmap f . fmap g`. Ou, escrevendo de outro jeito, para qualquer functor *F*, o seguinte deverá ser verdadeiro:
`fmap (f . g) F = fmap f (fmap g F)`.

If we can show that some type obeys both functor laws, we can rely on it having the same fundamental behaviors as other functors when it comes to mapping. We can know that when we use **`fmap`** on it, there won't be anything other than mapping going on behind the scenes and that it will act like a thing that can be mapped over, i.e. a functor. You figure out how the second law holds for some type by looking at the implementation of **`fmap`** for that type and then using the method that we used to check if **`Maybe`** obeys the first law.

If you want, we can check out how the second functor law holds for **`Maybe`**. If we do **`fmap (f . g)`** over **`Nothing`**, we get **`Nothing`**, because doing a **`fmap`** with any function over **`Nothing`** returns **`Nothing`**. If we do **`fmap f (fmap g Nothing)`**, we get **`Nothing`**, for the same reason. OK, seeing how the second law holds for **`Maybe`** if it's a **`Nothing`** value is pretty easy, almost trivial.

How about if it's a **`Just`** ***`something`*** value? Well, if we do **`fmap (f . g) (Just x)`**, we see from the implementation that it's implemented as **`Just ((f . g) x)`**, which is, of course, **`Just (f (g x))`**. If we do **`fmap f (fmap g (Just x))`**, we see from the implementation that **`fmap g (Just x)`** is **`Just (g x)`**. Ergo, **`fmap f (fmap g (Just x))`** equals **`fmap f (Just (g x))`** and from the implementation we see that this equals **`Just (f (g x))`**.

If you're a bit confused by this proof, don't worry. Be sure that you understand how function composition works. Many times, you can intuitively see how these laws hold because the types act like containers or functions. You can also just try them on a bunch of different values of a type and be able to say with some certainty that a type does indeed obey the laws.

Let's take a look at a pathological example of a type constructor being an instance of the **`Functor`** typeclass but not really being a functor, because it doesn't satisfy the laws. Let's say that we have a type:

```
data CMaybe a = CNothing | CJust Int a deriving (Show)
```

The C here stands for *counter*. It's a data type that looks much like **`Maybe a`**, only the **`Just`** part holds two fields instead of one. The first field in the **`CJust`** value constructor will always have a type of **`Int`**, and it will be some sort of counter and the second field is of type **a**, which comes from the type parameter and its type will, of course, depend on the concrete type that we choose for **CMaybe a**. Let's play with our new type to get some intuition for it.

```
ghci> CNothing
CNothing
ghci> CJust 0 "haha"
CJust 0 "haha"
ghci> :t CNothing
CNothing :: CMaybe a
ghci> :t CJust 0 "haha"
CJust 0 "haha" :: CMaybe [Char]
ghci> CJust 100 [1,2,3]
CJust 100 [1,2,3]
```

If we use the **CNothing** constructor, there are no fields, and if we use the **CJust** constructor, the first field is an integer and the second field can be any type. Let's make this an instance of **Functor** so that everytime we use **fmap**, the function gets applied to the second field, whereas the first field gets increased by 1.

```
instance Functor CMaybe where
   fmap f CNothing = CNothing
   fmap f (CJust counter x) = CJust (counter+1) (f x)
```

This is kind of like the instance implementation for **Maybe**, except that when we do **fmap** over a value that doesn't represent an empty box (a **CJust** value), we don't just apply the function to the contents, we also increase the counter by 1. Everything seems cool so far, we can even play with this a bit:

```
ghci> fmap (++"ha") (CJust 0 "ho")
CJust 1 "hoha"
ghci> fmap (++"he") (fmap (++"ha") (CJust 0 "ho"))
CJust 2 "hohahe"
ghci> fmap (++"blah") CNothing
CNothing
```

Does this obey the functor laws? In order to see that something doesn't obey a law, it's enough to find just one counter-example.

```
ghci> fmap id (CJust 0 "haha")
CJust 1 "haha"
ghci> id (CJust 0 "haha")
CJust 0 "haha"
```

Ah! We know that the first functor law states that if we map **id** over a functor, it should be the same as just calling **id** with the same functor, but as we've seen from this example, this is not true for our **CMaybe** functor. Even though it's part of the **Functor** typeclass, it doesn't obey the functor laws and is therefore not a functor. If someone used our **CMaybe** type as a functor, they would expect it to obey the functor laws like a good functor. But **CMaybe** fails at being a functor even though it pretends to be one, so using it as a functor might lead to some faulty code. When we use a functor, it shouldn't matter if we first compose a few functions and then map them over the functor or if we just map each function over a functor in succession. But with **CMaybe**, it matters, because it keeps track of how many times it's been mapped over. Not cool! If we wanted **CMaybe** to obey the functor laws, we'd have to make it so that the **Int** field stays the same when we use **fmap**.

At first, the functor laws might seem a bit confusing and unnecessary, but then we see that if we know that a type obeys both laws, we can make certain assumptions about how it will act. If a type obeys the functor laws, we know that calling `fmap` on a value of that type will only map the function over it, nothing more. This leads to code that is more abstract and extensible, because we can use laws to reason about behaviors that any functor should have and make functions that operate reliably on any functor.

All the `Functor` instances in the standard library obey these laws, but you can check for yourself if you don't believe me. And the next time you make a type an instance of `Functor`, take a minute to make sure that it obeys the functor laws. Once you've dealt with enough functors, you kind of intuitively see the properties and behaviors that they have in common and it's not hard to intuitively see if a type obeys the functor laws. But even without the intuition, you can always just go over the implementation line by line and see if the laws hold or try to find a counter-example.

We can also look at functors as things that output values in a context. For instance, `Just 3` outputs the value `3` in the context that it might or not output any values at all. `[1,2,3]` outputs three values—`1`, `2`, and `3`, the context is that there may be multiple values or no values. The function `(+3)` will output a value, depending on which parameter it is given.

If you think of functors as things that output values, you can think of mapping over functors as attaching a transformation to the output of the functor that changes the value. When we do `fmap (+3) [1,2,3]`, we attach the transformation `(+3)` to the output of `[1,2,3]`, so whenever we look at a number that the list outputs, `(+3)` will be applied to it. Another example is mapping over functions. When we do `fmap (+3) (*3)`, we attach the transformation `(+3)` to the eventual output of `(*3)`. Looking at it this way gives us some intuition as to why using `fmap` on functions is just composition (`fmap (+3) (*3)` equals `(+3) . (*3)`, which equals `\x -> ((x*3)+3)`), because we take a function like `(*3)` then we attach the transformation `(+3)` to its output. The result is still a function, only when we give it a number, it will be multiplied by three and then it will go through the attached transformation where it will be added to three. This is what happens with composition.

Applicative functors

In this section, we'll take a look at applicative functors, which are beefed up functors, represented in Haskell by the `Applicative` typeclass, found in the `Control.Applicative` module.

As you know, functions in Haskell are curried by default, which means that a function that seems to take several parameters actually takes just one parameter and returns a function that takes the next parameter and so on. If a function is of type `a -> b -> c`, we usually say that it takes two parameters and returns a `c`, but actually it takes an `a` and returns a function `b -> c`. That's why we can call a function as `f x y` or as `(f x) y`. This mechanism is what enables us to partially apply functions by just calling them with too few parameters, which results in functions that we can then pass on to other functions.

So far, when we were mapping functions over functors, we usually mapped functions that take only one parameter. But what happens when we map a function like `*`, which takes two parameters, over a functor? Let's take a look at a couple of concrete examples of this. If we have `Just 3` and we do `fmap (*) (Just 3)`, what do we get? From the instance implementation of `Maybe` for `Functor`, we know that if it's a `Just` *`something`* value, it will apply the function to the

something inside the **Just**. Therefore, doing **fmap (*) (Just 3)** results in **Just ((*) 3)**, which can also be written as **Just (* 3)** if we use sections. Interesting! We get a function wrapped in a **Just**!

```
ghci> :t fmap (++) (Just "hey")
fmap (++) (Just "hey") :: Maybe ([Char] -> [Char])
ghci> :t fmap compare (Just 'a')
fmap compare (Just 'a') :: Maybe (Char -> Ordering)
ghci> :t fmap compare "A LIST OF CHARS"
fmap compare "A LIST OF CHARS" :: [Char -> Ordering]
ghci> :t fmap (\x y z -> x + y / z) [3,4,5,6]
fmap (\x y z -> x + y / z) [3,4,5,6] :: (Fractional a) => [a -> a -> a]
```

If we map **compare**, which has a type of **(Ord a) => a -> a -> Ordering** over a list of characters, we get a list of functions of type **Char -> Ordering**, because the function **compare** gets partially applied with the characters in the list. It's not a list of **(Ord a) => a -> Ordering** function, because the first **a** that got applied was a **Char** and so the second **a** has to decide to be of type **Char**.

We see how by mapping "multi-parameter" functions over functors, we get functors that contain functions inside them. So now what can we do with them? Well for one, we can map functions that take these functions as parameters over them, because whatever is inside a functor will be given to the function that we're mapping over it as a parameter.

```
ghci> let a = fmap (*) [1,2,3,4]
ghci> :t a
a :: [Integer -> Integer]
ghci> fmap (\f -> f 9) a
[9,18,27,36]
```

But what if we have a functor value of **Just (3 *)** and a functor value of **Just 5** and we want to take out the function from **Just (3 *)** and map it over **Just 5**? With normal functors, we're out of luck, because all they support is just mapping normal functions over existing functors. Even when we mapped **\f -> f 9** over a functor that contained functions inside it, we were just mapping a normal function over it. But we can't map a function that's inside a functor over another functor with what **fmap** offers us. We could pattern-match against the **Just** constructor to get the function out of it and then map it over **Just 5**, but we're looking for a more general and abstract way of doing that, which works across functors.

Meet the **Applicative** typeclass. It lies in the **Control.Applicative** module and it defines two methods, **pure** and **<*>**. It doesn't provide a default implementation for any of them, so we have to define them both if we want something to be an applicative functor. The class is defined like so:

```
class (Functor f) => Applicative f where
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b
```

This simple three line class definition tells us a lot! Let's start at the first line. It starts the definition of the **Applicative** class and it also introduces a class constraint. It says that if we want to make a type constructor part of the **Applicative** typeclass, it has to be in **Functor** first. That's why if we know that if a type constructor is part of the **Applicative** typeclass, it's also in **Functor**, so we can use **fmap** on it.

The first method it defines is called **pure**. Its type declaration is **pure :: a -> f a**. **f** plays the role of our applicative functor instance here. Because Haskell has a very good type system and because everything a function can do is take some parameters and return some value, we can tell a lot from a type declaration and this is no exception. **pure** should take a value of any type and return an applicative functor with that value inside it. When we say *inside it*, we're using the box analogy again, even though we've seen that it doesn't always stand up to scrutiny. But the **a -> f a** type declaration is still pretty descriptive. We take a value and we wrap it in an applicative functor that has that value as the result inside it.

A better way of thinking about **pure** would be to say that it takes a value and puts it in some sort of default (or pure) context—a minimal context that still yields that value.

The **<*>** function is really interesting. It has a type declaration of **f (a -> b) -> f a -> f b**. Does this remind you of anything? Of course, **fmap :: (a -> b) -> f a -> f b**. It's a sort of a beefed up **fmap**. Whereas **fmap** takes a function and a functor and applies the function inside the functor, **<*>** takes a functor that has a function in it and another functor and sort of extracts that function from the first functor and then maps it over the second one. When I say *extract*, I actually sort of mean *run* and then extract, maybe even *sequence*. We'll see why soon.

Let's take a look at the **Applicative** instance implementation for **Maybe**.

```
instance Applicative Maybe where
    pure = Just
    Nothing <*> _ = Nothing
    (Just f) <*> something = fmap f something
```

Again, from the class definition we see that the **f** that plays the role of the applicative functor should take one concrete type as a parameter, so we write **instance Applicative Maybe where** instead of writing **instance Applicative (Maybe a) where**.

First off, **pure**. We said earlier that it's supposed to take something and wrap it in an applicative functor. We wrote **pure = Just**, because value constructors like **Just** are normal functions. We could have also written **pure x = Just x**.

Next up, we have the definition for **<*>**. We can't extract a function out of a **Nothing**, because it has no function inside it. So we say that if we try to extract a function from a **Nothing**, the result is a **Nothing**. If you look at the class definition for **Applicative**, you'll see that there's a **Functor** class constraint, which means that we can assume that both of **<*>**'s parameters are functors. If the first parameter is not a **Nothing**, but a **Just** with some function inside it, we say that we then want to map that function over the second parameter. This also takes care of the case where the second parameter is **Nothing**, because doing **fmap** with any function over a **Nothing** will return a **Nothing**.

So for **Maybe**, **<*>** extracts the function from the left value if it's a **Just** and maps it over the right value. If any of the parameters is **Nothing**, **Nothing** is the result.

OK cool great. Let's give this a whirl.

```
ghci> Just (+3) <*> Just 9
Just 12
```

```
ghci> pure (+3) <*> Just 10
Just 13
ghci> pure (+3) <*> Just 9
Just 12
ghci> Just (++"hahah") <*> Nothing
Nothing
ghci> Nothing <*> Just "woot"
Nothing
```

We see how doing **pure (+3)** and **Just (+3)** is the same in this case. Use **pure** if you're dealing with **Maybe** values in an applicative context (i.e. using them with **<*>**), otherwise stick to **Just**. The first four input lines demonstrate how the function is extracted and then mapped, but in this case, they could have been achieved by just mapping unwrapped functions over functors. The last line is interesting, because we try to extract a function from a **Nothing** and then map it over something, which of course results in a **Nothing**.

With normal functors, you can just map a function over a functor and then you can't get the result out in any general way, even if the result is a partially applied function. Applicative functors, on the other hand, allow you to operate on several functors with a single function. Check out this piece of code:

```
ghci> pure (+) <*> Just 3 <*> Just 5
Just 8
ghci> pure (+) <*> Just 3 <*> Nothing
Nothing
ghci> pure (+) <*> Nothing <*> Just 5
Nothing
```

What's going on here? Let's take a look, step by step. **<*>** is left-associative, which means that **pure (+) <*> Just 3 <*> Just 5** is the same as **(pure (+) <*> Just 3) <*> Just 5**. First, the **+** function is put in a functor, which is in this case a **Maybe** value that contains the function. So at first, we have **pure (+)**, which is **Just (+)**. Next, **Just (+) <*> Just 3** happens. The result of this is **Just (3+)**. This is because of partial application. Only applying **3** to the **+** function results in a function that takes one parameter and adds 3 to it. Finally, **Just (3+) <*> Just 5** is carried out, which results in a **Just 8**.

Isn't this awesome?! Applicative functors and the applicative style of doing **pure f <*> x <*> y <*> ...** allow us to take a function that expects parameters that aren't necessarily wrapped in functors and use that function to operate on several values that are in functor contexts. The function can take as many parameters as we want, because it's always partially applied step by step between occurences of **<*>**.

This becomes even more handy and apparent if we consider the fact that **pure f <*> x** equals **fmap f x**. This is one of the applicative laws. We'll take a closer look at them later, but for now, we can sort of intuitively see that this is so. Think about it, it makes sense. Like we said before, **pure** puts a value in a default context. If we just put a function in a default context and then extract and apply it to a value inside another applicative functor, we did the same as just mapping that function over that applicative functor. Instead of writing **pure f <*> x <*> y <*> ...**, we can write **fmap f x <*> y <*> ...**. This is why **Control.Applicative** exports a function called **<$>**, which is just **fmap** as an infix operator. Here's how it's defined:

```
(<$>) :: (Functor f) => (a -> b) -> f a -> f b
```

```
f <$> x = fmap f x
```

Yo! Quick reminder: type variables are independent of parameter names or other value names. The `f` in the function declaration here is a type variable with a class constraint saying that any type constructor that replaces `f` should be in the `Functor` typeclass. The `f` in the function body denotes a function that we map over `x`. The fact that we used `f` to represent both of those doesn't mean that they somehow represent the same thing.

By using `<$>`, the applicative style really shines, because now if we want to apply a function `f` between three applicative functors, we can write `f <$> x <*> y <*> z`. If the parameters weren't applicative functors but normal values, we'd write `f x y z`.

Let's take a closer look at how this works. We have a value of `Just "johntra"` and a value of `Just "volta"` and we want to join them into one `String` inside a `Maybe` functor. We do this:

```
ghci> (++) <$> Just "johntra" <*> Just "volta"
Just "johntravolta"
```

Before we see how this happens, compare the above line with this:

```
ghci> (++) "johntra" "volta"
"johntravolta"
```

Awesome! To use a normal function on applicative functors, just sprinkle some `<$>` and `<*>` about and the function will operate on applicatives and return an applicative. How cool is that?

Anyway, when we do `(++) <$> Just "johntra" <*> Just "volta"`, first `(++)`, which has a type of `(++) :: [a] -> [a] -> [a]` gets mapped over `Just "johntra"`, resulting in a value that's the same as `Just ("johntra"++)` and has a type of `Maybe ([Char] -> [Char])`. Notice how the first parameter of `(++)` got eaten up and how the `a`s turned into `Char`s. And now `Just ("johntra"++) <*> Just "volta"` happens, which takes the function out of the `Just` and maps it over `Just "volta"`, resulting in `Just "johntravolta"`. Had any of the two values been `Nothing`, the result would have also been `Nothing`.

So far, we've only used `Maybe` in our examples and you might be thinking that applicative functors are all about `Maybe`. There are loads of other instances of `Applicative`, so let's go and meet them!

Lists (actually the list type constructor, `[]`) are applicative functors. What a suprise! Here's how `[]` is an instance of `Applicative`:

```
instance Applicative [] where
    pure x = [x]
    fs <*> xs = [f x | f <- fs, x <- xs]
```

Earlier, we said that `pure` takes a value and puts it in a default context. Or in other words, a minimal context that still yields that value. The minimal context for lists would be the empty list, `[]`, but the empty list represents the lack of a value, so it can't hold in itself the value that we used **pure** on. That's why **pure** takes a value and puts it in a singleton list. Similarly, the minimal context for the **Maybe** applicative functor would be a **Nothing**, but it represents the lack of a value instead of a value, so **pure** is implemented as **Just** in the instance implementation for **Maybe**.

```
ghci> pure "Hey" :: [String]
["Hey"]
ghci> pure "Hey" :: Maybe String
Just "Hey"
```

What about **<*>**? If we look at what **<*>**'s type would be if it were limited only to lists, we get **(<*>) :: [a -> b] -> [a] -> [b]**. It's implemented with a list comprehension. **<*>** has to somehow extract the function out of its left parameter and then map it over the right parameter. But the thing here is that the left list can have zero functions, one function, or several functions inside it. The right list can also hold several values. That's why we use a list comprehension to draw from both lists. We apply every possible function from the left list to every possible value from the right list. The resulting list has every possible combination of applying a function from the left list to a value in the right one.

```
ghci> [(*0),(+100),(^2)] <*> [1,2,3]
[0,0,0,101,102,103,1,4,9]
```

The left list has three functions and the right list has three values, so the resulting list will have nine elements. Every function in the left list is applied to every function in the right one. If we have a list of functions that take two parameters, we can apply those functions between two lists.

```
ghci> [(+),(*)] <*> [1,2] <*> [3,4]
[4,5,5,6,3,4,6,8]
```

Because **<*>** is left-associative, **[(+),(*)] <*> [1,2]** happens first, resulting in a list that's the same as **[(1+),(2+),(1*),(2*)]**, because every function on the left gets applied to every value on the right. Then, **[(1+),(2+),(1*),(2*)] <*> [3,4]** happens, which produces the final result.

Using the applicative style with lists is fun! Watch:

```
ghci> (++) <$> ["ha","heh","hmm"] <*> ["?","!","."]
["ha?","ha!","ha.","heh?","heh!","heh.","hmm?","hmm!","hmm."]
```

Again, see how we used a normal function that takes two strings between two applicative functors of strings just by inserting the appropriate applicative operators.

You can view lists as non-deterministic computations. A value like **100** or **"what"** can be viewed as a deterministic computation that has only one result, whereas a list like **[1,2,3]** can be viewed as a computation that can't decide on which result it wants to have, so it presents us with all of the possible results. So when you do something like **(+) <$> [1,2,3] <*> [4,5,6]**, you can think of it as adding together two non-deterministic computations with **+**, only to produce another non-deterministic computation that's even less sure about its result.

Using the applicative style on lists is often a good replacement for list comprehensions. In the second chapter, we wanted to see all the possible products of **`[2,5,10]`** and **`[8,10,11]`**, so we did this:

```
ghci> [ x*y | x <- [2,5,10], y <- [8,10,11]]
[16,20,22,40,50,55,80,100,110]
```

We're just drawing from two lists and applying a function between every combination of elements. This can be done in the applicative style as well:

```
ghci> (*) <$> [2,5,10] <*> [8,10,11]
[16,20,22,40,50,55,80,100,110]
```

This seems clearer to me, because it's easier to see that we're just calling **`*`** between two non-deterministic computations. If we wanted all possible products of those two lists that are more than 50, we'd just do:

```
ghci> filter (>50) $ (*) <$> [2,5,10] <*> [8,10,11]
[55,80,100,110]
```

It's easy to see how **`pure f <*> xs`** equals **`fmap f xs`** with lists. **`pure f`** is just **`[f]`** and **`[f] <*> xs`** will apply every function in the left list to every value in the right one, but there's just one function in the left list, so it's like mapping.

Another instance of **`Applicative`** that we've already encountered is **`IO`**. This is how the instance is implemented:

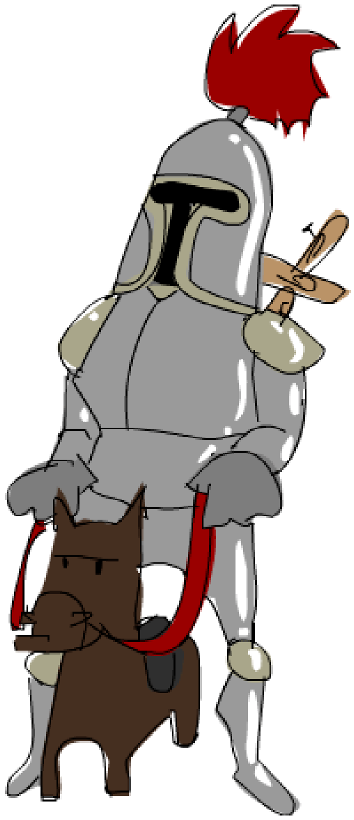
```
instance Applicative IO where
    pure = return
    a <*> b = do
        f <- a
        x <- b
        return (f x)
```

Since **`pure`** is all about putting a value in a minimal context that still holds it as its result, it makes sense that **`pure`** is just **`return`**, because **`return`** does exactly that; it makes an I/O action that doesn't do anything, it just yields some value as its result, but it doesn't really do any I/O operations like printing to the terminal or reading from a file.

If **`<*>`** were specialized for **`IO`** it would have a type of **`(<*>) :: IO (a -> b) -> IO a -> IO b`**. It would take an I/O action that yields a function as its result and another I/O action and create a new I/O action from those two that, when performed, first performs the first one to get the function and then performs the second one to get the value and then it would yield that function applied to the value as its result. We used *do* syntax to implement it here. Remember, *do* syntax is about taking several I/O actions and gluing them into one, which is exactly what we do here.

With **`Maybe`** and **`[]`**, we could think of **`<*>`** as simply extracting a function from its left parameter and then sort of applying it over the right one. With **`IO`**, extracting is still in the game, but now we also have a notion of *sequencing*, because we're taking two I/O actions and we're sequencing, or gluing, them into one. We have to extract the function from the first I/O action, but to extract a result from an I/O action, it has to be performed.

Consider this:



```
myAction :: IO String
myAction = do
    a <- getLine
    b <- getLine
    return $ a ++ b
```

This is an I/O action that will prompt the user for two lines and yield as its result those two lines concatenated. We achieved it by gluing together two **`getLine`** I/O actions and a **`return`**, because we wanted our new glued I/O action to hold the result of **`a ++ b`**. Another way of writing this would be to use the applicative style.

```
myAction :: IO String
myAction = (++) <$> getLine <*> getLine
```

What we were doing before was making an I/O action that applied a function between the results of two other I/O actions, and this is the same thing. Remember, **`getLine`** is an I/O action with the type **`getLine :: IO String`**. When we use **`<*>`** between two applicative functors, the result is an applicative functor, so this all makes sense.

If we regress to the box analogy, we can imagine **`getLine`** as a box that will go out into the real world and fetch us a string. Doing **`(++) <$> getLine <*> getLine`** makes a new, bigger box that sends those two boxes out to fetch lines from the terminal and then presents the concatenation of those two lines as its result.

The type of the expression **`(++) <$> getLine <*> getLine`** is **`IO String`**, which means that this expression is a completely normal I/O action like any other, which also holds a result value inside it, just like other I/O actions. That's why we can do stuff like:

```
main = do
    a <- (++) <$> getLine <*> getLine
    putStrLn $ "The two lines concatenated turn out to be: " ++ a
```

If you ever find yourself binding some I/O actions to names and then calling some function on them and presenting that as the result by using `return`, consider using the applicative style because it's arguably a bit more concise and terse.

Another instance of `Applicative` is `(->) r`, so functions. They are rarely used with the applicative style outside of code golf, but they're still interesting as applicatives, so let's take a look at how the function instance is implemented.

> If you're confused about what `(->) r` means, check out the previous section where we explain how `(->) r` is a functor.

```
instance Applicative ((->) r) where
    pure x = (\_ -> x)
    f <*> g = \x -> f x (g x)
```

When we wrap a value into an applicative functor with `pure`, the result it yields always has to be that value. A minimal default context that still yields that value as a result. That's why in the function instance implementation, `pure` takes a value and creates a function that ignores its parameter and always returns that value. If we look at the type for `pure`, but specialized for the `(->) r` instance, it's `pure :: a -> (r -> a)`.

```
ghci> (pure 3) "blah"
3
```

Because of currying, function application is left-associative, so we can omit the parentheses.

```
ghci> pure 3 "blah"
3
```

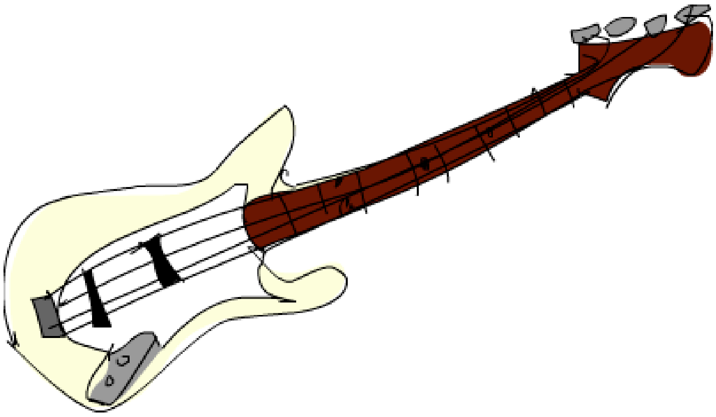
The instance implementation for `<*>` is a bit cryptic, so it's best if we just take a look at how to use functions as applicative functors in the applicative style.

```
ghci> :t (+) <$> (+3) <*> (*100)
(+) <$> (+3) <*> (*100) :: (Num a) => a -> a
ghci> (+) <$> (+3) <*> (*100) $ 5
508
```

Calling `<*>` with two applicative functors results in an applicative functor, so if we use it on two functions, we get back a function. So what goes on here? When we do `(+) <$> (+3) <*> (*100)`, we're making a function that will use `+` on the results of `(+3)` and `(*100)` and return that. To demonstrate on a real example, when we did `(+) <$> (+3) <*> (*100) $ 5`, the `5` first got applied to `(+3)` and `(*100)`, resulting in `8` and `500`. Then, `+` gets called with `8` and `500`, resulting in `508`.

```
ghci> (\x y z -> [x,y,z]) <$> (+3) <*> (*2) <*> (/2) $ 5
[8.0,10.0,2.5]
```

Same here. We create a function that will call the function `\x y z -> [x,y,z]` with the eventual results from `(+3)`, `(*2)` and `(/2)`. The `5` gets fed to each of the three functions and then `\x y z -> [x, y, z]` gets called with those results.



You can think of functions as boxes that contain their eventual results, so doing `k <$> f <*> g` creates a function that will call `k` with the eventual results from `f` and `g`. When we do something like `(+) <$> Just 3 <*> Just 5`, we're using `+` on values that might or might not be there, which also results in a value that might or might not be there. When we do `(+) <$> (+10) <*> (+5)`, we're using `+` on the future return values of `(+10)` and `(+5)` and the result is also something that will produce a value only when called with a parameter.

We don't often use functions as applicatives, but this is still really interesting. It's not very important that you get how the `(->) r` instance for `Applicative` works, so don't despair if you're not getting this right now. Try playing with the applicative style and functions to build up an intuition for functions as applicatives.

An instance of `Applicative` that we haven't encountered yet is `ZipList`, and it lives in `Control.Applicative`.

It turns out there are actually more ways for lists to be applicative functors. One way is the one we already covered, which says that calling `<*>` with a list of functions and a list of values results in a list which has all the possible combinations of applying functions from the left list to the values in the right list. If we do `[(+3),(*2)] <*> [1,2]`, `(+3)` will be applied to both `1` and `2` and `(*2)` will also be applied to both `1` and `2`, resulting in a list that has four elements, namely `[4,5,2,4]`.

However, `[(+3),(*2)] <*> [1,2]` could also work in such a way that the first function in the left list gets applied to the first value in the right one, the second function gets applied to the second value, and so on. That would result in a list with two values, namely `[4,4]`. You could look at it as `[1 + 3, 2 * 2]`.

Because one type can't have two instances for the same typeclass, the `ZipList a` type was introduced, which has one constructor `ZipList` that has just one field, and that field is a list. Here's the instance:

```
instance Applicative ZipList where
        pure x = ZipList (repeat x)
        ZipList fs <*> ZipList xs = ZipList (zipWith (\f x -> f x) fs xs)
```

`<*>` does just what we said. It applies the first function to the first value, the second function to the second value, etc. This is done with `zipWith (\f x -> f x) fs xs`. Because of how `zipWith` works, the resulting list will be as long as the shorter of the two lists.

`pure` is also interesting here. It takes a value and puts it in a list that just has that value repeating indefinitely. `pure "haha"` results in `ZipList (["haha","haha","haha"....` This might be a bit confusing since we said that `pure` should put a value in a minimal context that still yields that value. And you might be thinking that an infinite list of something is hardly minimal. But it makes sense with zip lists, because it has to produce the value on every position. This also satisfies the law that `pure f <*> xs` should equal `fmap f xs`. If `pure 3` just returned `ZipList [3]`,

pure (*2) <*> ZipList [1,5,10] would result in **ZipList [2]**, because the resulting list of two zipped lists has the length of the shorter of the two. If we zip a finite list with an infinite list, the length of the resulting list will always be equal to the length of the finite list.

So how do zip lists work in an applicative style? Let's see. Oh, the **ZipList a** type doesn't have a **Show** instance, so we have to use the `getZipList` function to extract a raw list out of a zip list.

```
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100,100]
[101,102,103]
ghci> getZipList $ (+) <$> ZipList [1,2,3] <*> ZipList [100,100..]
[101,102,103]
ghci> getZipList $ max <$> ZipList [1,2,3,4,5,3] <*> ZipList [5,3,1,2]
[5,3,3,4]
ghci> getZipList $ (,,) <$> ZipList "dog" <*> ZipList "cat" <*> ZipList "rat"
[('d','c','r'),('o','a','a'),('g','t','t')]
```

The **(,,)** function is the same as **\x y z -> (x,y,z)**. Also, the **(,)** function is the same as **\x y -> (x,y)**.

Aside from **zipWith**, the standard library has functions such as **zipWith3**, **zipWith4**, all the way up to 7. **zipWith** takes a function that takes two parameters and zips two lists with it. **zipWith3** takes a function that takes three parameters and zips three lists with it, and so on. By using zip lists with an applicative style, we don't have to have a separate zip function for each number of lists that we want to zip together. We just use the applicative style to zip together an arbitrary amount of lists with a function, and that's pretty cool.

Control.Applicative defines a function that's called `liftA2`, which has a type of **liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c**. It's defined like this:

```
liftA2 :: (Applicative f) => (a -> b -> c) -> f a -> f b -> f c
liftA2 f a b = f <$> a <*> b
```

Nothing special, it just applies a function between two applicatives, hiding the applicative style that we've become familiar with. The reason we're looking at it is because it clearly showcases why applicative functors are more powerful than just ordinary functors. With ordinary functors, we can just map functions over one functor. But with applicative functors, we can apply a function between several functors. It's also interesting to look at this function's type as **(a -> b -> c) -> (f a -> f b -> f c)**. When we look at it like this, we can say that **liftA2** takes a normal binary function and promotes it to a function that operates on two functors.

Here's an interesting concept: we can take two applicative functors and combine them into one applicative functor that has inside it the results of those two applicative functors in a list. For instance, we have **Just 3** and **Just 4**. Let's assume that the second one has a singleton list inside it, because that's really easy to achieve:

```
ghci> fmap (\x -> [x]) (Just 4)
Just [4]
```

OK, so let's say we have **Just 3** and **Just [4]**. How do we get **Just [3,4]**? Easy.

```
ghci> liftA2 (:) (Just 3) (Just [4])
```

```
Just [3,4]
ghci> (:) <$> Just 3 <*> Just [4]
Just [3,4]
```

Remember, **:** is a function that takes an element and a list and returns a new list with that element at the beginning. Now that we have **Just [3,4]**, could we combine that with **Just 2** to produce **Just [2,3,4]**? Of course we could. It seems that we can combine any amount of applicatives into one applicative that has a list of the results of those applicatives inside it. Let's try implementing a function that takes a list of applicatives and returns an applicative that has a list as its result value. We'll call it **sequenceA**.

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA [] = pure []
sequenceA (x:xs) = (:) <$> x <*> sequenceA xs
```

Ah, recursion! First, we look at the type. It will transform a list of applicatives into an applicative with a list. From that, we can lay some groundwork for an edge condition. If we want to turn an empty list into an applicative with a list of results, well, we just put an empty list in a default context. Now comes the recursion. If we have a list with a head and a tail (remember, **x** is an applicative and **xs** is a list of them), we call **sequenceA** on the tail, which results in an applicative with a list. Then, we just prepend the value inside the applicative **x** into that applicative with a list, and that's it!

So if we do **sequenceA [Just 1, Just 2]**, that's **(:) <$> Just 1 <*> sequenceA [Just 2]** . That equals **(:) <$> Just 1 <*> ((:) <$> Just 2 <*> sequenceA [])**. Ah! We know that **sequenceA []** ends up as being **Just []**, so this expression is now **(:) <$> Just 1 <*> ((:) <$> Just 2 <*> Just [])**, which is **(:) <$> Just 1 <*> Just [2]**, which is **Just [1,2]**!

Another way to implement **sequenceA** is with a fold. Remember, pretty much any function where we go over a list element by element and accumulate a result along the way can be implemented with a fold.

```
sequenceA :: (Applicative f) => [f a] -> f [a]
sequenceA = foldr (liftA2 (:)) (pure [])
```

We approach the list from the right and start off with an accumulator value of **pure []**. We do **liftA2 (:)** between the accumulator and the last element of the list, which results in an applicative that has a singleton in it. Then we do **liftA2 (:)** with the now last element and the current accumulator and so on, until we're left with just the accumulator, which holds a list of the results of all the applicatives.

Let's give our function a whirl on some applicatives.

```
ghci> sequenceA [Just 3, Just 2, Just 1]
Just [3,2,1]
ghci> sequenceA [Just 3, Nothing, Just 1]
Nothing
ghci> sequenceA [(+3),(+2),(+1)] 3
[6,5,4]
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2,3],[4,5,6],[3,4,4],[]]
[]
```

Ah! Pretty cool. When used on **`Maybe`** values, **`sequenceA`** creates a **`Maybe`** value with all the results inside it as a list. If one of the values was **`Nothing`**, then the result is also a **`Nothing`**. This is cool when you have a list of **`Maybe`** values and you're interested in the values only if none of them is a **`Nothing`**.

When used with functions, **`sequenceA`** takes a list of functions and returns a function that returns a list. In our example, we made a function that took a number as a parameter and applied it to each function in the list and then returned a list of results. **`sequenceA [(+3),(+2),(+1)] 3`** will call **`(+3)`** with **`3`**, **`(+2)`** with **`3`** and **`(+1)`** with **`3`** and present all those results as a list.

Doing **`(+) <$> (+3) <*> (*2)`** will create a function that takes a parameter, feeds it to both **`(+3)`** and **`(*2)`** and then calls **`+`** with those two results. In the same vein, it makes sense that **`sequenceA [(+3),(*2)]`** makes a function that takes a parameter and feeds it to all of the functions in the list. Instead of calling **`+`** with the results of the functions, a combination of **`:`** and **`pure []`** is used to gather those results in a list, which is the result of that function.

Using **`sequenceA`** is cool when we have a list of functions and we want to feed the same input to all of them and then view the list of results. For instance, we have a number and we're wondering whether it satisfies all of the predicates in a list. One way to do that would be like so:

```
ghci> map (\f -> f 7) [(>4),(<10),odd]
[True,True,True]
ghci> and $ map (\f -> f 7) [(>4),(<10),odd]
True
```

Remember, **`and`** takes a list of booleans and returns **`True`** if they're all **`True`**. Another way to achieve the same thing would be with **`sequenceA`**:

```
ghci> sequenceA [(>4),(<10),odd] 7
[True,True,True]
ghci> and $ sequenceA [(>4),(<10),odd] 7
True
```

`sequenceA [(>4),(<10),odd]` creates a function that will take a number and feed it to all of the predicates in **`[(>4),(<10),odd]`** and return a list of booleans. It turns a list with the type **`(Num a) => [a -> Bool]`** into a function with the type **`(Num a) => a -> [Bool]`**. Pretty neat, huh?

Because lists are homogenous, all the functions in the list have to be functions of the same type, of course. You can't have a list like **`[ord, (+3)]`**, because **`ord`** takes a character and returns a number, whereas **`(+3)`** takes a number and returns a number.

When used with **`[]`**, **`sequenceA`** takes a list of lists and returns a list of lists. Hmm, interesting. It actually creates lists that have all possible combinations of their elements. For illustration, here's the above done with **`sequenceA`** and then done with a list comprehension:

```
ghci> sequenceA [[1,2,3],[4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> [[x,y] | x <- [1,2,3], y <- [4,5,6]]
[[1,4],[1,5],[1,6],[2,4],[2,5],[2,6],[3,4],[3,5],[3,6]]
ghci> sequenceA [[1,2],[3,4]]
```

```
[[1,3],[1,4],[2,3],[2,4]]
ghci> [[x,y] | x <- [1,2], y <- [3,4]]
[[1,3],[1,4],[2,3],[2,4]]
ghci> sequenceA [[1,2],[3,4],[5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
ghci> [[x,y,z] | x <- [1,2], y <- [3,4], z <- [5,6]]
[[1,3,5],[1,3,6],[1,4,5],[1,4,6],[2,3,5],[2,3,6],[2,4,5],[2,4,6]]
```

This might be a bit hard to grasp, but if you play with it for a while, you'll see how it works. Let's say that we're doing **sequenceA [[1,2],[3,4]]**. To see how this happens, let's use the **sequenceA (x:xs) = (:) <$> x <*> sequenceA xs** definition of **sequenceA** and the edge condition **sequenceA [] = pure []**. You don't have to follow this evaluation, but it might help you if have trouble imagining how **sequenceA** works on lists of lists, because it can be a bit mind-bending.

- We start off with **sequenceA [[1,2],[3,4]]**
- That evaluates to **(:) <$> [1,2] <*> sequenceA [[3,4]]**
- Evaluating the inner **sequenceA** further, we get **(:) <$> [1,2] <*> ((:) <$> [3,4] <*> sequenceA [])**
- We've reached the edge condition, so this is now **(:) <$> [1,2] <*> ((:) <$> [3,4] <*> [[]])**
- Now, we evaluate the **(:) <$> [3,4] <*> [[]]** part, which will use **:** with every possible value in the left list (possible values are **3** and **4**) with every possible value on the right list (only possible value is **[]**), which results in **[3:[], 4:[]]**, which is **[[3],[4]]**. So now we have **(:) <$> [1,2] <*> [[3],[4]]**
- Now, **:** is used with every possible value from the left list (**1** and **2**) with every possible value in the right list (**[3]** and **[4]**), which results in **[1:[3], 1:[4], 2:[3], 2:[4]]**, which is **[[1,3],[1,4],[2,3],[2,4]**

Doing **(+) <$> [1,2] <*> [4,5,6]** results in a non-deterministic computation **x + y** where **x** takes on every value from **[1,2]** and **y** takes on every value from **[4,5,6]**. We represent that as a list which holds all of the possible results. Similarly, when we do **sequence [[1,2],[3,4],[5,6],[7,8]]**, the result is a non-deterministic computation **[x,y,z,w]**, where **x** takes on every value from **[1,2]**, **y** takes on every value from **[3,4]** and so on. To represent the result of that non-deterministic computation, we use a list, where each element in the list is one possible list. That's why the result is a list of lists.

When used with I/O actions, **sequenceA** is the same thing as **sequence**! It takes a list of I/O actions and returns an I/O action that will perform each of those actions and have as its result a list of the results of those I/O actions. That's because to turn an **[IO a]** value into an **IO [a]** value, to make an I/O action that yields a list of results when performed, all those I/O actions have to be sequenced so that they're then performed one after the other when evaluation is forced. You can't get the result of an I/O action without performing it.

```
ghci> sequenceA [getLine, getLine, getLine]
heyh
ho
woo
["heyh","ho","woo"]
```

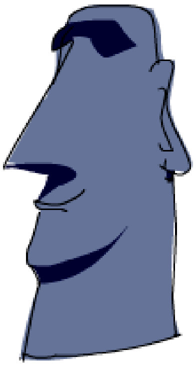
Like normal functors, applicative functors come with a few laws. The most important one is the one that we already mentioned, namely that `pure f <*> x = fmap f x` holds. As an exercise, you can prove this law for some of the applicative functors that we've met in this chapter.The other functor laws are:

- `pure id <*> v = v`
- `pure (.) <*> u <*> v <*> w = u <*> (v <*> w)`
- `pure f <*> pure x = pure (f x)`
- `u <*> pure y = pure ($ y) <*> u`

We won't go over them in detail right now because that would take up a lot of pages and it would probably be kind of boring, but if you're up to the task, you can take a closer look at them and see if they hold for some of the instances.

In conclusion, applicative functors aren't just interesting, they're also useful, because they allow us to combine different computations, such as I/O computations, non-deterministic computations, computations that might have failed, etc. by using the applicative style. Just by using **`<$>`** and **`<*>`** we can use normal functions to uniformly operate on any number of applicative functors and take advantage of the semantics of each one.

A palavra chave newtype



Até agora, nós aprendemos como criar tipos de dados algébricos utilizando a palavra chave **data**. Nós aprendemos também a dar sinônimos para tipos existentes com a palavra chave **type**. Nesta seção, nós iremos dar uma olhada em como criar novos tipos a partir de tipos de dados já existentes utilizando a palavra chave **newtype**, e porque nós iriamos querer fazer isso em primeiro lugar.

Na seção anterior, nós vimos que há na verdade mais formas do tipo lista ser um *applicative functor*. Uma delas é o **`<*>`** pegar cada função da lista a esquerda e aplicar a cada valor na lista a direita, resultando em todas as combinações possíveis ao aplicar as funções da esquerda com os valores da direita.

```
ghci> [(+1),(*100),(*5)] <*> [1,2,3]
[2,3,4,100,200,300,5,10,15]
```

A segunda forma é pegar a primeira função a esquerda do **`<*>`** e aplicar ao primeiro valor da direita, então pegar a segunda função da lista a esquerda e aplicar ao segundo valor da direita, e assim por diante. Por fim, é como se estivessemos mesclando as duas listas. Mas listas já são uma instância de **`Applicative`**, então como nós fazemos da lista uma instância de **`Applicative`** desta segunda forma? Se você ainda lembra, nós dissemos que o tipo **`ZipList a`** foi introduzido para esse propósito, que tem um construtor de valor **`ZipList`**, com apenas um campo. Colocamos a lista que estamos empacotando naquele campo. Então, **`ZipList`** se torna uma instância de **`Applicative`**, assim quando nós queremos usar listas como aplicatives para realizar mesclagem, nós apenas envolvemos elas com o construtor de valor **`ZipList`** e então uma vez feito, recuperamos elas com **`getZipList`**:

```
ghci> getZipList $ ZipList [(+1),(*100),(*5)] <*> ZipList [1,2,3]
[2,200,15]
```

Então, o que isso tem a ver com a palavra-chave *newtype*? Bem, penso sobre como nós podemos escrever a declaração dos dados para nosso tipo **`ZipList a`**. Uma forma seria fazer assim:

```
data ZipList a = ZipList [a]
```

Um tipo que tem apenas um construtor e que o construtor de valor tem apenas um campo que é uma lista de coisas. Nós podemos querer também usar a sintaxe do *record* onde nós automaticamente obtemos uma função que extrai uma lista de um **`ZipList`**:

```
data ZipList a = ZipList { getZipList :: [a] }
```

Parece bom e funciona muito bem. Nós temos duas formas de fazer um tipo já existente uma instância de um tipo de classe, uma forma é usarmos a palavra-chave *data* apenas para embalar o tipo envolta de outro tipo e fazer desse outro tipo uma instância.

A palavra-chave *newtype* em Haskell é exatamente para estes casos quando nós queremos apenas pegar um tipo e envolvê-lo em alguma coisa para aprensentar como outro tipo. Nas bibliotecas atuais, **`ZipList a`** é definido como algo assim:

```
newtype ZipList a = ZipList { getZipList :: [a] }
```

Ao invés da palavra-chave *data*, a palavra-chave *newtype* é usada. Por que motivo? Bem, *newtype* é rápido. Se você usa a palavra-chave *data* para envolver um tipo, há um custo ao envolver e recuperar os dados quando seu programa está sendo executado. Mas se você usa *newtype*, Haskell sabe que você apenas está usando isso para envolver um tipo existente em um novo tipo (portanto o nome), porque você quer que internamente seja a mesma coisa mas tenha um tipo diferente. Com isso em mente, Haskell consegue eliminar o trabalho de envolver e recuperar valores uma vez que ele sabe qual valor é de que tipo.

Então, porque não usar *newtype* em todos os casos ao invés de *data*? Bem, quando você cria um novo tipo a partir de um tipo existente usando a palavra-chave *newtype*, você pode ter apenas um construtor e este deve ter apenas um campo. Mas com *data*, você pode criar tipos de dados com mais de um construtor de valor e cada construtor de valor pode ter zero ou mais campos:

```
data Profession = Fighter | Archer | Accountant
data Race = Human | Elf | Orc | Goblin
data PlayerCharacter = PlayerCharacter Race Profession
```

Quando usamos *newtype*, estamos restritos apenas um construtor com apenas um campo.

Podemos usar também a palavra-chave *deriving* com *newtype* assim como usamos com *data*. Nós podemos derivar instâncias para **`Eq`**, **`Ord`**, **`Enum`**, **`Bounded`**, **`Show`** e **`Read`**. Se derivarmos a instância para um tipo de classe, o tipo que estamos envolvendo tem de estar nesses tipos também. Faz sentido, porque *newtype* apenas envolve um tipo existente. Então agora, se nós fizermos o seguinte, nós podemos imprimir e comparar valores do nosso tipo:

```
newtype CharList = CharList { getCharList :: [Char] } deriving (Eq, Show)
```

Vamos dar uma olhada:

```
ghci> CharList "this will be shown!"
```

```
CharList {getCharList = "this will be shown!"}
ghci> CharList "benny" == CharList "benny"
True
ghci> CharList "benny" == CharList "oisters"
False
```

Nesse *newtype* em particular, o valor do construtor tem o seguinte tipo:

```
CharList :: [Char] -> CharList
```

Ele recebe um valor do tipo **`[Char]`**, como [**"My Sharona"**] e retorna um valor do tipo **`CharList`**. A partir dos exemplos acima onde nós usamos o construtor **`CharList`**, nós vimos que esse é realmente um caso. Inversamente, a função **`getCharList`**, o qual foi gerada para nós já que usamos a sintaxe record em nosso *newtype*, tem este tipo:

```
getCharList :: CharList -> [Char]
```

Este recebe um valor do tipo **`CharList`** e converte para um valor do tipo **`[Char]`**. Você pode pensar nisso como envolvendo e recuperando, mas você também pode pensar nisso como uma conversão de valores de um tipo para outro.

Usando newtype para criar instâncias de um tipo de classe

Muitas vezes, nós queremos fazer dos nossos tipos instâncias de um certo tipo de classe, mas o tipo dos parâmetros não casam com o que nós queremos fazer. É fácil fazer de **`Maybe`** uma instância de **`Functor`**, porque o tipo de classe **`Functor`** é definido assim:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Então nós apenas começamos com:

```
instance Functor Maybe where
```

E então implementamos **`fmap`**. Todos os tipos de parâmetros fazem sentido aqui porque **`Maybe`** toma o lugar de **`f`** na definição do tipo de classe **`Functor`**, então se nós olharmos **`fmap`** como se ele apenas funcionasse com **`Maybe`**, ele acaba se comportando da seguinte maneira:

```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

Não é excelente? Agora, e se nós quisermos fazer da tupla uma instância de **`Functor`**, de uma forma que quando nós usarmos **`fmap`** em uma tupla este é aplicado ao primeiro item da tupla? Dessa forma, fazendo **`fmap (+3) (1,1)`** resultaria em **`(4,1)`**. Escrever uma instância para isso parece difícil. Com **`Maybe`** nós apenas fazemos **`instance Functor Maybe where`**, porque apenas construtores de valores de tipo que recebem exatamente um parâmetro podem se tornar uma instância de **`Functor`**. Mas parece que não tem uma forma de fazer algo assim com **`(a,b)`**, então o parâmetro do tipo **`a`** acaba sendo o parâmetro que muda quando nós usamos **`fmap`**. Para contornar isso, nós podemos usar *newtype* em nossa tupla de uma forma que o segundo parâmetro represente o tipo do primeiro item na tupla:

```
newtype Pair b a = Pair { getPair :: (a,b) }
```

E agora, nós podemos fazer disso uma instância de **`Functor`** da forma que a função é mapeada sobre o primeiro componente:

```
instance Functor (Pair c) where
    fmap f (Pair (x,y)) = Pair (f x, y)
```

Como você pode ver, nós podemos casar padrões de tipos definidos com *newtype*. Nós casamos padrões para obter a tupla subjacente, então nós aplicamos a função **`f`** ao primeiro componente da tupla e então usamos o construtor **`Pair`** para converter a tupla de volta para **`Pair b a`**. Se nós imaginarmos que o tipo **`fmap`** poderia ser se apenas trabalhasse nos nossos novos pares, este seria:

```
fmap :: (a -> b) -> Pair c a -> Pair c b
```

Novamente, escrevemos **`instance Functor (Pair c) where`** e assim **`Pair c`** toma o lugar de **`f`** na definição do tipo de classe para **`Functor`**:

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b
```

Então agora, se nós convertermos a tupla em **`Pair b a`**, nós podemos usar **`fmap`** nela e a função será mapeada sobre o primeiro componente:

```
ghci> getPair $ fmap (*100) (Pair (2,3))
(200,3)
ghci> getPair $ fmap reverse (Pair ("london calling", 3))
("gnillac nodnol",3)
```

A avaliação sob demanda de newtype

Nós mencionamos que *newtype* geralmente é mais rápido que *data*. A única coisa que pode ser feita com *newtype* é transformar um tipo existente em um novo tipo, então internamente, Haskell pode representar os valores dos tipos definidos com *newtype* como os originais, apenas tem de manter em mente que seus tipos agora são distintos. Isso significa que *newtype* não é apenas mais rápido, este é também avaliado sob demanda. Vamos dar uma olhada no que isso significa.

Como dissemos antes, Haskell é avaliado sob demanda por padrão, o que significa que apenas quando nós tentarmos exibir os resultados de nossas funções as computações serão feitas. Além disso, apenas as computações que são necessárias para nossa função exibir o resultado serão avaliadas. O valor **`undefined`** em Haskell representa uma computação errônea. Se tertarmos avaliar isso (ou seja, forçar Haskell a realizar esta computação) exibindo o resultado no terminal, Haskell irá ter um chilique (conhecido técnicamente como exceção):

```
ghci> undefined
*** Exception: Prelude.undefined
```

Entretanto, se nós criarmos uma lista com alguns valores **`undefined`** e apenas usarmos o topo da lista, que não é **`undefined`**, tudo vai funcionar porque Haskell de fato não precisará avaliar nenhum outro elemento na lista se nós apenas queremos ver o que o primeiro elemento é:

```
ghci> head [3,4,5,undefined,2,undefined]
3
```

Agora considere o seguinte tipo:

```
data CoolBool = CoolBool { getCoolBool :: Bool }
```

Este é o seu tipo de dado algébrico comum que foi definido usando a palavra chave *data*. Ele tem apenas um construtor, com um único campo que o tipo é **`Bool`**. Vamos criar uma função para fazer um casamento de padrões em **`CoolBool`** e retornar o valor **`"hello"`**, independente se **`Bool`** dentro de **`CoolBool`** for **`True`** ou **`False`**:

```
helloMe :: CoolBool -> String
helloMe (CoolBool _) = "hello"
```

Ao invés de aplicar esta função a um **`CoolBool`** normalmente, vamos fazer diferente e aplicar a **`undefined`**!

```
ghci> helloMe undefined
"*** Exception: Prelude.undefined
```

Caramba! Uma exceção! Agora, porque esta exceção acontece? Tipos definidos com a palavra chave *data* podem ter múltiplos construtores de valores (mesmo **`CoolBool`** tendo apenas um). Então para ver se o valor passado para nossa função obedece ao padrão **`(CoolBool _)`**, Haskell tem de avaliar o tipo apenas para ver que construtor de valor foi usado quando nós criamos o valor. E quando nós tentarmos avaliar um valor **`undefined`**, uma exceção será lançada.

Ao invés de usar a palavra chave *data* para **`CoolBool`**, vamos tentar usar *newtype*:

```
newtype CoolBool = CoolBool { getCoolBool :: Bool }
```

Nós não temos de mudar nossa função **helloMe**, porque a sintaxe do casamento de padrões é a mesma se você usar *newtype* ou *data* para definir nosso tipo. Vamos fazer a mesma coisa aqui e aplicar **helloMe** a um valor **undefined**:

```
ghci> helloMe undefined
"hello"
```

Agora funcionou! Hmmm, por quê? Bem, como dissemos antes, quando nós usamos *newtype*, Haskell pode internamente representar os valores de novos tipos da mesma forma que os valores originais. Ele não precisa adicionar uma caixa em torno deles, apenas precisa estar ciente de que os valores são de tipos diferentes. E porque Haskell sabe que os tipos criados com a palavra chave *newtype* podem ter apenas um construtor, ele não tem de avaliar o valor passado para a função para ter certeza que obedece aos padrões de **(CoolBool _)** porque os tipos criados com *newtype* podem ter apenas um construtor de valor possível e um campo!

Esta diferença de comportamento pode parecer trivial, mas é muito importante porque isso nos ajuda a perceber que mesmo tipos definidos com *data* e *newtype* se comportam de maneira similar do ponto de vista de programadores porque ambos tem construtores e campos, eles são apenas dois mecanismos diferentes. Enquanto que *data* pode ser usado para criar seus próprios tipos, *newtype* é para criar um novo tipo baseado em um tipo existente. Casar padrões em valores *newtype* não é como extrair algo de uma caixa (como é com *data*), é mais como fazer uma conversão direta entre um tipo e outro.

type vs. newtype vs. data

Até este ponto, você pode estar um pouco confuso sobre qual exatamente é a diferença entre *type*, *data* e *newtype*, então vamos refrescar nossas memórias um pouco.

A palavra chave **type** é para criar tipos sinônimos. O que significa é que nós podemos apenas dar outro nome para um tipo que já existe apenas para facilitar quando precisarmos referenciá-lo. Vamos dizer que fizemos o seguinte:

```
type IntList = [Int]
```

O que tudo isso faz é nos permitir referenciar o tipo **[Int]** como **IntList**. Eles podem ser usados em conjunto. Nós não temos um construtor de valor **IntList** ou nada parecido. Porque **[Int]** e **IntList** são apenas duas formas de referenciar o mesmo tipo, não importa qual nome nós usamos em nossas anotações de tipos:

```
ghci> ([1,2,3] :: IntList) ++ ([1,2,3] :: [Int])
[1,2,3,1,2,3]
```

Nós usamos tipos sinônimos quando nós queremos fazer nossas assinaturas de tipos mais descritivas, dando aos tipos nomes que nos dizem algo sobre seu propósito no contexto das funções onde estão sendo usadas. Por exemplo, quando nós usamos uma lista de associação do tipo **[(String,String)]** para representar uma agenda telefônica, nós demos a este tipo o sinônimo de **PhoneBook**, para que as assinaturas de nossas funções ficassem fáceis de ler.

A palavra chave **newtype** é para pegar tipos existentes e envolver eles em novos tipos, principalmente para facilitar fazer deles instâncias de certos tipos de classes. Quando nós usamos *newtype* para envolver um tipo existente, o tipo que nós obtemos é diferente do tipo original. Se criarmos o seguinte *newtype*:

```
newtype CharList = CharList { getCharList :: [Char] }
```

Nós não podemos usar **++** para juntar um **CharList** e uma lista do tipo **[Char]**. Nós não podemos nem mesmo usar **++** para juntar duas **CharList**, porque **++** trabalha apenas em listas e o tipo **CharList** não é uma lista, mesmo esta contendo uma. Nós podemos, entretando, converter duas **CharList** para listas, juntar elas com **++** e então converter de volta para um **CharList**.

Quando nós usamos a sintaxe record em nossas declarações *newtype*, nós obtemos funções para converter entre o novo tipo e o tipo original: nomeando o construtor de valor de nosso *newtype* e a função para extrair o valor do seu campo. O novo tipo também não se torna automaticamente uma instância dos tipos de classes que o tipo original pertence, então nós temos que derivar ou manualmente escrever elas.

Na prática, você pode pensar em declarações *newtype* como declarações *data* que possuem apenas um construtor e um campo. Se você se pegar escrevendo declarações *data* assim, considere usar *newtype*.

A palavra chave **data** é para criar seus próprios tipos e com eles, fazer o que você quiser. Eles podem ter quantos construtores e campos você desejar e podem ser usados para implementar qualquer tipo de dados algébricos que quiser. Qualquer coisa como listas e **Maybe** - como tipos de árvores.

Se você apenas quer que suas assinaturas de tipos pareçam claras e mais descritivas, você vai querer tipos sinônimos provavelmente. Se você quer pegar um tipo existente e envolver em um novo tipo para fazer deste uma instância de um tipo de classe, pode ser que esteje precisando de *newtype*. E se você quer fazer algo completamente novo, a probabilidade é de que você use a palavra chave *data*.

Monoids

Type classes in Haskell are used to present an interface for types that have some behavior in common. We started out with simple type classes like **Eq**, which is for types whose values can be equated, and **Ord**, which is for things that can be put in an order and then moved on to more interesting ones, like **Functor** and **Applicative**.

When we make a type, we think about which behaviors it supports, i.e. what it can act like and then based on that we decide which type classes to make it an instance of. If it makes sense for values of our type to be equated, we make it an instance of the **Eq** type class. If we see that our type is some kind of functor, we make it an instance of **Functor**, and so on.

Now consider the following: `*` is a function that takes two numbers and multiplies them. If we multiply some number with a `1`, the result is always equal to that number. It doesn't matter if we do `1 * x` or `x * 1`, the result is always `x`. Similarly, `++` is also a function which takes two things and returns a third. Only instead of multiplying numbers, it takes two lists and concatenates them. And much like `*`, it also has a certain value which doesn't change the other one when used with `++`. That value is the empty list: `[]`.

```
ghci> 4 * 1
4
ghci> 1 * 9
9
ghci> [1,2,3] ++ []
[1,2,3]
ghci> [] ++ [0.5, 2.5]
[0.5,2.5]
```

It seems that both `*` together with `1` and `++` along with `[]` share some common properties:

- The function takes two parameters.
- The parameters and the returned value have the same type.
- There exists such a value that doesn't change other values when used with the binary function.

There's another thing that these two operations have in common that may not be as obvious as our previous observations: when we have three or more values and we want to use the binary function to reduce them to a single result, the order in which we apply the binary function to the values doesn't matter. It doesn't matter if we do `(3 * 4) * 5` or `3 * (4 * 5)`. Either way, the result is `60`. The same goes for `++`:

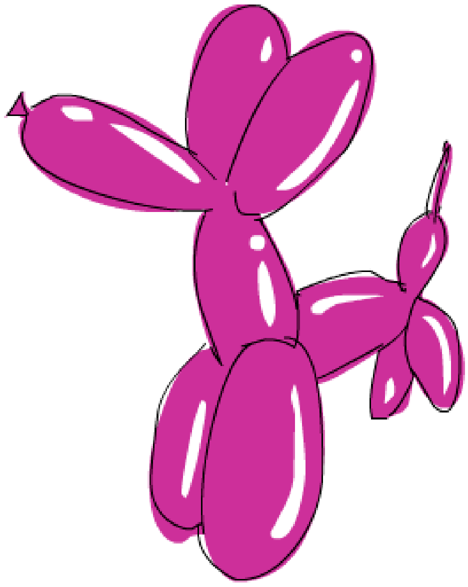
```
ghci> (3 * 2) * (8 * 5)
240
ghci> 3 * (2 * (8 * 5))
240
ghci> "la" ++ ("di" ++ "da")
"ladida"
ghci> ("la" ++ "di") ++ "da"
"ladida"
```

We call this property *associativity*. `*` is associative, and so is `++`, but `-`, for example, is not. The expressions `(5 - 3) - 4` and `5 - (3 - 4)` result in different numbers.

By noticing and writing down these properties, we have chanced upon *monoids*! A monoid is when you have an associative binary function and a value which acts as an identity with respect to that function. When something acts as an identity with respect to a function, it means that when called with that function and some other value, the result is always equal to that other value. `1` is the identity with respect to `*` and `[]` is the identity with respect to `++`. There are a lot of other monoids to be found in the world of Haskell, which is why the `Monoid` type class exists. It's for types which can act like monoids. Let's see how the type class is defined:

```
class Monoid m where
    mempty :: m
    mappend :: m -> m -> m
    mconcat :: [m] -> m
    mconcat = foldr mappend mempty
```

The **`Monoid`** type class is defined in **`import Data.Monoid`**. Let's take some time and get properly acquainted with it.



First of all, we see that only concrete types can be made instances of **`Monoid`**, because the **`m`** in the type class definition doesn't take any type parameters. This is different from **`Functor`** and **`Applicative`**, which require their instances to be type constructors which take one parameter.

The first function is **`mempty`**. It's not really a function, since it doesn't take parameters, so it's a polymorphic constant, kind of like **`minBound`** from **`Bounded`**. **`mempty`** represents the identity value for a particular monoid.

Next up, we have **`mappend`**, which, as you've probably guessed, is the binary function. It takes two values of the same type and returns a value of that type as well. It's worth noting that the decision to name **`mappend`** as it's named was kind of unfortunate, because it implies that we're appending two things in some way. While **`++`** does take two lists and append one to the other, **`*`** doesn't really do any appending, it just multiplies two numbers together. When we meet other instances of **`Monoid`**, we'll see that most of them don't append values either, so avoid thinking in terms of appending and just think in terms of **`mappend`** being a binary function that takes two monoid values and returns a third.

The last function in this type class definition is **`mconcat`**. It takes a list of monoid values and reduces them to a single value by doing **`mappend`** between the list's elements. It has a default implementation, which just takes **`mempty`** as a starting value and folds the list from the right with **`mappend`**. Because the default implementation is fine for most instances, we won't concern ourselves with **`mconcat`** too much from now on. When making a type an instance of **`Monoid`**, it suffices to just implement **`mempty`** and **`mappend`**. The reason **`mconcat`** is there at all is because for some instances, there might be a more efficient way to implement **`mconcat`**, but for most instances the default implementation is just fine.

Before moving on to specific instances of **`Monoid`**, let's take a brief look at the monoid laws. We mentioned that there has to be a value that acts as the identity with respect to the binary function and that the binary function has to be associative. It's possible to make instances of **`Monoid`** that don't follow these rules, but such instances are of no use to anyone because when using the **`Monoid`** type class, we rely on its instances acting like monoids. Otherwise, what's the point? That's why when making instances, we have to make sure they follow these laws:

- ``mempty `mappend` x = x``
- ``x `mappend` mempty = x``
- ``(x `mappend` y) `mappend` z = x `mappend` (y `mappend` z)``

The first two state that **`mempty`** has to act as the identity with respect to **`mappend`** and the third says that **`mappend`** has to be associative i.e. that it the order in which we use **`mappend`** to reduce several monoid values into one doesn't matter. Haskell doesn't enforce these laws, so we as the programmer have to be careful that our instances do indeed obey them.

Lists are monoids

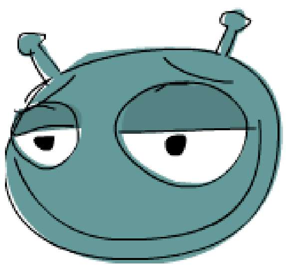
Yes, lists are monoids! Like we've seen, the **++** function and the empty list **[]** form a monoid. The instance is very simple:

```
instance Monoid [a] where
    mempty = []
    mappend = (++)
```

Lists are an instance of the **Monoid** type class regardless of the type of the elements they hold. Notice that we wrote **instance Monoid [a]** and not **instance Monoid []**, because **Monoid** requires a concrete type for an instance.

Giving this a test run, we encounter no surprises:

```
ghci> [1,2,3] `mappend` [4,5,6]
[1,2,3,4,5,6]
ghci> ("one" `mappend` "two") `mappend` "tree"
"onetwotree"
ghci> "one" `mappend` ("two" `mappend` "tree")
"onetwotree"
ghci> "one" `mappend` "two" `mappend` "tree"
"onetwotree"
ghci> "pang" `mappend` mempty
"pang"
ghci> mconcat [[1,2],[3,6],[9]]
[1,2,3,6,9]
ghci> mempty :: [a]
[]
```



Notice that in the last line, we had to write an explicit type annotation, because if we just did **mempty**, GHCi wouldn't know which instance to use, so we had to say we want the list instance. We were able to use the general type of **[a]** (as opposed to specifying **[Int]** or **[String]**) because the empty list can act as if it contains any type.

Because **mconcat** has a default implementation, we get it for free when we make something an instance of **Monoid**. In the case of the list, **mconcat** turns out to be just **concat**. It takes a list of lists and flattens it, because that's the equivalent of doing **++** between all the adjecent lists in a list.

The monoid laws do indeed hold for the list instance. When we have several lists and we **mappend** (or **++**) them together, it doesn't matter which ones we do first, because they're just joined at the ends anyway. Also, the empty list acts as the identity so all is well. Notice that monoids don't require that **a `mappend` b** be equal to **b `mappend` a**. In the case of the list, they clearly aren't:

```
ghci> "one" `mappend` "two"
"onetwo"
ghci> "two" `mappend` "one"
"twoone"
```

And that's okay. The fact that for multiplication **3 * 5** and **5 * 3** are the same is just a property of multiplication, but it doesn't hold for all (and indeed, most) monoids.

Product and **Sum**

We already examined one way for numbers to be considered monoids. Just have the binary function be `*` and the identity value `1`. It turns out that that's not the only way for numbers to be monoids. Another way is to have the binary function be `+` and the identity value `0`:

```
ghci> 0 + 4
4
ghci> 5 + 0
5
ghci> (1 + 3) + 5
9
ghci> 1 + (3 + 5)
9
```

The monoid laws hold, because if you add 0 to any number, the result is that number. And addition is also associative, so we get no problems there. So now that there are two equally valid ways for numbers to be monoids, which way do choose? Well, we don't have to. Remember, when there are several ways for some type to be an instance of the same type class, we can wrap that type in a *newtype* and then make the new type an instance of the type class in a different way. We can have our cake and eat it too.

The **`Data.Monoid`** module exports two types for this, namely **`Product`** and **`Sum`**. **`Product`** is defined like this:

```
newtype Product a =  Product { getProduct :: a }
    deriving (Eq, Ord, Read, Show, Bounded)
```

Simple, just a *newtype* wrapper with one type parameter along with some derived instances. Its instance for **`Monoid`** goes a little something like this:

```
instance Num a => Monoid (Product a) where
    mempty = Product 1
    Product x `mappend` Product y = Product (x * y)
```

`mempty` is just `1` wrapped in a **`Product`** constructor. **`mappend`** pattern matches on the **`Product`** constructor, multiplies the two numbers and then wraps the resulting number back. As you can see, there's a **`Num a`** class constraint. So this means that **`Product a`** is an instance of **`Monoid`** for all **`a`**'s that are already an instance of **`Num`**. To use **`Producta a`** as a monoid, we have to do some *newtype* wrapping and unwrapping:

```
ghci> getProduct $ Product 3 `mappend` Product 9
27
ghci> getProduct $ Product 3 `mappend` mempty
3
ghci> getProduct $ Product 3 `mappend` Product 4 `mappend` Product 2
24
ghci> getProduct . mconcat . map Product $ [3,4,2]
24
```

This is nice as a showcase of the **`Monoid`** type class, but no one in their right mind would use this way of multiplying numbers instead of just writing `3 * 9` and `3 * 1`. But a bit later, we'll see how these **`Monoid`** instances that may seem trivial at this time can come in handy.

`Sum` is defined like **`Product`** and the instance is similar as well. We use it in the same way:

```
ghci> getSum $ Sum 2 `mappend` Sum 9
11
ghci> getSum $ mempty `mappend` Sum 3
3
ghci> getSum . mconcat . map Sum $ [1,2,3]
6
```

`Any` and `All`

Another type which can act like a monoid in two distinct but equally valid ways is **`Bool`**. The first way is to have the *or* function **`||`** act as the binary function along with **`False`** as the identity value. The way *or* works in logic is that if any of its two parameters is **`True`**, it returns **`True`**, otherwise it returns **`False`**. So if we use **`False`** as the identity value, it will return **`False`** when *or*-ed with **`False`** and **`True`** when *or*-ed with **`True`**. The **`Any`** *newtype* constructor is an instance of **`Monoid`** in this fashion. It's defined like this:

```
newtype Any = Any { getAny :: Bool }
    deriving (Eq, Ord, Read, Show, Bounded)
```

Its instance looks goes like so:

```
instance Monoid Any where
        mempty = Any False
        Any x `mappend` Any y = Any (x || y)
```

The reason it's called **`Any`** is because **`` x `mappend` y ``** will be **`True`** if *any* one of those two is **`True`**. Even if three or more **`Any`** wrapped **`Bool`**s are **`mappend`**ed together, the result will hold **`True`** if any of them are **`True`**:

```
ghci> getAny $ Any True `mappend` Any False
True
ghci> getAny $ mempty `mappend` Any True
True
ghci> getAny . mconcat . map Any $ [False, False, False, True]
True
ghci> getAny $ mempty `mappend` mempty
False
```

The other way for **`Bool`** to be an instance of **`Monoid`** is to kind of do the opposite: have **`&&`** be the binary function and then make **`True`** the identity value. Logical *and* will return **`True`** only if both of its parameters are **`True`**. This is the *newtype* declaration, nothing fancy:

```
newtype All = All { getAll :: Bool }
        deriving (Eq, Ord, Read, Show, Bounded)
```

And this is the instance:

```
instance Monoid All where
        mempty = All True
        All x `mappend` All y = All (x && y)
```

When we **mappend** values of the **All** type, the result will be **True** only if *all* the values used in the **mappend** operations are **True**:

```
ghci> getAll $ mempty `mappend` All True
True
ghci> getAll $ mempty `mappend` All False
False
ghci> getAll . mconcat . map All $ [True, True, True]
True
ghci> getAll . mconcat . map All $ [True, True, False]
False
```

Just like with multiplication and addition, we usually explicitly state the binary functions instead of wrapping them in *newtype*s and then using **mappend** and **mempty**. **mconcat** seems useful for **Any** and **All**, but usually it's easier to use the **or** and **and** functions, which take lists of **Bool**s and return **True** if any of them are **True** or if all of them are **True**, respectively.

The Ordering monoid

Hey, remember the **Ordering** type? It's used as the result when comparing things and it can have three values: **LT**, **EQ** and **GT**, which stand for *less than*, *equal* and *greater than* respectively:

```
ghci> 1 `compare` 2
LT
ghci> 2 `compare` 2
EQ
ghci> 3 `compare` 2
GT
```

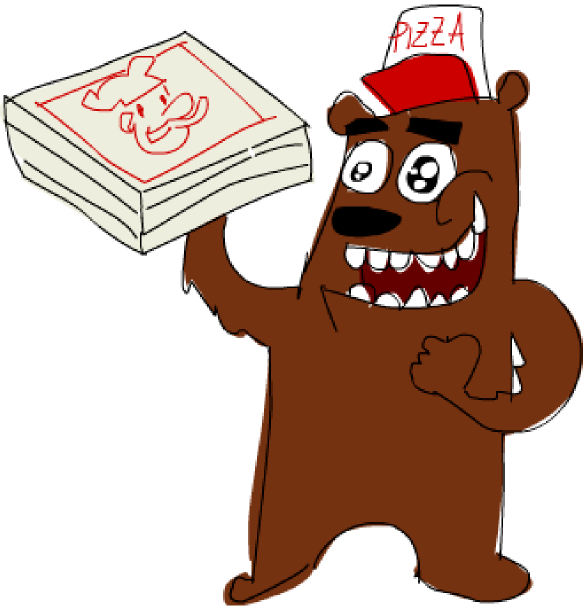
With lists, numbers and boolean values, finding monoids was just a matter of looking at already existing commonly used functions and seeing if they exhibit some sort of monoid behavior. With **Ordering**, we have to look a bit harder to recognize a monoid, but it turns out that its **Monoid** instance is just as intuitive as the ones we've met so far and also quite useful:

```
instance Monoid Ordering where
    mempty = EQ
    LT `mappend` _ = LT
    EQ `mappend` y = y
    GT `mappend` _ = GT
```

The instance is set up like this: when we **mappend** two **Ordering** values, the one on the left is kept, unless the value on the left is **EQ**, in which case the right one is the result. The identity is **EQ**. At first, this may seem kind of arbitrary, but it actually resembles the way we alphabetically compare words. We compare the first two letters and if they differ, we can already decide which word would go first in a dictionary. However, if the first two letters are equal, then we move on to comparing the next pair of letters and repeat the process.

For instance, if we were to alphabetically compare the words **"ox"** and **"on"**, we'd first compare the first two letters of each word, see that they are equal and then move on to comparing the second letter of each word. We see that **'x'** is alphabetically greater than **'n'**, and so we know how the words compare. To gain some intuition for **EQ** being the identity, we can notice that if we were to cram the same letter in the same position in both words, it wouldn't change their alphabetical ordering. **"oix"** is still alphabetically greater than and **"oin"**.

It's important to note that in the **`Monoid`** instance for **`Ordering`**, **`` x `mappend` y ``** doesn't equal **`` y `mappend` x ``**. Because the first parameter is kept unless it's **`EQ`**, **`` LT `mappend` GT ``** will result in **`LT`**, whereas **`` GT `mappend` LT ``** will result in **`GT`**:




```
ghci> LT `mappend` GT
LT
ghci> GT `mappend` LT
GT
ghci> mempty `mappend` LT
LT
ghci> mempty `mappend` GT
GT
```

OK, so how is this monoid useful? Let's say you were writing a function that takes two strings, compares their lengths, and returns an **`Ordering`**. But if the strings are of the same length, then instead of returning **`EQ`** right away, we want to compare them alphabetically. One way to write this would be like so:

```
lengthCompare :: String -> String -> Ordering
lengthCompare x y = let a = length x `compare` length y
                        b = x `compare` y
                    in  if a == EQ then b else a
```

We name the result of comparing the lengths **`a`** and the result of the alphabetical comparison **`b`** and then if it turns out that the lengths were equal, we return their alphabetical ordering.

But by employing our understanding of how **`Ordering`** is a monoid, we can rewrite this function in a much simpler manner:

```
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (x `compare` y)
```

We can try this out:

```
ghci> lengthCompare "zen" "ants"
LT
ghci> lengthCompare "zen" "ant"
GT
```

Remember, when we use **mappend**, its left parameter is always kept unless it's **EQ**, in which case the right one is kept. That's why we put the comparison that we consider to be the first, more important criterion as the first parameter. If we wanted to expand this function to also compare for the number of vowels and set this to be the second most important criterion for comparison, we'd just modify it like this:

```haskell
import Data.Monoid

lengthCompare :: String -> String -> Ordering
lengthCompare x y = (length x `compare` length y) `mappend`
                    (vowels x `compare` vowels y) `mappend`
                    (x `compare` y)
    where vowels = length . filter (`elem` "aeiou")
```

We made a helper function, which takes a string and tells us how many vowels it has by first filtering it only for letters that are in the string **"aeiou"** and then applying **length** to that.

```haskell
ghci> lengthCompare "zen" "anna"
LT
ghci> lengthCompare "zen" "ana"
LT
ghci> lengthCompare "zen" "ann"
GT
```

Very cool. Here, we see how in the first example the lengths are found to be different and so **LT** is returned, because the length of **"zen"** is less than the length of **"anna"**. In the second example, the lengths are the same, but the second string has more vowels, so **LT** is returned again. In the third example, they both have the same length and the same number of vowels, so they're compared alphabetically and **"zen"** wins.

The **Ordering** monoid is very cool because it allows us to easily compare things by many different criteria and put those criteria in an order themselves, ranging from the most important to the least.

Maybe the monoid

Let's take a look at the various ways that **Maybe a** can be made an instance of **Monoid** and what those instances are useful for.

One way is to treat **Maybe a** as a monoid only if its type parameter **a** is a monoid as well and then implement **mappend** in such a way that it uses the **mappend** operation of the values that are wrapped with **Just**. We use **Nothing** as the identity, and so if one of the two values that we're **mappend**ing is **Nothing**, we keep the other value. Here's the instance declaration:

```haskell
instance Monoid a => Monoid (Maybe a) where
    mempty = Nothing
    Nothing `mappend` m = m
    m `mappend` Nothing = m
    Just m1 `mappend` Just m2 = Just (m1 `mappend` m2)
```

Notice the class constraint. It says that **Maybe a** is an instance of **Monoid** only if **a** is an instance of **Monoid**. If we **mappend** something with a **Nothing**, the result is that something. If we **mappend** two **Just** values, the contents of the **Just**s get **mappended** and then wrapped back in a **Just**. We can do this because the class constraint ensures that the type of what's inside the **Just** is an instance of **Monoid**.

```
ghci> Nothing `mappend` Just "andy"
Just "andy"
ghci> Just LT `mappend` Nothing
Just LT
ghci> Just (Sum 3) `mappend` Just (Sum 4)
Just (Sum {getSum = 7})
```

This comes in use when you're dealing with monoids as results of computations that may have failed. Because of this instance, we don't have to check if the computations have failed by seeing if they're a **`Nothing`** or **`Just`** value; we can just continue to treat them as normal monoids.

But what if the type of the contents of the **`Maybe`** aren't an instance of **`Monoid`**? Notice that in the previous instance declaration, the only case where we have to rely on the contents being monoids is when both parameters of **`mappend`** are **`Just`** values. But if we don't know if the contents are monoids, we can't use **`mappend`** between them, so what are we to do? Well, one thing we can do is to just discard the second value and keep the first one. For this, the **`First a`** type exists and this is its definition:

```
newtype First a = First { getFirst :: Maybe a }
    deriving (Eq, Ord, Read, Show)
```

We take a **`Maybe a`** and we wrap it with a *newtype*. The **`Monoid`** instance is as follows:

```
instance Monoid (First a) where
    mempty = First Nothing
    First (Just x) `mappend` _ = First (Just x)
    First Nothing `mappend` x = x
```

Just like we said. **`mempty`** is just a **`Nothing`** wrapped with the **`First`** *newtype* constructor. If **`mappend`**'s first parameter is a **`Just`** value, we ignore the second one. If the first one is a **`Nothing`**, then we present the second parameter as a result, regardless of whether it's a **`Just`** or a **`Nothing`**:

```
ghci> getFirst $ First (Just 'a') `mappend` First (Just 'b')
Just 'a'
ghci> getFirst $ First Nothing `mappend` First (Just 'b')
Just 'b'
ghci> getFirst $ First (Just 'a') `mappend` First Nothing
Just 'a'
```

`First` is useful when we have a bunch of **`Maybe`** values and we just want to know if any of them is a **`Just`**. The **`mconcat`** function comes in handy:

```
ghci> getFirst . mconcat . map First $ [Nothing, Just 9, Just 10]
Just 9
```

If we want a monoid on **`Maybe a`** such that the second parameter is kept if both parameters of **`mappend`** are **`Just`** values, **`Data.Monoid`** provides a the **`Last a`** type, which works like **`First a`**, only the last non-**`Nothing`** value is kept when **`mappend`**ing and using **`mconcat`**:

```
ghci> getLast . mconcat . map Last $ [Nothing, Just 9, Just 10]
```

```
Just 10
ghci> getLast $ Last (Just "one") `mappend` Last (Just "two")
Just "two"
```

Using monoids to fold data structures

One of the more interesting ways to put monoids to work is to make them help us define folds over various data structures. So far, we've only done folds over lists, but lists aren't the only data structure that can be folded over. We can define folds over almost any data structure. Trees especially lend themselves well to folding.

Because there are so many data structures that work nicely with folds, the `Foldable` type class was introduced. Much like **`Functor`** is for things that can be mapped over, **`Foldable`** is for things that can be folded up! It can be found in **`Data.Foldable`** and because it export functions whose names clash with the ones from the **`Prelude`**, it's best imported qualified (and served with basil):

```
import qualified Foldable as F
```

To save ourselves precious keystrokes, we've chosen to import it qualified as **`F`**. Alright, so what are some of the functions that this type class defines? Well, among them are **`foldr`**, **`foldl`**, **`foldr1`** and **`foldl1`**. Huh? But we already know these functions, what's so new about this? Let's compare the types of **`Foldable`**'s **`foldr`** and the **`foldr`** from the **`Prelude`** to see how they differ:

```
ghci> :t foldr
foldr :: (a -> b -> b) -> b -> [a] -> b
ghci> :t F.foldr
F.foldr :: (F.Foldable t) => (a -> b -> b) -> b -> t a -> b
```

Ah! So whereas **`foldr`** takes a list and folds it up, the **`foldr`** from **`Data.Foldable`** accepts any type that can be folded up, not just lists! As expected, both **`foldr`** functions do the same for lists:

```
ghci> foldr (*) 1 [1,2,3]
6
ghci> F.foldr (*) 1 [1,2,3]
6
```

Okay then, what are some other data structures that support folds? Well, there's the **`Maybe`** we all know and love!

```
ghci> F.foldl (+) 2 (Just 9)
11
ghci> F.foldr (||) False (Just True)
True
```

But folding over a **`Maybe`** value isn't terribly interesting, because when it comes to folding, it just acts like a list with one element if it's a **`Just`** value and as an empty list if it's **`Nothing`**. So let's examine a data structure that's a little more complex then.

Remember the tree data structure from the [Making Our Own Types and Typeclasses](#) chapter? We defined it like this:

```
data Tree a = Empty | Node a (Tree a) (Tree a) deriving (Show, Read, Eq)
```

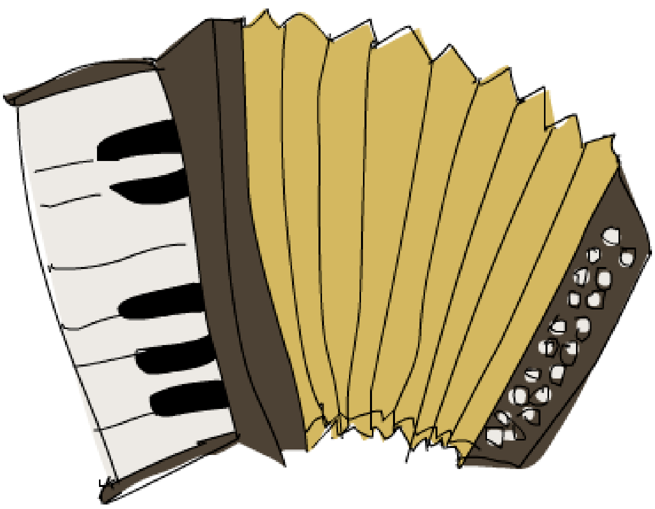
We said that a tree is either an empty tree that doesn't hold any values or it's a node that holds one value and also two other trees. After defining it, we made it an instance of **`Functor`** and with that we gained the ability to **`fmap`** functions over it. Now, we're going to make it an instance of **`Foldable`** so that we get the abilty to fold it up. One way to make a type constructor an instance of **`Foldable`** is to just directly implement **`foldr`** for it. But another, often much easier way, is to implement the **`foldMap`** function, which is also a part of the **`Foldable`** type class. The **`foldMap`** function has the following type:

```
foldMap :: (Monoid m, Foldable t) => (a -> m) -> t a -> m
```

Its first parameter is a function that takes a value of the type that our foldable structure contains (denoted here with **`a`**) and returns a monoid value. Its second parameter is a foldable structure that contains values of type **`a`**. It maps that function over the foldable structure, thus producing a foldable structure that contains monoid values. Then, by doing **`mappend`** between those monoid values, it joins them all into a single monoid value. This function may sound kind of odd at the moment, but we'll see that it's very easy to implement. What's also cool is that implementing this function is all it takes for our type to be made an instance of **`Foldable`**. So if we just implement **`foldMap`** for some type, we get **`foldr`** and **`foldl`** on that type for free!

This is how we make **`Tree`** an instance of **`Foldable`**:

```
instance F.Foldable Tree where
    foldMap f Empty = mempty
    foldMap f (Node x l r) = F.foldMap f l `mappend`
                             f x           `mappend`
                             F.foldMap f r
```



We think like this: if we are provided with a function that takes an element of our tree and returns a monoid value, how do we reduce our whole tree down to one single monoid value? When we were doing **`fmap`** over our tree, we applied the function that we were mapping to a node and then we recursively mapped the function over the left sub-tree as well as the right one. Here, we're tasked with not only mapping a function, but with also joining up the results into a single monoid value by using **`mappend`**. First we consider the case of the empty tree — a sad and lonely tree that has no values or sub-trees. It doesn't hold any value that we can give to our monoid-making function, so we just say that if our tree is empty, the monoid value it becomes is **`mempty`**.

The case of a non-empty node is a bit more interesting. It contains two sub-trees as well as a value. In this case, we recursively **`foldMap`** the same function **`f`** over the left and the right sub-trees. Remember, our **`foldMap`** results in a single monoid value. We also apply our function **`f`** to the value in the node. Now we have three monoid values (two from our sub-trees and one from applying **`f`** to the value in the node) and we just have to bang them together into a single value. For this purpose we use **`mappend`**, and naturally the left sub-tree comes first, then the node value and then the right sub-tree.

Notice that we didn't have to provide the function that takes a value and returns a monoid value. We receive that function as a parameter to **`foldMap`** and all we have to decide is where to apply that function and how to join up the resulting monoids from it.

Now that we have a **`Foldable`** instance for our tree type, we get **`foldr`** and **`foldl`** for free! Consider this tree:

```
testTree = Node 5
            (Node 3
                (Node 1 Empty Empty)
                (Node 6 Empty Empty)
            )
            (Node 9
                (Node 8 Empty Empty)
                (Node 10 Empty Empty)
            )
```

It has **`5`** at its root and then its left node is has **`3`** with **`1`** on the left and **`6`** on the right. The root's right node has a **`9`** and then an **`8`** to its left and a **`10`** on the far right side. With a **`Foldable`** instance, we can do all of the folds that we can do on lists:

```
ghci> F.foldl (+) 0 testTree
42
ghci> F.foldl (*) 1 testTree
64800
```

And also, **`foldMap`** isn't only useful for making new instances of **`Foldable`**; it comes in handy for reducing our structure to a single monoid value. For instance, if we want to know if any number in our tree is equal to **`3`**, we can do this:

```
ghci> getAny $ F.foldMap (\x -> Any $ x == 3) testTree
True
```

Here, **`\x -> Any $ x == 3`** is a function that takes a number and returns a monoid value, namely a **`Bool`** wrapped in **`Any`**. **`foldMap`** applies this function to every element in our tree and then reduces the resulting monoids into a single monoid with **`mappend`**. If we do this:

```
ghci> getAny $ F.foldMap (\x -> Any $ x > 15) testTree
False
```

All of the nodes in our tree would hold the value **`Any False`** after having the function in the lambda applied to them. But to end up **`True`**, **`mappend`** for **`Any`** has to have at least one **`True`** value as a parameter. That's why the final result is **`False`**, which makes sense because no value in our tree is greater than **`15`**.

We can also easily turn our tree into a list by doing a **`foldMap`** with the **`\x -> [x]`** function. By first projecting that function onto our tree, each element becomes a singleton list. The **`mappend`** action that takes place between all those singleton list results in a single list that holds all of the elements that are in our tree:

```
ghci> F.foldMap (\x -> [x]) testTree
[1,3,6,5,8,9,10]
```

What's cool is that all of these trick aren't limited to trees, they work on any instance of **Foldable**.

Resolvendo Problemas Funcionalmente Índice Um punhado de Monads