Haskell Monads

Bachelor's Degree Thesis

Academic year 2019/2020

UNIVERSIDAD COMPLUTENSE
MADRID

Faculty of Mathematics
Mathematics Degree

Student: *Ramiro Pastor Martín*

supervised by
Luis Fernando Llana Díaz

Madrid, February 25th of 2020

Contents

Chapter 1

Introduction

1.1 Motivation

Monads are an essential part of Haskell, starting with input/output. All I/O operations, from reading or writing to the console to editing files, are monadic actions in the IO monad. For example, reading a value from the console returns a type 'IO a' and there is no safe way to extract the value of type 'a' outside the IO type, thus it is mandatory to use the monad operators and techniques to operate with that value.

Furthermore, the monad typeclass is intimately related to the 'do' notation which enables imperative programming in Haskell, a declarative language. This means that anytime you have a monad, you can write do blocks, switching to imperative programming. This applies to every monad, from the IO type to all other monadic types, for example, do blocks can return HTML types, HTTP actions, or even LaTeX documents. The rules for do blocks are pretty simple so it is easy to sequence statements in the corresponding monadic type. Finally, whenever more than one monad are involved, they can be mixed with the 'lift' or the 'liftIO' functions, which correspond to the monad transformers typeclasses.

1.2 Objectives

The aim of this work is to explain the Monad typeclass of Haskell. It begins with a brief introduction to Category Theory, from where the concept of Monad is taken; however, it does not delve into the theory too much, because it is not a prerequisite to learn about Haskell's monads, which can be used without any knowledge of Category Theory. It is there merely for completition purposes (and to expose where the term 'monad' comes from)

Chapter two is dedicated to Haskell entirely, introducing the Monad typeclass and its superclasses Functor and Applicative, explaining the 'do' notation (a tool associated with monads), and ending with the MonadPlus typeclass and Monad Transformers.

Chapter three revisits the Functor, Applicative and Monad typeclasses to expose the similarities between them, and links with the first chapter thanks to the 'Hask' category. Finally, there are some appendixes which include exercises, random modelization in Haskell and more.

Chapter 2

Introduction to Categories

This is a very brief introduction to Category Theory. It features the definition of category, functor and natural transformation, as well as a quick insight into monads, without going into details - it is merely being exposed for completition purposes.

2.1 Category

2.1.1 Definition

Def: a **category** is a tern $\langle \mathbf{Obj}(\mathfrak{C}), \mathbf{Hom}(\mathfrak{C}), \odot \rangle$ where :

1. $Obj(\mathfrak{C})$ is a class (not necessarily a set) whose members are called **objects** of $\mathfrak{C}$. In practice one often abuses notation by denoting the class of objects of $\mathfrak{C}$ by the letter $\mathfrak{C}$ as well. In particular, the notation $X \in \mathfrak{C}$ is to be understood as "X is an object of $\mathfrak{C}$".

2. $Hom(\mathfrak{C})$ is a class (if $Obj(\mathfrak{C})$ is a class) or a set (if $Obj(\mathfrak{C})$ is a set) whose members are called **morphisms** of $\mathfrak{C}$. Each morphism f of $\mathfrak{C}$ is associated with a **departure object** X, and an **arrival object** Y, both from $Obj(\mathfrak{C})$; we write this as "f goes from X to Y" or $f : X \to Y$ or $X \xrightarrow{f} Y$.

   The (always a) set of morphisms from X to Y in the category $\mathfrak{C}$ is denoted as $\mathbf{Hom}_{\mathfrak{C}}(\mathbf{X}, \mathbf{Y})$. Also, instead of $Hom_{\mathfrak{C}}(X, X)$ we will write $Endo_{\mathfrak{C}}(X)$; its elements are called **endomorphisms** of X.

3. A composition law $\odot$ that $\forall\ X, Y, Z \in \mathfrak{C}$:

$$Hom_{\mathfrak{C}}(X, Y) \times Hom_{\mathfrak{C}}(Y, Z) \to Hom_{\mathfrak{C}}(X, Z)$$

$$(f, g) \mapsto g \odot f$$

   this is, for every $X \xrightarrow{f} Y \xrightarrow{g} Z$ there must exist a morphism $h : X \to Z$ assigned to $g \odot f$. It must verify:

   Associativity Composition of morphisms is associative. More precisely, given objects X, Y, Z, W of $\mathfrak{C}$ and morphisms $X \xrightarrow{f} Y \xrightarrow{g} Z \xrightarrow{h} W$ we require that $h \odot (g \odot f) = (h \odot g) \odot f$

Neutral elements Every object has an "identity endomorphism". More precisely, if $X \in \mathfrak{C}$, there exists an element $id_X \in End o_{\mathfrak{C}}(X)$ such that for every morphism $f : X \to Y$ in $Hom(\mathfrak{C})$, we have $f \odot id_X = f$, and for every morphism $g : Z \to X$ in $Hom(\mathfrak{C})$, we have $id_X \odot g = g$.

***Obs:* 1.** *In view of the associative axiom, whenever we have any composable sequence $f_1, \ldots, f_n$ of morphisms in a category, the expression $f_n \odot f_{n-1} \odot \ldots \odot f_2 \odot f_1$ is unambiguous.*

2.1.2 Unicity of neutral elements and examples

***Prop:* 1.** *For any category $\mathfrak{C}$ and any object $X \in \mathfrak{C}$, there is only one endomorphism of X satisfying the defining property of id_X. Thus one can really speak of the identity endomorphism of X.*

Proof. Given $X \in \mathfrak{C}$ suppose that there are two endomorphisms of X, $\widetilde{id_X}$ and $\widehat{id_X}$, with the property of neutral element.

This is, $\forall\ f \in Hom(A, X)$ and $\forall\ g \in Hom(X, B)$ occurs:

$$\tilde{id_X} \odot f = f \qquad\qquad \hat{id_X} \odot f = f$$
$$g \odot \tilde{id_X} = g \qquad\qquad g \odot \hat{id_X} = g$$

If we apply this to $\hat{id_X} \odot \tilde{id_X}$ it falls that:

$$\hat{id_X} = \hat{id_X} \odot \tilde{id_X} = \tilde{id_X} \implies \hat{id_X} = \tilde{id_X}$$

□

Examples: (note: $\odot$ is always the usual function composition $\circ$ unless said otherwise.)

$\mathfrak{Set}$: Obj($\mathfrak{Set}$) – the class of all sets. Hom($\mathfrak{Set}$) – functions between sets.

$\mathfrak{Grp}$: Obj($\mathfrak{Grp}$) – the class of all groups. Hom($\mathfrak{Grp}$) – group homomorphisms.

$\mathfrak{Top}$: Obj($\mathfrak{Top}$) – the class of all topological spaces. Hom($\mathfrak{Top}$) – continuous maps between topological spaces.

$\mathfrak{Vect}_{\mathbb{K}}$: Obj($\mathfrak{Vect}_{\mathbb{K}}$) – the class of all vector spaces over a given field $\mathbb{K}$. Hom($\mathfrak{Vect}_{\mathbb{K}}$) – linear maps between vector spaces.

$\mathfrak{Hask}$: Obj($\mathfrak{Hask}$) – the class of all Haskell types. Hom($\mathfrak{Hask}$) – Haskell functions. The composition law is the (.) operator.

$\leq$: any partially ordered set $\langle P, \leq \rangle$ defines a category where the objects are the elements of P, and there is a morphism (and only one) between any two objects A and B iff $A \leq B$.

This can be applied to the power set $\wp$(X) of any given set X, taking the inclusion as the partial order.

2.1.3 Isomorphisms and Automorphisms

Def: Let $\mathfrak{C}$ be a category and $X \xrightarrow{f} Y$ a morphism in $\mathfrak{C}$. We say that f is an **isomorphism**, or that f is **invertible**, if there exists a morphism $g : Y \to X$ in $\mathfrak{C}$ such that $g \odot f = id_X$ and $f \odot g = id_Y$. If this is the case, g is called an **inverse** of f, and viceversa.

***Prop:* 2.** *If f has an inverse, that inverse is unique (so, if f is an isomorphism, one can unambiguously denote its inverse by f^{-1}).*

Proof. given $f : X \to Y$ such that exists $g : Y \to X$ and $g* : Y \to X$ inverses of f, lets conclude that g = g*. We have:

$$g \odot f = id_X \qquad f \odot g = id_Y$$
$$g* \odot f = id_X \qquad f \odot g* = id_Y$$

beginning with the first ecuation, and composing with g* to the right, we have $g \odot f \odot g* = id_X \odot g*$ and applying the last ecuation, we get $g \odot id_Y = id_X \odot g* \implies g = g*$ □

***Prop:* 3.** *If $f : X \to Y$ and $g : Y \to Z$ are composable morphisms and are both invertible, then $g \odot f$ is also invertible, and $(g \odot f)^{-1} = f^{-1} \odot g^{-1}$.*

Proof. the existence of $(g \odot f)^{-1}$ will be given automatically as soon as we prove that $(g \odot f)^{-1} = f^{-1} \odot g^{-1}$ because we know that $f^{-1} \odot g^{-1}$ must exist.

In addition, with the already proved uniqueness of inverses (last proposition) and its consequent unambiguity in the use of the f^{-1} notation, we only need to prove that composing the function $g \odot f$ with the function $f^{-1} \odot g^{-1}$ gives us the indentity function in both X and Z. Indeed:

$$(g \odot f) \odot f^{-1} \odot g^{-1} = \overset{\text{associativity}}{\longrightarrow} = g \odot \left(f \odot f^{-1}\right) \odot g^{-1} = id_Z$$
$$f^{-1} \odot g^{-1} \odot (g \odot f) = \overset{\text{associativity}}{\longrightarrow} = f^{-1} \odot \left(g^{-1} \odot g\right) \odot f = id_X$$

□

***Obs:* 2.** *the reciprocal proposition (invertible composition implies invertible factors) is not true in general. As a counterexample, in the* **sets** *category take f,g : $\mathbb{N} \to \mathbb{N}$ with $f(x) = 2x$ and $g(x) = \lfloor x/2 \rfloor$, and compose $g \circ f$.*

Def: If X, Y are objects of a category $\mathfrak{C}$ such that there exists an isomorphism (i.e. invertible) $f : X \to Y$, we say that X and Y are **isomorphic**, and write $X \cong Y$ or f : X $\xrightarrow{\sim}$ Y.

***Obs:* 3.** *Usually, if an isomorphism between X and Y exists, it is non-unique. For example, there exists a bijection between two finite sets $\leftrightarrow$ both have the same number n of elements, and in that case there are exactly n! bijections between them.*

Def: If $\mathfrak{C}$ is a category and $X \in \mathfrak{C}$, the invertible endomorphisms of X are called **automorphisms**.

Def: If $\mathfrak{C}$ is a category and $X \in \mathfrak{C}$, the set of all invertible endomorphisms of X will be denoted by $\mathbf{Aut}_{\mathfrak{C}}(\mathbf{X})$.

***Prop:* 4.** *$Aut_{\mathfrak{C}}(X)$ is a group under the composition of morphisms. It is called the group of automorphisms of X.*

2.1.4 Groupoids and Subcategories

Def: A **groupoid** is a category in which every morphism is invertible.

Def: Let $\mathfrak{C}$ be any category, and let $\mathfrak{C}^{\mathbf{x}}$ be the category whose class of objects is that of $\mathfrak{C}$, and where the morphisms are defined as follows. If X, Y are two objects, then $Hom_{\mathfrak{C}^x}(X, Y)$ is the set of invertible elements of $Hom_{\mathfrak{C}}(X, Y)$. The composition of morphisms in $\mathfrak{C}^x$ is defined as the composition in $\mathfrak{C}$.

***Prop:* 5.** *$\mathfrak{C}^x$ is a category and is a groupoid.*

Def: Let $\mathfrak{C}$ be a category. We say that a category $\mathfrak{D}$ is a **subcategory** of $\mathfrak{C}$ if: the class of objects of $\mathfrak{D}$ is a subclass of the class of objects of $\mathfrak{C}$, i.e. $Obj(\mathfrak{D}) \subset Obj(\mathfrak{C})$; for every pair X, Y of objects of $\mathfrak{D}$, the set $Hom_{\mathfrak{D}}(X, Y)$ is a subset of $Hom_{\mathfrak{C}}(X, Y)$; and composition of two morphisms in $\mathfrak{D}$ is the same regardless of whether it is computed in $\mathfrak{D}$ or in $\mathfrak{C}$.

We say that $\mathfrak{D}$ is a **full subcategory** of $\mathfrak{C}$ if, for every pair of objects X, Y of $\mathfrak{D}$, one has $Hom_{\mathfrak{D}}(X, Y) = Hom_{\mathfrak{C}}(X, Y)$ (this is, we only lose objects and not morphisms).

2.2 Functors and natural transformations

From now on, the symbol $\odot$ will be replaced by $\circ$

2.2.1 Definition

Def: Let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be categories. A **covariant functor** $\Phi : \mathfrak{C}_1 \to \mathfrak{C}_2$ is a rule which to every object X of $\mathfrak{C}_1$ assigns an object $\Phi(X)$ of $\mathfrak{C}_2$, and to every morphism $f : X \to Y$ in $\mathfrak{C}_1$ assigns a morphism $\Phi(f) : \Phi(X) \to \Phi(Y)$ in $\mathfrak{C}_2$ such that $\Phi(g \circ f) = \Phi(g) \circ \Phi(f)$ whenever $X \xrightarrow{f} Y \xrightarrow{g} Z$ are morphisms in $\mathfrak{C}_1$, and such that $\Phi(id_X) = id_\Phi(X)$ for all objects X of $\mathfrak{C}_1$.

To get the notion of a **contravariant functor** $\Psi : \mathfrak{C}_1 \to \mathfrak{C}_2$ one has to make the following changes: $\Psi(f)$ should now be a morphism from $\Psi(Y)$ to $\Psi(X)$ (i.e., Ψ"reverses the directions of all arrows"), and the first requirement in the definition of a functor has to be replaced by $\Psi(g \circ f) = \Psi(f) \circ \Psi(g)$.

Usually, the word "functor" without any adjectives refers to a covariant functor. I will also use this convention from now on.

2.2.2 Examples of functors

There are plenty:

1. For any category $\mathfrak{C}$ we have the *identity functor* $Id_\mathfrak{C} : \mathfrak{C} \to \mathfrak{C}$.
2. If $\mathfrak{C}$ is a category and $\mathfrak{D} \subseteq \mathfrak{C}$ is a subcategory, one has the obvious "inclusion functor "$\mathfrak{D} \hookrightarrow \mathfrak{C}$. In particular, we have inclusion functors $\mathfrak{Vect}_\mathbb{K} \hookrightarrow \mathfrak{Grp} \hookrightarrow \mathfrak{Set}$. These functors are usually called the "forgetful functors".
3. The *power set functor* $\mathfrak{Set} \to \mathfrak{Set}$ which maps sets to their power sets; and maps functions $f : X \to Y$ to functions $\wp(X) \to \wp(Y)$ which take inputs $U \subseteq X$ and return $f(U)$, the image of U under f, defined by $f(U) \quad = \quad \{ \quad f(u) : u \in U \quad \}$
4. The fundamental group is a functor from $\mathfrak{Top}*$, the category of *pointed topological spaces*[1], to $\mathfrak{Grp}$. More precisely, check that if $f : (X, x) \to (Y, y)$ is a morphism in $\mathfrak{Top}*$, then one can use f to define a group homomorphism $\pi_1(X, x) \to \pi_1(Y, y)$, and this yields a functor $\mathfrak{Top}* \to \mathfrak{Grp}$.

2.2.3 The dual category

Def: Let $\mathfrak{C}$ be any category. The **dual category** $\mathfrak{C}^\circ$ of $\mathfrak{C}$ is informally speaking, obtained from $\mathfrak{C}$ by "reversing all the arrows". This is:

$$Obj(\mathfrak{C})^\circ := Obj(\mathfrak{C}) \quad ; \quad Hom_{\mathfrak{C}^\circ}(X, Y) := Hom_\mathfrak{C}(Y, X) \quad with \quad f^\circ \circ g^\circ = (g \circ f)^\circ$$

Prop: **6.** *The rule $X \mapsto X$, $f \mapsto f^\circ$ defines a contravariant functor $\mathfrak{C} \to \mathfrak{C}^\circ$*

Prop: **7.** *If $\mathfrak{C}_1$ and $\mathfrak{C}_2$ are two categories, a covariant functor $\Phi : \mathfrak{C}_1 \to \mathfrak{C}_2$ can also be though of as a contravariant functor ${\mathfrak{C}_1}^\circ \to \mathfrak{C}_2$, or a contravariant functor $\mathfrak{C}_1 \to {\mathfrak{C}_2}^\circ$, or a covariant functor ${\mathfrak{C}_1}^\circ \to {\mathfrak{C}_2}^\circ$. The same holds if we switch "covariant" and "contravariant" throughout the last sentence.*

[1]The objects of $\mathfrak{Top}*$ are pairs (X, x) consisting of a topological space X and a point $x \in X$. A morphism $f : (X, x) \to (Y, y)$ in $\mathfrak{Top}*$ is a continuous map $f : X \to Y$ such that $f(x) = y$. Composition of morphisms is defined as the composition of maps in the usual sense.

2.2.4 Natural transformations

Def: Let $\mathfrak{C}_1$ and $\mathfrak{C}_2$ be categories, and let $\Phi, \Psi : \mathfrak{C}_1 \to \mathfrak{C}_2$ be functors. A **morphism of functors**, or a **natural transformation**, $\alpha : \Phi \to \Psi$, is a rule which to every object $X \in \mathfrak{C}_1$ assigns a morphism $\alpha_X : \Phi(X) \to \Psi(X)$ such that for any morphism $X \xrightarrow{f} Y$ in $\mathfrak{C}_1$, the following diagram commutes:

$$\begin{array}{ccc} \Phi(X) & \xrightarrow{\Phi(f)} & \Phi(Y) \\ {\scriptstyle \alpha_X}\downarrow & & \downarrow{\scriptstyle \alpha_Y} \\ \Psi(X) & \xrightarrow{\Psi(f)} & \Psi(Y) \end{array}$$

Def: We say that the collection $(\alpha_X)_{X \in \mathfrak{C}}$ is an **isomorphism (of functors)** between Φ and Ψ if each morphism α_X is invertible. In this case the collection $\left({\alpha_X}^{-1}\right)$ defines a morphism of functors from Ψ to Φ. We call this collection α^{-1}.

2.2.5 Composition of functors and natural transformations

Def: Let $\mathfrak{C}_1, \mathfrak{C}_2, \mathfrak{C}_3$ be categories and let $\Phi : \mathfrak{C}_1 \to \mathfrak{C}_2$, $\Psi : \mathfrak{C}_2 \to \mathfrak{C}_3$ be functors. The **composed functor** $\Psi \circ \Phi : \mathfrak{C}_1 \to \mathfrak{C}_3$ assigns to each object $X \in \mathfrak{C}_1$ the object $\Psi(\Phi(X)) \in \mathfrak{C}_3$; and to each morphism f of $Hom_{\mathfrak{C}_1}(X, Y)$, the morphism $\Psi(\Phi(f)) \in Hom_{\mathfrak{C}_3}(\Psi(\Phi(X)), \Psi(\Phi(Y)))$

Similarly, if $\mathfrak{C}$ and $\mathfrak{D}$ are categories, $\Phi_1, \Phi_2, \Phi_3 : \mathfrak{C} \to \mathfrak{D}$ are three functors, and $\alpha : \Phi_1 \to \Phi_2$, $\beta : \Phi_2 \to \Phi_3$ are natural transformations, invent the definition of the composition $\beta \circ \alpha : \Phi_1 \to \Phi_3$. In fact, modulo some set-theoretical issues (which should be ignored at this point), one can define the category of functors $\mathfrak{Funct}(\mathfrak{C}, \mathfrak{D})$ whose objects are functors from $\mathfrak{C}$ to $\mathfrak{D}$ and whose morphisms are natural transformations.

2.3 Commutative diagram and monad definition

The concept of monad is found deep within the theory of Categories, far beyond the point in which we are now. In fact, the full theory can be built without set theory, with its own beauties such as expressing algebraic identities as commutative diagrams. The concept of *commutative diagram* is itself basic for this approach, so it will be exposed now.

Also, i will include the definition of *monad* from *Category Theory - Steve Awodey*

Def: A diagram (such as the ones below) is **commutative** when, for each pair of vertices c and c', any two paths formed from directed edges leading from c to c' yield, by composition of labels, equal morphisms from c to c'.

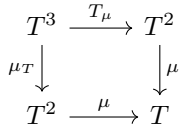
A considerable part of the effectiveness of categorical methods rests on the fact that such diagrams in each situation vividly represent the actions of the arrows at hand.

Def: A **monad** on a category $\mathfrak{C}$ consists of an endofunctor $T : \mathfrak{C} \to \mathfrak{C}$, and natural transformations $\eta : 1_{\mathfrak{C}} \to T$, and $\mu : T^2 \to T$ satisfying the two commutative diagrams below, that is,

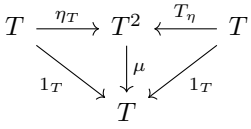
$$\mu \circ \mu_T = \mu \circ T_\mu$$

$$\mu \circ \eta_T = 1 = \mu \circ T_\eta$$

Note the formal analogy to the definition of a monoid. In fact, a monad is exactly the same thing as a *monoidal monoid* in the monoidal category $\mathfrak{C}^{\mathfrak{C}}$ with composition as the monoidal product, $G \otimes F = G \circ F$



$$\mu \circ \mu_T = \mu \circ T_\mu$$



$$\mu \circ \eta_T = 1_T = \mu \circ T_\eta$$

Chapter 3

Haskell Monad class

As seen in the previous chapter, monad definition in Mathematics lies beyond a long and winding path (we saw both ends, but the in-between theory was omitted); etimology doesn't help either, leading to:

Monad (n.): "Unity, arithmetical unit", 1610s, from Late Latin *monas* (genitive *monadis*), from Greek *monas* "unit", from *monos*"alone" (see *mono*). In Leibnitz's philosophy, "an ultimate unit of being" (1748). Related: *Monadic*.

So, as even more questions arise, lets sort them up:

3.1 Why? the IO monad

Beyond internally calculating values, we want our programs to interact with the world. The most common beginners' program in any language simply displays a "hello world" greeting on the screen. Here's a Haskell version:

```
Prelude> putStrLn "Hello, World!"
```

So now you should be thinking, "what is the type of the putStrLn function?" It takes a `String` and gives... um... what? What do we call that? The program doesn't get something back that it can use in another function. Instead, the result involves having the computer change the screen. In other words, it does something in the world outside of the program. What type could that have? Let's see what GHCi tells us:

```
Prelude> :t putStrLn
putStrLn :: String -> IO ()
```

"IO" stands for "input and output". Wherever there is `IO` in a type, interaction with the world outside the program is involved. We'll call these `IO` values *actions*. The other part of the `IO` type, in this case `()`, is the type of the return value of the action; that is, the type of what it gives back to the program (as opposed to what it does outside the program). `()` (pronounced as "unit") is a type that only contains one value also called `()` (effectively a tuple with zero elements). Since `putStrLn` sends output to the world but doesn't return anything to the program, `()` is used as a placeholder. We might read `IO ()` as "action which returns `()`".What makes IO actually work? Lots of things happen behind the scenes to take us from `putStrLn` to pixels in the screen, but we don't need to understand any of the details to write our programs. A complete Haskell program is actually a big IO action. In a compiled program, this action is called `main` and has type `IO ()`.

From this point of view, to write a Haskell program is to combine actions and functions to form the overall action main that will be executed when the program is run. The compiler takes care of instructing the computer on how to do this.

3.2 What?

Monads are by no means limited to input and output. Monads support a whole range of things like exceptions, state, non-determinism, continuations, coroutines, and more. In fact, thanks to the versatility of monads, none of these constructs needed to be built into Haskell as a language; instead, they are defined by the standard libraries.

3.2.1 Starring: Monad typeclass

In Haskell, the Monad type class is used to implement monads. It is provided by the Control.Monad module and included in the Prelude. The class has the following methods:[1]

```
class Monad m where
return :: a -> m a
(>>=)  :: m a -> (a -> m b) -> m b

(>>)   :: m a -> m b -> m b
fail   :: String -> m a
```

The core methods are return and (>>=) (which is pronounced "bind"). Aside from return and bind, notice the two additional functions (>>) and fail. The operator (>>) called "then" is a mere convenience and commonly implemented as

```
m >> n   =   m >>= \_ -> n
```

(>>) sequences two monadic actions when the second action does not involve the result of the first, which is common for monads like IO. The function fail handles pattern match failures in do notation. It's an unfortunate technical necessity and doesn't really have anything to do with monads. You are advised not to call fail directly in your code.

[1] For the full definition of Monad in the Prelude, look Appendix B

3.3 Pre-example with Maybe

For a concrete example, take the Maybe monad. The type constructor is m = Maybe, while return and (>>=) are defined like this:

```
return :: a -> Maybe a
return x  = Just x

(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
             Nothing -> Nothing
             Just x  -> g x
```

Maybe is the monad, and return brings a value into it by wrapping it with Just. As for (>>=), it takes a m :: Maybe a value and a $g :: a \rightarrow$ Maybe b function. If m is Nothing, there is nothing to do and the result is Nothing. Otherwise, in the Just x case, g is applied to x, the underlying value wrapped in Just, to give a Maybe b result, which might be Nothing, depending on what g does to x. To sum it all up, if there is an underlying value in m, we apply g to it, which brings the underlying value back into the Maybe monad.

The key first step to understand how return and (>>=) work is tracking which values and arguments are monadic and which ones aren't. As in so many other cases, type signatures are our guide to the process.

Motivation: Maybe

To see the usefulness of (>>=) and the Maybe monad, consider the following example: Imagine a family database that provides two functions

```
father :: Person -> Maybe Person
mother :: Person -> Maybe Person
```

These look up the name of someone's father or mother. In case our database is missing some information, Maybe allows us to return a Nothing value instead of crashing the program. Let's combine our functions to query various grandparents. For instance, the following function looks up the maternal grandfather:

```
maternalGrandfather :: Person -> Maybe Person
maternalGrandfather p =
case mother p of
    Nothing -> Nothing
    Just mom -> father mom
```

Or consider a function that checks whether both grandfathers are in the database:

```
bothGrandfathers :: Person -> Maybe (Person, Person)
bothGrandfathers p =
    case father p of
        Nothing -> Nothing
        Just dad ->
            case father dad of
                Nothing -> Nothing
                Just gf1 ->                    -- 1st grandfather
```

```
case mother p of
    Nothing -> Nothing
    Just mom ->
        case father mom of
            Nothing -> Nothing
            Just gf2 ->    -- 2nd grandfather
                Just (gf1, gf2)
```

What a mouthful! Every single query might fail by returning `Nothing` and the whole function must fail with `Nothing` if that happens. Clearly there as to be a better way to write that instead of repeating the case of `Nothing` again and again! Indeed, that's what the `Maybe` monad is set out to do. For instance, the function retrieving the maternal grandfather has exactly the same structure as the `(>>=)` operator, so we can rewrite it as:

```
maternalGrandfather p   =   mother p >>= father
```

With the help of lambda expressions and return, we can rewrite the two grandfathers function as well:

```
bothGrandfathers p =
    father p >>=
        (\dad -> father dad >>=
            (\gf1 -> mother p >>=
-- this line works as '\_ -> mother p', but naming gf1 allows later return
                (\mom -> father mom >>=
                    (\gf2 -> return (gf1,gf2) ))))
```

While these nested lambda expressions may look confusing to you, the thing to take away here is that `(>>=)` releases us from listing all the `Nothing`s, shifting the focus back to the interesting part of the code. To be a little more precise: The result of `father p` is a monadic value (in this case, either `Just dad` or `Nothing`, depending on whether `p`'s dad is in the database). As the `father` function takes a regular (non-monadic value), the `(>>=)` feeds `p`'s dad to it as a *non-monadic* value. The result of `father dad` is then monadic again, and the process continues.

So, `(>>=)` helps us pass non-monadic values to functions without leaving a monad. In the case of the `Maybe` monad, the monadic aspect is the qualifier that we don't know with certainty whether the value will be found.

3.3.1 Notions of Computation

We've seen how `(>>=)` and `return` are very handy for removing boilerplate code that crops up when using `Maybe`. That, however, is not enough to justify why monads matter so much. We will continue our monad studies by rewriting the two-grandfathers function using `do` notation with explicit braces and semicolons. Depending on your experience with other programming languages, you may find this very suggestive:

```
bothGrandfathers p = do {
    dad <- father p;
    gf1 <- father dad;
    mom <- mother p;
    gf2 <- father mom;
    return (gf1, gf2);
}
```

If this looks like a code snippet of an imperative programming language to you, that's because it is. In particular, this imperative language supports *exceptions*: father and mother are functions that might fail to produce results, i.e. raise an exception, and when that happens, the whole `do`-block will fail, i.e. terminate with an exception.

In other words, the expression `father p`, which has type `Maybe Person`, is interpreted as a statement of an imperative language that returns a `Person` as result**. This is true for all monads: a value of type `M a` is interpreted as a statement of an imperative language that returns a value of type `a` as result; and the semantics of this language are determined by the monad `M`.**[2]

Under this interpretation, the bind operator `(>>=)` is simply a function version of the semicolon. Just like a `let` expression can be written as a function application,

```
let x = foo in x + 3      corresponds to      (\x -> x + 3) foo
```

an assignment and semicolon can be written as the bind operator:

```
x <- foo; return (x + 3)
corresponds to
foo >>= (\x -> return (x + 3))
```

The `return` function lifts a value `a` to `M a`, a full-fledged statement of the imperative language corresponding to the monad `M`.

Different semantics of the imperative language correspond to different monads. The following table shows the classic selection that every Haskell programmer should know. If the idea behind monads is still unclear to you, studying each of the examples in the following chapters will not only give you a well-rounded toolbox but also help you understand the common abstraction behind them.

Monad	Imperative Semantics	Found in Prelude
`Maybe`	Exception (anonymous)	Yes
`Error`	Exception (with error description)	No
`State`	Global state	No
`IO`	Input/Output	Yes
`[ ]` (lists)	Nondeterminism	Yes
`Reader`	Environment	No
`Writer`	Logger	No

Furthermore, these different semantics need not occur in isolation. As we will see in a few chapters, it is possible to mix and match them by using monad transformers to combine the semantics of multiple monads in a single monad.

[2] By 'semantics', we mean what the language allows you to say. In the case of `Maybe`, the semantics allow us to express failure, as statements may fail to produce a result, leading to the statements that follow it being skipped.

3.4 Who?

The first observation when studying the monad definition in the Prelude is that its a type class, just like `Eq`, `Ord` or `Num`. As such, instead of *what is a monad?* we should be asking ourselves *what is TO BE monad?* - because that's how classes work and help us, enhancing types with new capabilities; for example, the `Eq` and `Ord` classes provide comparability between that type elements, and the `Num` class allows the use of `+` or `*`.

In fact, with a little help with the GHCi command `:kind` we can already answer the question *what types can be made instance of the Monad class?*. Check it yourself!

```
Prelude> :k Bool
Bool :: *
Prelude> :k Int
Int :: *
Prelude> :k []
[] :: * -> *
Prelude> :k [Int]
[Int] :: *
Prelude> :k Maybe
Maybe :: * -> *
Prelude> :k (,,,,)
(,,,,) :: * -> * -> * -> * -> * -> * -> *
Prelude> :k Eq
Eq :: * -> Constraint
Prelude> :k Ord
Ord :: * -> Constraint
Prelude> :k Num
Num :: * -> Constraint
Prelude> :k Show
Show :: * -> Constraint
Prelude> :k Functor
Functor :: (* -> *) -> Constraint
Prelude> :k Monad
Monad :: (* -> *) -> Constraint

Prelude> :t Constraint

<interactive>:1:1: Not in scope: data constructor ‘Constraint’
Prelude> :k Constraint

<interactive>:1:1:
    Not in scope: type constructor or class ‘Constraint’
Prelude> :m GHC.Prim
Prelude GHC.Prim> :k Constraint
Constraint :: BOX
Prelude GHC.Prim> :k BOX
BOX :: BOX

Prelude> :m Data.Monoid
Prelude Data.Monoid> :k Monoid
Monoid :: * -> Constraint
```

Looking closely the kind of Monad, we get that **only 1-parameterized types** are allowed to be instantiated in the `Monad` class. This is, types like `Maybe a`, `[a]` or `(a)`; but not `Int`, `Bool` or `Either a b` (however, `Either Int a` will do the trick). As soon as GHCi meets the "`instance Monad Int where`" line, the following error will be displayed:

```
The first argument of ‘Monad’ should have kind ‘* -> *’,
 but ‘Int’ has kind ‘*’
 In the instance declaration for ‘Monad Int’
```

You don't program with kinds: the compiler infers them for itself. But if you get parameterized types wrong then the compiler will report a kind error.

3.5 How?

3.5.1 The Rules

In Haskell, every instance of the `Monad` type class (and thus all implementations of bind `(>>=)` and `return`) must obey the following three laws:

```
m >>= return      =   m                          -- right unit
return x >>= f    =   f x                        -- left unit

(m >>= f) >>= g   =   m >>= (\x -> f x >>= g)  -- associativity
```

The behavior of `return` is specified by the left and right unit laws. They state that `return` doesn't perform any computation, it just collects values.

The law of associativity makes sure that (like the semicolon) the bind operator `(>>=)` only cares about the order of computations, not about their nesting. The associativity of the *then* operator `(>>)` is a special case:

```
(m >> n) >> o  =  m >> (n >> o)
```

3.5.2 Monadic composition

It is easier to picture the associativity of bind by recasting the law as

```
(f >=> g) >=> h  =  f >=> (g >=> h)
```

where `(>=>)` is the **monad composition operator**, a close analogue of the function composition operator `(.)`, only with flipped arguments. It is defined as:

```
(>=>) :: Monad m => (a -> m b) -> (b -> m c) -> a -> m c
f >=> g   =   \x -> f x >>= g
```

We can also flip monad composition to go the other direction using `(<=<)`.

3.5.3 Alternative definitions

Monads originally come from a branch of mathematics called Category Theory. Fortunately, it is entirely unnecessary to understand category theory in order to understand and use monads in Haskell. The definition of monads in Category Theory actually uses a slightly different presentation. Translated into Haskell, this presentation gives an alternative yet equivalent definition of a monad which can give us some additional insight.

So far, we have defined monads in terms of `(>>=)` and `return`. The alternative definition, instead, starts with monads as functors with two additional combinators:

```
fmap   :: (a -> b) -> M a -> M b  -- functor
return :: a -> M a
join   :: M (M a) -> M a
```

(As will be discussed in the section on the functor class, a functor `M` can be thought of as container, so that `M a` " contains" values of type `a`, with a corresponding mapping function, i.e. `fmap`, that allows functions to be applied to values inside it.) Under this interpretation, the functions behave as follows:

- `fmap` applies a given function to every element in a container

- return packages an element into a container
- join takes a container of containers and flattens it into a single container

With these functions, the bind combinator can be defined as follows:

```
m >>= g = join (fmap g m)
```

Likewise, we could give a definition of fmap and join in terms of (>>=) and return:

```
fmap f x = x >>= (return . f)
join x   = x >>= id
```

At this point we might, with good reason, conclude that all monads are by definition functors as well. That is indeed the case, both according to category theory and when programming in Haskell. A final observation is that Control.Monad defines liftM, a function with a strangely familiar type signature...

```
liftM :: (Monad m) => (a1 -> r) -> m a1 -> m r
```

As you might suspect, liftM is merely fmap implemented with (>>=) and return, just as we have done above. For a properly implemented monad with a matching Functor (that is, any *sensible* monad) liftM and fmap are interchangeable.

3.5.4 Note: avoiding the prerequisites

While following the next few chapters, you will likely want to write instances of Monad and try them out, be it to run the examples in this text or to do other experiments you might think of. However, Applicative being a superclass of Monad means that implementing Monad requires providing Functor and Applicative instances as well. At this point of the report, that would be somewhat of an annoyance, especially given that we have not discussed Applicative yet! As a workaround, once you have written the Monad instance you can use the functions in Control.Monad to fill in the Functor and Applicative implementations, as follows:

```
instance Functor Foo where
fmap = liftM

instance Applicative Foo where
pure = return
(<*>) = ap
```

We will find out what pure, (<*>) and ap are in due course.

3.6 Prerequisites: Functor and Applicative typeclasses

implementing Monad requires providing Functor and Applicative instances as well.

Functor class

Functor is a Prelude class for types which can be mapped over. It has a single method, called fmap. The class is defined as follows:

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

Some examples:

The Maybe functor

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

The List functor

```
instance Functor [] where
         fmap = map
```

The Tree functor

```
instance Functor Tree where
         fmap f (Leaf x) = Leaf (f x)
         fmap f (Branch left right) = Branch (fmap f left) (fmap f right)
```

The functor laws When providing a new instance of Functor, you should ensure it satisfies the two functor laws. There is nothing mysterious about these laws; their role is to guarantee fmap behaves sanely and actually performs a mapping operation (as opposed to some other nonsense).[3] The laws are:

```
fmap id   =   id
fmap (g . f)   =   fmap g . fmap f
```

[3]Some examples of nonsense that the laws rule out: removing or adding elements from a list, reversing a list, changing a Just-value into a Nothing

Applicative functors

Like monads, applicative functors are functors with extra laws and operations; in fact, Applicative is an intermediate class between Functor and Monad. It enables the *applicative style*, a convenient way of structuring functorial computations, and also provides means to express a number of important patterns.

Note: For extra convenience, fmap has an infix synonym, (<$>). It often helps readability, and also suggests how fmap can be seen as a different kind of function application.

```
Prelude> negate <$> Just 2
Just (-2)
```

As useful as it is, fmap isn't much help if we want to apply a function of two arguments to functorial values. For instance, how could we sum Just 2 and Just 3? The brute force approach would be extracting the values from the Maybe wrapper. That, however, would mean having to do tedious checks for Nothing. Even worse: in a different Functor extracting the value might not even be an option (just think about IO).

We could use fmap to partially apply (+) to the first argument:

```
Prelude> :t (+) <$> Just 2
(+) <$> Just 2 :: Num a => Maybe (a -> a)
```

But now we are stuck: we have a function and a value both wrapped in Maybe, and no way of applying one to the other. What we would like to have is an operator with a type akin to

```
f (a -> b) -> f a -> f b
```

to apply functions in the context of a functor. That operator is called (<*>), check this:

```
Prelude> (+) <$> Just 2 <*> Just 3
Just 5

Prelude> :t (<*>)
(<*>) :: Applicative f => f (a -> b) -> f a -> f b
```

(<*>) is one of the methods of Applicative the type class of *applicative functors* - functors that support function application within their contexts. Expressions such as

```
(+) <$> Just 2 <*> Just 3
```

are said to be written in *applicative style*, which is as close as we can get to regular function application while working with a functor. If you pretend for a moment the (<$>), (<*>) and Just aren't there, our example looks just like (+) 2 3.

3.6.1 Applicative functor laws

The definition of `Applicative` is:

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

Beyond `(<*>)`, the class has a second method, `pure`, which brings arbitrary values into the functor. As an example, let's have a look at the `Maybe` instance:

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

It doesn't do anything surprising: `pure` wraps the value with `Just`; `(<*>)` applies the function to the value if both exists, and results in `Nothing` otherwise.

Note For the lack of a better shorthand, in what follows we will use the word *morphism* to refer to the values to the left of `(<*>)`, which fit the type `Applicative f => f (a -> b)`; that is, the function-like things inserted into an applicative functor.

Just like `Functor`, `Applicative` has a set of laws which reasonable instances should follow. They are:

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

Those laws are a bit of a mouthful. They become easier to understand if you think of `pure` as a way to inject values into the functor in a default, featureless way, so that the result is as close as possible to the plain value. Thus:

- The identity law says that applying the `pure id` morphism does nothing, exactly like with the plain `id` function.
- The homomorphism law says that applying a "pure" function to a "pure" value is the same than applying the function to the value in the normal way and then using `pure` on the result. In a sense, that means `pure` preserves function application.
- The interchange law says that applying a morphism to a "pure" value `pure y` is the same as applying pure `($ y)` to the morphism. No surprises there - `($ y)` is the function that supplies `y` as argument to another function.
- The composition law says that if `(<*>)` is used to compose morphisms the composition is associative, like plain function composition.[4]

There is also a bonus law about the relation between `fmap` and `(<*>)`:

[4] With plain functions, we have `h . g . f = (h . g) . f = h . (g . f)` That is why we never bother to use parentheses in the middle of `(.)` chains.

```
fmap f x = pure f <*> x                    -- fmap
```

Applying a "pure" function with $(<*>)$ is equivalent to using fmap. **This law is a consequence of the other ones, so you need not bother with proving it when writing instances of Applicative**.

3.7 *do* notation

Using do blocks as an alternative monad syntax was introduced with an IO example. Since the following examples all involve IO, we will refer to the computations/monadic values as *actions* (as we did in the earlier parts of the report). Of course, do works with any monad; there is nothing specific about IO in how it works.

3.7.1 Translating the *then* operator

The (>>) *(then)* operator works almost identically in do notation and in unsugared code. For example, suppose we have a chain of actions like the following one:

```
putStr "Hello" >>
putStr " " >>
putStr "world!" >>
putStr "\n"
```

We can rewrite that in do notation as follows:

```
do putStr "Hello"
   putStr " "
   putStr "world!"
   putStr "\n"
```

This sequence of instructions nearly matches that in any imperative language. In Haskell, we can chain any actions as long as all of them are in the same monad. In the context of the IO monad, the actions include writing to a file, opening a network connection, or asking the user for input.

Here's the step-by-step translation of do notation to unsugared Haskell code:

```
do action1
   action2
   action3
```

becomes

```
action1 >>
do action2
   action3
```

and so on, until the do block is empty.

3.7.2 Translating the *bind* operator

The (>>=) is a bit more difficult to translate from and to do notation. (>>=) passes a value, namely the result of an action or function, downstream in the binding sequence. do notation assigns a variable name to the passed value using the `<-`.

```
do x1 <- action1
   x2 <- action2
   action3 x1 x2
```

x1 and x2 are the results of action1 and action2. If, for instance, action1 is an IO Integer then x1 will be bound to an Integer. The stored values are passed as arguments to action3, which returns a third action. The do block is broadly equivalent to the following vanilla Haskell snippet:

```
action1 >>= \ x1 -> action2 >>= \ x2 -> action3 x1 x2
```

The second argument of `(>>=)` is a function specifying what to do with the result of the action passed as first argument. Thus, chains of lambdas pass the results downstream. Remember that, without extra parentheses, a lambda extends all the way to the end of the expression. `x1` is still in scope at the point we call `action3`. We can rewrite the chain of lambdas more legibly by using separate lines and indentation:

```
action1
  >>=
    \ x1 -> action2
      >>=
        \ x2 -> action3 x1 x2
```

That shows the scope of each lambda function clearly. To group things more like the `do` notation, we could show it like this:

```
action1 >>= \ x1 ->
  action2 >>= \ x2 ->
    action3 x1 x2
```

These presentation differences are only a matter of assisting readability. Actually, the indentation isn't needed in this case. This is equally valid:

```
action1 >>= \ x1 ->
action2 >>= \ x2 ->
action3 x1 x2
```

3.7.3 The *fail* method

Above, we said the snippet with lambdas was "broadly equivalent" to the `do` block. The translation is not exact because the `do` notation adds special handling of pattern match failures. When placed at the left of either `<-` or `->`, `x1` and `x2` are patterns being matched. Therefore, if `action1` returned a `Maybe Integer` we could write a `do` block like this...

```
do Just x1 <- action1
   x2      <- action2
   action3 x1 x2
```

...and `x1` be an `Integer`. In such a case, what happens if `action1` returns `Nothing`? Ordinarily, the program would crash with a non-exhaustive patterns error, just like the one we get when calling `head` on an empty list. With `do` notation, however, failures are handled with the `fail` method for the relevant monad. The `do` block above translates to:

```
action1 >>= f
where f (Just x1) = do x2 <- action2
                       action3 x1 x2
      f _         = fail "..." -- A compiler-generated message.
```

What `fail` actually does depends on the monad instance. Though it will often rethrow the pattern matching error, monads that incorporate some sort of error handling may deal with the failure in their own specific ways. For instance, `Maybe` has `fail _ = Nothing`; analogously, for the list monad `fail _ = []`.[5]

[5]This explains why pattern matching failures in list comprehensions are silently ignored.

The fail method is an artifact of do notation. Rather than calling fail directly, you should rely on automatic handling of pattern match failures whenever you are sure that fail will do something sensible for the monad you are using.

3.7.4 Example: user-interactive program

Note for non-ghci users We are going to interact with the user, so we will use putStr and getLine alternately. To avoid unexpected results in the output, we must disable output buffering when importing System.IO.

To do this, put hSetBuffering stdout NoBuffering at the top of your code. To handle this otherwise, you would explicitly flush the output buffer before each interaction with the user (namely a getLine) using hFlush stdout. If you are testing this code with ghci, you don't have such problems.

Consider this simple program that asks the user for their first and last names:

```
nameDo :: IO ()
nameDo = do putStr "What is your first name? "
            first <- getLine
            putStr "And your last name? "
            last <- getLine
            let full = first ++ " " ++ last
            putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

A possible translation into vanilla monadic code:

```
nameLambda :: IO ()
nameLambda = putStr "What is your first name? " >>
             getLine >>= \ first ->
             putStr "And your last name? ">>
             getLine >>= \ last ->
             let full = first ++ " " ++ last
             in putStrLn ("Pleased to meet you, " ++ full ++ "!")
```

In cases like this, where we just want to chain several actions, the imperative style of do notation feels natural and convenient. In comparison, monadic code with explicit binds and lambdas is something of an acquired taste.

Notice that the first example above includes a let statement in the do block. The de-sugared version is simply a regular let expression where the in part is whatever follows from the do syntax.

3.7.5 Returning values

The last statement in do notation is the overall result of the do block. In the previous example, the result was of the type IO (), i.e. an empty value in the IO monad.

Suppose that we want to rewrite the example but return an IO String with the acquired name. All we need to do is add a return:

```
nameReturn :: IO String
nameReturn = do putStr "What is your first name? "
                first <- getLine
                putStr "And your last name? "
```

```
                last <- getLine
                let full = first ++ " " ++ last
                putStrLn ("Pleased to meet you, " ++ full ++ "!")
                return full
```

This example will "return" the full name as a string inside the IO monad, which can then be utilized downstream elsewhere:

```
greetAndSeeYou :: IO ()
greetAndSeeYou = do name <- nameReturn
                    putStrLn ("See you, " ++ name ++ "!")
```

Here, nameReturn will be run and the returned result (called "full" in the nameReturn function) will be assigned to the variable "name" in our new function. The greeting part of nameReturn will be printed to the screen because that is part of the calculation process. Then, the additional "see you" message will print as well, and the final returned value is back to being IO ().

If you know imperative languages like C, you might think return in Haskell matches return elsewhere. A small variation on the example will dispel that impression:

```
nameReturnAndCarryOn = do
  putStr "What is your first name? "
  first <- getLine
  putStr "And your last name? "
  last <- getLine
  let full = first++" "++last
  putStrLn ("Pleased to meet you, "++full++"!")
  return full
  putStrLn "I am not finished yet!"
```

The string in the extra line *will* be printed out because return is not a final statement interrupting the flow (as it would be in C and other languages). Indeed, the type of nameReturnAndCarryOn is IO (), - the type of the final putStrLn action. After the function is called, the IO String created by the return full will disappear without a trace.

3.7.6 Just sugar

As a syntactical convenience, do notation does not add anything essential, but it is often preferable for clarity and style. However, do is never used for a single action. The Haskell "Hello world" is simply:

```
main = putStrLn "Hello world!"
```

Snippets like this one are totally redundant:

```
fooRedundant = do x <- bar
                  return x
```

Thanks to the monad laws, we can and should write simply:

```
foo = bar
```

A subtle but crucial point relates to function composition: As we already know, the greetAndSeeYou action in the section just above could be rewritten as:

```
greetAndSeeYou :: IO ()
greetAndSeeYou =
  nameReturn >>= \ name -> putStrLn ("See you, " ++ name ++ "!")
```

While you might find the lambda a little unsightly, suppose we had a printSeeYou function defined elsewhere:

```
printSeeYou :: String -> IO ()
printSeeYou name = putStrLn ("See you, " ++ name ++ "!")
```

Now, we can have a clean function definition with neither lambdas or do:

```
greetAndSeeYou :: IO ()
greetAndSeeYou = nameReturn >>= printSeeYou
```

Or, if we have a *non-monadic* seeYou function:

```
seeYou :: String -> String
seeYou name = "See you, " ++ name ++ "!"
```

Then we can write:

```
-- Reminder: fmap f m == m >>= return . f == liftM f m
greetAndSeeYou :: IO ()
greetAndSeeYou = fmap seeYou nameReturn >>= putStrLn
```

Keep this last example with fmap in mind; we will soon return to using non-monadic functions in monadic code, and fmap will be useful there.

3.8 Additive monads (MonadPlus)

In our studies so far, we saw that the Maybe and list monads both represent the number of results a computation can have. That is, you use Maybe when you want to indicate that a computation can fail somehow (i.e. it can have 0 results or 1 result), and you use the list monad when you want to indicate a computation could have many valid answers ranging from 0 results to many results.

Given two computations in one of these monads, it might be interesting to amalgamate *all* valid solutions into a single result. For example, within the list monad, we can concatenate two lists of valid solutions.

3.8.1 *MonadPlus* definition

MonadPlus defines two methods. mzero is the monadic value standing for zero results; while mplus is a binary function which combines two computations.

```
class Monad m => MonadPlus m where
  mzero :: m a
  mplus :: m a -> m a -> m a
```

Here are the two instance declarations for Maybe and the list monad:

```
instance MonadPlus [] where
  mzero = []
  mplus = (++)


instance MonadPlus Maybe where
  mzero                   = Nothing
  Nothing `mplus` Nothing = Nothing -- 0 solutions + 0 solutions = 0 solutions
  Just x  `mplus` Nothing = Just x  -- 1 solution  + 0 solutions = 1 solution
  Nothing `mplus` Just x  = Just x  -- 0 solutions + 1 solution  = 1 solution
  Just x  `mplus` Just y  = Just x  -- 1 solution  + 1 solution  = 2 solutions,
                                    -- but Maybe can only have up to one solution,
                                    -- so we disregard the second one.
```

Also, if you import Control.Monad.Error, then (Either e) becomes an instance:

```
instance (Error e) => MonadPlus (Either e) where
  mzero             = Left noMsg
  Left _  `mplus` n = n
  Right x `mplus` _ = Right x
```

Like Maybe, (Either e) represents computations that can fail. Unlike Maybe, (Either e) allows the failing computations to include an error "message" (which is usually a String). Typically, Left s means a failed computation carrying an error message s, and Right x means a successful computation with result x.

3.8.2 Example: parallel parsing

Traditional input parsing involves functions which consume an input one character at a time. That is, a parsing function takes an input string and chops off (i.e. 'consumes') characters from the front if they satisfy certain criteria. For example, you could write a function which consumes

one uppercase character. If the characters on the front of the string don't satisfy the given criteria, the parser has *failed*; so such functions are candidates for Maybe.

Let's use mplus to run two parsers *in parallel*. That is, we use the result of the first one if it succeeds, and otherwise, we use the result of the second. If both fail, then our whole parser returns Nothing.

In the example below, we consume a digit in the input and return the digit that was parsed.

```
digit :: Int -> String -> Maybe Int
digit i s | i > 9 || i < 0 = Nothing
          | otherwise      = do
  let (c:_) = s
  if [c] == show i then Just i else Nothing
```

Our guards assure that the Int we are checking for is a single digit. Otherwise, we are just checking that the first character of our String matches the digit we are checking for. If it passes, we return the digit wrapped in a Just. The do-block assures that any failed pattern match will result in returning Nothing.

We can use our digit function with mplus to parse Strings of binary digits:

```
binChar :: String -> Maybe Int
binChar s = digit 0 s `mplus` digit 1 s
```

Parser libraries often make use of MonadPlus in this way. If you are curious, check the (+++) operator in Text.ParserCombinators.ReadP, or (<|>) in Text.ParserCombinators.Parsec.Prim.

3.8.3 The MonadPlus laws

Instances of MonadPlus are required to fulfill several rules, just as instances of Monad are required to fulfill the three monad laws. Unfortunately, the MonadPlus laws aren't fully agreed on. The most common approach says that mzero and mplus form a *monoid*. By that, we mean:

```
-- mzero is a neutral element
mzero `mplus` m  =  m
m `mplus` mzero  =  m
-- mplus is associative
-- (but not all instances obey this law because it makes some infinite structures impossible)
m `mplus` (n `mplus` o)  =  (m `mplus` n) `mplus` o
```

The Haddock documentation for Control.Monad quotes additional laws:

```
mzero >>= f  =  mzero
m >> mzero   =  mzero
```

And the HaskellWiki page cites another (with controversy):

```
(m `mplus` n) >>= k   =   (m >>= k) `mplus` (n >>= k)
```

There are even more sets of laws available. Sometimes monads like IO are used as a MonadPlus. Consult All About Monads and the Haskell Wiki page on MonadPlus for more information about such issues.

3.8.4 Useful functions

Beyond the basic mplus and mzero, there are two other general-purpose functions involving MonadPlus:

msum

A common task when working with MonadPlus: take a list of monadic values, e.g. [Maybe a] or [[a]], and fold it down with mplus. The function msum fulfills this role:

```
msum :: MonadPlus m => [m a] -> m a
msum = foldr mplus mzero
```

In a sense, msum generalizes the list-specific concat operation. Indeed, the two are equivalent when working on lists. For Maybe, msum finds the first Just x in the list and returns Nothing if there aren't any.

guard

When discussing the list monad we note how similar it is to list comprehensions, but we didn't discuss how to mirror list comprehension filtering. The guard function allows us to do exactly that.

Consider the following comprehension which retrieves all pythagorean triples (i.e. trios of integer numbers which work as the lengths of the sides for a right triangle). First we'll examine the brute-force approach. We'll use a boolean condition for filtering; namely, Pythagoras' theorem:

```
pythags = [ (x, y, z) | z <- [1..], x <- [1..z], y <- [x..z], x^2 + y^2 == z^2 ]
```

The translation of the comprehension above to the list monad is:

```
pythags = do
  z <- [1..]
  x <- [1..z]
  y <- [x..z]
  guard (x^2 + y^2 == z^2)
  return (x, y, z)
```

The guard function works like this:

```
guard :: MonadPlus m => Bool -> m ()
guard True  = return ()
guard _ = mzero
```

Concretely, guard will reduce a do-block to mzero if its predicate is False. Given the first law stated in the 'MonadPlus laws' section above, an mzero on the left-hand side of a (>>=) operation will produce mzero again. As do-blocks are decomposed to lots of expressions joined up by (>>=), an mzero at any point will cause the entire do-block to become mzero.

To further illustrate, we will examine guard in the special case of the list monad, extending on the pythags function above. First, here is guard defined for the list monad:

```
guard :: Bool -> [()]
guard True  = [()]
guard _ = []
```

Basically, guard *blocks off* a route. In pythags, we want to block off all the routes (or combinations of x, y and z) where x^2 + y^2 == z^2 is False. Let's look at the expansion of the above do-block to see how it works:

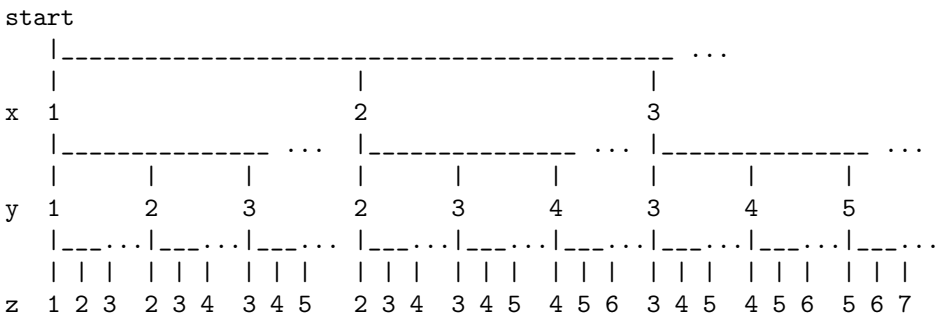
```
pythags =
  [1..] >>= \z ->
  [1..z] >>= \x ->
  [x..z] >>= \y ->
  guard (x^2 + y^2 == z^2) >>= \_ ->
  return (x, y, z)
```

Replacing (>>=) and return with their definitions for the list monad (and using some let-bindings to keep it readable), we obtain:

```
pythags =
 let ret x y z = [(x, y, z)]
     gd  z x y = concatMap (\_ -> ret x y z) (guard $ x^2 + y^2 == z^2)
     doY z x   = concatMap (gd  z x) [x..z]
     doX z     = concatMap (doY z  ) [1..z]
     doZ       = concatMap (doX    ) [1..]
 in doZ
```

Remember that guard returns the empty list in the case of its argument being False. Mapping across the empty list produces the empty list, no matter what function you pass in. So the empty list produced by the call to guard in the binding of gd will cause gd to be the empty list, and therefore ret to be the empty list.

To understand why this matters, think about list-computations as a tree. With our Pythagorean triple algorithm, we need a branch starting from the top for every choice of z, then a branch from each of these branches for every value of x, then from each of these, a branch for every value of y. So the tree looks like this:



Each combination of x, y and z represents a route through the tree. Once all the functions have been applied, each branch is concatenated together, starting from the bottom. Any route where our predicate doesn't hold evaluates to an empty list, and so has no impact on this concat operation.

3.8.5 Relationship with monoids

When discussing the MonadPlus laws, we alluded to the mathematical concept of monoids. It turns out that there is a Monoid class in Haskell, defined in Data.Monoid. A fuller presentation of is given in an appendix. For now, a minimal definition of Monoid implements two methods; namely, a neutral element (or 'zero') and an associative binary operation (or 'plus').

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

For example, lists form a simple monoid:

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

Sounds familiar, doesn't it? In spite of the uncanny resemblance to MonadPlus, there is a subtle yet key difference. Note the usage of [a] instead of [ ] in the instance declaration. Monoids are not necessarily "containers" of anything or parametrically polymorphic. For instance, the integer numbers on form a monoid under addition with 0 as neutral element.

In any case, MonadPlus instances look very similar to monoids, as both feature concepts of zero and plus. Indeed, we could even make MonadPlus a subclass of Monoid if it were worth the trouble:

```
instance MonadPlus m => Monoid (m a) where
  mempty  = mzero
  mappend = mplus
```

Note Due to the "free" type variable a in the instance definition, the snippet above is not valid Haskell 98. If you want to test it, you will have to enable the GHC *language extension* FlexibleInstances:

- If you are testing with GHCi, start it with the command line option `-XFlexibleInstances` or interactively type :set -XFlexibleInstances.
- Alternatively, if you are running a compiled program, add `{-# LANGUAGE FlexibleInstances #-}` to the top of your source file.

Again, Monoids and MonadPlus work at different levels. As noted before, there is no requirement for monoids to be parameterized in relation to "contained" or related type. More formally, monoids have kind $*$, but instances of MonadPlus (which are monads) have kind $* -> *$.

3.9 Monad transformers

We have seen how monads can help handling IO actions, Maybe, lists, and state. With monads providing a common way to use such useful general-purpose tools, a natural thing we might want to do is using the capabilities of *several* monads at once. For instance, a function could use both I/O and Maybe exception handling. While a type like IO (Maybe a) would work just fine, it would force us to do pattern matching within IO do-blocks to extract values, something that the Maybe monad was meant to spare us from.

Enter **monad transformers**: special types that allow us to roll two monads into a single one that shares the behavior of both.

3.9.1 Passphrase validation

Consider a real-life problem for IT staff worldwide: getting users to create strong passphrases. One approach: force the user to enter a minimum length with various irritating requirements (such as at least one capital letter, one number, one non-alphanumeric character, etc.)

Here's a Haskell function to acquire a passphrase from a user:

```
getPassphrase :: IO (Maybe String)
getPassphrase = do s <- getLine
                   if isValid s then return $ Just s
                                else return Nothing

-- The validation test could be anything we want it to be.
isValid :: String -> Bool
isValid s = length s >= 8
            && any isAlpha s
            && any isNumber s
            && any isPunctuation s
```

First and foremost, getPassphrase is an IO action, as it needs to get input from the user. We also use Maybe, as we intend to return Nothing in case the password does not pass the isValid. Note, however, that we aren't actually using Maybe as a monad here: the do block is in the IO monad, and we just happen to return a Maybe value into it.

Monad transformers not only make it easier to write getPassphrase but also simplify all the code instances. Our passphrase acquisition program could continue like this:

```
askPassphrase :: IO ()
askPassphrase = do putStrLn "Insert your new passphrase:"
                   maybe_value <- getPassphrase
                   if isJust maybe_value
                     then do putStrLn "Storing in database..." -- do stuff
                     else putStrLn "Passphrase invalid."
```

The code uses one line to generate the maybe_value variable followed by further validation of the passphrase.With monad transformers, we will be able to extract the passphrase in one go - without any pattern matching or equivalent bureaucracy like isJust. The gains for our simple example might seem small but will scale up for more complex situations.

3.9.2 A simple monad transformer: MaybeT

To simplify getPassphrase and all the code that uses it, we will define a *monad transformer* that gives the IO monad some characteristics of the Maybe monad; we will call it MaybeT. That follows a convention where monad transformers have a "T" appended to the name of the monad whose characteristics they provide.

MaybeT is a wrapper around m (Maybe a), where m can be any monad (IO in our example):

```
newtype MaybeT m a = MaybeT { runMaybeT :: m (Maybe a) }
```

This data type definition specifies a MaybeT type constructor, parameterized over m, with a term constructor, also called MaybeT, and a convenient accessor function runMaybeT, with which we can access the underlying representation.

The whole point of monad transformers is that *they are monads themselves*; and so we need to make MaybeT m an instance of the Monad class:

```
instance Monad m => Monad (MaybeT m) where
    return  = MaybeT . return . Just
```

It would also have been possible (though arguably less readable) to write return = MaybeT . return . return.

As in all monads, the bind operator is the heart of the transformer.

```
-- The signature of (>>=), specialized to MaybeT m
(>>=) :: MaybeT m a -> (a -> MaybeT m b) -> MaybeT m b

x >>= f = MaybeT $ do maybe_value <- runMaybeT x
                      case maybe_value of
                           Nothing    -> return Nothing
                           Just value -> runMaybeT $ f value
```

Starting from the first line of the do block:

1. First, the runMaybeT accessor unwraps x into an m (Maybe a) computation. That shows us that the whole do block is in m.
2. Still in the first line, $<-$ extracts a Maybe a value from the unwrapped computation.
3. The case statement tests maybe_value:
   - With Nothing, we return Nothing into m;
   - With Just, we apply f to the value from the Just. Since f has MaybeT m b as result type, we need an extra runMaybeT to put the result back into the m monad.
4. Finally, the do block as a whole has m (Maybe b) type; so it is wrapped with the MaybeT constructor.

It may look a bit complicated; but aside from the copious amounts of wrapping and unwrapping, the implementation does the same as the familiar bind operator of Maybe:

```
-- (>>=) for the Maybe monad
maybe_value >>= f = case maybe_value of
                        Nothing -> Nothing
                        Just value -> f value
```

Why use the MaybeT constructor before the do block while we have the accessor runMaybeT within do? Well, the do block must be in the m monad, not in MaybeT m (which lacks a defined bind operator at this point).

Note The chained functions in the definition of return suggest a metaphor, which you may find either useful or confusing. Consider the combined monad as a *sandwich*. This metaphor might suggest three layers of monads in action, but there are only two really: the inner monad and the combined monad (there are no binds or returns done in the base monad; it only appears as part of the implementation of the transformer). If you like this metaphor at all, think of the transformer and the base monad as two parts of the same thing - the *bread* - which wraps the inner monad.

Technically, this is all we need; however, it is convenient to make MaybeT an instance of a few other classes:

```
instance Monad m => MonadPlus (MaybeT m) where
    mzero     = MaybeT $ return Nothing
    mplus x y = MaybeT $ do maybe_value <- runMaybeT x
                            case maybe_value of
                                 Nothing    -> runMaybeT y
                                 Just _     -> return maybe_value

instance MonadTrans MaybeT where
    lift = MaybeT . (liftM Just)
```

MonadTrans implements the lift function, so we can take functions from the m monad and bring them into the MaybeT m monad in order to use them in do blocks. As for MonadPlus, since Maybe is an instance of that class it makes sense to make the MaybeT an instance too.

Application to the passphrase example

With all this done, here is what the previous example of passphrase management looks like:

```
askPassword :: MaybeT IO ()
askPassword = do lift $ putStrLn "Insert your new password:"
                 value <- msum $ repeat getValidPassphrase
                 lift $ putStrLn "Storing in database..."
```

The code is now simpler, especially in the user function askPassphrase. Most importantly, we do not have to manually check whether the result is Nothing or Just: the bind operator takes care of that for us.

Note how we use lift to bring the functions getLine and putStrLn into the MaybeT IO monad. Also, since MaybeT IO is an instance of MonadPlus, checking for passphrase validity can be taken care of by a guard statement, which will return mzero (i.e. IO Nothing) in case of a bad passphrase.Incidentally, with the help of MonadPlus it also becomes very easy to ask the user *ad infinitum* for a valid passphrase:

```
askPassword :: MaybeT IO ()
askPassword = do lift $ putStrLn "Insert your new password:"
                 value <- msum $ repeat getValidPassphrase
                 lift $ putStrLn "Storing in database..."
```

3.9.3 A plethora of transformers

The transformers package provides modules with transformers for many common monads (MaybeT, for instance, can be found in Control.Monad.Trans.Maybe). These are defined consistently with their non-transformer versions; that is, the implementation is basically the same except with the extra wrapping and unwrapping needed to thread the other monad. From this point on, we will use **base monad** to refer to the non-transformer monad (e.g. Maybe in MaybeT) on which a transformer is based and **inner monad** to refer to the other monad (e.g. IO in MaybeT IO) on which the transformer is applied.

To pick an arbitrary example, ReaderT Env IO String is a computation which involves reading values from some environment of type Env (the semantics of Reader, the base monad) and performing some IO in order to give a value of type String. Since the (>>=) operator and return for the transformer mirror the semantics of the base monad, a do block of type ReaderT Env IO String will, from the outside, look a lot like a do block of the Reader monad, except that IO actions become trivial to embed by using lift.

Type juggling

We have seen that the type constructor for MaybeT is a wrapper for a Maybe value in the inner monad. So, the corresponding accessor runMaybeT gives us a value of type m (Maybe a) - i.e. a value of the base monad returned in the inner monad. Similarly, for the ListT and ExceptT transformers, which are built around lists and Either respectively:

```
runListT :: ListT m a -> m [a]
```

and

```
runExceptT :: ExceptT e m a -> m (Either e a)
```

Not all transformers are related to their base monads in this way, however. Unlike the base monads in the two examples above, the Writer, Reader, State, and Cont monads have neither multiple constructors nor constructors with multiple arguments. For that reason, they have run... functions which act as simple unwrappers, analogous to the run...T of the transformer versions. The table below shows the result types of the run... and run...T functions in each case, which may be thought of as the types wrapped by the base and transformed monads respectively.[6]

Base Monad	**Transformer**	Original Type "wrapped" by base	Combined Type "wrapped" by transformer
Writer	WriterT	`(a, w)`	`m (a, w)`
Reader	ReaderT	`r -> a`	`r -> m a`
State	StateT	`s -> (a, s)`	`s -> m (a, s)`
Cont	ContT	`(a -> r) -> r`	`(a -> m r) -> m r`

[6]The wrapping interpretation is only literally true for versions of the mtl package older than 2.0.0.0 .

Notice that the base monad is absent in the combined types. Without interesting constructors (of the sort for Maybe or lists), there is no reason to retain the base monad type after unwrapping the transformed monad. It is also worth noting that in the latter three cases we have function types being wrapped. StateT, for instance, turns state-transforming functions of the form $s \to (a, s)$ into state-transforming functions of the form $s \to$ m (a, s); only the result type of the wrapped function goes into the inner monad. ReaderT is analogous. ContT is different because of the semantics of Cont (the *continuation* monad): the result types of both the wrapped function and its function argument must be the same, and so the transformer puts both into the inner monad. In general, there is no magic formula to create a transformer version of a monad; the form of each transformer depends on what makes sense in the context of its non-transformer type.

3.9.4 Lifting

We will now have a more detailed look at the lift function, which is critical in day-to-day use of monad transformers. The first thing to clarify is the name "lift". One function with a similar name that we already know is liftM. As we already know, it is a monad-specific version of fmap:

```
liftM :: Monad m => (a -> b) -> m a -> m b
```

liftM applies a function $(a -> b)$ to a value within a monad m. We can also look at it as a function of just one argument:

```
liftM :: Monad m => (a -> b) -> (m a -> m b)
```

liftM converts a plain function into one that acts within m. By "lifting", we refer to bringing something into something else – in this case, a function into a monad.

liftM allows us to apply a plain function to a monadic value without needing do-blocks or other such tricks:

```
.-------------------------------------------------------------------.
|                                |                                  |
|         do notation            |            liftM                 |
|--------------------------------|----------------------------------|
|     do x <- monadicValue       |     liftM f monadicValue         |
|        return (f x)            |                                  |
|________________________________|__________________________________|
```

The lift function plays an analogous role when working with monad transformers. It brings (or, to use another common word for that, *promotes*) inner monad computations to the combined monad. By doing so, it allows us to easily insert inner monad computations as part of a larger computation in the combined monad.

lift is the single method of the MonadTrans class, found in Control.Monad.Trans.Class. All monad transformers are instances of MonadTrans, and so lift is available for them all.

```
class MonadTrans t where
    lift :: (Monad m) => m a -> t m a
```

There is a variant of lift specific to IO operations, called liftIO, which is the single method of the MonadIO class in Control.Monad.IO.Class.

```
class (Monad m) => MonadIO m where
   liftIO :: IO a -> m a
```

liftIO can be convenient when multiple transformers are stacked into a single combined monad. In such cases, IO is always the innermost monad, and so we typically need more than one lift to bring IO values to the top of the stack. liftIO is defined for the instances in a way that allows us to bring an IO value from any depth while writing the function a single time.

Implementing lift

Implementing lift is usually pretty straightforward. Consider the MaybeT transformer:

```
instance MonadTrans MaybeT where
    lift m = MaybeT (liftM Just m)
```

We begin with a monadic value of the inner monad. With liftM (fmap would have worked just as fine), we slip the base monad (through the Just constructor) underneath, so that we go from m a to m (Maybe a)). Finally, we use the MaybeT constructor to wrap up the monadic sandwich. Note that the liftM here works in the inner monad, just like the do-block wrapped by MaybeT in the implementation of (>>=) we saw early on was in the inner monad.

3.9.5 Implementing transformers

The State transformer

As an additional example, we will now have a detailed look at the implementation of StateT. You might want to review the appendix on the State monad before continuing.

Just as the State monad might have been built upon the definition

```
newtype State s a = State { runState :: (s -> (a,s)) }
```

, the StateT transformer is built upon the definition:

```
newtype StateT s m a = StateT { runStateT :: (s -> m (a,s)) }
```

StateT s m will have the following Monad instance, here shown alongside the one for the base state monad:

```
State                                    | StateT
-----------------------------------------|-------------------------------------------------
                                         |
newtype State s a =                      | newtype StateT s m a =
 State { runState :: (s -> (a,s)) }      |  StateT { runStateT :: (s -> m (a,s)) }
                                         |
instance Monad (State s) where           | instance (Monad m) => Monad (StateT s m) where
 return a        = State $ \s -> (a,s)   |  return a         = StateT $ \s -> return (a,s)
 (State x) >>= f = State $ \s ->         |  (StateT x) >>= f = StateT $ \s -> do
  let (v,s') = x s                       |   (v,s') <- x s          -- get new value and state
  in runState (f v) s'                   |   runStateT (f v) s'     -- pass them to f
                                         |
-----------------------------------------|-------------------------------------------------
```

Our definition of return makes use of the return function of the inner monad. (>>=) uses a do-block to perform a computation in the inner monad.

Note Incidentally, we can now finally explain why, in the appendix about State, there is a state function instead of a State constructor. In the transformers and mtl packages, State s is implemented as a type synonym for StateT s Identity, with Identity being the dummy monad introduced in an exercise of the previous section. The resulting monad is equivalent to the one defined using newtype that we have used up to now.

If the combined monads StateT s m are to be used as state monads, we will certainly want the all-important get and put operations. Here, we will show definitions in the style of the mtl package. In addition to the monad transformers themselves, mtl provides type classes for the essential operations of common monads. For instance, the MonadState class, found in Control.Monad.State, has get and put as methods:

```
instance (Monad m) => MonadState s (StateT s m) where
  get   = StateT $ \s -> return (s,s)
  put s = StateT $ \_ -> return ((),s)
```

Note The first line should be read as: "For any type s and any instance of Monad m; s and StateT s m together form an instance of MonadState". s and m correspond to the state and the inner monad, respectively. s is an independent part of the instance specification so that the methods can refer to it - for instance, the type of put is s → StateT s ().

There are MonadState instances for state monads wrapped by other transformers, such as

```
MonadState s m => MonadState s (MaybeT m)
```

They bring us extra convenience by making it unnecessary to lift uses of get and put explicitly, as the MonadState instance for the combined monads handles the lifting for us.

It can also be useful to lift instances that might be available for the inner monad to the combined monad. For instance, all combined monads in which StateT is used with an instance of MonadPlus can be made instances of MonadPlus:

```
instance (MonadPlus m) => MonadPlus (StateT s m) where
  mzero = StateT $ \_ -> mzero
  (StateT x1) `mplus` (StateT x2) = StateT $ \s -> (x1 s) `mplus` (x2 s)
```

The implementations of mzero and mplus do the obvious thing; that is, delegating the actual work to the instance of the inner monad.

Lest we forget, the monad transformer must have a MonadTrans instance, so that we can use lift:

```
instance MonadTrans (StateT s) where
  lift c = StateT $ \s -> c >>= (\x -> return (x,s))
```

The lift function creates a StateT state transformation function that binds the computation in the inner monad to a function that packages the result with the input state. If, for instance, we apply StateT to the List monad, a function that returns a list (i.e., a computation in the List monad) can be lifted into StateT s [ ] where it becomes a function that returns a StateT s → [(a,s)]). I.e. the lifted computation produces *multiple* (value,state) pairs from its input state. This "forks" the computation in StateT, creating a different branch of the computation for each value in the list returned by the lifted function. Of course, applying StateT to a different monad will produce different semantics for the lift function.

Chapter 4

Last Steps

In this chapter we revisit everything we have seen in previous chapters, linking the Haskell concepts to what we saw on chapter two on category theory.

4.1 Revisiting the *Applicative* class

A more-in-depth look at the Applicative class. The first subsection is just the same text as in chapter 2.

4.1.1 *Applicative* recap

The definition of Applicative is:

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

Beyond $(<*>)$, the class has a second method, pure, which brings arbitrary values into the functor. As an example, let's have a look at the Maybe instance:

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

It doesn't do anything surprising: pure wraps the value with Just; $(<*>)$ applies the function to the value if both exists, and results in Nothing otherwise.

Note For the lack of a better shorthand, in what follows we will use the word *morphism* to refer to the values to the left of $(<*>)$, which fit the type `Applicative f => f (a -> b)`; that is, the function-like things inserted into an applicative functor.

Just like Functor, Applicative has a set of laws which reasonable instances should follow. They are:

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

Those laws are a bit of a mouthful. They become easier to understand if you think of `pure` as a way to inject values into the functor in a default, featureless way, so that the result is as close as possible to the plain value. Thus:

- The identity law says that applying the `pure id` morphism does nothing, exactly like with the plain `id` function.
- The homomorphism law says that applying a "pure" function to a "pure" value is the same than applying the function to the value in the normal way and then using `pure` on the result. In a sense, that means `pure` preserves function application.
- The interchange law says that applying a morphism to a "pure" value `pure y` is the same as applying pure `($ y)` to the morphism. No surprises there - `($ y)` is the function that supplies `y` as argument to another function.
- The composition law says that if `(<*>)` is used to compose morphisms the composition is associative, like plain function composition.[1]

There is also a bonus law about the relation between `fmap` and `(<*>)`:

```
fmap f x = pure f <*> x                    -- fmap
```

Applying a "pure" function with `(<*>)` is equivalent to using `fmap`. **This law is a consequence of the other ones, so you need not bother with proving it when writing instances of `Applicative`.**

4.1.2 Deja vu

Does `pure` remind you of anything?

```
pure :: Applicative f => a -> f a
```

The only difference between that and...

```
return :: Monad m => a -> m a
```

... is the class constraint. `pure` and `return` serve the same purpose; that is, bringing values into functors. The uncanny resemblances do not stop here. In the appendix about `State` we mention a function called `ap`...

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

... which could be used to make functions with many arguments less painful to handle in monadic code:

```
allTypes :: GeneratorState (Int, Float, Char, Integer, Double, Bool, Int)
allTypes = liftM (,,,,,,) getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
```

[1] With plain functions, we have `h . g . f = (h . g) . f = h . (g . f)` That is why we never bother to use parentheses in the middle of `(.)` chains.

`ap` looks a lot like `(<*>)`.

Those, of course, are not coincidences. `Monad` inherits from `Applicative`...

```
Prelude> :info Monad
class Applicative m => Monad (m :: * -> *) where
--etc.
```

... because `return` and `(>>=)` are enough to implement `pure` and `(<*>)`[2]

```
pure = return
(<*>) = ap

ap u v = do
    f <- u
    x <- v
    return (f x)
```

Several other monadic functions have more general applicative versions. Here are a few of them:

Monadic	**Applicative**	Module (where to find the applicative version)
`(>>)`	`(*>)`	Prelude (GHC 7.10+); Control.Applicative
`liftM2`	`liftA2`	Control.Applicative
`mapM`	`traverse`	Prelude (GHC 7.10+); Data.Traversable
`sequence`	`sequenceA`	Data.Traversable
`forM_`	`for_`	Data.Foldable

4.1.3 *ZipList*

Lists are applicative functors as well. Specialised to lists, the type of `(<*>)` becomes...

```
[a -> b] -> [a] -> [b]
```

... and so `(<*>)` applies a list of functions to another list. But exactly how is that done?

The standard instance of `Applicative` for lists, which follows from the `Monad` instance, applies every function to every element, like an explosive version of `map`.

```
Prelude> [(2*),(5*),(9*)] <*> [1,4,7]
[2,8,14,5,20,35,9,36,63]
```

Interestingly, there is another reasonable way of applying a list of functions. Instead of using every combination of functions and values, we can match each function with the value in the corresponding position in the other list. A Prelude function which can be used for that is `zipWith`:

```
Prelude> :t zipWith
zipWith :: (a -> b -> c) -> [a] -> [b] -> [c]
Prelude> zipWith ($) [(2*),(5*),(9*)] [1,4,7]
[2,20,63]
```

[2] And if the `Monad` instance follows the monad laws, the resulting `pure` and `(<*>)` will automatically follow the applicative laws.

When there are two useful possible instances for a single type, the dilemma is averted by creating a newtype which implements one of them. In this case, we have ZipList, which lives in Control.Applicative:

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

We have already seen what (<*>) should be for zip-lists; all that is needed is to add the newtype wrappers:

```
instance Applicative ZipList where
    (ZipList fs) <*> (ZipList xs) = ZipList (zipWith ($) fs xs)
    pure x                        = undefined -- TODO
```

As for pure, it is tempting to use pure x = ZipList [x], following the standard list instance. We can't do that, however, as it violates the applicative laws. According to the identity law:

```
pure id <*> v = v
```

Substituting (<*>) and the suggested pure, we get:

```
ZipList [id] <*> ZipList xs = ZipList xs
ZipList (zipWith ($) [id] xs) = ZipList xs
```

Now, suppose xs is the infinite list [1..]:

```
ZipList (zipWith ($) [id] [1..]) = ZipList [1..]
ZipList [1] = ZipList [1..]
[1] = [1..] -- Obviously false!
```

The problem is that zipWith produces lists whose length is that of the shortest list passed as argument, and so `(ZipList [id] <*>)` will cut off all elements of the other zip-list after the first. The only way to ensure `zipWith ($) fs` never removes elements is making fs infinite. The correct pure follows from that:

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

The ZipList applicative instance offers an alternative to all the zipN and zipWithN functions in Data.List which can be extended to any number of arguments:

```
>>> import Control.Applicative
>>> ZipList [(2*),(5*),(9*)] <*> ZipList [1,4,7]
ZipList {getZipList = [2,20,63]}
>>> (,,) <$> ZipList [1,4,9] <*> ZipList [2,8,1] <*> ZipList [0,0,9]
ZipList {getZipList = [(1,2,0),(4,8,0),(9,1,9)]}
>>> liftA3 (,,) (ZipList [1,4,9]) (ZipList [2,8,1]) (ZipList [0,0,9])
ZipList {getZipList = [(1,2,0),(4,8,0),(9,1,9)]}
```

4.1.4 Sequencing of effects

As we have just seen, the standard Applicative instance for lists applies every function in one list to every element of the other. That, however, does not specify (<*>) unambiguously. To see why, try to guess what is the result of

```
[(2*),(3*)]<*>[4,5]
```

without looking at the example above or the answer just below.

```
Prelude> [(2*),(3*)] <*> [4,5]


--- ...


[8,10,12,15]
```

Unless you were paying very close attention or had already analysed the implementation of $(<*>)$, the odds of getting it right were about even. The other possibility would be [8,12,10,15]. The difference is that for the first (and correct) answer the result is obtained by taking the skeleton of the first list and replacing each element by all possible combinations with elements of the second list, while for the other possibility the starting point is the second list.

In more general terms, the difference between is one of *sequencing of effects*. Here, by effects we mean the functorial context, as opposed to the values within the functor (some examples: the skeleton of a list, actions performed in the real world in IO, the existence of a value in Maybe). The existence of two legal implementations of $(<*>)$ for lists which only differ in the sequencing of events indicates that [ ] is a non-commutative applicative functor. A *commutative* applicative functor, by contrast, leaves no margin for ambiguity in that respect. More formally, a commutative applicative functor is one for which the following holds:

```
liftA2 f u v = liftA2 (flip f) v u -- Commutativity
```

Or, equivalently,

```
f <$> u <*> v = flip f <$> v <*> u
```

By the way, if you hear about *commutative monads* in Haskell, the concept involved is the same, only specialised to Monad.

Commutativity (or the lack thereof) affects other functions which are derived from $(<*>)$ as well. $(*>)$ is a clear example:

```
(*>) :: Applicative f => f a -> f b -> f b
```

$(*>)$ combines effects while preserving only the values of its second argument. For monads, it is equivalent to $(>>)$. Here is a demonstration of it using Maybe, which is commutative:

```
Prelude> Just 2 *> Just 3
Just 3
Prelude> Just 3 *> Just 2
Just 2
Prelude> Just 2 *> Nothing
Nothing
Prelude> Nothing *> Just 2
Nothing
```

Swapping the arguments does not affect the effects (that is, the being and nothingness of wrapped values). For IO, however, swapping the arguments does reorder the effects:

```
Prelude> (print "foo" *> pure 2) *> (print "bar" *> pure 3)
"foo"
"bar"
3
Prelude> (print "bar" *> pure 3) *> (print "foo" *> pure 2)
"bar"
"foo"
2
```

The convention in Haskell is to always implement (<*>) and other applicative operators using left-to-right sequencing. Even though this convention helps reducing confusion, it also means appearances sometimes are misleading. For instance, the (<*) function is *not* `flip (*>)`, as it sequences effects from left to right just like (*>):

```
Prelude> (print "foo" *> pure 2) <* (print "bar" *> pure 3)
"foo"
"bar"
2
```

For the same reason, `(<**>) :: Applicative f => f a -> f (a -> b) -> f b` from Control.Applicative is not `flip (<*>)`. That means it provides a way of inverting the sequencing:

```
>>> [(2*),(3*)] <*> [4,5]
[8,10,12,15]
>>> [4,5] <**> [(2*),(3*)]
[8,12,10,15]
```

An alternative is the Control.Applicative.Backwards module from transformers, which offers a newtype for flipping the order of effects:

```
newtype Backwards f a = Backwards { forwards :: f a }
```

```
>>> Backwards [(2*),(3*)] <*> Backwards [4,5]
Backwards [8,12,10,15]
```

4.1.5 A sliding scale of power

Functor, Applicative, Monad. Three closely related functor type classes; three of the most important classes in Haskell. Though we have seen many examples of Functor and Monad in use, and a few of Applicative, we have not compared them head to head yet. If we ignore pure/return for a moment, the characteristic methods of the three classes are:

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

While those look like disparate types, we can change the picture with a few aesthetic adjustments. Let's replace fmap by its infix synonym, (<$>); (>>=) by its flipped version, (=<<); and tidy up the signatures a bit:

```
(<$>) :: Functor t     =>   (a -> b) -> (t a -> t b)
(<*>) :: Applicative t => t (a -> b) -> (t a -> t b)
(=<<) :: Monad t       => (a -> t b) -> (t a -> t b)
```

Suddenly, the similarities are striking. fmap, (<*>) and (=<<) are all mapping functions over Functors.[3] The differences between them are in what is being mapped over in each case:

- fmap maps arbitrary functions over functors.
- (<*>) maps `t (a -> b)` morphisms over (applicative) functors.
- (=<<) maps `a -> t b` functions over (monadic) functors.

The day-to-day differences in uses of Functor, Applicative and Monad follow from what the types of those three mapping functions allow you to do. As you move from fmap to (<*>) and then to (>>=), you gain in power, versatility and control, at the cost of guarantees about the results. We will now slide along this scale. While doing so, we will use the contrasting terms *values* and *context* to refer to plain values within a functor and to the whatever surrounds them, respectively.

The type of fmap ensures that it is impossible to use it to change the context, no matter which function it is given. In

```
(a -> b) -> t a -> t b
```

, the `(a -> b)` function has nothing to do with the t context of the t a functorial value, and so applying it cannot affect the context. For that reason, if you do fmap f xs on some list xs the number of elements of the list will never change.

```
Prelude> fmap (2*) [2,5,6]
[4,10,12]
```

That can be taken as a safety guarantee or as an unfortunate restriction, depending on what you intend. In any case, (<*>) is clearly able to change the context:

```
Prelude> [(2*),(3*)] <*> [2,5,6]
[4,10,12,6,15,18]
```

The `t (a -> b)` morphism carries a context of its own, which is combined with that of the t a functorial value. (<*>), however, is subject to a more subtle restriction. While `t (a -> b)` morphisms carry context, within them there are plain `(a -> b)` functions, which are still unable to modify the context. That means the changes to the context (<*>) performs are fully determined by the context of its arguments, and the values have no influence over the resulting context.

```
Prelude> (print "foo" *> pure (2*)) <*> (print "bar" *> pure 3)
"foo"
"bar"
6
Prelude> (print "foo" *> pure 2) *> (print "bar" *> pure 3)
"foo"
"bar"
3
Prelude> (print "foo" *> pure undefined) *> (print "bar" *> pure 3)
"foo"
"bar"
3
```

[3] It is not just a question of type signatures resembling each other: the similarity has theoretical ballast. One aspect of the connection is that it is no coincidence that all three type classes have identity and composition laws.

Thus with list (<*>) you know that the length of the resulting list will be the product of the lengths of the original lists, with IO (<*>) you know that all real world effect will happen as long as the evaluation terminates, and so forth.

With Monad, however, we are in a very different game. (>>=) takes an `a -> t b` function, and so it is able to create context from values. That means a lot of flexibility:

```
Prelude> [1,2,5] >>= \x -> replicate x x
[1,2,2,5,5,5,5,5]
Prelude> [0,0,0] >>= \x -> replicate x x
[]
Prelude> return 3 >>= \x -> print $ if x < 10 then 'Too small' else 'OK'
'Too small'
Prelude> return 42 >>= \x -> print $ if x < 10 then 'Too small' else 'OK'
'OK'
```

Taking advantage of the extra flexibility, however, might mean having less guarantees about, for instance, whether your functions are able to unexpectedly erase parts of a data structure for pathological inputs, or whether the control flow in your application remains intelligible. In some situations there might be performance implications as well, as the complex data dependencies monadic code makes possible might prevent useful refactorings and optimisations.

All in all, it is a good idea to only use as much power as needed for the task at hand. If you do need the extra capabilities of Monad, go right ahead; however, it is often worth it to check whether Applicative or Functor are sufficient.

4.1.6 The monoidal presentation

Back in last chapter, we saw how the Monad class can be specified using either (>=>) or join instead of (>>=). In a similar way, Applicative also has an alternative presentation, which might be implemented through the following type class:

```
class Functor f => Monoidal f where
    unit  :: f ()
    (*&*) :: f a -> f b -> f (a,b)
```

There are deep theoretical reasons behind the name "monoidal".[4] In any case, we can informally say that it does look a lot like a monoid: unit provides a default functorial value whose context wraps nothing of interest, and (*&*) combines functorial values by pairing values and combining effects. The Monoidal formulation provides a clearer view of how Applicative manipulates functorial contexts. Naturally, unit and (*&*) can be used to define pure and (<*>), and vice-versa.

The Applicative laws are equivalent to the following set of laws, stated in terms of Monoidal:

```
fmap snd $ unit *&* v = v                    -- Left identity
fmap fst $ u *&* unit = u                    -- Right identity
fmap asl $ u *&* (v *&* w) = (u *&* v) *&* w -- Associativity
-- asl (x, (y, z)) = ((x, y), z)
```

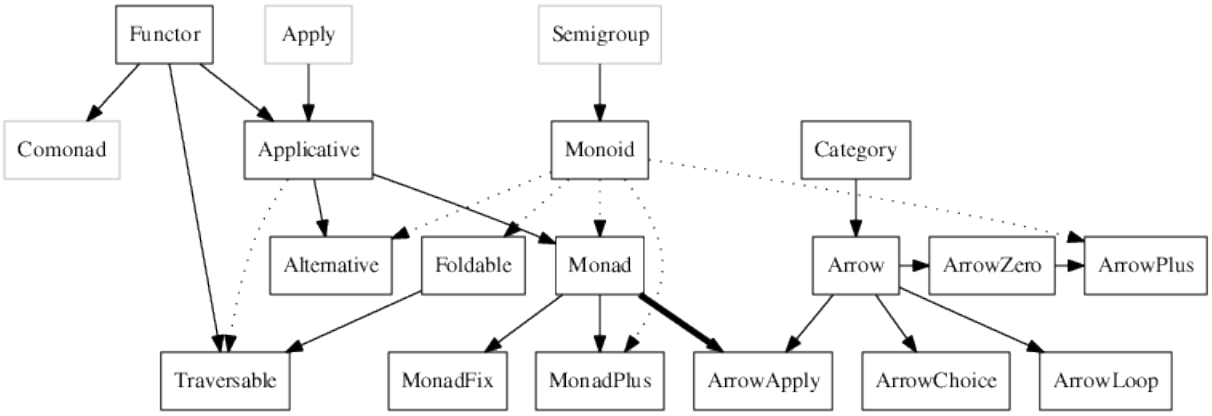
The functions to the left of the ($) are just boilerplate to convert between equivalent types, such as b and ((), b). If you ignore them, the laws are a lot less opaque than in the usual Applicative

[4] For extra details, follow the leads from the corresponding section of the Typeclasseopedia (https://wiki.haskell.org/Typeclassopedia#Alternative_formulation) and the blog post by Edward Z. Yang which inspired it. (http://blog.ezyang.com/2012/08/applicative-functors/)

formulation. By the way, just like for Applicative there is a bonus law, which is guaranteed to hold in Haskell:

```
fmap (g *** h) (u *&* v) = fmap g u *&* fmap h v -- Naturality
-- g *** h = \(x, y) -> (g x, h y)
```

4.1.7 Class heritage



4.2 Still for the curious: The Hask Category

In this section, we will dive in the **Hask** category, identifying some Haskell funcstions with their mathematical equivalent in Category Theory. Recalling from the first chapter:

$\mathfrak{Hask}$: Obj($\mathfrak{Hask}$) – the class of all Haskell types. Hom($\mathfrak{Hask}$) – Haskell functions. The composition law is the (.) operator.

4.2.1 Checking that Hask is a category

We can check the first and second law easily: we know (.) is an associative function, and clearly, for any f and g, f . g is another function.

In Hask, the identity morphism is id, and we have trivially:

```
id . f = f . id = f
```

This isn't an exact translation of the law above, though; we're missing subscripts. The function id in Haskell is *polymorphic*– it can take many different types for its domain and range, or, in category-speak, can have many different source and target objects. But morphisms in category theory are by definition *monomorphic*– each morphism has one specific source object and one specific target object. A polymorphic Haskell function can be made monomorphic by specifying its type (*instantiating* with a monomorphic type), so it would be more precise if we said that the identity morphism from **Hask** on a type A is `(id :: A -> A)`. With this in mind, the above law would be rewritten as:

```
(id :: B -> B) . f = f . (id :: A -> A) = f
```

However, for simplicity, we will ignore this distinction when the meaning is clear.

Actually, there is a subtlety here: because (.) is a lazy function, if f is undefined, we have that `id . f = \_ -> ⊥` . Now, while this may seem equivalent to ⊥ for all intents and purposes, you can actually tell them apart using the strictifying function seq, meaning that the last category law is broken. We can define a new strict composition function,

```
f .! g = ((.) $! f) $! g
```

that makes **Hask** a category. We proceed by using the normal (.), though, and attribute any discrepancies to the fact that seq breaks an awful lot of the nice language properties anyway.

4.2.2 Functors on Hask

The Functor typeclass you have probably seen in Haskell does in fact tie in with the categorical notion of a functor. Remember that a functor has two parts: it maps objects in one category to objects in another and morphisms in the first category to morphisms in the second. Functors in Haskell are from **Hask** to *func*, where *func* is the subcategory of **Hask** defined on just that functor's types. E.g. the list functor goes from **Hask** to **Lst**, where **Lst** is the category containing only *list types*, that is, [T] for any type T. The morphisms in **Lst** are functions defined on list types, that is, functions `[T] -> [U]` for types T, U. How does this tie into the Haskell typeclass Functor? Recall its definition:

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

Let's have a sample instance, too:

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

Here's the key part: the *type constructor* Maybe takes any type T to a new type, Maybe T. Also, fmap restricted to Maybe types takes a function `a -> b` to a function `Maybe a -> Maybe b`. But that's it! We've defined two parts, something that takes objects in **Hask** to objects in another category (that of Maybe types and functions defined on Maybe types), and something that takes morphisms in **Hask** to morphisms in this category. So Maybe is a functor.

A useful intuition regarding Haskell functors is that they represent types that can be mapped over. This could be a list or a Maybe, but also more complicated structures like trees. A function that does some mapping could be written using fmap, then any functor structure could be passed into this function. E.g. you could write a generic function that covers all of Data.List.map, Data.Map.map, Data.Array.IArray.amap, and so on.

What about the functor axioms? The polymorphic function id takes the place of id_A for any A, so the first law states:

```
fmap id = id
```

With our above intuition in mind, this states that mapping over a structure doing nothing to each element is equivalent to doing nothing overall. Secondly, morphism composition is just (.), so

```
fmap (f . g) = fmap f . fmap g
```

This second law is very useful in practice. Picturing the functor as a list or similar container, the right-hand side is a two-pass algorithm: we map over the structure, performing g, then map over it again, performing f. The functor axioms guarantee we can transform this into a single-pass algorithm that performs f . g. This is a process known as *fusion*.

Translating categorical concepts into Haskell

Functors provide a good example of how category theory gets translated into Haskell. The key points to remember are that:

- We work in the category **Hask** and its subcategories.
- Objects are types.
- Morphisms are functions. Things that take a type and return another type are type constructors.
- Things that take a function and return another function are higher-order functions.
- Typeclasses, along with the polymorphism they provide, make a nice way of capturing the fact that in category theory things are often defined over a number of objects at once.

4.2.3 Monads

Monads are obviously an extremely important concept in Haskell, and in fact they originally came from category theory.[5] A monad is a special type of functor, from a category to that same category, that supports some additional structure. So, down to definitions. A monad is a functor $M : C \to C$, along with two morphisms for every object X in C:

[5]Experienced category theorists will notice that we're simplifying things a bit here; instead of presenting *unit* and *join* as natural transformations, we treat them explicitly as morphisms, and require naturality as extra axioms

- ${unit_X}^M : X \to M(X)$
- ${join_X}^M M(M(X)) \to M(X)$

When the monad under discussion is obvious, we'll leave out the ^{M} superscript for these functions and just talk about $unit_X$ and $join_X$ for some X.

Translating

Let's see how this translates to the Haskell typeclass Monad, then.

```
class Functor m => Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

The class constraint of `Functor m` ensures that we already have the functor structure: a mapping of objects and of morphisms. `return` is the (polymorphic) analogue to $unit_X$ for any X. But we have a problem.

Although `return`'s type looks quite similar to that of *unit*; the other function, `(>>=)`, often called *bind*, bears no resemblance to *join*. There is however another monad function,

```
join :: Monad m => m (m a) -> m a
```

that looks quite similar. Indeed, we can recover `join` and `(>>=)` from each other:

```
join :: Monad m => m (m a) -> m a
join x = x >>= id

(>>=) :: Monad m => m a -> (a -> m b) -> m b
x >>= f = join (fmap f x)
```

So specifying a monad's `return`, `fmap`, and `join` is equivalent to specifying its `return` `fmap`, and `(>>=)`. It just turns out that the normal way of defining a monad in category theory is to give `unit` and `join`, whereas Haskell programmers like to give `return` and `bind.`[6] Often, the categorical way makes more sense. Any time you have some kind of structure M and a natural way of taking any object X into $M(X)$, as well as a way of taking $M(M(X))$ into $M(X)$, you probably have a monad. We can see this in the following example section.

Example: the powerset functor is also a monad

The power set functor P : **Set** $\to$ **Set** forms a monad. For any set S you have a $unit_S(x) = \{x\}$, mapping elements to their singleton set. Note that each of these singleton sets are trivially a subset of S, so $unit_S$ returns elements of the powerset of S, as is required. Also, you can define a function $join_S$ as follows: we receive an input $L \in \mathcal{P}(\mathcal{P}(S))$. This is:

- A member of the powerset of the powerset of S.

alongside the standard monad laws (laws 3 and 4). The reasoning is simplicity; we are not trying to teach category theory as a whole, simply give a categorical background to some of the structures in Haskell. You may also notice that we are giving these morphisms names suggestive of their Haskell analogues, because the names η and μ don't provide much intuition.

[6] This is perhaps due to the fact that Haskell programmers like to think of monads as a way of sequencing computations with a common feature, whereas in category theory the container aspect of the various structures is emphasised. `join` pertains naturally to containers (squashing two layers of a container down into one), but `(>>=)` is the natural sequencing operation (do something, feeding its results into something else).

- So a member of the set of all subsets of the set of all subsets of S.
- So a set of subsets of S.

We then return the union of these subsets, giving another subset of S. Symbolically,

$$join_S(L) = \bigcup L$$

Hence $\mathcal{P}$ is a monad.[7]

In fact, $\mathcal{P}$ is almost equivalent to the list monad; with the exception that we're talking lists instead of sets, they're almost the same. Compare:

Power set functor on **Set**, given a set S and a morphism $f : A \to B$

Function type	**Definition**
$P(f) : \mathcal{P}(A) \to \mathcal{P}(B)$	$(P(f))(S) = \{f(a) : a \in S\}$
$unit_S : S \to \mathcal{P}(S)$	$unit_S(x) = \{x\}$
$join_S : \mathcal{P}(\mathcal{P}(S)) \to \mathcal{P}(S)$	$join_S(L) = \bigcup L$

List monad from Haskell, given a type `T` and a function `f :: A -> B`

Function type	**Definition**
`fmap f :: [A] -> [B]`	`fmap f xs = [ f a | a <- xs ]`
`return :: T -> [T]`	`return x = [x]`
`join :: [[T]] -> [T]`	`join xs = concat xs`

4.2.4 The monad laws and their importance

Just as functors had to obey certain axioms in order to be called functors, monads have a few of their own. We'll first list them, then translate to Haskell, then see why they're important.

Given a monad $M : C \to C$ and a morphism $f : A \to B$ for $A, B \in C$,

1. $join \circ M(join) = join \circ join$
2. $join \circ M(unit) = join \circ unit = id$
3. $unit \circ f = M(f) \circ unit$
4. $join \circ M(M(f) = M(f) \circ join$

By now, the Haskell translations should be hopefully self-explanatory:

1. `join . fmap join = join . join`
2. `join . fmap return = join . return = id`

[7] If you can prove that certain laws hold, which we'll explore with lists in the next subsection.

3. `return . f = fmap f . return`

4. `join . fmap (fmap f) = fmap f . join`

(Remember that `fmap` is the part of a functor that acts on morphisms.) These laws seem a bit impenetrable at first, though. What on earth do these laws mean, and why should they be true for monads? Let's explore the laws.

The first law

`join . fmap join = join . join`

In order to understand this law, we'll first use the example of lists. The first law mentions two functions, `join . fmap join` (the left-hand side) and `join . join` (the right-hand side). What will the types of these functions be? Remembering that `join`'s type is `[[a]] -> [a]` (we're talking just about lists for now), the types are both `[[[a]]] -> [a]` (the fact that they're the same is handy; after all, we're trying to show they're completely the same function!). So we have a list of lists of lists. The left-hand side, then, performs `fmap join` on this 3-layered list, then uses `join` on the result. `fmap` is just the familiar `map` for lists, so we first map across each of the list of lists inside the top-level list, concatenating them down into a list each. Now we have a list of lists, which we then run through `join`. In summary, we 'enter' the top level, collapse the second and third levels down, then collapse this new level with the top level.

What about the right-hand side? We first run `join` on our list of list of lists. Although this is three layers, and you normally apply a two-layered list to `join`, this will still work, because a `[[[a]]]` is just `[[b]]`, where `b = [a]`, so in a sense, a three-layered list is just a two layered list, but rather than the last layer being 'flat', it is composed of another list. So if we apply our list of lists (of lists) to `join`, it will flatten those outer two layers into one. As the second layer wasn't flat but instead contained a third layer, we will still end up with a list of lists, which the other join flattens. Summing up, the left-hand side will flatten the inner two layers into a new layer, then flatten this with the outermost layer. The right-hand side will flatten the outer two layers, then flatten this with the innermost layer. These two operations should be equivalent. It's sort of like a law of associativity for `join`.

`Maybe` is also a monad, with

```
return :: a -> Maybe a
return x = Just x

join :: Maybe (Maybe a) -> Maybe a
join Nothing         = Nothing
join (Just Nothing)  = Nothing
join (Just (Just x)) = Just x
```

So if we had a *three*-layered Maybe (i.e., it could be `Nothing`, `Just Nothing`, `Just (Just Nothing)` or `Just (Just (Just x))`), the first law says that collapsing the inner two layers first, then that with the outer layer is exactly the same as collapsing the outer layers first, then that with the innermost layer.

The second law

`join . fmap return = join . return = id`

What about the second law, then? Again, we'll start with the example of lists. Both functions mentioned in the second law are functions `[a] -> [a]`. The left-hand side expresses a function that maps over the list, turning each element `x` into its singleton list `[x]`, so that at the end we're left with a list of singleton lists. This two-layered list is flattened down into a single-layer list again using the `join`. The right hand side, however, takes the entire list `[x, y, z, ...]`, turns it into the singleton list of lists `[[x, y, z, ...]]`, then flattens the two layers down into one again. This law is less obvious to state quickly, but it essentially says that applying `return` to a monadic value, then `join`ing the result should have the same effect whether you perform the `return` from inside the top layer or from outside it.

The third and fourth laws

`return . f = fmap f . return`

`join . fmap (fmap f) = fmap f . join`

The last two laws express more self evident fact about how we expect monads to behave. The easiest way to see how they are true is to expand them to use the expanded form:

```
1.   \x -> return (f x) = \x -> fmap f (return x)

2.   \x -> join (fmap (fmap f) x) = \x -> fmap f (join x)
```

Application to do-blocks

Well, we have intuitive statements about the laws that a monad must support, but why is that important? The answer becomes obvious when we consider do-blocks. Recall that a do-block is just syntactic sugar for a combination of statements involving `(>>=)` as witnessed by the usual translation:

```
do { x }                 -->  x
do { let { y = v }; x }  -->  let y = v in do { x }
do { v <- y; x }         -->  y >>= \v -> do { x }
do { y; x }              -->  y >>= \_ -> do { x }
```

Also notice that we can prove what are normally quoted as the monad laws using `return` and `(>>=)` from our above laws (the proofs are a little heavy in some cases, feel free to skip them if you want to):

```
1.  return x >>= f = f x   -- First Law
```

Proof:

```
  return x >>= f
= join (fmap f (return x)) -- By the definition of (>>=)
= join (return (f x))      -- By law 3
= (join . return) (f x)
= id (f x)                 -- By law 2
= f x
```

```
2.  m >>= return = m   -- Second Law
```

Proof:

```
 m >>= return
= join (fmap return m)    -- By the definition of (>>=)
= (join . fmap return) m
= id m                    -- By law 2
= m
```

3. `(m >>= f) >>= g = m >>= (\x -> f x >>= g)`

Proof (recall that `fmap f . fmap g = fmap (f . g)`):

```
  (m >>= f) >>= g
= (join (fmap f m)) >>= g                          -- By the definition of (>>=)
= join (fmap g (join (fmap f m)))                  -- By the definition of (>>=)
= (join . fmap g) (join (fmap f m))
= (join . fmap g . join) (fmap f m)
= (join . join . fmap (fmap g)) (fmap f m)         -- By law 4
= (join . join . fmap (fmap g) . fmap f) m
= (join . join . fmap (fmap g . f)) m              -- By the distributive law of functors
= (join . join . fmap (\x -> fmap g (f x))) m
= (join . fmap join . fmap (\x -> fmap g (f x))) m -- By law 1
= (join . fmap (join . (\x -> fmap g (f x)))) m    -- By the distributive law of functors
= (join . fmap (\x -> join (fmap g (f x)))) m
= (join . fmap (\x -> f x >>= g)) m                -- By the definition of (>>=)
= join (fmap (\x -> f x >>= g) m)
= m >>= (\x -> f x >>= g)                          -- By the definition of (>>=)
```

These new monad laws, using `return` and `(>>=)`, can be translated into do-block notation.

Points-free style	**Do-block style**
`return x >>= f = f x`	`do { v <- return x; f v } = do { f x }`
`m >>= return = m`	`do { v <- m; return v } = do { m }`

`(m >>= f) >>= g = m >>= (\x -> f x >>= g)`	`do { y <- do { x <- m; f x }; g y}`
	`=`
	`do { x <- m; y <- f x; g y }`

The monad laws are now common-sense statements about how do-blocks should function. If one of these laws were invalidated, users would become confused, as you couldn't be able to manipulate things within the do-blocks as would be expected. The monad laws are, in essence, usability guidelines.

Chapter 5

Conclusion

To wrap everything up: we learned the basics of some essential Haskell typeclasses, from the simple Monoid typeclass to the central piece of the report, the Monad typeclass, not forgetting Functor, Applicative, MonadPlus and monad transformers; and we have also seen how it applies to different datatypes, such as the Maybe type, the IO type or even lists.

We also learned the mechanics of the very useful do notation, which enables imperative programming in Haskell; and reviewed the basics of Category Theory tiding everything up with the Hask category. Moreover, the rules for each of these typeclasses were also presented.

In conclusion, we can now apply this knowledge to our every day Haskell programming, and hopefully feel that monads have been demystified for us. Just remember, monad is to one what triad is to three!

Chapter 6

Bibliography

Saunders Mac Lane. Categories for the Working Mathematician. University of Chicago, Springer, 1998

Steve Awodey. Category Theory. Oxford University, Oxford University Press, 2010

B. O'Sullivan, D. Stewart, and J. Goerzen. Real world Haskell. O'Reilly, 2008

B. C. Ruiz, F. Gutiérrez, P. Guerrero, and J. Gallardo. Razonando con Haskell (Un curso sobre programación funcional). Thompson, 2004.

José A. Alonso Jiménez, Ma José Hidalgo Doblado. Piensa en Haskell (Ejercicios de programación funcional con Haskell). Universidad de Sevilla, 2012

Appendix A

Appendix: the Monoid type class

Not to be confused with the Monad class, the more pleasant Monoid class, with kind `* -> Constraint`, found in the Data.Monoid module, modelizes the semigroups or monoids.

A **monoid** in Mathematics is an algebraic structure consisting of a set of objects with an operation between them, being this operation *associative* and with a *neutral element*. Phew! But what is the meaning of this? By *associative* we mean that, if you have three elements a, b and c, then $a * (b * c) = (a * b) * c$. A *neutral element* is the one that does not worth to operate with, because it does nothing! To say, e is a *neutral element* if $e*a = a*e = a$, given any object a. As an example, you may take the real numbers as objects and the ordinary multiplication as operation.

Now that you know the math basics behind the Monoid class, let's see its definition:

```
class Monoid m where
  mempty :: m
  mappend :: m -> m -> m
  mconcat :: [m] -> m
  mconcat = foldr mappend mempty
  (<>) :: m -> m -> m    -- infix synonym for mappend
```

See that **mappend** corresponds to the monoid operation and **mempty** to its neutral element. The names of the methods may seem unsuitable, but they correspond to an example of monoid: the lists with the appending `(++)` operation. Who is the neutral element here? The empty list:

```
xs ++ [] = [] ++ xs = xs
```

Some examples:

The list monoid

```
instance Monoid [a] where
         mempty  = []
         mappend = (++)
         mconcat xss = [x | xs <- xss, x <- xs]
```

The monoid of functions with range a monoid

```
instance Monoid b => Monoid (a -> b) where
         mempty _ = mempty
         mappend f g x = f x `mappend` g x
```

The Unit monoid

```
instance Monoid () where
         mempty        = ()
         _ `mappend` _ = ()
         mconcat _     = ()
```

The cartesian product of two monoids

```
instance (Monoid a, Monoid b) => Monoid (a,b) where
         mempty = (mempty, mempty)
         (a1,b1) `mappend` (a2,b2) =
                 (a1 `mappend` a2, b1 `mappend` b2)
```

Lexicographical ordering

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

Lift a semigroup into 'Maybe' forming a 'Monoid'

```
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)
```

As you can see in all the examples, the following rules are verified:

```
  (x <> y) <> z = x <> (y <> z)   -- associativity
  mempty <> x = x                 -- left identity
  x <> mempty = x                 -- right identity
```

Appendix B

Appendix: the Maybe monad

We introduced monads using `Maybe` as an example. The `Maybe` monad represents computations which might "go wrong" by not returning a value. For reference, here are our definitions of `return` and `(>>=)` for `Maybe` as we saw in the main body:[1]

```
return :: a -> Maybe a
return x  = Just x

(>>=)  :: Maybe a -> (a -> Maybe b) -> Maybe b
m >>= g = case m of
             Nothing -> Nothing
             Just x  -> g x
```

B.1 Safe functions

The `Maybe` datatype provides a way to make a safety wrapper around *partial functions*, that is, functions which can fail to work for a range of arguments. For example, `head` and `tail` only work with non-empty lists. Another typical case, which we will explore in this section, are mathematical functions like `sqrt` and `log`; (as far as real numbers are concerned) these are only defined for non-negative arguments.

```
> log 1000
6.907755278982137
> log (-1000)
''ERROR'' -- runtime error
```

To avoid this crash, a "safe" implementation of log could be:

```
safeLog :: (Floating a, Ord a) => a -> Maybe a
safeLog x
    | x > 0     = Just (log x)
    | otherwise = Nothing

> safeLog 1000
Just 6.907755278982137
```

[1] The definitions in the actual instance in `Data.Maybe` are written a little differently, but are fully equivalent to these.

```
> safeLog -1000
Nothing
```

We could write similar "safe functions" for all functions with limited domains such as division, square-root, and inverse trigonometric functions (safeDiv, safeSqrt, safeArcSin, etc. all of which would have the same *type* as safeLog but definitions specific to their constraints)

If we wanted to combine these monadic functions, the cleanest approach is with monadic composition and point-free style:

```
safeLogSqrt = safeLog <=< safeSqrt
```

Written in this way, safeLogSqrt resembles a lot its unsafe, non-monadic counterpart:

```
unsafeLogSqrt = log . sqrt
```

B.2 Lookup tables

A lookup table relates *keys* to *values*. You look up a value by knowing its key and using the lookup table. For example, you might have a phone book application with a lookup table where contact names are keys to corresponding phone numbers. An elementary way of implementing lookup tables in Haskell is to use a list of pairs: [(a, b)]. Here a is the type of the keys, and b the type of the values.[2]Here's how the phone book lookup table might look like:

```
phonebook :: [(String, String)]
phonebook = [ ('Bob',   '01788 665242'),
              ('Fred',  '01624 556442'),
              ('Alice', '01889 985333'),
              ('Jane',  '01732 187565') ]
```

The most common thing you might do with a lookup table is look up values. Everything is fine if we try to look up "Bob", "Fred", "Alice" or "Jane" in our phone book, but what if we were to look up "Zoe"? Zoe isn't in our phone book, so the lookup would fail. Hence, the Haskell function to look up a value from the table is a Maybe computation (it is available from Prelude):

```
lookup :: Eq a => a  -- a key
       -> [(a, b)]   -- the lookup table to use
       -> Maybe b    -- the result of the lookup
```

Let us explore some of the results from lookup:

```
Prelude> lookup 'Bob' phonebook
Just '01788 665242'
Prelude> lookup 'Jane' phonebook
Just '01732 187565'
Prelude> lookup 'Zoe' phonebook
Nothing
```

Now let's expand this into using the full power of the monadic interface. Say, we're now working for the government, and once we have a phone number from our contact, we want to look up this phone number in a big, government-sized lookup table to find out the registration number of their car. This, of course, will be another Maybe-computation. But if the person we're looking for

[2]Check Data.Map for a different, and potentially more useful, implementation.

isn't in our phone book, we certainly won't be able to look up their registration number in the governmental database.

What we need is a function that will take the results from the first computation and put it into the second lookup *only* if we get a successful value in the first lookup. Of course, our final result should be `Nothing` if we get `Nothing` from either of the lookups.

```
getRegistrationNumber :: String       -- their name
                      -> Maybe String -- their Reg.Num.
getRegistrationNumber name =
  lookup name phonebook >>=
    (\number -> lookup number governmentDatabase)
```

If we then wanted to use the result from the governmental database lookup in a third lookup (say we want to look up their registration number to see if they owe any car tax), then we could extend our `getRegistrationNumber` function:

```
getTaxOwed :: String       -- their name
           -> Maybe Double -- the amount of tax they owe
getTaxOwed name =
  lookup name phonebook >>=
    (\number -> lookup number governmentDatabase) >>=
      (\registration -> lookup registration taxDatabase)
```

Or, using the `do`-block style:

```
getTaxOwed name = do
  number       <- lookup name phonebook
  registration <- lookup number governmentDatabase
  lookup registration taxDatabase
```

Let's just pause here and think about what would happen if we got a `Nothing` anywhere. By definition, when the first argument to `(>>=)` is `Nothing`, it just returns `Nothing` while ignoring whatever function it is given. Thus, a `Nothing` at any stage in the large computation will result in a `Nothing` overall, regardless of the other functions. After the first `Nothing` hits, all `(>>=)`s will just pass it to each other, skipping the other function arguments. The technical description says that the structure of the `Maybe` monad **propagates failures**.

B.3 Open monads

Another trait of the `Maybe` monad is that it is **open**: if we have a `Just` value, we can see the contents and extract the associated values through pattern matching.

```
zeroAsDefault :: Maybe Int -> Int
zeroAsDefault mx = case mx of
    Nothing -> 0
    Just x -> x
```

This usage pattern of replacing `Nothing` with a default is captured by the `fromMaybe` function in `Data.Maybe`.

```
zeroAsDefault :: Maybe Int -> Int
zeroAsDefault mx = fromMaybe 0 mx
```

The maybe Prelude function allows us to do it in a more general way, by supplying a function to modify the extracted value.

```
displayResult :: Maybe Int -> String
displayResult mx = maybe s1 ((s2++).show) mx
  where
    s1 = 'There was no result'
    s2 = 'The result was'


Prelude> :t maybe
maybe :: b -> (a -> b) -> Maybe a -> b
Prelude> displayResult (Just 10)
'The result was 10'
Prelude> displayResult Nothing
'There was no result'
```

This possibility makes sense for Maybe, as it allows us to recover from failures. Not all monads are open in this way; often, they are designed to hide unnecessary details. return and (>>=) alone do not allow us to extract the underlying value from a monadic computation, and so it is perfectly possible to make a "no-exit" monad, from which it is never possible to extract values. The most obvious example of that is the IO monad.

B.4 Maybe and safety

We have seen how Maybe can make code safer by providing a graceful way to deal with failure that does not involve runtime errors. Does that mean we should always use Maybe for everything? Not really.

When you write a function, you are able to tell whether it might fail to produce a result during normal operation of the program[3], either because the functions you use might fail (as in the examples in this chapter) or because you know some of the argument or intermediate result values do not make sense (for instance, imagine a calculation that is only meaningful if its argument is less than 10). If that is the case, by all means use Maybe to signal failure; it is far better than returning an arbitrary default value or throwing an error.

Now, adding Maybe to a result type without a reason would only make the code more confusing and no safer. The type signature of a function with unnecessary Maybe would tell users of the code that the function could fail when it actually can't. Of course, that is not as bad a lie as the opposite one (that is, claiming that a function will not fail when it actually can), but we really want honest code in *all* cases. Furthermore, using Maybe forces us to propagate failure (with fmap or monadic code) and eventually handle the failure cases (using pattern matching, the maybe function, or fromMaybe from Data.Maybe). If the function cannot actually fail, coding for failure is an unnecessary complication.

[3] With "normal operation" we mean to exclude failure caused by uncontrollable circumstances in the real world, such as memory exhaustion or a dog chewing the printer cable.

Appendix C

Appendix: The List monad

Lists are a fundamental part of Haskell, and we've used them extensively before getting to this chapter. The novel insight is that the list type is a monad too!

As monads, lists are used to model *nondeterministic* computations which may return an arbitrary number of results. There is a certain parallel with how Maybe represented computations which could return zero or one value; but with lists, we can return zero, one, or many values (the number of values being reflected in the length of the list).

C.1 List instantiated as monad

The return function for lists simply injects a value into a list:

```
return x  =  [x]
```

In other words, return here makes a list containing one element, namely the single argument it took. The type of the *list return* is return :: $a \to [a]$, or, equivalently, return :: $a \to [\,]\,a$. The latter style of writing it makes it more obvious that we are replacing the generic type constructor (which we had called M in Understanding monads) by the list type constructor [ ] (which is distinct from but easy to confuse with the empty list!).

The binding operator is less trivial. We will begin by considering its type, which for the case of lists should be:

```
[a] -> (a -> [b]) -> [b]
```

This is just what we'd expect: it pulls out the value from the list to give to a function that returns a new list.

The actual process here involves first mapping a given function over a given list to get back a list of lists, i.e. type [[b]] (of course, many functions which you might use in mapping do not return lists; but, as shown in the type signature above, **monadic binding for lists only works with functions that return lists**). To get back to a regular list, we then concatenate the elements of our list of lists to get a final result of type [b]. Thus, we can define the list version of (>>=):

```
xs >>= f = concat (map f xs)
```

The bind operator is key to understanding how different monads do their jobs, and its definition indicates the chaining strategy for working with the monad.

For the list monad, non-determinism is present because different functions may return any number of different results when mapped over lists.

Bunny invasion

```
Prelude> let generation = replicate 2
Prelude> ['bunny'] >>= generation
['bunny','bunny']
Prelude> ['bunny'] >>= generation >>= generation
['bunny','bunny','bunny','bunny']
```

In this silly example all elements are equal, but the same overall logic could be used to model radioactive decay, or chemical reactions, or any phenomena that produces a series of elements starting from a single one.

C.2 Board game example

Suppose we are modeling a turn-based board game and want to find all the possible ways the game could progress. We would need a function to calculate the list of options for the next turn, given a current board state:

```
nextConfigs :: Board -> [Board]
nextConfigs bd = undefined -- details not important
```

To figure out all the possibilities after two turns, we would again apply our function to each of the elements of our new list of board states. Our function takes a single board state and returns a list of possible new states. Thus, we can use monadic binding to map the function over each element from the list:

```
nextConfigs bd >>= nextConfigs
```

In the same fashion, we could bind the result back to the function yet again (ad infinitum) to generate the next turn's possibilities. Depending on the particular game's rules, we may reach board states that have no possible next-turns; in those cases, our function will return the empty list.

On a side note, we could translate several turns into a do block (like we did for the grandparents example in Understanding monads):

```
threeTurns :: Board -> [Board]
threeTurns bd = do
  bd1 <- nextConfigs bd
  bd2 <- nextConfigs bd1
  nextConfigs bd2
```

If the above looks too magical, keep in mind that `do` notation is syntactic sugar for `(>>=)` operations. To the right of each left-arrow, there is a function with arguments that evaluate to a list; the variable to the left of the arrow stands for the list elements. After a left-arrow assignment line, there can be later lines that call the assigned variable as an argument for a function. This later function will be performed for *each* of the elements from within the list that came from the left-arrow line's function. This per-element process corresponds to the 'map' in the definition of `(>>=)`. A resulting list of lists (one per element of the original list) will be flattened into a single list (the 'concat' in the definition of `(>>=)`).

C.3 List comprehensions

The list monad works in a way that has uncanny similarity to list comprehensions. Let's slightly modify the `do` block we just wrote for `threeTurns` so that it ends with a `return`...

```
threeTurns bd = do
  bd1 <- nextConfigs bd
  bd2 <- nextConfigs bd1
  bd3 <- nextConfigs bd2
  return bd3
```

This mirrors exactly the following list comprehension:

```
threeTurns bd =
  [ bd3 | bd1 <- nextConfigs bd, bd2 <- nextConfigs bd1, bd3 <- nextConfigs bd2 ]
```

(In a list comprehension, it is perfectly legal to use the elements drawn from one list to define the following ones, like we did here.)

The resemblance is no coincidence: list comprehensions are, behind the scenes, defined in terms of concatMap

```
concatMap f xs = concat (map f xs)
```

. That's just the list monad binding definition again! To summarize the nature of the list monad: binding for the list monad is a combination of concatenation and mapping, and so the combined function concatMap is effectively the same as (>>=) for lists (except for different syntactic order).

For the correspondence between list monad and list comprehension to be complete, we need a way to reproduce the filtering that list comprehensions can do. Search for Additive Monads (MonadPlus).

Appendix D

Appendix: The IO (Input/Output) monad

Haskell is a *functional* and *lazy* language. However, the real world effects of input/output operations can't be expressed through pure functions. Furthermore, in most cases I/O can't be done lazily. Since lazy computations are only performed when their values become necessary, unfettered lazy I/O would make the order of execution of the real world effects unpredictable. Haskell addresses these issues through the IO monad.

D.1 Input/output and purity

Haskell functions are *pure*: when given the same arguments, they return the same results. Pure functions are reliable and predictable; they ease debugging and validation. Test cases can also be set up easily since we can be sure that nothing other than the arguments will influence a function's result. Being entirely contained within the program, the Haskell compiler can evaluate functions thoroughly in order to optimize the compiled code.

So, how do we manage actions like opening a network connection, writing a file, reading input from the outside world, or anything else that does something more than returning a calculated result? Well, the key is: *these actions are not functions.* The IO monad is a means to represent actions as Haskell values, so that we can manipulate them with pure functions.

D.2 Combining functions and I/O actions

Let's combine functions with I/O to create a full program that will:

1. Ask the user to insert a string
2. Read their string
3. Use fmap to apply a function shout that capitalizes all the letters from the string
4. Write the resulting string

```
module Main where

import Data.Char (toUpper)
import Control.Monad
```

```
main = putStrLn "Write your string: " >> fmap shout getLine >>= putStrLn

shout = map toUpper
```

We have a full-blown program, but we didn't include any type definitions. Which parts are functions and which are IO actions or other values? We can load our program in GHCi and check the types:

```
main :: IO ()
putStrLn :: String -> IO ()
"Write your string: " :: [Char]
(>>) :: Monad m => m a -> m b -> m b
fmap :: Functor m => (a -> b) -> m a -> m b
shout :: [Char] -> [Char]
getLine :: IO String
(>>=) :: Monad m => m a -> (a -> m b) -> m b
```

Whew, that is a lot of information there. We've seen all of this before, but let's review.

`main` is `IO ()`. That's not a function. Functions are of types $a \to b$. Our entire program is an IO action.

`putStrLn` is a function, but it results in an IO action. The "Write your string:" text is a `String` (remember, that's just a synonym for `[Char]`). It is used as an argument for `putStrLn` and is incorporated into the IO action that results. So, `putStrLn` is a function, but `putStrLn x` evaluates to an IO action. The `()` part of the IO type indicates that nothing is available to be passed on to any later functions or actions.That last part is key. We sometimes say informally that an IO action "returns" something; however, taking that too literally leads to confusion. It is clear what we mean when we talk about *functions* returning results, but IO actions are not functions. Let's skip down to `getLine` - an IO action that *does* provide a value. `getLine` is not a function that returns a `String` because `getLine` *isn't a function.* Rather, `getLine` is an IO action which, when evaluated, will materialize a `String`, which can then be passed to later functions through, for instance, `fmap` and `(>>=)`.When we use `getLine` to get a `String`, the value is monadic because it is wrapped in `IO` functor (which happens to be a monad). We cannot pass the value directly to a function that takes plain (non-monadic, or non-functorial) values. `fmap` does the work of taking a non-monadic function while passing in and returning monadic values.

As we've seen already, `(>>=)` does the work of passing a monadic value into a function that takes a non-monadic value and returns a monadic value. It may seem inefficient for `fmap` to take the non-monadic result of its given function and return a monadic value only for `(>>=)` to then pass the underlying non-monadic value to the next function. It is precisely this sort of chaining, however, that creates the reliable sequencing that make monads so effective at integrating pure functions with IO actions.

do notation review

Given the emphasis on sequencing, the `do` notation can be especially appealing with the `IO` monad. Our program

```
putStrLn "Write your string: " >> fmap shout getLine >>= putStrLn
```

could be written as:

```
do putStrLn "Write your string: "
   string <- getLine
   putStrLn (shout string)
```

D.3 The universe as part of our program

One way of viewing the IO monad is to consider IO a as a computation which provides a value of type a while changing *the state of the world* by doing input and output. Obviously, you cannot literally set the state of the world; it is hidden from you, as the IO functor is abstract (that is, you cannot dig into it to see the underlying values; it is closed in a way opposite to that in which Maybe can be said to be open). Seen this way, IO is roughly analogous to the State monad, which we will meet shortly. With State, however, the state being changed is made of normal Haskell values, and so we can manipulate it directly with pure functions.

Understand that this idea of the universe as an object affected and affecting Haskell values through IO is only a metaphor; a loose interpretation at best. The more mundane fact is that IO simply brings some very base-level operations into the Haskell language.[1] Remember that Haskell is an abstraction, and that Haskell programs must be compiled to machine code in order to actually run. The actual workings of IO happen at a lower level of abstraction, and are wired into the very definition of the Haskell language.[2]

D.4 Pure and impure

Consider the following snippet:

```
speakTo :: (String -> String) -> IO String
  speakTo fSentence = fmap fSentence getLine

  -- Usage example.
  sayHello :: IO String
  sayHello = speakTo (\name -> "Hello, " ++ name ++ "!")
```

In most other programming languages, which do not have separate types for I/O actions, speakTo would have a type akin to:

```
  speakTo :: (String -> String) -> String
```

With such a type, however, speakTo would not be a function at all! Functions produce the same results when given the same arguments; the String delivered by speakTo, however, also depends on whatever is typed at the terminal prompt. In Haskell, we avoid that pitfall by returning an IO String, which is not a String but a promise that *some* String will be delivered by carrying out certain instructions involving I/O (in this case, the I/O consists of getting a line of input from the terminal). Though the String can be different each time speakTo is evaluated, the I/O instructions are always the same.

[1] The technical term is "primitive", as in primitive operations.

[2] The same can be said about all higher-level programming languages, of course. Incidentally, Haskell's IO operations can actually be extended via the *Foreign Function Interface* (FFI) which can make calls to C libraries. As C can use inline assembly code, Haskell can indirectly engage with anything a computer can do. Still, Haskell functions manipulate such outside operations only *indirectly* as values in IO functors.

When we say Haskell is a purely functional language, we mean that all of its functions are *really* functions, which is not the case in most other languages. To be precise, Haskell expressions are always *referentially transparent*; that is, you can always replace an expression (such as speakTo) with its value (in this case, \fSentence -¿ fmap fSentence getLine) without changing the behaviour of the program. The String delivered by getLine, in contrast, is opaque; its value is not specified and can't be discovered in advance by the program. If speakTo had the problematic type we mentioned above, sayHello would be a String; however, replacing it by any specific string would break the program.

In spite of Haskell being purely functional, IO actions can be said to be *impure* because their impact on the outside world are *side effects* (as opposed to the regular effects that are entirely contained within Haskell). Programming languages that lack purity may have side-effects in many other places connected with various calculations. Purely functional languages, however, assure that *even expressions with impure values are referentially transparent.* That means we can talk about, reason about and handle impurity in a purely functional way, using purely functional machinery such as functors and monads. While IO actions are impure, all of the Haskell functions that manipulate them remain pure.

Functional purity, coupled to the fact that I/O shows up in types, benefit Haskell programmers in various ways. The guarantees about referential transparency increase a lot the potential for compiler optimizations. IO values being distinguishable through types alone make it possible to immediately tell where we are engaging with side effects or opaque values. As IO itself is just another functor, we maintain to the fullest extent the predictability and ease of reasoning associated with pure functions.

D.5 Functional and imperative

When we introduced monads, we said that a monadic expression can be interpreted as a statement of an imperative language. That interpretation is immediately compelling for IO, as the language around IO actions looks a lot like a conventional imperative language. It must be clear, however, that we are talking about an *interpretation.* We are not saying that monads or do notation turn Haskell into an imperative language. The point is merely that you can view and understand monadic code in terms of imperative statements. The semantics may be imperative, but the implementation of monads and (>>=) is still purely functional. To make this distinction clear, let's look at a little illustration:

```
int x;
 scanf("%d", &x);
 printf("%d\n", x);
```

This is a snippet of C, a typical imperative language. In it, we declare a variable x, read its value from user input with scanf and then print it with printf. We can, within an IO do block, write a Haskell snippet that performs the same function and looks quite similar:

```
x <- readLn
 print x
```

Semantically, the snippets are nearly equivalent.[3] In the C code, however, the statements directly

[3]One difference is that x is a mutable variable in C, and so it is possible to declare it in one statement and set its value in the next; Haskell never allows such mutability. If we wanted to imitate the C code even more closely, we

correspond to instructions to be carried out by the program. The Haskell snippet, on the other hand, is desugared to:

```
readLn >>= \x -> print x
```

The desugared version has no statements, only functions being applied. We tell the program the order of the operations indirectly as a simple consequence of *data dependencies*: when we chain monadic computations with (>>=), we get the later results by applying functions to the results of the earlier ones. It just happens that, for instance, evaluating print x leads to a string to be printed in the terminal.

When using monads, Haskell allows us to write code with imperative semantics while keeping the advantages of functional programming.

D.6 I/O in the libraries

So far the only I/O primitives we have used were putStrLn and getLine and small variations thereof. The standard libraries, however, offer many other useful functions and actions involving IO. We present some of the most important ones in the next appendix, including the basic functionality needed for reading from and writing to files.

D.7 monadic control structures

Given that monads allow us to express sequential execution of actions in a wholly general way, could we use them to implement common iterative patterns, such as loops? In this section, we will present a few of the functions from the standard libraries which allow us to do precisely that. While the examples are presented here applied to IO, keep in mind that the following ideas apply to *every* monad.

Remember, there is nothing magical about monadic values; we can manipulate them just like any other values in Haskell. Knowing that, we might think to try the following function to get five lines of user input:

```
fiveGetLines = replicate 5 getLine
```

That won't do, however (try it in GHCi!). The problem is that replicate produces, in this case, a list of actions, while we want an action which returns a list (that is, IO [String] rather than [IO String]). What we need is a *fold* to run down the list of actions, executing them and combining the results into a single list. As it happens, there is a Prelude function which does that: sequence.

```
sequence :: (Monad m) => [m a] -> m [a]
```

And so, we get the desired action with:

```
fiveGetLines = sequence $ replicate 5 getLine
```

---

could have used an IORef, which is a cell that contains a value which can be destructively updated. For obvious reasons, IORefs can only be used within the IO monad.

replicate and sequence form an appealing combination; so Control.Monad offers a replicateM function for repeating an action an arbitrary number of times. Control.Monad provides a number of other convenience functions in the same spirit - monadic zips, folds, and so on.

```
fiveGetLinesAlt = replicateM 5 getLine
```

A particularly important combination is map and sequence. Together, they allow us to make actions from a list of values, run them sequentially, and collect the results. mapM, a Prelude function, captures this pattern:

```
mapM :: (Monad m) => (a -> m b) -> [a] -> m [b]
```

We also have variants of the above functions with a trailing underscore in the name, such as sequence_, mapM_ and replicateM_. These discard any final values and so are appropriate when you are only interested in performing actions. Compared with their underscore-less counterparts, these functions are like the distinction between (>>=) and (>>=). mapM_ for instance has the following type:

```
mapM_ :: (Monad m) => (a -> m b) -> [a] -> m ()
```

Finally, it is worth mentioning that Control.Monad also provides forM and forM_, which are flipped versions of mapM and mapM_. forM_ happens to be the idiomatic Haskell counterpart to the imperative for-each loop; and the type signature suggests that neatly:

```
forM_ :: (Monad m) => [a] -> (a -> m b) -> m ()
```

Appendix E

Appendix: The IO library

Here, we'll explore the most commonly used elements of the System.IO module.

```
data IOMode  = ReadMode   | WriteMode
             | AppendMode | ReadWriteMode

openFile     :: FilePath -> IOMode -> IO Handle
hClose       :: Handle -> IO ()

hIsEOF       :: Handle -> IO Bool

hGetChar     :: Handle -> IO Char
hGetLine     :: Handle -> IO String
hGetContents :: Handle -> IO String

getChar      :: IO Char
getLine      :: IO String
getContents  :: IO String

hPutChar     :: Handle -> Char -> IO ()
hPutStr      :: Handle -> String -> IO ()
hPutStrLn    :: Handle -> String -> IO ()

putChar      :: Char -> IO ()
putStr       :: String -> IO ()
putStrLn     :: String -> IO ()

readFile     :: FilePath -> IO String
writeFile    :: FilePath -> String -> IO ()
```

Note FilePath is a *type synonym* for String. So, for instance, the readFile function takes a String (the file to read) and returns an action that, when run, produces the contents of that file.

Most of the IO functions are self-explanatory. The openFile and hClose functions open and close a file, respectively. The IOMode argument determines the mode for opening the file. hIsEOF tests for end-of file. hGetChar and hGetLine read a character or line (respectively) from a file. hGetContents reads the entire file. The getChar, getLine, and getContents

variants read from standard input. hPutChar prints a character to a file; hPutStr prints a string; and hPutStrLn prints a string with a newline character at the end. The variants without the h prefix work on standard output. The readFile and writeFile functions read and write an entire file without having to open it first.

E.1 Bracket

The bracket function comes from the Control.Exception module. It helps perform actions safely.

```
bracket :: IO a -> (a -> IO b) -> (a -> IO c) -> IO c
```

Consider a function that opens a file, writes a character to it, and then closes the file. When writing such a function, one needs to be careful to ensure that, if there were an error at some point, the file is still successfully closed. The bracket function makes this easy. It takes three arguments: The first is the action to perform at the beginning. The second is the action to perform at the end, regardless of whether there's an error or not. The third is the action to perform in the middle, which might result in an error. For instance, our character-writing function might look like:

```
writeChar :: FilePath -> Char -> IO ()
writeChar fp c =
    bracket
      (openFile fp WriteMode)
      hClose
      (\h -> hPutChar h c)
```

This will open the file, write the character, and then close the file. However, if writing the character fails, hClose will still be executed, and the exception will be reraised afterwards. That way, you don't need to worry too much about catching the exceptions and about closing all of your handles.

E.2 A file reading program

We can write a simple program that allows a user to read and write files. The interface is admittedly poor, and it does not catch all errors (such as reading a non-existent file). Nevertheless, it should give a fairly complete example of how to use IO. Enter the following code into "FileRead.hs", and compile/run:

```
module Main
      where

  import System.IO
  import Control.Exception

  main = doLoop

  doLoop = do
    putStrLn "Enter a command rFN wFN or q to quit:"
    command <- getLine
    case command of
      'q':_ -> return ()
      'r':filename -> do putStrLn ("Reading " ++ filename)
                         doRead filename
                         doLoop
```

```
    'w':filename -> do putStrLn ("Writing " ++ filename)
                       doWrite filename
                       doLoop
    _            -> doLoop

doRead filename =
    bracket (openFile filename ReadMode) hClose
            (\h -> do contents <- hGetContents h
                      putStrLn "The first 100 chars:"
                      putStrLn (take 100 contents))

doWrite filename = do
  putStrLn "Enter text to go into the file:"
  contents <- getLine
  bracket (openFile filename WriteMode) hClose
          (\h -> hPutStrLn h contents)
```

What does this program do? First, it issues a short string of instructions and reads a command. It then performs a **case** switch on the command and checks first to see if the first character is a 'q'. If it is, it returns a value of unit type.

Note The return function is a function that takes a value of type a and returns an action of type IO a. Thus, the type of return () is IO ().

If the first character of the command wasn't a 'q', the program checks to see if it was an 'r' followed by some string that is bound to the variable filename. It then tells you that it's reading the file, does the read and runs doLoop again. The check for 'w' is nearly identical. Otherwise, it matches _, the wildcard character, and loops to doLoop.

The doRead function uses the bracket function to make sure there are no problems reading the file. It opens a file in ReadMode, reads its contents and prints the first 100 characters.

The doWrite function asks for some text, reads it from the keyboard, and then writes it to the specified file.

Note Both doRead and doWrite could have been made simpler by using readFile and writeFile, but they were written in the extended fashion to show how the more complex functions are used.

The program has one major problem: it will die if you try to read a file that doesn't already exist or if you specify some bad filename like `*bs^#_@`. You may think that the calls to bracket in doRead and doWrite should take care of this, but they don't. They only catch exceptions within the main body, not within the startup or shutdown functions (openFile and hClose, in these cases). To make this completely reliable, we would need a way to catch exceptions raised by openFile.

Appendix F

Appendix: The State monad (Random Number Generation)

If you have programmed in any other language before, you likely wrote some functions that "kept state". For those new to the concept, a *state* is one or more variables that are required to perform some computation but are not among the arguments of the relevant function. Object-oriented languages, like C++, suggest extensive use of state variables within objects in the form of member variables. Programs written in procedural languages, like C, typically use variables declared outside the current scope to keep track of state.

In Haskell, however, such techniques are not as straightforward to apply. They require mutable variables and imply functions will have hidden dependencies, which is at odds with Haskell's functional purity. Fortunately, in most cases it is possible to avoid such extra complications and keep track of state in a functionally pure way. We do so by passing the state information from one function to the next, thus making the hidden dependencies explicit. The `State` type is a tool crafted to make this process of threading state through functions more convenient. In this chapter, we will see how it can assist us in a typical problem involving state: generating pseudo-random numbers.

F.1 Pseudo-Random Numbers

Generating actual random numbers is far from easy. Computer programs almost always use *pseudo*-random numbers instead. They are called "pseudo" because they are not truly random. Rather, they are genererated by algorithms (the pseudo-random number generators) which take an initial state (commonly called the seed) and produce from it a sequence of numbers that have the appearance of being random.[1]

Every time a pseudo-random number is requested, state somewhere must be updated, so that the generator can be ready for producing a fresh, different random number. Sequences of pseudo-random numbers can be replicated exactly if the initial seed and the generating algorithm are known.

F.1.1 Implementation in Haskell

Producing a pseudo-random number in most programming languages is very simple: there is a function somewhere in the libraries that provides a pseudo-random value (perhaps even a truly ran-

[1] A common source of seeds is the current date and time as given by the internal clock of the computer. Assuming the clock is functioning correctly, it can provide unique seeds suitable for most day-to-day needs (as opposed to applications which demand high-quality randomness, as in cryptography or statistics)

dom one, depending on how it is implemented). Haskell has a similar one in the System.Random module from the random package:

```
GHCi> :m System.Random
GHCi> :t randomIO
randomIO :: Random a => IO a
GHCi> randomIO
-1557093684
GHCi> randomIO
1342278538
```

randomIO is an IO action. It couldn't be otherwise, as it makes use of mutable state, which is kept out of reach from our Haskell programs. Thanks to this hidden dependency, the pseudo-random values it gives back can be different every time.

F.1.2 Example: rolling dice

Suppose we are coding a game in which at some point we need an element of chance. In real-life games that is often obtained by means of dice. So, let's create a dice-throwing function. We'll use the IO function randomIO, which allows us to specify a range from which the pseudo-random values will be taken. For a 6 die, the call will be randomIO (1,6).

```
import Control.Monad
import System.Random

rollDiceIO :: IO (Int, Int)
rollDiceIO = liftM2 (,) (randomRIO (1,6)) (randomRIO (1,6))
```

That function rolls two dice. Here, liftM2 is used to make the non-monadic two-argument function (,) work within a monad. The (,) is the non-infix version of the tuple constructor. Thus, the two die rolls will be returned as a tuple in IO.

Getting rid of *IO*

A disadvantage of randomIO is that it requires us to use IO and store our state outside the program, where we can't control what happens to it. We would rather only use I/O when there is an unavoidable reason to interact with the outside world.

To avoid bringing IO into play, we can build a *local* generator. The random and mkStdGen functions in System.Random allow us to generate tuples containing a pseudo-random number together with an updated generator to use the next time the function is called.

```
GHCi> :m System.Random
GHCi> let generator = mkStdGen 0
  -- ``0'' is our seed
GHCi> :t generator
generator :: StdGen
GHCi> generator
1 1
GHCi> :t random
random :: (RandomGen g, Random a) => g -> (a, g)
GHCi> random generator :: (Int, StdGen)
(2092838931,1601120196 1655838864)
```

Note In `random generator :: (Int, StdGen)`, we use the `::` to introduce a *type annotation*, which is essentially a type signature that we can put in the middle of an expression. Here, we are saying that the expression to the right, `random generator` has type `(Int, StdGen)`. It makes sense to use a type annotation here because, as we will discuss later, `random` can produce values of different types, so if we want it to give us an `Int` we'd better specify it in some way.

While we managed to avoid `IO`, there are new problems. First and foremost, if we want to use `generator` to get random numbers, the obvious definition...

```
GHCi> let randInt = fst . random $ generator :: Int
GHCi> randInt
2092838931
```

... is useless. It will always give back the same value, `2092838931`, as the same generator in the same state will be used every time. To solve that, we can take the second member of the tuple (that is, the new generator) and feed it to a *new* call to `random`:

```
GHCi> let (randInt, generator') = random generator :: (Int, StdGen)
GHCi> randInt
  -- Same value
2092838931
GHCi> random generator' :: (Int, StdGen)
  -- Using new generator' returned from ``random generator''
(-2143208520,439883729 1872071452)
```

That, of course, is clumsy and rather tedious, as we now need to deal with the fuss of carefully passing the generator around.

F.1.3 Dice without IO

We can re-do our dice throw with our new approach using the `randomR` function:

```
GHCi> randomR (1,6) (mkStdGen 0)
(6, 40014 40692)
```

The resulting tuple combines the result of throwing a single die with a new generator. A simple implementation for throwing two dice is then:

```
clumsyRollDice :: (Int, Int)
clumsyRollDice = (n, m)
        where
        (n, g) = randomR (1,6) (mkStdGen 0)
        (m, _) = randomR (1,6) g
```

The implementation of `clumsyRollDice` works as an one-off, but we have to manually pass the generator `g` from one `where` clause to the other. This approach becomes increasingly cumbersome as our programs get more complex, which means we have more values to shift around. It is also error-prone: what if we pass one of the middle generators to the wrong line in the `where` clause?

What we really need is a way to automate the extraction of the second member of the tuple (i.e. the new generator) and feed it to a new call to `random`. This is where the `State` comes into the picture.

F.2 Introducing *State*

Note In this chapter we will use the state monad provided by the module Control.Monad.Trans.State of the transformers package. By reading Haskell code in the wild, you will soon meet Control.Monad.State, a module of the closely related mtl package. The differences between these two modules need not concern us at the moment; everything we discuss here also applies to the mtl variant.

The Haskell type State describes functions that consume a state and produce both a result and an updated state, which are given back in a tuple.

The state function is wrapped by a data type definition which comes along with a runState accessor so that pattern matching becomes unnecessary. For our current purposes, the State type might be defined as:

```
newtype  State s a  =  State { runState :: s -> (a, s) }
```

Here, s is the type of the state, and a the type of the produced result. Calling the type State is arguably a bit of a misnomer because the wrapped value is not the state itself but a *state processor*.

newtype

Note that we defined the data type with the newtype keyword, rather than the usual data. newtype can be used only for types with just one constructor and just one field.

It ensures that the trivial wrapping and unwrapping of the single field is eliminated by the compiler. For that reason, simple wrapper types such as State are usually defined with newtype. Would defining a synonym with type be enough in such cases? Not really, because type does not allow us to define instances for the new data type, which is what we are about to do...

F.2.1 Where did the *State* constructor go?

When you start using Control.Monad.Trans.State, you will quickly notice there is no State constructor available. The transformers package implements the State type in a somewhat different way. The differences do not affect how we use or understand State; except that, instead of a State constructor, Control.Monad.Trans.State exports a state function,

```
state :: (s -> (a, s)) -> State s a
```

which does the same job. As for *why* the implementation is not the obvious one we presented above, we will get back to that a few chapters down the road.

F.2.2 Instantiating the monad

So far, all we have done was to wrap a function type and give it a name. There is another ingredient, however: State is a monad, and that gives us very handy ways of using it. Unlike the instances of Functor or Monad we have seen so far, State has *two* type parameters. Since the type class only allows one parametrised parameter, the last one, we have to indicate the other one, s, will be fixed.

```
instance Monad (State s) where
```

That means there are actually *many* different State monads, one for each possible type of state - State String, State Int, State SomeLargeDataStructure, and so forth. Naturally, we only

need to write one implementation of `return` and `(>>=)`; the methods will be able to deal with all choices of `s`.

The `return` function is implemented as:

```
return :: a -> State s a
return x = state ( \ st -> (x, st) )
```

Giving a value (`x`) to `return` produces a function which takes a state (`st`) and returns it unchanged, together with value we want to be returned. As a finishing step, the function is wrapped up with the `state` function.

Binding is a bit intricate:

```
(>>=) :: State s a -> (a -> State s b) -> State s b
pr >>= k = state $ \ st ->
   let (x, st') = runState pr st
       -- Running the first processor on st.
   in runState (k x) st'
       -- Running the second processor on st'.
```

`(>>=)` is given a state processor (`pr`) and a function (`k`) that is used to create another processor from thGe result of the first one. The two processors are combined into a function that takes the `initial` state (`st`) and returns the *second* result and the *third* state (i.e. the output of the second processor). Overall, `(>>=)` here allows us to run two state processors in sequence, while allowing the result of the first stage to influence what happens in the second one.

One detail in the implementation is how `runState` is used to undo the `State` wrapping, so that we can reach the function that will be applied to the states. The type of `runState pr`, for instance, is `s -> (a, s)`.

F.2.3 Setting and accessing the State

The monad instance allows us to manipulate various state processors, but you may at this point wonder where exactly the *original* state comes from in the first place. That issue is handily dealt with by the function `put`:

```
put newState = state $ \_ -> ((), newState)
```

Given a state (the one we want to introduce), `put` generates a state processor which ignores whatever state it receives, and gives back the state we originally provided to `put`. Since we don't care about the result of this processor (all we want to do is to replace the state), the first element of the tuple will be `()`, the universal placeholder value.

As a counterpart to `put`, there is `get`:

```
get = state $ \st -> (st, st)
```

The resulting state processor gives back the state `st` it is given in both as a result and as a state. That means the state will remain unchanged, and that a copy of it will be made available for us to manipulate.

F.2.4 Getting Values and State

As we have seen in the implementation of (>>=), runState is used to unwrap the State a b value to get the actual state processing function, which is then applied to some initial state. Other functions which are used in similar ways are evalState and execState. Given a State a b and an initial state, the function evalState will give back only the result value of the state processing, whereas execState will give back just the new state.

```
evalState :: State s a -> s -> a
evalState pr st = fst (runState pr st)

execState :: State s a -> s -> s
execState pr st = snd (runState pr st)
```

F.2.5 Dice and state

Time to use the State monad for our dice throw examples.

```
import Control.Monad.Trans.State
import System.Random
```

We want to generate Int dice throw results from a pseudo-random generator of type StdGen. Therefore, the type of our state processors will be State StdGen Int, which is equivalent to $StdGenn; \rightarrow (Int, StdGen)$ bar the wrapping.

We can now implement a processor that, given a StdGen generator, produces a number between 1 and 6. Now, the type of randomR is:

```
-- The StdGen type we are using is an instance of RandomGen.
randomR :: (Random a, RandomGen g) => (a, a) -> g -> (a, g)
```

Doesn't it look familiar? If we assume a is Int and g is StdGen it becomes:

```
randomR (1, 6) :: StdGen -> (Int, StdGen)
```

We already have a state processing function! All that is missing is to wrap it with state:

```
rollDie :: State StdGen Int
rollDie = state $ randomR (1, 6)
```

For illustrative purposes, we can use get, put and do-notation to write rollDie in a very verbose way which displays explicitly each step of the state processing:

```
rollDie :: State StdGen Int
rollDie = do generator <- get
             let (value, newGenerator) = randomR (1,6) generator
             put newGenerator
             return value
```

Let's go through each of the steps:

1. First, we take out the pseudo-random generator from the monadic context with $<-$, so that we can manipulate it.
2. Then, we use the randomR function to produce an integer between 1 and 6 using the generator we took. We also store the new generator graciously returned by randomR.

3. We then set the state to be the newGenerator using put, so that any further randomR in the do-block, or further on in a (>>=) chain, will use a different pseudo-random generator.

4. Finally, we inject the result back into the State StdGen monad using return.

We can finally use our monadic die. As before, the initial generator state itself is produced by the mkStdGen function.

```
GHCi> evalState rollDie (mkStdGen 0)
6
```

Why have we involved monads and built such an intricate framework only to do exactly what fst $ randomR (1,6) already does? Well, consider the following function:

```
rollDice :: State StdGen (Int, Int)
rollDice = liftM2 (,) rollDie rollDie
```

We obtain a function producing *two* pseudo-random numbers in a tuple. Note that these are in general different:

```
GHCi> evalState rollDice (mkStdGen 666)
 (6,1)
```

Under the hood, state is being passed through (>>=) from one rollDie computation to the other. Doing that was previously very clunky using randomR (1,6) alone because we had to pass state manually. Now, the monad instance is taking care of that for us. Assuming we know how to use the lifting functions, constructing intricate combinations of pseudo-random numbers (tuples, lists, whatever) has suddenly become much easier.

F.3 Pseudo-random values of different types

Until now, we have used only Int as type of the value produced by the pseudo-random generator. However, looking at the type of randomR shows we are not restricted to Int. It can generate values of any type in the Random class from System.Random. There already are instances for Int, Char, Integer, Bool, Double and Float, so you can immediately generate any of those.

Because State StdGen is "agnostic" in regard to the type of the pseudo-random value it produces, we can write a similarly "agnostic" function that provides a pseudo-random value of unspecified type (as long as it is an instance of Random):

```
getRandom :: Random a => State StdGen a
getRandom = state random
```

Compared to rollDie, this function does not specify the Int type in its signature and uses random instead of randomR; otherwise, it is just the same. getRandom can be used for any instance of Random:

```
GHCi> evalState getRandom (mkStdGen 0) :: Bool
True
GHCi> evalState getRandom (mkStdGen 0) :: Double
0.9872770354820595
GHCi> evalState getRandom (mkStdGen 0) :: Integer
2092838931
```

Indeed, it becomes quite easy to conjure all these at once:

```
someTypes :: State StdGen (Int, Float, Char)
someTypes = liftM3 (,,) getRandom getRandom getRandom

allTypes :: State StdGen (Int, Float, Char, Integer, Double, Bool, Int)
allTypes = liftM (,,,,,,) getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
```

For allTypes, since there is no liftM7 (the standard libraries only go to liftM5) we have used the ap function from Control.Monad instead. ap fits multiple computations into an application of a multiple argument function, which here is the (lifted) 7-element-tuple constructor. To understand ap further, look at its signature:

```
ap :: (Monad m) => m (a -> b) -> m a -> m b
```

Remember then that the type variable a in Haskell can be replaced by a function type as well as a regular value one, and compare to:

```
GHCi>:t liftM (,,,,,,) getRandom
liftM (,,,,,,) getRandom :: (Random a1) =>
                            State StdGen (b -> c -> d -> e -> f -> g
                                -> (a1, b, c, d, e, f, g))
```

The monad m obviously becomes State StdGen, while ap's first argument is a function

```
b -> c -> d -> e -> f -> g -> (a1, b, c, d, e, f, g)
```

Applying ap over and over (in this case 6 times), we finally get to the point where b is an actual value (in our case, a 7-element tuple), not another function. To sum it up, ap applies a function-in-a-monad to a monadic value (compare with liftM / fmap, which applies a function *not* in a monad to a monadic value).

So much for understanding the implementation. Function allTypes provides pseudo-random values for all default instances of Random; an additional Int is inserted at the end to prove that the generator is not the same, as the two Int s will be different.

```
GHCi> evalState allTypes (mkStdGen 0)
GHCi>(2092838931,9.953678e-4,'a',-868192881,0.4188001483955421,False,316817438)
```

Appendix G

The System.Random library

This library deals with the common task of pseudo-random number generation.[1]

The library makes it possible to generate repeatable results, by starting with a specified initial random number generator, or to get different results on each run by using the system-initialised generator or by supplying a seed from some other source.[2]

The library is split into two layers:

- A core *random number generator* provides a supply of bits. The class RandomGen provides a common interface to such generators. The library provides one instance of RandomGen, the abstract data type StdGen. Programmers may, of course, supply their own instances of RandomGen.
- The class Random provides a way to extract values of a particular type from a random number generator. For example, the Float instance of Random allows one to generate random values of type Float.

G.1 The *RandomGen* class

The class RandomGen provides a common interface to random number generators. The most common approach is using the StdGen type, presented in the next subsection.

```
class RandomGen g where

Minimal complete definition
  next, split

Methods
  next     :: g -> (Int , g)
  split    :: g -> (g , g)
  genRange :: g -> (Int , Int)
  genRange _ = (minBound , maxBound)

Instances
  RandomGen StdGen
```

[1] This implementation uses the Portable Combined Generator of L'Ecuyerfor 32-bit computers, transliterated by Lennart Augustsson. It has a period of roughly 2.30584e18.

[2] For example, the third decimal of the internal clock

The next operation returns an Int that is uniformly distributed in the range returned by genRange (including both end points), and a new generator.

The genRange operation yields the range of values returned by the generator. The default definition spans the full range of Int.

It is required that:

- If `(a,b) = genRange g`, then `a < b`.
- genRange always returns a pair of defined Int s.

The second condition ensures that genRange cannot examine its argument, and hence the value it returns can be determined only by the instance of RandomGen. That in turn allows an implementation to make a single call to genRange to establish a generator's range, without being concerned that the generator returned by (say) next might have a different range to the generator passed to next.

The split operation allows one to obtain two distinct random number generators. This is very useful in functional programs (for example, when passing a random number generator down to recursive calls), but very little work has been done on statistically robust implementations of split.

G.2 The type *StdGen* and the global number generator

G.2.1 *StdGen*

```
data StdGen

Instances
  Read StdGen
  Show StdGen
  RandmGen StdGen

mkStdGen :: Int -> StdGen
```

The StdGen instance of RandomGen has a genRange of at least 30 bits.

The result of repeatedly using next should be statistically robust.

The Show and Read instances of StdGen provide a primitive way to save the state of a random number generator. It is required that `read (show g) == g`.

In addition, reads may be used to map an arbitrary string (not necessarily one produced by show) onto a value of type StdGen. In general, the Read instance of StdGen has the following properties:

- It guarantees to succeed on any string.
- It guarantees to consume only a finite portion of the string.
- Different argument strings are likely to result in different results.

The function mkStdGen provides an alternative way of producing an initial generator, by mapping an Int into a generator. Again, distinct arguments should be likely to produce distinct generators.

G.2.2 The global number generator

There is a single, implicit, global random number generator of type StdGen, held in some global variable maintained by the IO monad. It is initialised automatically in some system-dependent fashion, for example, by using the time of day, or Linux's kernel random number generator. To get deterministic behaviour, use setStdGen.

```
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

Uses the supplied function to get a value from the current global random generator, and updates the global generator with the new generator returned by the function.

```
getStdGen :: IO StdGen
```

Gets the global random number generator.

```
setStdGen :: StdGen -> IO ()
```

Sets the global random number generator.

```
newStdGen :: IO StdGen
```

Applies split to the current global random generator, updates it with one of the results, and returns the other.

G.3 Random vaues of other types: the *Random* class

With a source of random number supply in hand, the Random class allows the programmer to extract random values of a variety of types.

```
class Random a where

Minimal complete definition
  randomR, random

Methods
  randomR   :: RandomGen g => (a, a) -> g -> (a, g)
  random    :: RandomGen g => g -> (a, g)
  randomRs  :: RandomGen g => (a, a) -> g -> [a]
  randoms   :: RandomGen g => g -> [a]
  randomRIO :: (a, a) -> IO a
  randomIO  :: IO a

Instances
  Random Bool
  Random Char
  Random Double
  Random Float
  Random Int
  ...
```

randomR takes a range *(lo,hi)* and a random number generator g, and returns a random value uniformly distributed in the closed interval *[lo,hi]*, together with a new generator. It is unspecified what happens if $lo > hi$. For continuous types there is no requirement that the values *lo* and *hi* are ever produced, but they may be, depending on the implementation and the interval.

random is the same as randomR, but using a default range determined by the type:

- For bounded types (instances of Bounded, such as Char), the range is normally the whole type.
- For fractional types, the range is normally the semi-closed interval *[0,1)*.
- For Integer, the range is (arbitrarily) the range of Int.

randomRs is a plural variant of randomR, producing an infinite list of random values instead of returning a new generator.

randoms is a plural variant of random, producing an infinite list of random values instead of returning a new generator.

randomRIO is a variant of randomR that uses the global random number generator.

randomIO is a variant of random that uses the global random number generator.

G.4 Other functions (that are not exported)

The following code is found in System.Random[3] but not exported.

G.4.1 The global number generator coding

First, some note found early in the module

```
-- The standard nhc98 implementation of Time.ClockTime does not match
-- the extended one expected in this module, so we lash-up a quick
-- replacement here.
#ifdef __NHC__
foreign import ccall "time.h time" readtime :: Ptr CTime -> IO CTime
getTime :: IO (Integer, Integer)
getTime = do CTime t <- readtime nullPtr;  return (toInteger t, 0)
#else
getTime :: IO (Integer, Integer)
getTime = do
  utc <- getCurrentTime
  let daytime = toRational $ utctDayTime utc
  return $ quotRem (numerator daytime) (denominator daytime)
#endif
```

The function getTime is used in:

```
mkStdRNG :: Integer -> IO StdGen
mkStdRNG o = do
    ct          <- getCPUTime
    (sec, psec) <- getTime
    return (createStdGen (sec * 12345 + psec + ct + o))
```

Which finally gives us

```
theStdGen :: IORef StdGen
theStdGen  = unsafePerformIO $ do
   rng <- mkStdRNG 0
   newIORef rng
```

[3] http://hackage.haskell.org/package/random-1.1/docs/src/System-Random.html

Appendix H

Appendix: Summary of functions

Everything has been taken from the Haskell documentation

H.1 Functor context

```
class Functor f where
  The Functor class is used for types that can be mapped over.

  Instances of Functor should satisfy the following laws:
    fmap id  ==  id
    fmap (f . g)  ==  fmap f . fmap g

  Minimal complete definition
    fmap

  Methods
    fmap :: (a -> b) -> f a -> f b
    (<$) :: a -> f b -> f a                              infixl 4

  Predefined functions  (in Data.Functor)
    (<$>) :: Functor f => (a -> b) -> f a -> f b         infixl 4
    (<$>) = fmap
    ($>) :: Functor f => f a -> b -> f b                 infixl 4
    ($>) = flip (<$)
    void :: Functor f => f a -> f ()
    void x = () <$ x
```

(< $ >) is an infix synonym for fmap.

The method (< $) replaces all locations in the input with the same value. The default definition is fmap . const, but this may be overriden with a more efficient version.

($ >) is a flipped version of (< $).

void value discards or ignores the result of evaluation, such as the return value of an System.IO.IO action.

H.2 Applicative context

The Control.Applicative module describes a structure intermediate between a functor and a monad (technically, a strong lax monoidal functor). Compared with monads, this interface lacks the full power of the binding operation (>>=), but

1. it has more instances.
2. it is sufficient for many uses, e.g. context-free parsing, or the Traversable class.
3. instances can perform analysis of computations before they are executed, and thus produce shared optimizations.

```
class Functor f => Applicative f where
  A functor with application, providing operations to
  embed pure expressions (pure), and
  sequence computations and combine their results: (<*>).

  Instances of Functor should satisfy the following laws:
    pure id <*> v = v                                    -- identity
    pure (.) <*> u <*> v <*> w = u <*> (v <*> w)         -- composition
    pure f <*> pure x = pure (f x)                       -- homomorphism
    u <*> pure y = pure ($ y) <*> u                      -- interchange

    As a consequence of these laws, the Functor instance for f will satisfy
      fmap f x = pure f <*> x

    If f is also a Monad, it should satisfy
      pure = return
      (<*>) = ap
      (which implies that pure and <*> satisfy the applicative functor laws).

  Minimal complete definition
    pure, (<*>)

  Methods
    pure :: a -> f a
    (<*>) :: f (a -> b) -> f a -> f b                    infixl 4
    (*>) :: f a -> f b -> f b                            infixl
    u *> v = pure (const id) <*> u <*> v
    (<*) :: f a -> f b -> f a                            infixl 4
    u <* v = pure const <*> u <*> v

  Utility functions
    (<**>) :: Applicative f => f a -> f (a -> b) -> f b   infixl 4
    (<**>) = liftA2 (flip ($))
    liftA  :: Applicative f => (a -> b) -> f a -> f b
    liftA f a = pure f <*> a
    liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
    liftA2 f a b = fmap f a <*> b
    liftA3 :: Applicative f => (a -> b -> c -> d) -> f a -> f b -> f c -> f d
    liftA3 f a b c = fmap f a <*> b <*> c
    when   :: (Applicative f) => Bool -> f () -> f ()
    when p s = if p then s else pure ()
```

`(* >)` and `(< *)` are already defined, but may be overriden with equivalent specialized implementations.

`(< ** >)` is a variant of `(< * >)` with the arguments reversed.

`liftA` lifts a function to actions. This function may be used as a value for `fmap` in a `Functor` instance.

`liftA2` lifts a binary function to actions.

`liftA3` lifts a ternary function to actions.

`when` is a conditional execution of `Applicative` expressions.

H.3 Monad context

```
class Applicative m => Monad m where

  Instances of Monad should satisfy the following laws:
    return a >>= k  =  k a
    m >>= return  =  m
    m >>= (x -> k x >>= h)  =  (m >>= k) >>= h

  Furthermore, the Monad and Applicative operations should relate as follows:
    pure = return
    (<*>) = ap
  The above laws imply:
    fmap f xs  =  xs >>= return . f
    (>>) = (*>)
    pure and (<*>) satisfy the applicative functor laws.

  Minimal complete definition
    (>>=)

  Methods
    (>>=)    :: m a -> (a -> m b) -> m b                    infixl 1
    (>>)     :: m a -> m b -> m b                           infixl 1
    m >> k   =   m >>= \_ -> k
    return   :: a -> m a
    return   =   pure
    fail     :: String -> m a
    fail s   =   error s

  Utility functions
    join     :: (Monad m) => m (m a) -> m a
    join x            =  x >>= id
    (=<<)    :: Monad m => (a -> m b) -> m a -> m b
    f =<< x    =    x >>= f
    sequence :: Monad m => [m a] -> m [a]
    sequence = mapM id
    mapM     :: Monad m => (a -> m b) -> [a] -> m [b]
    mapM f as = foldr k (return []) as
    where
      k a r = do { x <- f a; xs <- r; return (x:xs) }
    liftM    :: (Monad m) => (a1 -> r) -> m a1 -> m r
    liftM f m1         = do { x1 <- m1; return (f x1) }
    liftM2   :: (Monad m) => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
    liftM2 f m1 m2     = do { x1 <- m1; x2 <- m2; return (f x1 x2) }
    liftM3   :: (Monad m) => (a1 -> a2 -> a3 -> r) -> m a1 -> m a2 -> m a3 -> m r
    liftM3 f m1 m2 m3  = do { x1 <- m1; x2 <- m2; x3 <- m3; return (f x1 x2 x3) }
    ap       :: (Monad m) => m (a -> b) -> m a -> m b
    ap m1 m2           = do { x1 <- m1; x2 <- m2; return (x1 x2) }
```

H.4 Alternative context

```
class Applicative f => Alternative f where
  A monoid on applicative functors.

  If defined, some and many should be the least solutions of the equations:
    some v = (:) <$> v <*> many v
    many v = some v <|> pure []

  Minimal complete definition
    empty, (<|>)
  Methods
    empty :: f a
      The identity of <|>
    (<|>) :: f a -> f a -> f a                              infixl 3
      An associative binary operation
    some :: f a -> f [a]
      One or more
    many :: f a -> f [a]
      Zero or more.

  Utility functions
    optional :: Alternative f => f a -> f (Maybe a)
      One or none.
```

H.5 Module System.Random

```
class RandomGen g where

Minimal complete definition
  next, split

Methods
  next     :: g -> (Int , g)
  split    :: g -> (g , g)
  genRange :: g -> (Int , Int)
  genRange _ = (minBound , maxBound)

Instances
  RandomGen StdGen


data StdGen

Instances
  Read StdGen
  Show StdGen
  RandmGen StdGen

mkStdGen :: Int -> StdGen


class Random a where

Minimal complete definition
  randomR, random

Methods
  randomR   :: RandomGen g => (a, a) -> g -> (a, g)
  random    :: RandomGen g => g -> (a, g)
  randomRs  :: RandomGen g => (a, a) -> g -> [a]
  randoms   :: RandomGen g => g -> [a]
  randomRIO :: (a, a) -> IO a
  randomIO  :: IO a

Instances
  Random Bool
  Random Char
  Random Double
  Random Float
  Random Int
  ...
```

And the global random number generator

```
getStdRandom :: (StdGen -> (a, StdGen)) -> IO a
```

Uses the supplied function to get a value from the current global random generator, and updates the global generator with the new generator returned by the function.

```
getStdGen :: IO StdGen
```

Gets the global random number generator.

```
setStdGen :: StdGen -> IO ()
```

Sets the global random number generator.

```
newStdGen :: IO StdGen
```

Applies split to the current global random generator, updates it with one of the results, and returns the other.

H.6 Module Control.Monad

The Functor, Monad and MonadPlus classes, with some useful operations on monads. Some of the information is already exposed in the previous sections.

```
class Functor f where
  already seen

class Applicative m => Monad m where
  already seen

class (Alternative m, Monad m) => MonadPlus m where
  Monads that also support choice and failure.

  Instances of MonadPlus should satisfy the following laws:
    mzero `mplus` m = m
    m `mplus` mzero = m
    associativity of mplus
    mzero >>= f     = mzero
    m >> mzero      = mzero

  Minimal complete definition
    Nothing
  Methods
    mzero :: m a
    mplus :: m a -> m a -> m a

Basic Monad functions
  mapM      :: (Traversable t, Monad m) => (a -> m b) -> t a -> m (t b)
  mapM_     :: (Foldable t, Monad m)    => (a -> m b) -> t a -> m ()
  forM      :: (Traversable t, Monad m) => t a -> (a -> m b) -> m (t b)
  forM_     :: (Foldable t, Monad m)    => t a -> (a -> m b) -> m ()
  sequence  :: (Traversable t, Monad m) => t (m a) -> m (t a)
  sequence_ :: (Foldable t, Monad m)    => t (m a) -> m ()
  (=<<)     :: Monad m   => (a -> m b) -> m a -> m b                infixr 1
  (>=>)     :: Monad m   => (a -> m b) -> (b -> m c) -> a -> m c    infixr 1
  (<=<)     :: Monad m   => (b -> m c) -> (a -> m b) -> a -> m c    infixr 1
  forever   :: Monad m   => m a -> m b
  void      :: Functor f => f a -> f ()

Generalisations of list functions
  join      :: Monad m => m (m a) -> m a
  msum      :: (Foldable t, MonadPlus m) => t (m a) -> m a
  mfilter      :: MonadPlus m => (a -> Bool) -> m a -> m a
  filterM      :: Monad m => (a -> m Bool) -> [a] -> m [a]
  mapAndUnzipM :: Monad m => (a -> m (b, c)) -> [a] -> m ([b], [c])
  zipWithM     :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m [c]
  zipWithM_    :: Monad m => (a -> b -> m c) -> [a] -> [b] -> m ()
  foldM     :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m b
  foldM_    :: (Foldable t, Monad m) => (b -> a -> m b) -> b -> t a -> m ()
  replicateM   :: Monad m => Int -> m a -> m [a]
  replicateM_  :: Monad m => Int -> m a -> m ()
```

```
Conditional execution of monadic expressions
  guard     :: Alternative f => Bool -> f ()
  when      :: Applicative f => Bool -> f () -> f ()
  unless    :: Applicative f => Bool -> f () -> f ()

Monadic lifting operators
  liftM  :: Monad m => (a1 -> r) -> m a1 -> m r
  liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r
  liftM3 :: Monad m => (a1 -> a2 -> a3 -> r) -> m a1 -> m a2 -> m a3 -> m r
  liftM4 :: Monad m => (a1 -> a2 -> a3 -> a4 -> r) -> m a1 -> m a2 -> m a3 -> m a4 -> m r
  liftM5 :: Monad m => (a1 -> a2 -> a3 -> a4 -> a5 -> r) -> m a1 -> m a2 -> m a3 -> m a4 -> m a5 -> m
  ap :: Monad m => m (a -> b) -> m a -> m b

Strict monadic functions
  (<$!>)    :: Monad m => (a -> b) -> m a -> m b                    infixl 4
```

Naming conventions The functions in this library use the following naming conventions:

- A postfix 'M' always stands for a function in the Kleisli category: The monad type constructor m is added to function results (modulo currying) and nowhere else. So, for example,

  ```
  filter  ::              (a ->   Bool) -> [a] ->   [a]
  filterM :: (Monad m) => (a -> m Bool) -> [a] -> m [a]
  ```

- A postfix '_' changes the result type from m a to m (). Thus, for example:

  ```
  sequence  :: Monad m => [m a] -> m [a]
  sequence_ :: Monad m => [m a] -> m ()
  ```

- A prefix 'm' generalizes an existing function to a monadic form. Thus, for example:

  ```
  sum  :: Num a       => [a]   -> a
  msum :: MonadPlus m => [m a] -> m a
  ```

Appendix I

Exercises

These exercises have been taken from several different sources, and are not necessarily sorted by any criteria

I.1 Basic *Functor* and *Applicative* exercises

1. Define instances of `Functor` for the following types:
   - A rose tree, defined as: `data Tree a = Node a [Tree a]`
   - `Either e` for a fixed `e`.
   - The function type `((→) r)`. In this case, `f a` will be $(r \to a)$
2. Check that the Applicative laws hold for the instance for `Maybe` presented in the main body:

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

3. Write `Applicative` instances for
   - `Either e`, for a fixed `e`
   - `((→) r)`, for a fixed `r`

I.2 Advanced *Monad* and *Applicative* exercises

1. What is the expected behavior of `sequence` for the `Maybe` monad?

2. Write a definition of `(<*>)` using `(>>=)` and `fmap`. Do not use do-notation.

3. Implement

```
liftA5 :: Applicative f => (a -> b -> c -> d -> e -> k)
-> f a -> f b -> f c -> f d -> f e -> f k
```

4. For the list functor, implement from scratch (that is, without using anything from `Applicative` or `Monad` directly) both `(<*>)` and its version with the "wrong" sequencing of effects,

```
(<|*|>) :: Applicative f => f (a -> b) -> f a -> f b
```

5. Rewrite the definition of commutativity for a `Monad`;

```
liftA2 f u v = liftA2 (flip f) v u -- Commutativity
   -- Or, equivalently,
f <$> u <*> v = flip f <$> v <*> u
```

   using do-notation instead of `ap` or `liftM2`.

6. Are the following applicative functors commutative?

   - `ZipList`
   - `((→) r)`
   - `State s` (Use the `newtype` definition from the `State` appendix).

   Hint: You may find the answer to exercise 5 (in this section) useful.

7. What is the result of `[2,7,8] *> [3,9]`? Try to guess without writing.)

8. Implement `(<**>)` in terms of other `Applicative` functions.

9. As we have just seen, some functors allow two legal implementations of `(<*>)` which are only different in the sequencing of effects. Why there is not an analogous issue involving `(>>=)`?

The next few exercises concern the following tree data structure:[1]

```
data AT a = L a | B (AT a) (AT a)
```

10. Write `Functor`, `Applicative` and `Monad` instances for `AT`. Do not use shortcuts such as `pure = return`. The `Applicative` and `Monad` instances should match; in particular, `(>>=)` should be equivalent to `ap`, which follows from the `Monad` instance.

[1] In case you are wondering, "AT" stands for "apple tree".

11. Implement the following functions, using either the Applicative instance, the Monad one or neither of them, if neither is enough to provide a solution. Between Applicative and Monad, choose the *least* powerful one which is still good enough for the task. Justify your choice for each case in a few words.

    - `fructify :: AT a -> AT a`, which grows the tree by replacing each leaf L with a branch B containing two copies of the leaf.
    - `prune :: a -> (a -> Bool) -> AT a -> AT a`, with prune z p t replacing a branch of t with a leaf carrying the default value z whenever any of the leaves directly on it satisfies the test p.
    - `reproduce :: (a -> b) -> (a -> b) -> AT a -> AT b`, with reproduce f g t resulting in a new tree with two modified copies of t on the root branch. The left copy is obtained by applying f to the values in t, and the same goes for g and the right copy.

12. There is another legal instance of Applicative for AT (the reversed sequencing version of the original one doesn't count). Write it.

    Hint: this other instance can be used to implement

    ```
    sagittalMap :: (a -> b) -> (a -> b) -> AT a -> AT b
    ```

    which, when given a branch, maps one function over the left child tree and the other over the right child tree.

13. Write implementations for unit and $(*\&*)$ in terms of pure and $(<*>)$, and vice-versa.

14. Formulate the law of commutative applicative functors,

    ```
    liftA2 f u v = liftA2 (flip f) v u -- Commutativity
       -- Or, equivalently,
    f <$> u <*> v = flip f <$> v <*> u
    ```

    in terms of the Monoidal methods.

15. Write from scratch Monoidal instances for:

    - ZipList
    - $((\rightarrow)\ r)$

I.3 *State* exercises

1. Implement a function `rollNDiceIO :: Int -> IO [Int]` that, given an integer (a number of die rolls), returns a list of that number of pseudo-random integers between 1 and 6.

2. Implement a function `rollDice :: StdGen -> ((Int, Int), StdGen)` that, given a generator, returns a tuple with our random numbers as first element and the last generator as the second.

3. Similarly to what was done for rollNDiceIO, implement a function

   ```
   rollNDice :: Int -> State StdGen [Int]
   ```

   that, given an integer, returns a list with that number of pseudo-random integers between 1 and 6.

4. Write an instance of Functor for State s. Your final answer should not use anything that mentions Monad in its type (that is, return, (>>=), etc.). Then, explain in a few words what the fmap you wrote does.

   (Hint: If you get stuck, have another look at the comments about liftM in the main body.)

5. Besides put and get, there are also

   ```
   modify :: (s -> s) -> State s ()
   ```

   which modifies the current state using a function, and

   ```
   gets :: (s -> a) -> State s a
   ```

   which produces a modified copy of the state while leaving the state itself unchanged. Write implementations for them.

6. If you are not convinced that State is worth using, try to implement a function equivalent to evalState allTypes without making use of monads, i.e. with an approach similar to clumsyRollDice above.

I.4 *MonadPlus* exercises

1. Prove the MonadPlus laws for Maybe and the list monad.
2. We could augment our above parser to involve a parser for any character:

```
-- | Consume a given character in the input, and return the character we
--   just consumed, paired with rest of the string. We use a do-block  so that
--   if the pattern match fails at any point, fail of the Maybe monad (i.e.
--   Nothing) is returned.
char :: Char -> String -> Maybe (Char, String)
char c s = do
  let (c':s') = s
  if c == c' then Just (c, s') else Nothing
```

   It would then be possible to write a `hexChar` function which parses any valid hexadecimal character (0-9 or a-f). Try writing this function

   (hint: `map digit [0..9] :: [String -> Maybe Int]`).

I.5 Monad transformers exercises'

1. Why is it that the `lift` function has to be defined separately for each monad, where as `liftM` can be defined in a universal way?
2. `Identity` is a trivial functor, defined in `Data.Functor.Identity` as:

```
newtype Identity a = Identity { runIdentity :: a }
```

   It has the following Monad instance:

```
instance Monad Identity where
    return a = Identity a
    m >>= k  = k (runIdentity m)
```

   Implement a monad transformer `IdentityT`, analogous to `Identity` but wrapping values of type `m a` rather than `a`. Write at least its `Monad` and `MonadTrans` instances.
3. Implement `state :: MonadState s m => (s -> (a, s)) -> m a` in terms of `get` and `put`.
4. Are `MaybeT (State s)` and `StateT s Maybe` equivalent? (Hint: one approach is comparing what the `run...T` unwrappers produce in each case.)

I.6 Hask category exercises

1. As was mentioned, any partial order $(P, \leq$ is a category with objects the elements of P and a morphism between elements a and b iff $a \leq b$. Which of the above laws guarantees the transitivity of $\leq$?

2. Check the functor laws for the Maybe and list functors.

3. Verify that the list and Maybe monads do in fact obey the first monad law,

   ```
   join . fmap join = join . join
   ```

   with some examples to see precisely how the layer flattening works.

4. Prove the second monad law, join . fmap return = join . return = id for the Maybe monad.

5. Convince yourself that the 3rd and 4th laws should hold true for any monad by exploring what they mean, in a similar style to how the first and second laws were explored.

6. In fact, the two versions of the laws we gave:

   ```
   -- Categorical:
   join . fmap join = join . join
   join . fmap return = join . return = id
   return . f = fmap f . return
   join . fmap (fmap f) = fmap f . join

   -- Functional:
   m >>= return = m
   return m >>= f = f m
   (m >>= f) >>= g = m >>= (\x -> f x >>= g)
   ```

   are entirely equivalent. We showed that we can recover the functional laws from the categorical ones. Go the other way; show that starting from the functional laws, the categorical laws hold. It may be useful to remember the following definitions:

   ```
   join m = m >>= id
   fmap f m = m >>= return . f
   ```

Appendix J

My solutions for the exercises

Solutions will be given in packs, one for each section

J.1 Basic *Functor* and *Applicative* solutions

1. Define instances of `Functor` for the following types:
   - A rose tree, defined as: `data Tree a = Node a [Tree a]`
   - `Either e` for a fixed `e`.
   - The function type `((→ ) r)`. In this case, `f a` will be $(r \to a)$
2. Check that the Applicative laws hold for the instance for `Maybe` presented in the main body:

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

3. Write `Applicative` instances for
   - `Either e`, for a fixed `e`
   - `((→ ) r)`, for a fixed `r`

```
{-# LANGUAGE TypeSynonymInstances #-}

import Control.Applicative



-------------------------------------------------------------------------
--      FIRST EXERCISE OF BASIC FUNCTOR AND APPLICATIVE SECTION       --
-------------------------------------------------------------------------

data Tree a = Node a [Tree a]
 deriving (Eq, Show)


instance Functor Tree where
  fmap f (Node n ts)  = Node (f n) ( map (fmap f) ts )




mapTree :: (a -> b) -> Tree a -> Tree b
mapTree f (Node n [])     = Node (f n) []
mapTree f (Node n (t:ts)) = Node (f n) list
  where
    list  = (mapTree f t) : mapp (mapTree f) ts
    mapp _ [] = []
    mapp g (x:xs) = (g x):(mapp g xs)

-- NO HAY FORMA DE ESCRIBIR LOS DOS BUCLES RECURSIVOS EN UNA SOLA LINEA?

--  fmap f (Node n [])     = Node (f n) []
--  fmap f (Node n (t:ts)) = Node (f n) ????


n1 = Node 1 []
n2 = Node 2 []
n3 = Node 3 []
n4 = Node 4 []
n5 = Node 5 []
n6 = Node 6 []
n7 = Node 7 []

t1 = Node 10 [n1,n2,n3]
t2 = Node 11 [n4,n5]
t3 = Node 12 [n6]
t4 = Node 13 [n7]

t10 = Node 20 [t1,t2,t3,t4]


ej1 :: IO (Tree Int)
```

```
ej1 = do
  let t10' = fmap (100+) t10
  putStrLn (show t10)
  putStrLn (show t10')
  return t10'

-- Checking the correctness of the solution:
-- https://hackage.haskell.org/package/containers-0.5.7.1/docs/src/Data.Tree.html#line-74


-----------------------------------------------------------------------


data Either' a b = Left' a | Right' b

instance Functor (Either' a) where
  fmap _ (Left' x)  = Left' x
  fmap f (Right' y) = Right' (f y)


-- Checking the laws:

-- fmap id (Left x) = Left x == Left x                        OK
-- fmap id (Right y) = Right (id y) = Right y == Right y      OK

-- fmap (f.g) (Left x) = Left x  == Left x = fmap f (Left x) = fmap f (fmap g (Left x))  OK
-- fmap (f.g) (Right y) = Right ((f.g) y)
--   ==          ---------------------------------> because   (f.g) y  ==  f (g y)        OK
--  Right (f (g y)) = fmap f (Right (g y)) = fmap f (fmap g (Right y))


-----------------------------------------------------------------------

type FuncsWithFixedDomain r = (->) r

-- *Main> :k FuncsWithFixedDomain
-- FuncsWithFixedDomain :: * -> * -> *
-- *Main> :k FuncsWithFixedDomain Int
-- FuncsWithFixedDomain Int :: * -> *


-- instance Functor (FuncsWithFixedDomain r) where  ==>  ERROR

-- [1 of 1] Compiling Main             ( BasicFunctorAndApp.hs, interpreted )

-- BasicFunctorAndApp.hs:88:10:
--     Duplicate instance declarations:
--       instance Functor (FuncsWithFixedDomain r)
--         -- Defined at BasicFunctorAndApp.hs:88:10
--       instance Functor ((->) r) -- Defined in ‘GHC.Base’
-- Failed, modules loaded: none.
```

```
-- http://hackage.haskell.org/package/base-4.8.2.0/docs/src/GHC.Base.html#line-612


mapF :: (a -> b) -> (r -> a) -> (r -> b)
mapF alpha f =  alpha . f


-- *Main> (mapF (2+) length ) [1,2,3]
-- 5
-- (0.00 secs, 0 bytes)


-- Checking the laws:

-- mapF id f = id . f  ==  f                                                    OK

-- mapF (alpha.beta) f = (alpha.beta).f
--       ==          --------------------> because   (alpha.beta).f  == alpha.(beta.f)  OK
--  alpha.(beta.f) = mapF alpha (beta.f) = mapF alpha (mapF beta f)




-------------------------------------------------------------------------
--     SECOND EXERCISE OF BASIC FUNCTOR AND APPLICATIVE SECTION       --
-------------------------------------------------------------------------

data Maybe' a = Just' a | Nothing'

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


instance Applicative Maybe' where
  pure = Just'
  (Just' f) <*> (Just' x) = Just' (f x)
  _        <*> _          = Nothing'



-- Checking the laws:

-- Identity:
--   (pure id) <*> Nothing   ==  Nothing
--   (pure id) <*> (Just x) = (Just id) <*> (Just x) = Just (id x)  ==  Just x
-- Homomorphism
--   (pure f) <*> (pure x) = (Just f) <*> (Just x) == Just (f x) == pure (f x)
-- Interchange
--   Nothing <*> (pure x) = Nothing <*> (Just x) == Nothing ==  (Just ($ y)) <*> Nothing
--   (Just f) <*> (pure x) = Just (f x)  ==  Just (f $ x) = (Just ($ x)) <*> (Just f)
```

```
-- Composition
--   pure (.) <*> Just u <*> Just v <*> Just a = Just (u.v) <*> Just a  ==  Just ((u.v) a)
--     ==
--    Just u <*> Just (v a) = Just u <*> (Just v <*> Just a)



------------------------------------------------------------------------
--      THIRD EXERCISE OF BASIC FUNCTOR AND APPLICATIVE SECTION       --
------------------------------------------------------------------------


instance Applicative (Either' e) where
  pure x = Right' x
  (Right' f) <*> (Right' x) = Right' (f x)
  _          <*> (Left' x)  = Left' x


-- Checking the laws:
--   analogous to the Maybe instance


------------------------------------------------------------------------



-- instance Applicative (FuncsWithFixedDomain r) where
--   pure x = const x
--   (alpha <*> f) rVal = alpha r (f rVal)


-- http://hackage.haskell.org/package/base-4.8.2.0/docs/src/GHC.Base.html#line-616
```

J.2 Advanced *Monad* and *Applicative* solutions

1. What is the expected behavior of `sequence` for the `Maybe` monad?

2. Write a definition of `(<*>)` using `(>>=)` and `fmap`. Do not use do-notation.

3. Implement

```
liftA5 :: Applicative f => (a -> b -> c -> d -> e -> k)
-> f a -> f b -> f c -> f d -> f e -> f k
```

4. For the list functor, implement from scratch (that is, without using anything from `Applicative` or `Monad` directly) both `(<*>)` and its version with the "wrong" sequencing of effects,

```
(<|*|>) :: Applicative f => f (a -> b) -> f a -> f b
```

5. Rewrite the definition of commutativity for a `Monad`;

```
liftA2 f u v = liftA2 (flip f) v u -- Commutativity
   -- Or, equivalently,
f <$> u <*> v = flip f <$> v <*> u
```

   using do-notation instead of `ap` or `liftM2`.

6. Are the following applicative functors commutative?

   - `ZipList`
   - `((→) r)`
   - `State s` (Use the `newtype` definition from the `State` appendix).

   Hint: You may find the answer to exercise 5 (in this section) useful.

7. What is the result of `[2,7,8] *> [3,9]`? Try to guess without writing.)

8. Implement `(<**>)` in terms of other `Applicative` functions.

9. As we have just seen, some functors allow two legal implementations of `(<*>)` which are only different in the sequencing of effects. Why there is not an analogous issue involving `(>>=)`?

The next few exercises concern the following tree data structure:[1]

```
data AT a = L a | B (AT a) (AT a)
```

10. Write `Functor`, `Applicative` and `Monad` instances for `AT`. Do not use shortcuts such as `pure = return`. The `Applicative` and `Monad` instances should match; in particular, `(>>=)` should be equivalent to `ap`, which follows from the `Monad` instance.

[1] In case you are wondering, "AT" stands for "apple tree".

11. Implement the following functions, using either the Applicative instance, the Monad one or neither of them, if neither is enough to provide a solution. Between Applicative and Monad, choose the *least* powerful one which is still good enough for the task. Justify your choice for each case in a few words.

    - `fructify :: AT a -> AT a`, which grows the tree by replacing each leaf L with a branch B containing two copies of the leaf.
    - `prune :: a -> (a -> Bool) -> AT a -> AT a`, with prune z p t replacing a branch of t with a leaf carrying the default value z whenever any of the leaves directly on it satisfies the test p.
    - `reproduce :: (a -> b) -> (a -> b) -> AT a -> AT b`, with reproduce f g t resulting in a new tree with two modified copies of t on the root branch. The left copy is obtained by applying f to the values in t, and the same goes for g and the right copy.

12. There is another legal instance of Applicative for AT (the reversed sequencing version of the original one doesn't count). Write it.

    Hint: this other instance can be used to implement

    ```
    sagittalMap :: (a -> b) -> (a -> b) -> AT a -> AT b
    ```

    which, when given a branch, maps one function over the left child tree and the other over the right child tree.

13. Write implementations for unit and $(*\&*)$ in terms of pure and $(<*>)$, and vice-versa.

14. Formulate the law of commutative applicative functors,

    ```
    liftA2 f u v = liftA2 (flip f) v u -- Commutativity
       -- Or, equivalently,
    f <$> u <*> v = flip f <$> v <*> u
    ```

    in terms of the Monoidal methods.

15. Write from scratch Monoidal instances for:

    - ZipList
    - $((\rightarrow)\ r)$

```
import Control.Applicative
import Control.Monad


-----------------------------------------------------------------------
--   FIRST EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION     --
-----------------------------------------------------------------------


-- sequence :: Monad m => [m a] -> m [a]

-- Looking at the type signature, the expected behavior could be:
-- sequence [Just 5, Just 6, Just 7, Nothing] == Just [5,6,7]
-- sequence [Nothing, Nothing, Nothing] == Nothing

-- However, taking a closer look,
-- sequence :: Monad m => [m a] -> m [a]
-- sequence = mapM id
--
-- mapM :: Monad m => (a -> m b) -> [a] -> m [b]
-- mapM f as = foldr k (return []) as
--   where
--     k a r = do { x <- f a; xs <- r; return (x:xs) }

-- The base case of the foldr call inside mapM is (return []), so it should be:
-- sequence [Nothing, Nothing, Nothing] == Just []


-- Finally, the real behavior is:
-- *Main> sequence [Nothing, Nothing, Nothing]
-- Nothing
-- (0.00 secs, 0 bytes)
-- *Main> sequence [Nothing, Nothing, Nothing, Just 5, Just 7]
-- Nothing
-- (0.00 secs, 0 bytes)
-- *Main> sequence [Just 5, Just 7]
-- Just [5,7]
-- (0.00 secs, 0 bytes)

-- To understand this:
--   in (MapM id) ,  we have  id :: m b -> m b
--   essentially, as soon as we get a Nothing in the do block, we have  Nothing >>= ...
--   which always ends up being Nothing




-----------------------------------------------------------------------
--   SECOND EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
-----------------------------------------------------------------------
```

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


myApply :: (Functor m, Monad m) => m (a -> b) -> m a -> m b
myApply phi m =   phi >>= (\f -> fmap f m)


-- *Main> myApply (Just (2+)) (Just 3)
-- Just 5

-- *Main> myApply [(1+), (2*), id] [10,20,30]
-- [11,21,31,20,40,60,10,20,30]




------------------------------------------------------------------------
--   THIRD EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION     --
------------------------------------------------------------------------


liftA5 :: Applicative f => (a -> b -> c -> d -> e -> k)
  -> f a -> f b -> f c -> f d -> f e -> f k

liftA5 func a b c d e = fmap func a <*> b <*> c <*> d <*> e
--                      (          )     )     )     )     )




------------------------------------------------------------------------
--   FOURTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
------------------------------------------------------------------------


myListApply :: [ a -> b ] -> [a] -> [b]
myListApply fs as  =  concatMap (\f -> map f as) fs


-- *Main> myListApply [(1+), (2*), id] [10,20,30]
-- [11,21,31,20,40,60,10,20,30]


fs <|*|> xs = concatMap (\x -> fmap ($ x) fs) xs
```

```
-------------------------------------------------------------------------
--   FIFTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION     --
-------------------------------------------------------------------------


-- Commutativity:
--   liftA2 f u v  ==  liftA2 (flip f) v u
--  or equivalently
--   f <$> u <*> v  ==  flip f <$> v <*> u

-- Or equivalently:
--   do {x <- u; y <- v; return (f x y)}
--     ==
--   do {y <- v; x <- u; return ((flip f) y x)}
--     ==
--   do {y <- v; x <- u; return (f x y)}


-- *Main> let aux f u v = do {x <- u; y <- v; return (f x y)}
-- (0.02 secs, 0 bytes)
-- *Main> :t aux
-- aux :: Monad m => (t -> t1 -> b) -> m t -> m t1 -> m b
-- *Main> :t liftA2
-- liftA2 :: Applicative f => (a -> b -> c) -> f a -> f b -> f c
-- *Main> :t liftM2
-- liftM2 :: Monad m => (a1 -> a2 -> r) -> m a1 -> m a2 -> m r




-------------------------------------------------------------------------
--   SIXTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION     --
-------------------------------------------------------------------------


-- Is  ZipList  (found in Control.Applicative) a commutative applicative functor?
--   newtype ZipList a = ZipList { getZipList :: [a] }
--   instance Applicative ZipList where
--     (ZipList fs) <*> (ZipList xs) = ZipList (zipWith ($) fs xs)
--     pure x                        = ZipList (repeat x)

-- (f <$> (ZipList l1)) <*> (ZipList l2)
--   ==  (ZipList (map f l1)) <*> ZipList l2
--   ==  ZipList (zipWith ($) (map f l1) l2)
--   ==  ZipList (zipWith ($) (map (flip f) l2) l1)
--   ==  (ZipList (map (flip f) l2)) <*> ZipList l1
--   == ( (flip f) <$> (ZipList l2) ) <*> (ZipList l1)


-- Is  ((->) r)  a commutative applicative functor?
-- instance Applicative ((->) a) where
--     pure = const
```

```
--     (<*>) f g x = f x (g x)

-- We have
--   f :: a -> b -> c
--   g :: r -> a
--   h :: r -> b
-- So
-- (f <$> g <*> h) x
--   ==  ((f.g) <*> h) x
--   ==  (f.g) x (h x)
--   ==  ((flip f).h) x (g x)
--   ==  ( ((flip f).h) <*>  g ) x
--   ==  ( ((flip f) <$> h) <*> g ) x


-- Is  State s  a commutative applicative functor?
--   No, because the order of computations affects the result

-- https://en.wikibooks.org/wiki/Haskell/Solutions/Applicative_functors



-------------------------------------------------------------------------
--  SEVENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
-------------------------------------------------------------------------


-- [2,7,8] *> [3,9] ?

-- *Main> :t (*>)
-- (*>) :: Applicative f => f a -> f b -> f b
-- *Main> :t (<*>)
-- (<*>) :: Applicative f => f (a -> b) -> f a -> f b
-- *Main> :t ($>)

-- <interactive>:1:1:
--     Not in scope: ‘$>’
--     Perhaps you meant one of these:
--       ‘>>’ (imported from Control.Monad), ‘$!’ (imported from Prelude),
--       ‘>’ (imported from Prelude)
-- *Main> :t (>>)
-- (>>) :: Monad m => m a -> m b -> m b

-- So it could be:
--   [2,7,8] *> [3,9]  ==  [3,9]
-- However:
--   *Main> [2,7,8] *> [3,9]
--   [3,9,3,9,3,9]
--   (0.02 secs, 0 bytes)
--   *Main>  [2,7,8] >> [3,9]
--   [3,9,3,9,3,9]
--   (0.00 secs, 0 bytes)
```

```
-- Because:
--   (*>) u v = pure (const id) <*> u <*> v
-- *Main> :t const
-- const :: a -> b -> a
-- *Main> :t id
-- id :: a -> a
-- *Main> :t (const id)
-- (const id) :: b -> a -> a
-- So:
--   [2,7,8] *> [3,9]
--   ==  [ (const id) ] <*> [2,7,8] <*> [3,9]
--   ==  [const id 2 , const id 7 , const id 8 ] <*> [3,9]
--   ==  [const id 2 3 , const id 2 9 , ... , const id 8 9]
--   ==  [3,9,3,9,3,9]

-- *Main> [2,7,8] <* [3,9]
-- [2,2,7,7,8,8]
-- (0.00 secs, 0 bytes)
-- *Main> [3,9] *> [2,7,8]
-- [2,7,8,2,7,8]
-- (0.00 secs, 0 bytes)

-- In conclusion, for lists:
--   l1 (*>) l2  repeats l2 as many times as length l1
--   l1 (<*) l2  repeats each element of l1 as many times as length l2




-------------------------------------------------------------------------
--   EIGHTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
-------------------------------------------------------------------------


-- (<**>) :: Applicative f => f a -> f (a -> b) -> f b
--   Recalling:  (<**>)  is NOT  flip (<*>)


myInvertedApply :: Applicative f => f a -> f (a -> b) -> f b
myInvertedApply = liftA2 (flip ($))


-- (searched hoogle because i was lazy)

-- Recalling:
-- Commutativity:
--   liftA2 f u v  ==  liftA2 (flip f) v u




-------------------------------------------------------------------------
```

```
--    NINTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
-------------------------------------------------------------------------


-- from the Haskell documentation:
--   (=<<) :: Monad m => (a -> m b) -> m a -> m b
--   f =<< x         = x >>= f

-- *Main> :t (*>)
-- (*>) :: Applicative f => f a -> f b -> f b
-- *Main> :t (<*)
-- (<*) :: Applicative f => f a -> f b -> f a
-- *Main> :t (<*>)
-- (<*>) :: Applicative f => f (a -> b) -> f a -> f b
-- *Main> :t (<**>)
-- (<**>) :: Applicative f => f a -> f (a -> b) -> f b

-- It does not happen because the order of computations is fixed:
--   First, the monadic action x
--   Second, retrieve the hidden value in x and apply f
-- It doesn't make sense to think that we can evaluate any monadic action in f
-- first, because we need a value  a  from  m a  before




-------------------------------------------------------------------------
--    TENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION    --
-------------------------------------------------------------------------


data AT a = L a | B (AT a) (AT a)
  deriving (Show)

at1 :: AT Int
at1 = B (L 5) (B (L 2) (L 100))

at2 :: AT (Int -> Char)
at2 = let f=toEnum.(100+) in B (B (L f) (L f)) (B (L f) (B (L f) (L f)))

at3 :: AT (Int -> Int)
at3 = B (L (2*)) ( B (L (1+)) (L (3*)) )


-- Working with finite trees


mapAT :: (a -> b) -> AT a -> AT b
mapAT f (L a) = L (f a)
mapAT f (B l r) = B (mapAT f l) (mapAT f r)
```

```
instance Functor AT where
  fmap = mapAT

-- fmap id (L x) = L (id x)  ==  L x
-- fmap id (B l r) = B (fmap id l) (fmap id r)  ==  B l r
--   should be proved by induction (over the depth of the tree?)

-- fmap (f.g) (L x) = L ( f (g x) )
--   ==
-- fmap f (L (g x)) = (fmap f).(fmap g) (L x)



-- For each leaf in the tree of functions, substitute it with the tree
-- resulting from applying that function to the tree of values:
applyAT :: AT (a -> b) -> AT a -> AT b
applyAT at_fs at_xs = case at_fs of
  (L f)   -> mapAT f at_xs
  (B l r) -> B (applyAT l at_xs) (applyAT r at_xs)


instance Applicative AT where
  pure x = L x
  fs <*> xs = applyAT fs xs

-- pure id <*> at
--   = (L id) <*> at
--   = mapAT id at
--   = at           (checked in the Functor part)
--   == at          (as we wanted)

-- (pure f) <*> (pure x)
--   = (L f) <*> (L x)
--   = mapAT f (L x)
--   = L (f x)
--   == pure (f x)  (as we wanted)

-- pure (.) <*> gs <*> fs <*> xs
--   ==   ????
-- gs <*> (fs <*> xs)
--
-- pure (.) <*> gs <*> fs <*> xs
--   = (L (.)) <*> gs <*> fs <*> xs
--   = (mapAT (.) gs) <*> fs <*> xs
--   = case gs of
--     (L g) =>
--       = L (g.) <*> fs <*> xs
--         = case fs of
--           (L f) =>
--             = L (g.f) <*> xs
--             = mapAT (g.f) xs
--             == mapAT g (mapAT f xs)
```

```
--               = gs <*> (fs <*> xs)
-- and the other cases are 'trivial':
--   applyAT (B l r) xs = B (applyAT l xs) (applyAT r xs)
-- so everything is decided in the leaves

-- fs <*> (pure y)
--   = fs <*> (L y)
--   == substitute each leaf (L f) in fs by [L (f y)]
--   = mapAT ($ y) fs
--   = L ($ y) <*> fs
--   = pure ($ y) <*> fs



bindAT :: AT a -> (a -> AT b) -> AT b
bindAT (L x) f   = f x
bindAT (B l r) f = B (bindAT l f) (bindAT r f)


instance Monad AT where
  return x = (L x)
  (>>=)    = bindAT


-- return a >>= f
--   = (L a) >>= f
--   = f a         (by the def of >>=)
--   == f a        (as we wanted)

-- at >>= return
--   = case at of
--     (L a) =>
--       = (L a) >>= return
--       = return a
--       == (L a)  (as we wanted)
--     (B l r) =>
--       = B (l>>=return) (r>>=return)
--         (reduced to the leaf case)

-- at >>= (\x -> f x >>= g)
--   ==   ????
-- (at >>= f) >>= g
--
-- case at of (L a):
--   = (L a) >>= (\x -> f x >>= g)
--   = (\x -> f x >>= g) a
--   = (f a) >>= g
--   == (f a) >>= g
--   = ((L a) >>= f) >>= g
--   = (at >>= f) >>= g
```

```
-- Is (<*>) equal to ap?
--   ap :: (Monad m) => m (a -> b) -> m a -> m b
--   ap m1 m2 = do {x1 <- m1; x2 <- m2; return (x1 x2)}
-- Translating the do block first
--   ap m1 m2  =  m1 >>= ( \x1 -> m2 >>= \x2 -> return (x1 x2) )
-- So:
--   ap at_fs at_xs
--     = at_fs >>= ( \x1 -> at_xs >>= \x2 -> return (x1 x2) )
--     case  at_fs of (L f)
--       = ( \x1 -> at_xs >>= \x2 -> return (x1 x2) ) f
--       = at_xs >>= (\x2 -> return (f x2))
--         case at_xs of (L x)
--           = (\x2 -> return (f x2)) x
--           = return (f x)
--           = L (f x)
--           == (L f) <*> (L x)




-------------------------------------------------------------------------
--  ELEVENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION   --
-------------------------------------------------------------------------


-- Grow the tree by replacing each leaf with a branch
-- containing two copies of the leaf.

fructify1 :: AT a -> AT a
fructify1 at =  at >>= (\a -> B (L a) (L a))

fructify1' :: AT a -> AT a
fructify1' at =  join ( fmap (\a -> B (L a) (L a)) at )

fructify2 :: AT a -> AT a
fructify2 at =  (fmap g at) <*> (B (L 0) (L 0))
  where g a = \_ -> a

fructify2' :: AT a -> AT a
fructify2' at =  (fmap const at) <*> (B (L "hi") (L "bye"))

fructify2'' :: AT a -> AT a
fructify2'' at = at <* (B (L True) (L False))

wrongFructify at = at *> (B (L 0) (L 0))



-- Replace a branch of a tree with a leaf carrying the default
-- value z whenever any of the leaves directly on it satisfies the test p

prune :: a -> (a -> Bool) -> AT a -> AT a
```

```
-- prune z p t = boolT ???
--   where boolT = fmap p t
-- None of the instances above allows to cut parts of the tree because
-- each of them only grows the tree(s).
prune z p (L x)              = if p x then (L z) else (L x)
prune z p t@(B (L x) (L y) ) = if (p x) || (p y) then (L z) else t
prune z p (B (L x) t)        = if p x then (L z) else B (L x) (prune z p t)
prune z p (B t (L y))        = if p y then (L z) else B (prune z p t) (L y)
prune z p (B l r)            = B (prune z p l) (prune z p r)


-- *Main> at1
-- B (L 5) (B (L 2) (L 100))
-- (0.00 secs, 12846520 bytes)
-- *Main> prune (-1) (<3) at1
-- B (L 5) (L (-1))
-- (0.00 secs, 0 bytes)
-- *Main> prune (-1) (<10) at1
-- L (-1)
-- (0.00 secs, 0 bytes)



-- Duplicate a tree applying two different functions

reproduce :: (a -> b) -> (a -> b) -> AT a -> AT b
reproduce f g t = B (fmap f t) (fmap g t)

reproduce2 :: (a -> b) -> (a -> b) -> AT a -> AT b
reproduce2 f g t = (B (L f) (L g)) <*> t

reproduce2' :: (a -> b) -> (a -> b) -> AT a -> AT b
reproduce2' f g t =  (B (L f) (L g)) >>= (\f -> fmap f t)

reproduce2'' :: (a -> b) -> (a -> b) -> AT a -> AT b
reproduce2'' f g t = (B (L f) (L g)) `ap` t




-------------------------------------------------------------------------
--   TWELFTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION   --
-------------------------------------------------------------------------

-- First, the reversed sequencing version of (<*>)
--   (first, search for the leaves in the tree of values
--    second, pass each value to every function in the tree of funcs)
applyAT'' :: AT (a -> b) -> AT a -> AT b
applyAT'' at_fs (L x) = mapAT ($ x) at_fs
applyAT'' at_fs (B l r) = B (applyAT'' at_fs l) (applyAT'' at_fs r)

-- *Main> applyAT at3 at1
```

```
-- B (B (L 10) (B (L 4) (L 200))) (B (B (L 6) (B (L 3) (L 101))) (B (L 15) (B (L 6) (L 300))))
-- (0.00 secs, 0 bytes)
-- *Main> applyAT'' at3 at1
-- B (B (L 10) (B (L 6) (L 15))) (B (B (L 4) (B (L 3) (L 6))) (B (L 200) (B (L 101) (L 300))))
-- (0.00 secs, 0 bytes)


-- And the other version:

applyAT' :: AT (a -> b) -> AT a -> AT b
applyAT' (L f) at   = mapAT f at
applyAT' (B l r) at = B (applyAT' r at) (applyAT' l at)


-- *Main> applyAT at3 at1
-- B (B (L 10) (B (L 4) (L 200))) (B (B (L 6) (B (L 3) (L 101))) (B (L 15) (B (L 6) (L 300))))
-- (0.00 secs, 0 bytes)
-- *Main> applyAT' at3 at1
-- B (B (B (L 15) (B (L 6) (L 300))) (B (L 6) (B (L 3) (L 101)))) (B (L 10) (B (L 4) (L 200)))
-- (0.00 secs, 0 bytes)

-- pure id <*> t == t ??
-- pure id <*> t
--   = (L id) <*> t
--   = mapAT f t
--   == t

-- pure f <*> pure x == pure (f x) ??
-- pure f <*> pure x
--   = (L f) <*> (L x)
--   = mapAT f (L x)
--   = L (f x)
--   == pure (f x)

-- fs <*> pure x == pure ($ x) <*> fs ??
-- fs <*> pure x == pure ($ x) <*> fs
--   iff fs <*> (L x) == L ($ x) <*> fs
--   iff fs <*> (L x) == mapAT ($ x) fs
-- Now, fs <*> (L x) =
--   case fs of
--     (L f):
--       = (L f) <*> (L x)
--       = L (f x)
--       == L ($ x) <*> (L f)
--     (B l r):
--       = (B l r) <*> (L x)
--       = B (r <*> (L x)) (l <*> (L x))
--       =/= B (mapAT ($ x) l) (mapAT ($ x) r) ?!?!?!?!?
--       = mapAT ($ x) (B l r)
--       = L ($ x) <*> (B l r)

-- To solve that problem, i thought about:
```

```
-- -- First, we need an illegal version of fmap:

mapAT' :: (a -> b) -> AT a -> AT b
mapAT' f (L x)   = L (f x)
mapAT' f (B l r) = B (mapAT' f r) (mapAT' f l)

-- Occurs that
-- mapAT' id (B (L 1) (L 2))
--   = B (mapAT' id (L 2)) (mapAT' id (L 1))
--   = B (L 2) (L 1)
--   =/= B (L 1) (L 2)
-- However, the applicative instance seems ok   <-- NO:
--   pure id <*> t  =/=  t

-- So i went to
-- https://en.wikibooks.org/wiki/Haskell/Solutions/Applicative_functors#111
-- And found:

-- instance Applicative AT where
--     pure x                  = L x
--     L f       <*> tx        = fmap f tx
--     tf        <*> L x       = fmap ($ x) tf
--     B tfl tfr <*> B txl txr = B (tfl <*> txl) (tfr <*> txr)

-- " It only combines subtrees with matching positions in the tree structures.
-- The resulting behaviour is similar to that of ZipLists,
-- except that when the subtree shapes are different:
-- it inserts missing branches rather than removing extra ones
-- (and it couldn't be otherwise, since there are no empty ATs).
-- By the way, sagittalMap would have the exact same implementation of reproduce,
-- only using this instance. "

-- And seems ok:

-- pure id <*> t == t  OK
-- pure f <*> pure x == pure (f x)  OK
-- fs <*> pure x == pure ($ x) <*> fs  OK
-- pure (.) <*> gs <*> fs <*> as == gs <*> (fs <*> as)  OK


data AT' a = L' a | B' (AT' a) (AT' a)
  deriving (Show)

at1' :: AT' Int
at1' = B' (L' 5) (B' (L' 2) (L' 100))

at2' :: AT' (Int -> Char)
at2' = let f=toEnum.(100+) in B' (B' (L' f) (L' f)) (B' (L' f) (B' (L' f) (L' f)))

at3' :: AT' (Int -> Int)
at3' = B' (L' (2*)) ( B' (L' (1+)) (L' (3*)) )
```

```
instance Functor AT' where
  fmap f (L' x) = L' (f x)
  fmap f (B' l r) = B' (fmap f l) (fmap f r)

instance Applicative AT' where
  pure = L'
  L' f       <*> tx        = fmap f tx
  tf        <*> L' x       = fmap ($ x) tf
  B' tfl tfr <*> B' txl txr = B' (tfl <*> txl) (tfr <*> txr)



-- *Main> at3 <*> at1
-- B (B (L 10) (B (L 4) (L 200))) (B (B (L 6) (B (L 3) (L 101))) (B (L 15) (B (L 6) (L 300))))
-- (0.00 secs, 0 bytes)
-- *Main> at3' <*> at1'
-- B' (L' 10) (B' (L' 3) (L' 300))
-- (0.02 secs, 0 bytes)




-------------------------------------------------------------------------
-- THIRTEENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION  --
-------------------------------------------------------------------------

-- Recalling:
--   unit  :: Monoidal f => f ()
--   (*&*) :: Monoidal f => f a -> f b -> f (a,b)
--   pure  :: Applicative f => a -> f a
--   (<*>) :: Applicative f => f (a -> b) -> f a -> f b

-- IMPLEMENTING THE MONOIDAL CLASS:
-- (because its not defined in the Prelude)

class Functor f => Monoidal f where
  unit  :: f ()
  (*&*) :: f a -> f b -> f (a,b)


myUnit :: Applicative f => f ()
myUnit = pure ()

myOP :: Applicative f => f a -> f b -> f (a,b)
myOP a b = (fmap (,) a) <*> b


-- *Main> myOP [1,2,3] [4,5,6]
-- [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]
-- (0.02 secs, 0 bytes)
```

```
instance Monoidal [] where
  unit = [()]
  (*&*) l1 l2 = concatMap (((flip zip) l2) . repeat) l1

-- *Main> [1,2,3] *&* [4,5,6]
-- [(1,4),(1,5),(1,6),(2,4),(2,5),(2,6),(3,4),(3,5),(3,6)]



pureGivenMonoidal :: Monoidal f => a -> f a
pureGivenMonoidal x = fmap (\_ -> x) unit

applyGivenMonoidal :: Monoidal f => f (a -> b) -> f a -> f b
applyGivenMonoidal fs as = fmap (\(f,a) -> f a) (fs *&* as)




-----------------------------------------------------------------------
-- FOURTEENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION  --
-----------------------------------------------------------------------

-- Commutativity:
--   liftA2 f u v  ==  liftA2 (flip f) v u
--  or equivalently
--   f <$> u <*> v  ==  flip f <$> v <*> u
-- Or equivalently:
--   do {x <- u; y <- v; return (f x y)}
--     ==
--   do {y <- v; x <- u; return ((flip f) y x)}
--     ==
--   do {y <- v; x <- u; return (f x y)}

-- Or equivalently:
--   fmap (\(f,a) -> f a) ((f <$> u) *&* v)
--     ==
--   fmap (\(f,a) -> f a) ((flip f) <$> v *&* u)




-----------------------------------------------------------------------
--  FIFTEENTH EXERCISE OF THE ADVANCED MONAD AND APPLICATIVE SECTION  --
-----------------------------------------------------------------------

instance Monoidal ZipList where
  unit = ZipList [()]
  (*&*) (ZipList l1) (ZipList l2) = ZipList (zip l1 l2)


-- *Main> (,) <$> (ZipList [1,2,3,4]) <*> (ZipList [10,11,12])
```

```
-- ZipList {getZipList = [(1,10),(2,11),(3,12)]}
-- (0.00 secs, 0 bytes)
-- *Main> (ZipList [1,2,3,4]) *&* (ZipList [10,11,12])
-- ZipList {getZipList = [(1,10),(2,11),(3,12)]}
-- (0.00 secs, 0 bytes)



instance Monoidal ((->) r) where
  unit = const ()
  (*&*) f g = \r -> (f r, g r)


-- *Main> ((,) <$> (\n -> 2*n) <*> (\n -> 3*n)) 5
-- (10,15)
-- (0.00 secs, 0 bytes)
-- *Main> ((\n -> 2*n) *&* (\n -> 3*n)) 5
-- (10,15)
-- (0.00 secs, 0 bytes)
```

J.3 *State* exercises

1. Implement a function `rollNDiceIO :: Int -> IO [Int]` that, given an integer (a number of die rolls), returns a list of that number of pseudo-random integers between 1 and 6.

2. Implement a function `rollDice :: StdGen -> ((Int, Int), StdGen)` that, given a generator, returns a tuple with our random numbers as first element and the last generator as the second.

3. Similarly to what was done for rollNDiceIO, implement a function

   ```
   rollNDice :: Int -> State StdGen [Int]
   ```

   that, given an integer, returns a list with that number of pseudo-random integers between 1 and 6.

4. Write an instance of Functor for State s. Your final answer should not use anything that mentions Monad in its type (that is, return, (>>=), etc.). Then, explain in a few words what the fmap you wrote does.

   (Hint: If you get stuck, have another look at the comments about liftM in the main body.)

5. Besides put and get, there are also

   ```
   modify :: (s -> s) -> State s ()
   ```

   which modifies the current state using a function, and

   ```
   gets :: (s -> a) -> State s a
   ```

   which produces a modified copy of the state while leaving the state itself unchanged. Write implementations for them.

6. If you are not convinced that State is worth using, try to implement a function equivalent to evalState allTypes without making use of monads, i.e. with an approach similar to clumsyRollDice above.

```
{-# LANGUAGE TypeSynonymInstances #-}

import Control.Monad
import Control.Monad.State
import Control.Applicative
import System.Random



-------------------------------------------------------------------------
--                FIRST EXERCISE OF THE STATE SECTION                  --
-------------------------------------------------------------------------

rollNDiceIO :: Int -> IO [Int]
rollNDiceIO n = (take n) <$> (getStdGen >>= f)
  where
    f = \ gen -> return (randomRs (1,6) gen)


{-

*Main> rollNDiceIO 10
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package bytestring-0.10.4.0 ... linking ... done.
Loading package Win32-2.3.0.2 ... linking ... done.
Loading package old-locale-1.0.0.6 ... linking ... done.
Loading package time-1.4.2 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
[5,5,1,6,6,5,1,6,5,3]
(0.11 secs, 5620896 bytes)
*Main> rollNDiceIO 6
[5,5,1,6,6,5]
(0.00 secs, 0 bytes)
*Main> rollNDiceIO 10
[5,5,1,6,6,5,1,6,5,3]
(0.02 secs, 0 bytes)
*Main> rollNDiceIO 10
[5,5,1,6,6,5,1,6,5,3]
(0.00 secs, 0 bytes)
*Main> getStdGen
700125431 1
(0.00 secs, 0 bytes)
*Main> getStdGen
700125431 1
(0.00 secs, 0 bytes)
*Main> newStdGen
895916699 2147483398
(0.00 secs, 0 bytes)
*Main> getStdGen
700125432 40692
(0.00 secs, 0 bytes)
```

```
*Main> getStdGen
700125432 40692
(0.00 secs, 0 bytes)
*Main> newStdGen
895956713 40691
(0.00 secs, 0 bytes)
*Main> getStdGen
700125433 1655838864
(0.00 secs, 0 bytes)
*Main> rollNDiceIO 10
[4,5,4,3,5,6,6,5,2,5]
(0.00 secs, 0 bytes)

-}


rollNDiceIO' :: Int -> IO [Int]
rollNDiceIO' n = sequence (fmap randomRIO (replicate n (1,6)))

-- does exactly the same




--------------------------------------------------------------------------
--                SECOND EXERCISE OF THE STATE SECTION                  --
--------------------------------------------------------------------------

{-
roll2Dice :: StdGen -> ((Int,Int) , StdGen)
roll2Dice gen = ((n1,n2) , b) where
  (gen1, gen2) = split gen
  n1 = fst $ randomR (1,6) gen1
  n2 = fst $ randomR (1,6) gen2
  b  = snd $ randomR (1,6) gen2

This version seems correct but gives an "ambiguous type" error
-}


roll2Dice' :: StdGen -> ((Int,Int) , StdGen)
roll2Dice' gen = ((n1,n2) , b) where
  (gen1, gen2) = split gen
  n1 = fst $ aux gen1
  n2 = fst $ aux gen2
  b  = snd $ aux gen2
  aux = (\g -> randomR (1,6) g)  :: StdGen -> (Int,StdGen)


{-

*Main> roll2Dice' (mkStdGen 0)
```

```
Loading package array-0.5.0.0 ... linking ... done.
Loading package deepseq-1.3.0.2 ... linking ... done.
Loading package bytestring-0.10.4.0 ... linking ... done.
Loading package Win32-2.3.0.2 ... linking ... done.
Loading package old-locale-1.0.0.6 ... linking ... done.
Loading package time-1.4.2 ... linking ... done.
Loading package random-1.0.1.1 ... linking ... done.
((6,1),1601120196 2147442707)
(0.08 secs, 6278672 bytes)
*Main> roll2Dice' (mkStdGen 0)
((6,1),1601120196 2147442707)
(0.02 secs, 0 bytes)
*Main> roll2Dice' (mkStdGen 38)
((6,2),166664317 2147442707)
(0.02 secs, 8181848 bytes)
*Main> roll2Dice' (mkStdGen 123)
((6,5),970416508 2147442707)
(0.00 secs, 0 bytes)
*Main> roll2Dice' (mkStdGen 3456789)
((2,1),1125608184 2147442707)
(0.00 secs, 0 bytes)
*Main> roll2Dice' (mkStdGen 4689273)
((3,4),1290960903 2147442707)
(0.00 secs, 0 bytes

-}


roll2Dice'' :: StdGen -> ((Int,Int) , StdGen)
roll2Dice'' gen = ((n1,n2) , res) where
  (n1,aux) = randomR (1,6) gen
  (n2,res) = randomR (1,6) aux

-- *Main> roll2Dice' (mkStdGen 0)
-- ((6,1),1601120196 2147442707)
-- (0.00 secs, 0 bytes)
-- *Main> roll2Dice'' (mkStdGen 0)
-- ((6,6),1601120196 1655838864)
-- (0.02 secs, 8290224 bytes)




--------------------------------------------------------------------------
--                THIRD EXERCISE OF THE STATE SECTION                   --
--------------------------------------------------------------------------

rollNDice :: Int -> State StdGen [Int]
rollNDice n = state ( \gen -> ((take n) $ (randomRs (1,6) gen) , (snd.next) gen) )


{-
```

```
*Main> rollNDice 10

<interactive>:21:1:
    No instance for (Show (State StdGen [Int]))
      arising from a use of ‘print’
    In a stmt of an interactive GHCi command: print it
(0.02 secs, 0 bytes)
*Main> runState (rollNDice 10) (mkStdGen 0)
Loading package transformers-0.4.3.0 ... linking ... done.
Loading package mtl-2.2.1 ... linking ... done.
([6,6,4,1,5,2,4,2,2,1],40014 40692)
(0.02 secs, 0 bytes)
*Main> runState (rollNDice 10) (mkStdGen 0)
([6,6,4,1,5,2,4,2,2,1],40014 40692)
(0.00 secs, 0 bytes)
*Main> runState (rollNDice 11) (mkStdGen 0)
([6,6,4,1,5,2,4,2,2,1,6],40014 40692)
(0.00 secs, 0 bytes)
*Main> runState (rollNDice 11) (mkStdGen 534621)
([3,1,5,6,2,4,3,3,6,1,2],2065012641 40692)
(0.00 secs, 0 bytes)

-}




--------------------------------------------------------------------------
--               FOURTH EXERCISE OF THE STATE SECTION                   --
--------------------------------------------------------------------------


-- instance Functor (State s) where

{-
StateExercises.hs:188:10:
    Illegal instance declaration for ‘Functor (State s)’
      (All instance types must be of the form (T t1 ... tn)
       where T is not a synonym.
       Use TypeSynonymInstances if you want to disable this.)
    In the instance declaration for ‘Functor (State s)’
-}

-- After adding that at the top of this file:

{-
StateExercises.hs:190:10:
    Illegal instance declaration for ‘Functor (State s)’
      (All instance types must be of the form (T a1 ... an)
       where a1 ... an are *distinct type variables*,
       and each type variable appears at most once in the instance head.
```

```
       Use FlexibleInstances if you want to disable this.)
    In the instance declaration for ‘Functor (State s)’
-}

-- Redefining the State data (to avoid these problems):

data State' s a = St (s -> (a,s))

instance Functor (State' s) where
  fmap f (St g) = St ( \s -> ( f $ fst $ g s , snd $ g s) )


-- fmap id x == x ??
-- fmap id (St g)
--   = St ( \s -> ( fst $ g s , snd $ g s) )
--   = St ( \s -> g s)
--   == (St g)

-- fmap (g.f) x == (fmap g) . (fmap f) x ??
-- fmap (g.f) (St pr)
--   = ST ( \s -> ((f.g) $ fst $ pr s , snd $ pr s) )
--   == fmap g (St (\s -> (f $ fst $ pr s , snd $ pr s)))
--   = ((fmap g) . (fmap f)) x

-- This Functor instance works as follow:
--   fmap f (St pr)
--     with  f :: a -> b , pr :: s -> (a,s)
--   applies f to the first value of the pair returned by pr




-------------------------------------------------------------------------
--               FIFTH EXERCISE OF THE STATE SECTION                   --
-------------------------------------------------------------------------

myModify :: (s -> s) -> State s ()
myModify f = state ( \s -> (() , f s) )

-- *Main> runState (myModify (2*)) 5
-- ((),10)
-- (0.00 secs, 0 bytes)



myGets :: (s -> a) -> State s a
myGets f = state ( \s -> (f s , s) )

-- *Main> runState (myGets (toEnum :: Int -> Char)) 90
-- ('Z',90)
-- (0.00 secs, 0 bytes)
```

```
-------------------------------------------------------------------------
--               SIXTH EXERCISE OF THE STATE SECTION                   --
-------------------------------------------------------------------------


-- evalState :: State s a -> s -> a

allTypes :: State StdGen (Int, Float, Char, Integer, Double, Bool, Int)
allTypes = liftM (,,,,,,) getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom
                      `ap` getRandom

getRandom :: Random a => State StdGen a
getRandom = state random



monsterRandom :: StdGen -> (Int, Float, Char, Integer, Double, Bool, Int)
monsterRandom gen = (n1,f1,c1,n2,d1,b1,n3)
  where
    (n1,g1) = random gen
    (f1,g2) = random g1
    (c1,g3) = random g2
    (n2,g4) = random g3
    (d1,g5) = random g4
    (b1,g6) = random g5
    (n3,_) = random g6



-- *Main> evalState allTypes (mkStdGen 0)
-- (-117157315039303149,0.4883204,'\260381',-2598893763451025729,0.30447780927171453,False,-525544148
-- (0.02 secs, 0 bytes)
-- *Main> monsterRandom (mkStdGen 0)
-- (-117157315039303149,0.4883204,'\260381',-2598893763451025729,0.30447780927171453,False,-525544148
-- (0.00 secs, 0 bytes)
```

J.4 *MonadPlus* exercises

1. Prove the MonadPlus laws for Maybe and the list monad.
2. We could augment our above parser to involve a parser for any character:

```
-- | Consume a given character in the input, and return the character we
--   just consumed, paired with rest of the string. We use a do-block  so that
--   if the pattern match fails at any point, fail of the Maybe monad (i.e.
--   Nothing) is returned.
char :: Char -> String -> Maybe (Char, String)
char c s = do
  let (c':s') = s
  if c == c' then Just (c, s') else Nothing
```

   It would then be possible to write a `hexChar` function which parses any valid hexadecimal character (0-9 or a-f). Try writing this function

   (hint: `map digit [0..9] :: [String -> Maybe Int]`).

```
import Control.Monad

-------------------------------------------------------------------------
--              FIRST EXERCISE OF THE MONADPLUS SECTION                --
-------------------------------------------------------------------------


-- instance MonadPlus [] where
--   mzero = []
--   mplus = (++)

-- neutral element:
--  mzero `mplus` m = [] ++ m == m
--  m `mplus` mzero = m ++ [] == m

-- associativity
-- (l1 `mplus` l2) `mplus` l3
--   = (l1 ++ l2) ++ l3
--   == l1 ++ (l2 ++ l3)
--   = l1 `mplus` (l2 `mplus` l3)

-- interaction with the monad part

-- mzero >>= f  ==  mzero  ??
-- mzero >>= f
--   = [] >>= f
--   = concatMat f []
--   = []
--   == mzero

-- l >> mzero  ==  mzero  ??
-- l >> mzero
--   = l >> []
--   = l >>= (\_ -> [])
--   = concatMap (\_ -> []) l
--   = []
--   == mzero

-- (l1 `mplus` l2) >>= k
--   ==  ????
-- (l1 >>= k) `mplus` (l2 >>= k)

-- (l1 `mplus` l2) >>= k
--   = (l1 ++ l2) >>= k
--   = concatMap k (l1++l2)
--   == (concatMap k l1) ++ (concatMap k l2)
--   = (l1 >>= k) `mplus` (l2 >>= k)



-------------------------------------------------------------------------
```

```
--             SECOND EXERCISE OF THE MONADPLUS SECTION               --
-----------------------------------------------------------------------


char :: Char -> String -> Maybe (Char, String)
char c s = do
  let (c':s') = s
  if c == c' then Just (c,s') else Nothing


digit :: Int -> String -> Maybe Int
digit i s | i > 9 || i < 0 = Nothing
          | otherwise      = do
  let (c:_) = s
  if [c] == show i then Just i else Nothing


hexChar :: String -> Maybe Char
hexChar s = (fmap (head . show) (isDigit s)) `mplus` isValidChar
  where
    funcList    = map digit [0..9]
    isDigit x   = msum (map ($ x) funcList)
    char' c s   = fmap fst (char c s)
    funcList'   = map char' ['a','b','c','d','e','f']
    isValidChar = msum (map ($ s) funcList')
```

J.5 Monad transformers exercises'

1. Why is it that the lift function has to be defined separately for each monad, where as liftM can be defined in a universal way?

2. Identity is a trivial functor, defined in Data.Functor.Identity as:

   ```
   newtype Identity a = Identity { runIdentity :: a }
   ```

   It has the following Monad instance:

   ```
   instance Monad Identity where
       return a = Identity a
       m >>= k  = k (runIdentity m)
   ```

   Implement a monad transformer IdentityT, analogous to Identity but wrapping values of type m a rather than a. Write at least its Monad and MonadTrans instances.

3. Implement `state :: MonadState s m => (s -> (a, s)) -> m a` in terms of get and put.

4. Are MaybeT (State s) and StateT s Maybe equivalent? (Hint: one approach is comparing what the run...T unwrappers produce in each case.)

```
-------------------------------------------------------------------------
--         FIRST EXERCISE OF THE MONAD TRANSFORMERS SECTION            --
-------------------------------------------------------------------------
```

J.6 Hask category exercises

1. As was mentioned, any partial order $(P, \leq$ is a category with objects the elements of P and a morphism between elements a and b iff $a \leq b$. Which of the above laws guarantees the transitivity of $\leq$?

2. Check the functor laws for the Maybe and list functors.

3. Verify that the list and Maybe monads do in fact obey the first monad law,

   ```
   join . fmap join = join . join
   ```

   with some examples to see precisely how the layer flattening works.

4. Prove the second monad law, join . fmap return = join . return = id for the Maybe monad.

5. Convince yourself that the 3rd and 4th laws should hold true for any monad by exploring what they mean, in a similar style to how the first and second laws were explored.

6. In fact, the two versions of the laws we gave:

   ```
   -- Categorical:
   join . fmap join = join . join
   join . fmap return = join . return = id
   return . f = fmap f . return
   join . fmap (fmap f) = fmap f . join

   -- Functional:
   m >>= return = m
   return m >>= f = f m
   (m >>= f) >>= g = m >>= (\x -> f x >>= g)
   ```

   are entirely equivalent. We showed that we can recover the functional laws from the categorical ones. Go the other way; show that starting from the functional laws, the categorical laws hold. It may be useful to remember the following definitions:

   ```
   join m = m >>= id
   fmap f m = m >>= return . f
   ```

```
-------------------------------------------------------------------------
--            FIRST EXERCISE OF THE HASK CATEGORY SECTION              --
-------------------------------------------------------------------------

-- Given a partially ordered set (P, <=), we can define a category whose
-- objects are the elements of P, and there is a morphism between elements
-- a and b  iff  a <= b

-- The transitivity of <= guarantees the existence of the composition law,
-- this is:
--   if f and g exist, with  f : a -> b  ,  g : b -> c
--   ==> a <= b  and  b <= c
--   ==> a <= c   (transitivity)
--   ==> exists h : a -> c
--         (so we can asign g.f = h)




-------------------------------------------------------------------------
--           SECOND EXERCISE OF THE HASK CATEGORY SECTION              --
-------------------------------------------------------------------------

-- Maybe functor:
{-
instance  Functor Maybe  where
    fmap _ Nothing       = Nothing
    fmap f (Just a)      = Just (f a)
-}

-- fmap id Nothing == Nothing
-- fmap id (Just x) = Just (id x) == (Just x)

-- fmap (g.f) Nothing
--   = Nothing
--   == fmap g Nothing
--   = fmap g (fmap f Nothing)

-- fmap (g.f) (Just x)
--   = Just ( g (f x) )
--   == fmap g (Just (f x))
--   = fmap g (fmap f (Just x))


-- List functor:
{-
instance Functor [] where
    {-# INLINE fmap #-}
    fmap = map
-}
{-
```

```
map _ []     = []
map f (x:xs) = f x : map f xs
-}

-- map id [] == []
-- map id (x:xs)
--   = (id x) : map id xs
--   = x : map id xs
--   == (x:xs)

-- map (g.f) []
--   == []
--   = map g []
--   = map g (map f [])

-- map (g.f) (x:xs)
--   = (g (f x)) : map (g.f) xs
--   == map g ((f x):xs)
--   = map g (map f (x:xs))




-------------------------------------------------------------------------
--            THIRD EXERCISE OF THE HASK CATEGORY SECTION              --
-------------------------------------------------------------------------
{-
join x = x >>= id
-}

-- Maybe monad:
{-
instance  Monad Maybe  where
    (Just x) >>= k      = k x
    Nothing  >>= _      = Nothing
-}
{- occurs that
join :: Maybe (Maybe a) -> Maybe a
join Nothing         = Nothing
join (Just Nothing)  = Nothing
join (Just (Just x)) = Just x
-}

-- (join . fmap join) Nothing
--   = join Nothing
--   == Nothing
--   = join (join Nothing)

-- (join . fmap join) (Just Nothing)
--   == Nothing
--   = (join . join) (Just Nothing)
```

```
-- (join . fmap join) (Just (Just Nothing))
--   = join (Just (join (Just Nothing)))
--   = join (Just Nothing)
--   == Nothing
--   = join (Just Nothing)
--   = (join . join) (Just (Just Nothing))

-- (join . fmap join) (Just (Just (Just x)))
--   = join (Just (join (Just (Just x))))
--   = join (Just (Just x))
--   = Just x
--   == join (Just (Just x))
--   = (join . join) (Just (Just (Just x)))


-- List monad:
--   Some examples

{-
Prelude Control.Monad> (join . fmap join) [[]]
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . fmap join) []
[]
(0.00 secs, 2959128 bytes)
Prelude Control.Monad> (join . fmap join) [[]]
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . fmap join) [[[]]]
[]
(0.00 secs, 3707032 bytes)
Prelude Control.Monad> (join . fmap join) [[[[]]]]
[[]]
(0.02 secs, 0 bytes)
Prelude Control.Monad> (join . join) []
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . join) [[]]
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . join) [[[]]]
[]
(0.00 secs, 0 bytes)
Prelude Control.Monad> (join . join) [[[[]]]]
[[]]
(0.00 secs, 0 bytes)
-}
-- Prelude Control.Monad> fmap join [ [[1,2],[3]] , [[4,5],[6,7]] ]
-- [[1,2,3],[4,5,6,7]]     <-- CHECK THIS
-- (0.00 secs, 0 bytes)
-- Prelude Control.Monad> (join . fmap join) [ [[1,2],[3]] , [[4,5],[6,7]] ]
-- [1,2,3,4,5,6,7]
```

```
-- (0.00 secs, 0 bytes)
-- Prelude Control.Monad>  join [ [[1,2],[3]] , [[4,5],[6,7]] ]
-- [[1,2],[3],[4,5],[6,7]] <- AND THIS
-- (0.00 secs, 0 bytes)
-- Prelude Control.Monad> (join . join) [ [[1,2],[3]] , [[4,5],[6,7]] ]
-- [1,2,3,4,5,6,7]
-- (0.02 secs, 0 bytes)




-------------------------------------------------------------------------
--            FOURTH EXERCISE OF THE HASK CATEGORY SECTION             --
-------------------------------------------------------------------------

-- Second law:
--   join . fmap return = join . return = id

-- (join . fmap return) Nothing
--   = join Nothing
--   = Nothing
--   == join (Just Nothing)
--   = (join . return) Nothing

-- (join . fmap return) (Just x)
--   = join (Just (Just x))
--   = Just x
--   == join (Just (Just x))
--   = (join . return) (Just x)




-------------------------------------------------------------------------
--            FIFTH EXERCISE OF THE HASK CATEGORY SECTION              --
-------------------------------------------------------------------------

-- Third law:
--   return . f = fmap f . return

-- f :: a -> b        , (return . f)      = a -> m b
-- return :: a -> m a , (fmap f . return) = a -> m b

-- States that applying a function to a value and then
-- embedding the result into the monad is the same as
-- embedding the value into the monad and then mapping the function.



-- Fourth law:
--   join . fmap (fmap f) = fmap f . join
```

```
-- (fmap f) :: m a -> m b ,
-- fmap (fmap f) :: m (m a) -> m (m b),
-- join . fmap (fmap f) :: m (m a) -> m b
-- AND on the other side:
-- join :: m (m a) -> m a ,
-- fmap f . join :: m (m a) -> m b

-- States that, when having a 2 layer monadic value m (m a)
-- its the same joining and then mapping or
-- mapping to the inner layer and joining afterwards.




-------------------------------------------------------------------------
--            SIXTH EXERCISE OF THE HASK CATEGORY SECTION              --
-------------------------------------------------------------------------

-- Recalling:
--   join m   =  m >>= id
--   fmap f m =  m >>= return . f



-- First Law:
--   join . fmap join == join . join  ??

-- (join . fmap join) m
--   = join (fmap join m)
--   = join (m >>= return . join)
--   = (m >>= return . join) >>= id
--   = (m >>= id) >>= id     *****
--   = join (m >>= id)
--   = join (join m)
--   = (join . join) m

-- ***** OBS: we need to prove that return.join == id
--            (which is the second law)


-- Second Law:
--   join . fmap return == join . return == id ??

-- (join . fmap return) m
--   = join (fmap return m)
--   = join (m >>= return . return)
--   = join (return m)

-- (join . return) m
--   = join (return m)
--   = return m >>= id
--   = id m
```

```
--   == m


-- Third Law:
--   return . f = fmap f . return

-- (return  . f) x
--   = return (f x)

-- (fmap f . return) x
--   = fmap f (return x)
--   = (return x) >>= return . f
--   = (return . f) x


-- Fourth Law:
--   join . fmap (fmap f) = fmap f . join

-- (fmap f . join) m
--   = fmap f (join m)
--   = fmap f (m >>= id)
--   = (m >>= id) >>= return . f
--   = (m >>= id) >>= fmap f . return

--   = m >>= (\m' -> id m' >>= return . f)
--   = m >>= (\m' -> m' >>= return . f)

-- (join . fmap (fmap f)) m
--   = join (fmap (fmap f) m)
--   = join (m >>= return . (fmap f))
--   = join (m >>= \x -> return (fmap f x))
--   = (m >>= \x -> return (fmap f x)) >>= id
```