# A Fistful of Monads

When we first talked about functors, we saw that they were a useful concept for values that can be mapped over. Then, we took that concept one step further by introducing applicative functors, which allow us to view values of certain data types as values with contexts and use normal functions on those values while preserving the meaning of those contexts.

In this chapter, we'll learn about monads, which are just beefed up applicative functors, much like applicative functors are only beefed up functors.

ファンクターについて最初に説明したとき、ファンクターはマッピング可能な値のための便利な概念であることがわかった。このファンクターによって、特定のデータ型の値をコンテキストを持つ値として見ることができ、コンテキストの意味を保持したままその値に対して通常の関数を使うことができるようになります。

この章では、モナドについて学びます。モナドは、アプリケーティブ・ファンクタを強化したものに過ぎず、アプリケーティブ・ファンクタがファンクタを強化したものに過ぎないのと同じです。

When we started off with functors, we saw that it's possible to map functions over various data types. We saw that for this purpose, the `Functor` type class was introduced and it had us asking the question: when we have a function of type `a -> b` and some data type `f a`, how do we map that function over the data type to end up with `f b`? We saw how to map something over a `Maybe a`, a list `[a]`, an `IO a` etc. We even saw how to map a function `a -> b` over other functions of type `r -> a` to get functions of type `r -> b`. To answer this question of how to map a function over some data type, all we had to do was look at the type of `fmap`:

ファンクターを使い始めたとき、さまざまなデータ型に関数をマッピングできることを知った。この目的のためにファンクター型クラスが導入され、a -> b 型の関数とあるデータ型 f a があるとき、その関数をそのデータ型にマッピングして f b を得るにはどうすればよいかという疑問が生じた。多分 a、リスト[a]、IO a などに何かをマッピングする方法を見た。関数 a -> b を他の r -> a 型の関数にマッピングして r -> b 型の関数を得る方法も見た:

```
1. fmap :: (Functor f) => (a -> b) -> f a -> f b
```

And then make it work for our data type by writing the appropriate `Functor` instance.
Then we saw a possible improvement of functors and said, hey, what if that function `a -> b` is already wrapped inside a functor value? Like, what if we have `Just (*3)`, how do we apply that to `Just 5`? What if we don't want to apply it to `Just 5` but to a `Nothing` instead? Or if we have `[(*2),(+4)]`, how would we apply that to `[1,2,3]`? How would that work even? For this, the `Applicative` type class was introduced, in which we wanted the answer to the following type:

そして、適切なファンクターのインスタンスを書くことで、私たちのデータ型に対応させる。
そして、ファンクターの改良の可能性を見て、a -> b という関数がすでにファンクターの値の中に包まれていたらどうだろう？例えば、Just (*3)があったとして、それを Just 5 に適用するにはどうすればいいのか？例えば、Just (*3)があったとして、それを Just 5 に適用するのはどうだろう？あるいは、[(*2),(+4)]があったとして、それをどうやって[1,2,3]に適用す

るのか？どうすればいいのだろう？そのために、適用型クラスが導入され、次のような型の答えが求められるようになった：

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

We also saw that we can take a normal value and wrap it inside a data type. For instance, we can take a `1` and wrap it so that it becomes a `Just 1`. Or we can make it into a `[1]`. Or an I/O action that does nothing and just yields `1`. The function that does this is called `pure`.

Like we said, an applicative value can be seen as a value with an added context. A *fancy* value, to put it in technical terms. For instance, the character `'a'` is just a normal character, whereas `Just 'a'` has some added context. Instead of a `Char`, we have a `Maybe Char`, which tells us that its value might be a character, but it could also be an absence of a character.

It was neat to see how the `Applicative` type class allowed us to use normal functions on these values with context and how that context was preserved. Observe:

また、通常の値をデータ型にラップすることもできる。例えば、1 をラップしてただの 1 にすることもできるし、[1]にすることもできる。これを行う関数は pure と呼ばれる。

さっきも言ったように、応用的な値というのは、コンテキストが追加された値と見なすことができる。専門用語で言えば、空想的な値だ。例えば、文字'a'は普通の文字である。Char の代わりに Maybe Char があり、これはその値が文字である可能性があることを示すが、文字の不在である可能性もある。

Applicative 型クラスによって、コンテキストを持つこれらの値に対して通常の関数を使用することができ、そのコンテキストがどのように保持されるかを見ることができたのは、素晴らしいことだった。観察してみよう：

```
ghci> (*) <$> Just 2 <*> Just 8
Just 16
ghci> (++) <$> Just "klingon" <*> Nothing
Nothing
ghci> (-) <$> [3,4] <*> [1,2,3]
[2,1,0,3,2,1]
```

Ah, cool, so now that we treat them as applicative values, `Maybe a` values represent computations that might have failed, `[a]` values represent computations that have several results (non-deterministic computations), `IO a` values represent values that have side-effects, etc.

Monads are a natural extension of applicative functors and with them we're concerned with this: if you have a value with a context, `m a`, how do you apply to it a function that takes a normal `a` and returns a value with a context? That is, how do you apply a function of type `a -> m b` to a value of type `m a`? So essentially, we will want this function:

つまり、a 値は失敗したかもしれない計算を表し、[a]値はいくつかの結果を持つ計算（非決定論的計算）を表し、IO a 値は副作用を持つ値を表す、といった具合だ。

モナドは適用型ファンクタの自然な拡張であり、モナドでは次のようなことが問題になる：コンテキストを持つ値 m a がある場合、通常の a を受け取りコンテキストを持つ値を返す関数をどのように適用するか？つまり、a -> m b 型の関数を m a 型の値に適用するにはどうすればいいのか？つまり、本質的にはこの関数が必要なのである：

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

**If we have a fancy value and a function that takes a normal value but returns a fancy value, how do we feed that fancy value into the function?** This is the main question that we will concern ourselves when dealing

with monads. We write `m a` instead of `f a` because the `m` stands for `Monad`, but monads are just applicative functors that support `>>=`. The `>>=` function is pronounced as *bind*.
When we have a normal value `a` and a normal function `a -> b` it's really easy to feed the value to the function — you just apply the function to the value normally and that's it. But when we're dealing with values that come with certain contexts, it takes a bit of thinking to see how these fancy values are fed to functions and how to take into account their behavior, but you'll see that it's easy as one two three.

空想的な値と、普通の値を受け取って空想的な値を返す関数があったとして、その空想的な値をどうやって関数に送り込めばいいのだろうか？これがモナドを扱う際の主な疑問である。f a の代わりに m a と書くのは、m がモナドの略だからだが、モナドは>>=をサポートする単なる適用ファンクターである。
普通の値 a と普通の関数 a -> b があるとき、その値を関数に渡すのはとても簡単だ。しかし、ある特定の文脈を持つ値を扱う場合、これらの空想的な値がどのように関数に供給され、どのようにその振る舞いを考慮するのか、少し考える必要がありますが、1・2・3 のように簡単であることがわかるでしょう。

## Getting our feet wet with Maybe

Now that we have a vague idea of what monads are about, let's see if we can make that idea a bit less vague.

Much to no one's surprise, `Maybe` is a monad, so let's explore it a bit more and see if we can combine it with what we know about monads.
Make sure you understand applicatives at this point. It's good if you have a feel for how the various `Applicative` instances work and what kind of computations they represent, because monads are nothing more than taking our existing applicative knowledge and upgrading it.
A value of type `Maybe a` represents a value of type `a` with the context of possible failure attached. A value of `Just "dharma"` means that the string `"dharma"` is there whereas a value of `Nothing` represents its absence, or if you look at the string as the result of a computation, it means that the computation has failed.
When we looked at `Maybe` as a functor, we saw that if we want to `fmap` a function over it, it gets mapped over the insides if it's a `Just` value, otherwise the `Nothing` is kept because there's nothing to map it over!
Like this:

さて、モナドが何であるかについて漠然とした考えを持つことができたので、その考えをもう少し漠然としたものでなくすることができるか見てみよう。誰も驚かないだろうが、Maybe はモナドの一種なのだ。だから、もう少しモナドについて調べて、私たちがモナドについて知っていることと組み合わせられるか見てみよう。この時点でアプリカティブを理解していることを確認してください。様々な Applicative インスタンスがどのように機能し、どのような計算を表しているのか、感覚的に理解しておくと良いでしょう。モナドは既存の Applicative の知識をアップグレードしたものに他ならないからです。

Maybe a 型の値は、失敗する可能性のあるコンテキストが付加された a 型の値を表す。ただ "dharma "という値は "dharma "という文字列が存在することを意味し、Nothing という値はその文字列が存在しないことを表す。

Maybe をファンクタとして見たとき、関数を fmap する場合、それが Just 値であればその内側にマップされ、そうでなければマップするものがないので Nothing が保持されることがわかった！

このように:

```
ghci> fmap (++"!") (Just "wisdom")
Just "wisdom!"
ghci> fmap (++"!") Nothing
Nothing
```

As an applicative functor, it functions similarly. However, applicatives also have the function wrapped. `Maybe` is an applicative functor in such a way that when we use `<*>` to apply a function inside a `Maybe` to a value that's inside a `Maybe`, they both have to be `Just` values for the result to be a `Just` value, otherwise the result is `Nothing`. It makes sense because if you're missing either the function or the thing you're applying it to, you can't make something up out of thin air, so you have to propagate the failure:

アプリケイティブ・ファンクターとしても、同様の働きをする。しかし、アプリケータは関数をラップすることもできる。Maybe の中にある関数を Maybe の中にある値に適用するために<*>を使うとき、その結果が Just 値であるためには、両方とも Just 値でなければならない。これは理にかなっている。なぜなら、関数かそれを適用するもののどちらかが欠落している場合、何もないところから何かを作り出すことはできないので、失敗を伝播させなければならないからだ:

```
ghci> Just (+3) <*> Just 3
Just 6
ghci> Nothing <*> Just "greed"
Nothing
ghci> Just ord <*> Nothing
Nothing
```

When we use the applicative style to have normal functions act on `Maybe` values, it's similar. All the values have to be `Just` values, otherwise it's all for `Nothing`!

アプリケーティブ・スタイルを使って、通常の関数が Maybe 値に対して作用するようにする場合も同様だ。すべての値が Just 値でなければならない！

```
ghci> max <$> Just 3 <*> Just 6
Just 6
ghci> max <$> Just 3 <*> Nothing
Nothing
```

And now, let's think about how we would do `>>=` for `Maybe`. Like we said, `>>=` takes a monadic value, and a function that takes a normal value and returns a monadic value and manages to apply that function to the monadic value. How does it do that, if the function takes a normal value? Well, to do that, it has to take into account the context of that monadic value.
In this case, `>>=` would take a `Maybe a` value and a function of type `a -> Maybe b` and somehow apply the function to the `Maybe a`. To figure out how it does that, we can use the intuition that we have from `Maybe` being an applicative functor. Let's say that we have a function `¥x -> Just (x+1)`. It takes a number, adds `1` to it and wraps it in a `Just`:

次に、>>=を Maybe に対してどのように適用するかを考えてみよう。さっきも言ったように、>>=はモナド値を受け取り、普通の値を受け取ってモナド値を返す関数で、その関数をモナド値に適用するように管理する。その関数が通常の値を取る場合、どのようにそれを行うのだろうか？そのためには、モナド値のコンテキストを考慮しなければならない。
この場合、>>=は Maybe a という値と a -> Maybe b という型の関数を受け取り、その関数を Maybe a に適用する。例えば、関数 ¥x -> Just (x+1) があるとしよう。これは数値を受け取り、それに 1 を加え、Just で包む：

```
ghci> (¥x -> Just (x+1)) 1
Just 2
ghci> (¥x -> Just (x+1)) 100
Just 101
```

If we feed it **1**, it evaluates to **Just 2**. If we give it the number **100**, the result is **Just 101**. Very straightforward. Now here's the kicker: how do we feed a **Maybe** value to this function? If we think about how **Maybe** acts as an applicative functor, answering this is pretty easy. If we feed it a **Just** value, take what's inside the **Just** and apply the function to it. If give it a **Nothing**, hmm, well, then we're left with a function but **Nothing** to apply it to. In that case, let's just do what we did before and say that the result is **Nothing**.
Instead of calling it **>>=**, let's call it **applyMaybe** for now. It takes a **Maybe a** and a function that returns a **Maybe b** and manages to apply that function to the **Maybe a**. Here it is in code:

1 を与えればジャスト 2 と評価され、100 を与えればジャスト 101 となる。非常にわかりやすい。さて、ここからが重要なのだが、この関数に Maybe の値を与えるにはどうすればいいのだろうか？Maybe がどのように適用ファンクターとして機能するかを考えれば、この答えはとても簡単だ。もし Just 値を与えれば、Just の中にあるものを取り出し、それに関数を適用する。もし Nothing を与えたら、うーん、関数はあるけどそれを適用するのは Nothing ということになる。その場合は、さっきと同じように、結果は Nothing であるとしよう。
これを >>= と呼ぶ代わりに、とりあえず applyMaybe と呼ぶことにしよう。これは Maybe a と Maybe b を返す関数を受け取り、その関数を Maybe a に適用する：

```
applyMaybe :: Maybe a -> (a -> Maybe b) -> Maybe b
applyMaybe Nothing f  = Nothing
applyMaybe (Just x) f = f x
```

Okay, now let's play with it for a bit. We'll use it as an infix function so that the **Maybe** value is on the left side and the function on the right:

では、ちょっと遊んでみよう。Maybe の値が左側に、関数が右側に来るように、これを infix 関数として使うことにしよう：

```
ghci> Just 3 `applyMaybe` ¥x -> Just (x+1)
Just 4
ghci> Just "smile" `applyMaybe` ¥x -> Just (x ++ " :)")
Just "smile :)"
ghci> Nothing `applyMaybe` ¥x -> Just (x+1)
Nothing
ghci> Nothing `applyMaybe` ¥x -> Just (x ++ " :)")
Nothing
```

In the above example, we see that when we used **applyMaybe** with a **Just** value and a function, the function simply got applied to the value inside the **Just**. When we tried to use it with a **Nothing**, the whole result was **Nothing**. What about if the function returns a **Nothing**? Let's see:

上の例では、applyMaybe を Just の値と関数で使った場合、関数は単に Just の中の値に適用された。これを Nothing で使おうとすると、結果はすべて Nothing でした。関数が Nothing を返す場合はどうでしょうか？見てみよう:

```
ghci> Just 3 `applyMaybe` ¥x -> if x > 2 then Just x else Nothing
Just 3
ghci> Just 1 `applyMaybe` ¥x -> if x > 2 then Just x else Nothing
Nothing
```

Just what we expected. If the monadic value on the left is a **`Nothing`**, the whole thing is **`Nothing`**. And if the function on the right returns a **`Nothing`**, the result is **`Nothing`** again. This is very similar to when we used **`Maybe`** as an applicative and we got a **`Nothing`** result if somewhere in there was a **`Nothing`**.
It looks like that for **`Maybe`**, we've figured out how to take a fancy value and feed it to a function that takes a normal value and returns a fancy one. We did this by keeping in mind that a **`Maybe`** value represents a computation that might have failed.
You might be asking yourself, how is this useful? It may seem like applicative functors are stronger than monads, since applicative functors allow us to take a normal function and make it operate on values with contexts. We'll see that monads can do that as well because they're an upgrade of applicative functors, and that they can also do some cool stuff that applicative functors can't.

We'll come back to **`Maybe`** in a minute, but first, let's check out the type class that belongs to monads.

予想通りだ。左側のモナド値が Nothing であれば、全体が Nothing になる。そして、右側の関数が Nothing を返せば、結果はまた Nothing だ。これは、Maybe をアプリケーティブとして使ったときに、どこかに Nothing があれば Nothing の結果が出たのとよく似ている。
Maybe では、派手な値を受け取って、それを普通の値を受け取って派手な値を返す関数に渡す方法がわかったようだ。Maybe の値は、失敗したかもしれない計算を表しているということを念頭に置いて、これを実行した。
これはどう便利なのだろう？というのも、アプリケーティブファンクタを使えば、通常の関数をコンテキストを持つ値に対して操作することができるからです。モナドはアプリケイティブ・ファンクターのアップグレード版なので、モナドにもそれができるし、アプリケイティブ・ファンクターにはできないクールなこともできる。
その前に、モナドに属する型クラスを見てみよう。

## The Monad type class

Just like functors have the **`Functor`** type class and applicative functors have the **`Applicative`** type class, monads come with their own type class: **`Monad`**! Wow, who would have thought? This is what the type class looks like:

ファンクターにファンクター型クラスがあり、アプリケーティブ・ファンクターにアプリケーティブ型クラスがあるように、モナドにも独自の型クラスがある：モナドだ！すごい！誰が想像できただろう？型クラスはこんな感じだ:

```
class Monad m where
    return :: a -> m a

    (>>=) :: m a -> (a -> m b) -> m b

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

```
7.      x >> y = x >>= ¥_ -> y
8.
9.      fail :: String -> m a
10.     fail msg = error msg
```

Let's start with the first line. It says `class Monad m where`. But wait, didn't we say that monads are just beefed up applicative functors? Shouldn't there be a class constraint in there along the lines of `class (Applicative m) = > Monad m where` so that a type has to be an applicative functor first before it can be made a monad? Well, there should, but when Haskell was made, it hadn't occured to people that applicative functors are a good fit for Haskell so they weren't in there. But rest assured, every monad is an applicative functor, even if the `Monad` class declaration doesn't say so.
The first function that the `Monad` type class defines is `return`. It's the same as `pure`, only with a different name. Its type is `(Monad m) => a -> m a`. It takes a value and puts it in a minimal default context that still holds that value. In other words, it takes something and wraps it in a monad. It always does the same thing as the `pure` function from the `Applicative` type class, which means we're already acquainted with `return`. We already used `return` when doing I/O. We used it to take a value and make a bogus I/O action that does nothing but yield that value. For `Maybe` it takes a value and wraps it in a `Just`.
Just a reminder: `return` is nothing like the `return` that's in most other languages. It doesn't end function execution or anything, it just takes a normal value and puts it in a context.

最初の行から見ていこう。ここにはクラス・モナド m と書かれている。しかし、モナドとはアプリケーティブ・ファンクターを強化したものだと言わなかったか？クラス（Applicative m）= > Monad m where のようなクラス制約があって、型がモナドになる前にまずアプリケーティブ・ファンクターである必要があるのでは？まあ、あるはずなんだけど、Haskell が作られたときには、アプリケーティブ・ファンクターが Haskell に適しているなんて思いもよらなかったから、入っていなかったんだ。しかし、モナド・クラスの宣言にそう書かれていなくても、すべてのモナドはアプリケーティブ・ファンクターなのでご安心を。
モナド型クラスが定義する最初の関数は return だ。これは名前が違うだけで pure と同じだ。この関数は値を受け取り、その値を保持する最小限のデフォルト・コンテキストに置く。つまり、何かをモナドに包むのだ。これは常に Applicative 型クラスの純粋関数と同じことをする。return は I/O のときにすでに使った。ある値を受け取って、その値を返すだけのインチキ I/O アクションを作るために使ったのだ。Maybe の場合は、値を受け取ってそれを Just で包む。
注意：return は他の言語にある return とは違う。関数の実行を終了するわけでも何でもなく、通常の値を受け取ってそれをコンテキストに置くだけだ。

The next function is `>>=`, or bind. It's like function application, only instead of taking a normal value and feeding it to a normal function, it takes a monadic value (that is, a value with a context) and feeds it to a function that takes a normal value but returns a monadic value.
Next up, we have `>>`. We won't pay too much attention to it for now because it comes with a default implementation and we pretty much never implement it when making `Monad` instances.
The final function of the `Monad` type class is `fail`. We never use it explicitly in our code. Instead, it's used by Haskell to enable failure in a special syntactic construct for monads that we'll meet later. We don't need to concern ourselves with `fail` too much for now.

Now that we know what the **Monad** type class looks like, let's take a look at how **Maybe** is an instance of **Monad**!

次の関数は>>=（バインド）だ。これは関数の応用のようなもので、通常の値を受け取って通常の関数に渡す代わりに、モナド値（つまりコンテキストを持つ値）を受け取り、通常の値を受け取ってモナド値を返す関数に渡します。
次は>>だ。これはデフォルトの実装であり、モナドのインスタンスを作るときに実装することはほとんどないからだ。
モナド型クラスの最後の関数は fail だ。私たちのコードでは明示的に使うことはない。その代わり、後で説明するモナドのための特別な構文で失敗を可能にするために Haskell によって使われる。今は fail をあまり気にする必要はない。
モナド型クラスがどのようなものかわかったところで、Maybe がどのようにモナドのインスタンスになるのかを見てみよう！

```
instance Monad Maybe where
    return x = Just x
    Nothing >>= f = Nothing
    Just x >>= f  = f x
    fail _ = Nothing
```

**return** is the same as **pure**, so that one's a no-brainer. We do what we did in the **Applicative** type class and wrap it in a **Just**.
The **>>=** function is the same as our **applyMaybe**. When feeding the **Maybe a** to our function, we keep in mind the context and return a **Nothing** if the value on the left is **Nothing** because if there's no value then there's no way to apply our function to it. If it's a **Just** we take what's inside and apply **f** to it.
We can play around with **Maybe** as a monad:

return は pure と同じである。Applicative 型クラスでやったように、それを Just で包みます。
関数 >>= は applyMaybe と同じだ。Maybe a を関数に渡すとき、コンテキストを考慮し、左の値が Nothing の場合は Nothing を返す。もし Just であれば、その中にあるものを受け取り、それに f を適用する。
モナドとして Maybe を使って遊ぶことができる：

```
ghci> return "WHAT" :: Maybe String
Just "WHAT"
ghci> Just 9 >>= ¥x -> return (x*10)
Just 90
ghci> Nothing >>= ¥x -> return (x*10)
Nothing
```

Nothing new or exciting on the first line since we already used **pure** with **Maybe** and we know that **return** is just **pure** with a different name. The next two lines showcase **>>=** a bit more.
Notice how when we fed **Just 9** to the function **¥x -> return (x*10)**, the **x** took on the value **9** inside the function. It seems as though we were able to extract the value from a **Maybe** without pattern-matching. And we still didn't lose the context of our **Maybe** value, because when it's **Nothing**, the result of using **>>=** will be **Nothing** as well.

最初の行では、Maybe ですでに pure を使っているし、return は pure に別の名前をつけただけのものだとわかっているからだ。次の 2 行では >>= をもう少し詳しく説明する。
ちょうど 9 を関数に与えたとき、関数内で x が値 9 になったことに注目してください。パターンマッチングなしで Maybe から値を取り出すことができたようだ。Nothing の場合、>>=を使った結果も Nothing になるからだ。

## Walk the line

Now that we know how to feed a `Maybe a` value to a function of type `a -> Maybe b` while taking into account the context of possible failure, let's see how we can use `>>=` repeatedly to handle computations of several `Maybe a` values.
Pierre has decided to take a break from his job at the fish farm and try tightrope walking. He's not that bad at it, but he does have one problem: birds keep landing on his balancing pole! They come and they take a short rest, chat with their avian friends and then take off in search of breadcrumbs. This wouldn't bother him so much if the number of birds on the left side of the pole was always equal to the number of birds on the right side. But sometimes, all the birds decide that they like one side better and so they throw him off balance, which results in an embarrassing tumble for Pierre (he's using a safety net).

Let's say that he keeps his balance if the number of birds on the left side of the pole and on the right side of the pole is within three. So if there's one bird on the right side and four birds on the left side, he's okay. But if a fifth bird lands on the left side, then he loses his balance and takes a dive.

We're going to simulate birds landing on and flying away from the pole and see if Pierre is still at it after a certain number of birdy arrivals and departures. For instance, we want to see what happens to Pierre if first one bird arrives on the left side, then four birds occupy the right side and then the bird that was on the left side decides to fly away.

We can represent the pole with a simple pair of integers. The first component will signify the number of birds on the left side and the second component the number of birds on the right side:

失敗する可能性のあるコンテキストを考慮しながら、a -> Maybe b 型の関数に Maybe a 値を与える方法がわかったので、複数の Maybe a 値の計算を処理するために>>=を繰り返し使う方法を見てみよう。

ピエールは養魚場の仕事を休んで綱渡りに挑戦することにした。彼はそれほど下手ではないが、ひとつ問題がある！鳥たちはやってきては少し休憩し、鳥仲間とおしゃべりをした後、パンくずを探して飛び立つ。ポールの左側にいる鳥の数と右側にいる鳥の数がいつも同じであれば、それほど気にならないだろう。しかし時々、すべての鳥が片方の方が好きだと判断し、ピエールのバランスを崩してしまう。

ポールの左側にいる鳥と右側にいる鳥の数が 3 羽以内ならバランスを保つとしよう。つまり、右側に 1 羽、左側に 4 羽の鳥がいれば大丈夫だ。しかし、5 羽目が左側に着地すると、バランスを崩して急降下する。

鳥がポールに着地したり、ポールから離れたりするのをシミュレートし、ある数の鳥が着地したり離れたりした後、ピエールがまだやっているかどうかを確認するつもりだ。例えば、まず 1 羽の鳥が左側に到着し、次に 4 羽の鳥が右側を占め、そして左側にいた鳥が飛び去ることにした場合、ピエールがどうなるかを見たい。

極は単純な整数の組で表すことができる。最初の要素は左側の鳥の数を表し、2 番目の要素は右側の鳥の数を表す：

```
type Birds = Int
type Pole = (Birds,Birds)
```

First we made a type synonym for `Int`, called `Birds`, because we're using integers to represent how many birds there are. And then we made a type synonym `(Birds,Birds)` and we called it `Pole` (not to be confused with a person of Polish descent).
Next up, how about we make a function that takes a number of birds and lands them on one side of the pole. Here are the functions:

まず Int のシノニムとして Birds という型を作った。そして同義語（Birds,Birds）を作り、それを Pole（ポーランド系の人と混同しないように）と呼んだ。

次に、鳥の数を受け取り、それらをポールの片側に着陸させる関数を作ってみよう。以下がその関数である：

```
landLeft :: Birds -> Pole -> Pole
landLeft n (left,right) = (left + n,right)

landRight :: Birds -> Pole -> Pole
landRight n (left,right) = (left,right + n)
```

Pretty straightforward stuff. Let's try them out:

とても簡単なことだ。試してみよう：

```
ghci> landLeft 2 (0,0)
(2,0)
ghci> landRight 1 (1,2)
(1,3)
ghci> landRight (-1) (1,2)
(1,1)
```

To make birds fly away we just had a negative number of birds land on one side. Because landing a bird on the `Pole` returns a `Pole`, we can chain applications of `landLeft` and `landRight`:

鳥を飛び立たせるためには、負の数の鳥を片側に着陸させればよい。Pole に鳥を着陸させると Pole を返すので、landLeft と landRight のアプリケーションを連鎖させることができる：

```
ghci> landLeft 2 (landRight 1 (landLeft 1 (0,0)))
(3,1)
```

When we apply the function `landLeft 1` to `(0,0)` we get `(1,0)`. Then, we land a bird on the right side, resulting in `(1,1)`. Finally two birds land on the left side, resulting in `(3,1)`. We apply a function to something by first writing the function and then writing its parameter, but here it would be better if the pole went first and then the landing function. If we make a function like this:

関数 landLeft 1 を(0,0)に適用すると(1,0)となる。次に右側に鳥を 1 羽着地させると(1,1)となる。最後に 2 羽の鳥が左側に着地し、(3,1)となる。関数を何かに適用するには、まず関数を書き、次にそのパラメータを書く。このように関数を作ると

```
x -: f = f x
```

We can apply functions by first writing the parameter and then the function:

関数を適用するには、まずパラメータを記述し、次に関数を記述する：

```
ghci> 100 -: (*3)
300
ghci> True -: not
False
ghci> (0,0) -: landLeft 2
(2,0)
```

By using this, we can repeatedly land birds on the pole in a more readable manner:

これを使うことで、より読みやすくポールへの着地を繰り返すことができる：

```
ghci> (0,0) -: landLeft 1 -: landRight 1 -: landLeft 2
(3,1)
```

Pretty cool! This example is equivalent to the one before where we repeatedly landed birds on the pole, only it looks neater. Here, it's more obvious that we start off with `(0,0)` and then land one bird one the left, then one on the right and finally two on the left.
So far so good, but what happens if 10 birds land on one side?

なかなかクールだ！この例は、鳥をポールに繰り返し着地させる前の例と同じだが、見た目がよりすっきりしている。ここでは、(0,0)から始めて、左側に 1 羽、右側に 1 羽、最後に左側に 2 羽を着地させることがより明確になっている。

ここまではいいのですが、10 羽が片側に着地したらどうなるでしょうか？

```
ghci> landLeft 10 (0,3)
(10,3)
```

10 birds on the left side and only 3 on the right? That's sure to send poor Pierre falling through the air! This is pretty obvious here but what if we had a sequence of landings like this:

左側に 10 羽、右側に 3 羽だけ？これではかわいそうなピエールが宙を舞うのは間違いない！これはかなり明白なことだが、もしこのような着地の連続があったとしたらどうだろう：

```
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

It might seem like everything is okay but if you follow the steps here, you'll see that at one time there are 4 birds on the right side and no birds on the left! To fix this, we have to take another look at our `landLeft` and `landRight` functions. From what we've seen, we want these functions to be able to fail. That is, we want them to return a new pole if the balance is okay but fail if the birds land in a lopsided manner. And what better way to add a context of failure to value than by using `Maybe`! Let's rework these functions:

一見すべてがうまくいっているように見えるかもしれないが、ここのステップをたどると、あるとき右側に 4 羽の鳥がいて、左側には鳥がいないことがわかるだろう！これを修正するには、landLeft 関数と landRight 関数をもう一度見てみる必要があります。今まで見てきたところでは、これらの関数は失敗できるようにしたい。つまり、バランスに問題がなければ新しいポールを返すが、鳥が偏って着地した場合は失敗するようにしたい。そして、Maybe を使うこと以上に、失敗というコンテクストを価値に加える良い方法があるだろうか！これらの関数を作り直そう：

```
landLeft :: Birds -> Pole -> Maybe Pole
landLeft n (left,right)
    | abs ((left + n) - right) < 4 = Just (left + n, right)
    | otherwise                    = Nothing

landRight :: Birds -> Pole -> Maybe Pole
landRight n (left,right)
    | abs (left - (right + n)) < 4 = Just (left, right + n)
    | otherwise                    = Nothing
```

Instead of returning a `Pole` these functions now return a `Maybe Pole`. They still take the number of birds and the old pole as before, but then they check if landing that many birds on the pole would throw Pierre off balance. We use guards to check if the difference between the number of birds on the new pole is less than 4. If it is, we wrap the new pole in a `Just` and return that. If it isn't, we return a `Nothing`, indicating failure.
Let's give these babies a go:

これらの関数は Pole を返す代わりに Maybe Pole を返すようになった。前と同じように鳥の数と古いポールを受け取るが、ポールにそれだけの数の鳥を着地させるとピエールのバランスが崩れるかどうかをチェックする。もしそうなら、新しいポールを Just で包んでそれを返す。もしそうでなければ、失敗を示す Nothing を返す。

では、この子たちを試してみよう：

```
ghci> landLeft 2 (0,0)
Just (2,0)
ghci> landLeft 10 (0,3)
```

```
4. Nothing
```

Nice! When we land birds without throwing Pierre off balance, we get a new pole wrapped in a `Just`. But when many more birds end up on one side of the pole, we get a `Nothing`. This is cool, but we seem to have lost the ability to repeatedly land birds on the pole. We can't do `landLeft 1 (landRight 1 (0,0))` anymore because when we apply `landRight 1` to `(0,0)`, we don't get a `Pole`, but a `Maybe Pole`. `landLeft 1` takes a `Pole` and not a `Maybe Pole`.

We need a way of taking a `Maybe Pole` and feeding it to a function that takes a `Pole` and returns a `Maybe Pole`. Luckily, we have `>>=`, which does just that for `Maybe`. Let's give it a go:

いいね！ピエールのバランスを崩さずに鳥をランディングさせると、ジャストに包まれた新しいポールが手に入る。しかし、多くの鳥が竿の片側に止まると、Nothing になる。これはクールだが、ポールに鳥を繰り返し着地させる機能は失われているようだ。landLeft 1 (landRight 1 (0,0))はもうできない。landRight 1 を(0,0)に適用すると、Pole ではなく Maybe Pole が得られるからだ。

Maybe Pole を受け取り、それを Pole を受け取って Maybe Pole を返す関数に渡す方法が必要だ。幸運なことに、私たちには >>= がある。やってみよう:

```
1. ghci> landRight 1 (0,0) >>= landLeft 2
2. Just (2,1)
```

Remember, `landLeft 2` has a type of `Pole -> Maybe Pole`. We couldn't just feed it the `Maybe Pole` that is the result of `landRight 1 (0,0)`, so we use `>>=` to take that value with a context and give it to `landLeft 2`. `>>=` does indeed allow us to treat the `Maybe` value as a value with context because if we feed a `Nothing` into `landLeft 2`, the result is `Nothing` and the failure is propagated:

landLeft 2 は Pole -> Maybe Pole という型を持っていることを思い出してほしい。landRight 1 の結果（0,0）である Maybe Pole をそのまま landLeft 2 に与えるわけにはいかないので、>>=を使ってコンテキスト付きの値を landLeft 2 に与える。>>= landLeft2 に Nothing を入力すると、結果は Nothing となり、失敗が伝播されるからだ:

```
1. ghci> Nothing >>= landLeft 2
2. Nothing
```

With this, we can now chain landings that may fail because `>>=` allows us to feed a monadic value to a function that takes a normal one.

Here's a sequence of birdy landings:

これによって、失敗する可能性のある着地を連鎖させることができる。なぜなら、>>=によって、通常の値を受け取る関数にモナド値を与えることができるからだ。

以下は鳥のような着陸の連続である:

```
1. ghci> return (0,0) >>= landRight 2 >>= landLeft 2 >>= landRight 2
2. Just (2,4)
```

At the beginning, we used `return` to take a pole and wrap it in a `Just`. We could have just applied `landRight 2` to `(0,0)`, it would have been the same, but this way we can be more consistent by using `>>=` for every

function. `Just (0,0)` gets fed to `landRight 2`, resulting in `Just (0,2)`. This, in turn, gets fed to `landLeft 2`, resulting in `Just (2,2)`, and so on.
Remember this example from before we introduced failure into Pierre's routine:

冒頭では、return を使ってポールを取り出し、それを Just で囲んだ。ただ landRight 2 を(0,0)に適用しても同じだったが、こうしてすべての関数に>>=を使うことで、より一貫性を持たせることができる。ジャスト(0,0)はランドライト 2 に送られ、ジャスト(0,2)になる。これが今度は landLeft 2 に送られ、Just (2,2)となる。

ピエールのルーチンに失敗を導入する前のこの例を思い出してほしい：

```
ghci> (0,0) -: landLeft 1 -: landRight 4 -: landLeft (-1) -: landRight (-2)
(0,2)
```

It didn't simulate his interaction with birds very well because in the middle there his balance was off but the result didn't reflect that. But let's give that a go now by using monadic application (`>>=`) instead of normal application:

というのも、途中で彼のバランスが崩れていたのだが、結果はそれを反映していなかったからだ。しかし、通常のアプリケーションの代わりにモナド・アプリケーション（>>=）を使うことで、それを試してみよう：

```
ghci> return (0,0) >>= landLeft 1 >>= landRight 4 >>= landLeft (-1) >>= landRight (-
2)
Nothing
```

Awesome. The final result represents failure, which is what we expected. Let's see how this result was obtained. First, `return` puts `(0,0)` into a default context, making it a `Just (0,0)`. Then, `Just (0,0) >>= landLeft 1` happens. Since the `Just (0,0)` is a `Just` value, `landLeft 1` gets applied to `(0,0)`, resulting in a `Just (1,0)`, because the birds are still relatively balanced. Next, `Just (1,0) >>= landRight 4` takes place and the result is `Just (1,4)` as the balance of the birds is still intact, although just barely. `Just (1,4)` gets fed to `landLeft (-1)`. This means that `landLeft (-1) (1,4)` takes place. Now because of how `landLeft` works, this results in a `Nothing`, because the resulting pole is off balance. Now that we have a `Nothing`, it gets fed to `landRight (-2)`, but because it's a `Nothing`, the result is automatically `Nothing`, as we have nothing to apply `landRight (-2)` to.
We couldn't have achieved this by just using `Maybe` as an applicative. If you try it, you'll get stuck, because applicative functors don't allow for the applicative values to interact with each other very much. They can, at best, be used as parameters to a function by using the applicative style. The applicative operators will fetch their results and feed them to the function in a manner appropriate for each applicative and then put the final applicative value together, but there isn't that much interaction going on between them. Here, however, each step relies on the previous one's result. On every landing, the possible result from the previous one is examined and the pole is checked for balance. This determines whether the landing will succeed or fail.
We may also devise a function that ignores the current number of birds on the balancing pole and just makes Pierre slip and fall. We can call it `banana`:

素晴らしい。最終的な結果は失敗を表している。この結果がどのようにして得られたかを見てみよう。まず、return は(0,0)をデフォルトのコンテキストに入れ、Just (0,0)にする。そして、Just (0,0) >>= landLeft 1 が起こる。ジャスト(0,0)は

ジャスト値なので、landLeft 1 が(0,0)に適用され、鳥はまだ比較的バランスが取れているので、ジャスト(1,0)になります。次に、Just (1,0) >>= landRight 4 が行われ、結果は Just (1,4)となる。Just (1,4)は landLeft (-1)に送られる。つまり、landLeft (-1) (1,4)が行われる。landLeft がどのように機能するかによって、結果的にポールのバランスが崩れるため、この結果は Nothing になる。Nothing になったので、それを landRight (-2)に送るが、Nothing なので、landRight (-2)を適用するものがないので、結果は自動的に Nothing になる。
Maybe をアプリケーティブとして使うだけでは、これを実現することはできなかった。なぜなら、アプリケーティブ・ファンクターではアプリケーティブの値同士があまり相互作用できないからです。せいぜい、アプリケーティブスタイルを使って関数のパラメータとして使うことができる程度です。アプリケ-ション演算子は、それぞれのアプリケ-ションに適した方法で結果をフェッチして関数に送り、最終的なアプリケ-ションの値をまとめますが、その間にそれほど多くの相互作用があるわけではありません。しかし、ここでは、それぞれのステップは前のステップの結果に依存します。着地のたびに、前のステップの結果が調べられ、ポールのバランスがチェックされる。これによって着地が成功するか失敗するかが決まる。
バランスポールの上にいる現在の鳥の数を無視して、ピエールを滑らせて落下させるだけの関数を考案することもできる。それをバナナと呼ぶこともできる:

```
banana :: Pole -> Maybe Pole
banana _ = Nothing
```

Now we can chain it together with our bird landings. It will always cause our walker to fall, because it ignores whatever's passed to it and always returns a failure. Check it:

これで鳥の着地と連鎖させることができる。ウォーカーに渡されたものはすべて無視され、常に失敗を返すからだ。確認してみよう:

```
ghci> return (0,0) >>= landLeft 1 >>= banana >>= landRight 1
Nothing
```

The value `Just (1,0)` gets fed to `banana`, but it produces a `Nothing`, which causes everything to result in a `Nothing`. How unfortunate!
Instead of making functions that ignore their input and just return a predetermined monadic value, we can use the `>>` function, whose default implementation is this:

値 Just (1,0)が banana に入力されるが、Nothing を生成する。なんと不幸なことだろう！
入力を無視して決められたモナド値を返すだけの関数を作る代わりに、デフォルトの実装がこうなっている>>関数を使うことができる:

```
(>>) :: (Monad m) => m a -> m b -> m b
m >> n = m >>= ¥_ -> n
```

Normally, passing some value to a function that ignores its parameter and always just returns some predetermined value would always result in that predetermined value. With monads however, their context and meaning has to be considered as well. Here's how `>>` acts with `Maybe`:

通常、パラメータを無視して常に決められた値を返す関数に何らかの値を渡すと、常にその決められた値が返されることになる。しかし、モナドの場合は、その文脈と意味も考慮しなければならない。ここでは、>>が Maybe でどのように振る舞うかを説明する:

```
ghci> Nothing >> Just 3
Nothing
ghci> Just 3 >> Just 4
Just 4
ghci> Just 3 >> Nothing
Nothing
```

If you replace `>>` with `>>= ¥_ ->`, it's easy to see why it acts like it does.
We can replace our `banana` function in the chain with a `>>` and then a `Nothing`:

を >>= ¥_ -> に置き換えてみると、なぜそのように動作するのかが簡単にわかる。
鎖の中のバナナ関数を >>に、そして Nothing に置き換えることができる:

```
ghci> return (0,0) >>= landLeft 1 >> Nothing >>= landRight 1
Nothing
```

There we go, guaranteed and obvious failure!

It's also worth taking a look at what this would look like if we hadn't made the clever choice of treating `Maybe` values as values with a failure context and feeding them to functions like we did. Here's how a series of bird landings would look like:

ほら、明らかな失敗だ！
もし、Maybe 値を失敗のコンテキストを持つ値として扱い、関数に与えるという賢い選択をしていなかったら、これがどうなっていたかを見てみる価値もある。一連の鳥の着陸がどのようになるかを見てみよう:

```
routine :: Maybe Pole
routine = case landLeft 1 (0,0) of
    Nothing -> Nothing
    Just pole1 -> case landRight 4 pole1 of
        Nothing -> Nothing
        Just pole2 -> case landLeft 2 pole2 of
            Nothing -> Nothing
            Just pole3 -> landLeft 1 pole3
```

We land a bird on the left and then we examine the possibility of failure and the possibility of success. In the case of failure, we return a `Nothing`. In the case of success, we land birds on the right and then do the same thing all over again. Converting this monstrosity into a neat chain of monadic applications with `>>=` is a classic example of how the `Maybe` monad saves us a lot of time when we have to successively do computations that are based on computations that might have failed.
Notice how the `Maybe` implementation of `>>=` features exactly this logic of seeing if a value is `Nothing` and if it is, returning a `Nothing` right away and if it isn't, going forward with what's inside the `Just`.
In this section, we took some functions that we had and saw that they would work better if the values that they returned supported failure. By turning those values into `Maybe` values and replacing normal function application with `>>=`, we got a mechanism for handling failure pretty much for free, because `>>=` is supposed to preserve the

context of the value to which it applies functions. In this case, the context was that our values were values with failure and so when we applied functions to such values, the possibility of failure was always taken into account.

左の鳥を着地させ、失敗の可能性と成功の可能性を調べる。失敗の場合は Nothing を返す。成功した場合は、右側に鳥を着陸させ、同じことを繰り返す。この怪物を >>= を使ってモナド・アプリケーションのきれいな連鎖に変換することは、失敗したかもしれない計算を基にした計算を連続して行わなければならないときに、Maybe モナドがいかに多くの時間を節約してくれるかを示す典型的な例である。
Maybe 実装の>>=が、値が Nothing かどうかを確認し、Nothing であればすぐに Nothing を返し、Nothing でなければ Just の中身を処理するという、まさにこのロジックを備えていることに注目してほしい。
このセクションでは、私たちが持っているいくつかの関数を取り上げ、それらが返す値が失敗をサポートしていれば、よりうまく機能することを確認した。これらの値を Maybe 値に変え、通常の関数の適用を>>=に置き換えることで、失敗を処理するメカニズムが無料で手に入った。>>=は関数を適用する値のコンテキストを保持することになっているからだ。というのも、>>=は関数を適用する値のコンテキストを保持することになっているからだ。この場合、コンテキストとは、私たちの値が失敗を伴う値であるということであり、そのような値に関数を適用する際には、失敗の可能性が常に考慮されることになる。