do notation

Monads in Haskell are so useful that they got their own special syntax called `do` notation. We've already encountered `do` notation when we were doing I/O and there we said that it was for gluing together several I/O actions into one. Well, as it turns out, `do` notation isn't just for `IO`, but can be used for any monad. Its principle is still the same: gluing together monadic values in sequence. We're going to take a look at how `do` notation works and why it's useful.
Consider this familiar example of monadic application:

Haskell のモナドはとても便利なので、do 記法と呼ばれる独自の特別な構文を持つようになった。do 記法はすでに I/O のときに使ったことがあるが、そのときはいくつかの I/O アクションをひとつにまとめるためのものだと言った。しかし、do 記法は I/O だけでなく、どんなモナドにも使えることがわかった。その原理は同じで、モナドの値を順番につなぎ合わせることだ。do 記法がどのように機能し、なぜ便利なのかを見ていこう。

身近なモナド応用例を考えてみよう:

```
ghci> Just 3 >>= (¥x -> Just (show x ++ "!"))
Just "3!"
```

Been there, done that. Feeding a monadic value to a function that returns one, no big deal. Notice how when we do this, `x` becomes `3` inside the lambda. Once we're inside that lambda, it's just a normal value rather than a monadic value. Now, what if we had another `>>=` inside that function? Check this out:

経験済みだ。を返す関数にモナド値を与えても、大したことはない。これを実行すると、ラムダの中では x が 3 になることに注目してほしい。ラムダの中に入ってしまえば、それはモナド値ではなく普通の値だ。では、もしこの関数の中に別の >>=があったらどうなるだろうか？これをチェックしてみよう:

```
ghci> Just 3 >>= (¥x -> Just "!" >>= (¥y -> Just (show x ++ y)))
Just "3!"
```

Ah, a nested use of `>>=`! In the outermost lambda, we feed `Just "!"` to the lambda `¥y -> Just (show x ++ y)`. Inside this lambda, the `y` becomes `"!"`. `x` is still `3` because we got it from the outer lambda. All this sort of reminds me of the following expression:

ああ、>>=の入れ子使用だ！一番外側のラムダで、Just "!"を Just (show x ++ y)というラムダに渡している。このラムダの中で、y は"!"になる。このようなことから、次のような式を思い出す:

```
ghci> let x = 3; y = "!" in show x ++ y
"3!"
```

The main difference between these two is that the values in the former example are monadic. They're values with a failure context. We can replace any of them with a failure:

この 2 つの主な違いは、前者の例の値がモナドであることだ。失敗のコンテキストを持つ値なのだ。そのいずれかを失敗で置き換えることができる:

```
ghci> Nothing >>= (¥x -> Just "!" >>= (¥y -> Just (show x ++ y)))
Nothing
```

```
ghci> Just 3 >>= (¥x -> Nothing >>= (¥y -> Just (show x ++ y)))
Nothing
ghci> Just 3 >>= (¥x -> Just "!" >>= (¥y -> Nothing))
Nothing
```

In the first line, feeding a `Nothing` to a function naturally results in a `Nothing`. In the second line, we feed `Just 3` to a function and the `x` becomes `3`, but then we feed a `Nothing` to the inner lambda and the result of that is `Nothing`, which causes the outer lambda to produce `Nothing` as well. So this is sort of like assigning values to variables in `let` expressions, only that the values in question are monadic values.
To further illustrate this point, let's write this in a script and have each `Maybe` value take up its own line:

最初の行では、関数に Nothing を与えると当然 Nothing になる。2 行目では、関数に Just 3 を与えると x は 3 になるが、次に内側のラムダに Nothing を与えると、その結果は Nothing になり、外側のラムダも同様に Nothing を生成する。つまり、これは let 式で変数に値を代入するようなもので、問題の値がモナド値であるということだけだ。
この点をさらに説明するために、これをスクリプトに書いて、それぞれの Maybe 値がそれ自身の行を占めるようにしてみよう:

```
foo :: Maybe String
foo = Just 3   >>= (¥x ->
      Just "!" >>= (¥y ->
      Just (show x ++ y)))
```

To save us from writing all these annoying lambdas, Haskell gives us `do` notation. It allows us to write the previous piece of code like this:

このような煩わしいラムダを書く手間を省くために、Haskell は do 記法を提供している。これにより、先ほどのコードをこのように書くことができる:

```
foo :: Maybe String
foo = do
    x <- Just 3
    y <- Just "!"
    Just (show x ++ y)
```

It would seem as though we've gained the ability to temporarily extract things from `Maybe` values without having to check if the `Maybe` values are `Just` values or `Nothing` values at every step. How cool! If any of the values that we try to extract from are `Nothing`, the whole `do` expression will result in a `Nothing`. We're yanking out their (possibly existing) values and letting `>>=` worry about the context that comes with those values. It's important to remember that `do` expressions are just different syntax for chaining monadic values.
In a `do` expression, every line is a monadic value. To inspect its result, we use `<-`. If we have a `Maybe String` and we bind it with `<-` to a variable, that variable will be a `String`, just like when we used `>>=` to feed monadic values to lambdas. The last monadic value in a `do` expression, like `Just (show x ++ y)` here, can't be used with `<-` to bind its result, because that wouldn't make sense if we translated the `do` expression back to a chain of `>>=` applications. Rather, its result is the result of the whole glued up monadic value, taking into account the possible failure of any of the previous ones.
For instance, examine the following line:

Maybe 値が Just 値なのか Nothing 値なのかをいちいちチェックすることなく、Maybe 値から一時的に何かを抽出できるようになったようだ。なんともクールだ！取り出そうとする値のどれかが Nothing であれば、do 式全体が Nothing という結果になる。既存の）値を取り出して、その値に付随するコンテキストを >>= に心配させているのだ。do 式は、モナド値を連鎖させるための異なる構文であることを覚えておくことが重要だ。

do 式では、すべての行がモナド値である。その結果を調べるには、<-を使う。Maybe 文字列があり、それを<-で変数にバインドすると、その変数は文字列となる。do 式の最後のモナド値は、例えば Just (show x ++ y)のように、<-を使ってその結果をバインドすることはできない。というのも、do 式を>>=アプリケーションの連鎖に戻したのでは意味がないからだ。そうではなく、その結果は、前のどれかが失敗する可能性を考慮に入れて、糊付けされたモナド値全体の結果となる。

例えば、次の行を見てみよう:

```
ghci> Just 9 >>= (¥x -> Just (x > 8))
Just True
```

Because the left parameter of `>>=` is a `Just` value, the lambda is applied to `9` and the result is a `Just True`. If we rewrite this in `do` notation, we get:

の左パラメータは Just 値なので、ラムダは 9 に適用され、結果は Just True となる。これを do 記法で書き直すとこうなる:

```
marySue :: Maybe Bool
marySue = do
    x <- Just 9
    Just (x > 8)
```

If we compare these two, it's easy to see why the result of the whole monadic value is the result of the last monadic value in the `do` expression with all the previous ones chained into it.
Our tightwalker's routine can also be expressed with `do` notation. `landLeft` and `landRight` take a number of birds and a pole and produce a pole wrapped in a `Just`, unless the tightwalker slips, in which case a `Nothing` is produced. We used `>>=` to chain successive steps because each one relied on the previous one and each one had an added context of possible failure. Here's two birds landing on the left side, then two birds landing on the right and then one bird landing on the left:

この 2 つを比較すれば、なぜモナド全体の値が、do 式の最後のモナドに前のモナドをすべて連鎖させた結果になるのかが簡単にわかる。
landLeft と landRight は鳥の数とポールを受け取り、Just に包まれたポールを生成する。我々は連続したステップを連鎖させるために >>= を使ったが、これは各ステップが前のステップに依存しており、各ステップには失敗の可能性があるというコンテキストが追加されていたからである。ここで 2 羽が左側に着地し、次に 2 羽が右側に着地し、そして 1 羽が左側に着地する:

```
routine :: Maybe Pole
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
```

```
    second <- landRight 2 first
    landLeft 1 second
```

Let's see if he succeeds:

彼が成功するかどうか見てみよう:

```
ghci> routine
Just (3,2)
```

He does! Great. When we were doing these routines by explicitly writing `>>=`, we usually said something like `return (0,0) >>= landLeft 2`, because `landLeft 2` is a function that returns a `Maybe` value. With `do` expressions however, each line must feature a monadic value. So we explicitly pass the previous `Pole` to the `landLeft` `landRight` functions. If we examined the variables to which we bound our `Maybe` values, `start` would be `(0,0)`, `first` would be `(2,0)` and so on.
Because `do` expressions are written line by line, they may look like imperative code to some people. But the thing is, they're just sequential, as each value in each line relies on the result of the previous ones, along with their contexts (in this case, whether they succeeded or failed).
Again, let's take a look at what this piece of code would look like if we hadn't used the monadic aspects of `Maybe`:

彼はそうする！すごい。というのも、landLeft 2 は Maybe 値を返す関数だからだ。しかし do 式では、各行にモナド値を記述しなければならない。そこで、landLeft landRight 関数に前の Pole を明示的に渡している。Maybe 値をバインドする変数を調べると、start は(0,0)、first は(2,0)となる。
do 式は一行ずつ書かれているため、人によっては命令形のコードに見えるかもしれない。各行の各値は、前の値の結果とそのコンテキスト（この場合は成功か失敗か）に依存しているからだ。
もう一度、Maybe のモナド的な側面を使わなかった場合、このコードがどのようになるかを見てみよう:

```
routine :: Maybe Pole
routine =
    case Just (0,0) of
        Nothing -> Nothing
        Just start -> case landLeft 2 start of
            Nothing -> Nothing
            Just first -> case landRight 2 first of
                Nothing -> Nothing
                Just second -> landLeft 1 second
```

See how in the case of success, the tuple inside `Just (0,0)` becomes `start`, the result of `landLeft 2 start` becomes `first`, etc.
If we want to throw the Pierre a banana peel in `do` notation, we can do the following:

成功した場合、Just (0,0)の中のタプルが start になり、landLeft 2 start の結果が first になるなど、どのようになるか見てみよう。
ピエールにバナナの皮を do 記法で投げたい場合は、次のようにすればいい:

```
routine :: Maybe Pole
```

```
routine = do
    start <- return (0,0)
    first <- landLeft 2 start
    Nothing
    second <- landRight 2 first
    landLeft 1 second
```

When we write a line in `do` notation without binding the monadic value with `<-`, it's just like putting `>>` after the monadic value whose result we want to ignore. We sequence the monadic value but we ignore its result because we don't care what it is and it's prettier than writing `_ <- Nothing`, which is equivalent to the above. When to use `do` notation and when to explicitly use `>>=` is up to you. I think this example lends itself to explicitly writing `>>=` because each step relies specifically on the result of the previous one. With `do` notation, we had to specifically write on which pole the birds are landing, but every time we used that came directly before. But still, it gave us some insight into `do` notation.

In `do` notation, when we bind monadic values to names, we can utilize pattern matching, just like in `let` expressions and function parameters. Here's an example of pattern matching in a `do` expression:

モナド値を<-でバインドせずに do 記法で行を書くと、結果を無視したいモナド値の後に>>を置くのと同じことになる。モナド値を並べるが、その結果は無視する。なぜなら、それが何であるかは気にしないし、_ <- Nothing と書くよりきれいだからだ。

いつ do 記法を使うか、いつ>>=を明示的に使うかはあなた次第だ。この例では、各ステップが前のステップの結果に特に依存しているので、明示的に>>=と書くのに適していると思う。do 記法では、鳥がどのポールに着地するかを具体的に書かなければならなかったが、それを使うたびに直前まで来ていた。しかしそれでも、do 記法についての洞察は得られた。

do 記法では、モナド値を名前にバインドするとき、let 式や関数のパラメータと同じようにパターン・マッチを利用することができる。以下は、do 記法におけるパターン・マッチの例である：

```
justH :: Maybe Char
justH = do
    (x:xs) <- Just "hello"
    return x
```

We use pattern matching to get the first character of the string `"hello"` and then we present it as the result. So `justH` evaluates to `Just 'h'`.

What if this pattern matching were to fail? When matching on a pattern in a function fails, the next pattern is matched. If the matching falls through all the patterns for a given function, an error is thrown and our program crashes. On the other hand, failed pattern matching in `let` expressions results in an error being produced right away, because the mechanism of falling through patterns isn't present in `let` expressions. When pattern matching fails in a `do` expression, the `fail` function is called. It's part of the `Monad` type class and it enables failed pattern matching to result in a failure in the context of the current monad instead of making our program crash. Its default implementation is this:

パターン・マッチを使って文字列 "hello "の最初の文字を取得し、それを結果として提示する。つまり、justH は単に'h'と評価される。

このパターン・マッチが失敗したらどうなるでしょうか？関数内のパターンのマッチングが失敗すると、次のパターンがマッチングされます。もしマッチングがある関数のすべてのパターンを通過してしまうと、エラーが投げられ、プログラムはクラッシュします。一方、let 式でパターンマッチングに失敗すると、すぐにエラーが発生します。let 式にはパターンを通過するメカニズムが存在しないからです。do 式でパターンマッチングに失敗すると、fail 関数が呼び出される。この関数はモナド型クラスの一部で、パターン・マッチに失敗したときにプログラムをクラッシュさせるのではなく、現在のモナドのコンテキストで失敗するようにします。デフォルトの実装はこうなっている：

```
fail :: (Monad m) => String -> m a
fail msg = error msg
```

So by default it does make our program crash, but monads that incorporate a context of possible failure (like `Maybe`) usually implement it on their own. For `Maybe`, its implemented like so:

そのため、デフォルトではプログラムがクラッシュしてしまうのだが、（Maybe のように）失敗の可能性があるコンテキストを組み込んでいるモナドは、通常それを独自に実装している。Maybe の場合は以下のように実装されている：

```
fail _ = Nothing
```

It ignores the error message and makes a `Nothing`. So when pattern matching fails in a `Maybe` value that's written in `do` notation, the whole value results in a `Nothing`. This is preferable to having our program crash. Here's a `do` expression with a pattern that's bound to fail:

エラーメッセージは無視され、Nothing になる。つまり、do 記法で書かれた Maybe 値でパターン・マッチが失敗した場合、値全体が Nothing になる。これはプログラムがクラッシュするよりも望ましい。以下は、失敗するに決まっているパターンを使った do 式である：

```
wopwop :: Maybe Char
wopwop = do
    (x:xs) <- Just ""
    return x
```

The pattern matching fails, so the effect is the same as if the whole line with the pattern was replaced with a `Nothing`. Let's try this out:

パターン・マッチは失敗するので、パターンを含む行全体を Nothing に置き換えたのと同じ効果が得られる。試してみよう：

```
ghci> wopwop
Nothing
```

The failed pattern matching has caused a failure within the context of our monad instead of causing a program-wide failure, which is pretty neat.

パターン・マッチの失敗は、プログラム全体の失敗を引き起こすのではなく、モナドのコンテキスト内での失敗を引き起こした。

The list monad

So far, we've seen how `Maybe` values can be viewed as values with a failure context and how we can incorporate failure handling into our code by using `>>=` to feed them to functions. In this section, we're going to take a look at how to use the monadic aspects of lists to bring non-determinism into our code in a clear and readable manner.
We've already talked about how lists represent non-deterministic values when they're used as applicatives. A value like `5` is deterministic. It has only one result and we know exactly what it is. On the other hand, a value like `[3,8,9]` contains several results, so we can view it as one value that is actually many values at the same time. Using lists as applicative functors showcases this non-determinism nicely:

ここまでは、Maybe 値を失敗のコンテキストを持つ値と見なす方法と、>>=を使って関数に渡すことで失敗処理をコードに組み込む方法について見てきた。このセクションでは、リストのモナド的な側面を利用して、非決定性をコードにわかりやすく、読みやすく取り入れる方法を見ていきます。
リストがアプリケ-ションとして使われるとき、どのように非決定論的な値を表現するかについてはすでにお話ししました。5 のような値は決定論的だ。結果は 1 つだけで、それが何であるかも正確に分かっている。一方、[3,8,9]のような値にはいくつかの結果が含まれるため、1 つの値でありながら実際には同時に多くの値であるとみなすことができます。応用ファンクターとしてリストを使うことで、この非決定性をうまく見せることができる:

```
ghci> (*) <$> [1,2,3] <*> [10,100,1000]
[10,100,1000,20,200,2000,30,300,3000]
```

All the possible combinations of multiplying elements from the left list with elements from the right list are included in the resulting list. When dealing with non-determinism, there are many choices that we can make, so we just try all of them, and so the result is a non-deterministic value as well, only it has many more results.

This context of non-determinism translates to monads very nicely. Let's go ahead and see what the `Monad` instance for lists looks like:

左のリストの要素と右のリストの要素を掛け合わせた結果のリストには、すべての可能な組み合わせが含まれる。非決定性を扱う場合、選択できる選択肢はたくさんあるので、それらをすべて試すだけで、結果も非決定的な値になる。
このような非決定性という文脈は、モナドに非常にうまく翻訳される。リストのモナド・インスタンスを見てみよう:

```
instance Monad [] where
    return x = [x]
    xs >>= f = concat (map f xs)
    fail _ = []
```

`return` does the same thing as `pure`, so we should already be familiar with `return` for lists. It takes a value and puts it in a minimal default context that still yields that value. In other words, it makes a list that has only that one value as its result. This is useful for when we want to just wrap a normal value into a list so that it can interact with non-deterministic values.
To understand how `>>=` works for lists, it's best if we take a look at it in action to gain some intuition first. `>>=` is about taking a value with a context (a monadic value) and feeding it to a function that takes a normal value and returns one that has context. If that function just produced a normal value instead of one with a

context, `>>=` wouldn't be so useful because after one use, the context would be lost. Anyway, let's try feeding a non-deterministic value to a function:

return は pure と同じことをするので、リストに対する return はすでによく知っているはずだ。return は値を受け取り、その値が得られる最小限のデフォルト・コンテキストに置く。言い換えれば、結果としてその値だけを持つリストを作るということだ。これは、非決定的な値とやりとりできるように、通常の値をリストにラップしたい場合に便利です。
リストに対して >>= がどのように機能するかを理解するためには、まず直感的に理解するために、実際に >>= を使ってみるのが一番だ。>>= これは、コンテキストを持つ値（モナド値）を受け取り、それを通常の値を受け取ってコンテキストを持つ値を返す関数に渡すというものだ。もしその関数がコンテキストを持つ値ではなく、普通の値を返すだけだとしたら、>>=はそれほど便利ではないだろう。とにかく、関数に非決定論的な値を与えてみよう:

```
1. ghci> [3,4,5] >>= ¥x -> [x,-x]
2. [3,-3,4,-4,5,-5]
```

When we used `>>=` with `Maybe`, the monadic value was fed into the function while taking care of possible failures. Here, it takes care of non-determinism for us. `[3,4,5]` is a non-deterministic value and we feed it into a function that returns a non-deterministic value as well. The result is also non-deterministic, and it features all the possible results of taking elements from the list `[3,4,5]` and passing them to the function `¥x -> [x,-x]`. This function takes a number and produces two results: one negated and one that's unchanged. So when we use `>>=` to feed this list to the function, every number is negated and also kept unchanged. The `x` from the lambda takes on every value from the list that's fed to it.
To see how this is achieved, we can just follow the implementation. First, we start off with the list `[3,4,5]`. Then, we map the lambda over it and the result is the following:

Maybe で>>=を使うと、失敗の可能性を考慮しながらモナド値が関数に入力される。ここでは、非決定性を考慮している。[3,4,5]は非決定論的な値であり、同様に非決定論的な値を返す関数に送り込む。その結果もまた非決定論的であり、リスト[3,4,5]から要素を取り出し、それを関数 ¥x -> [x,-x]に渡した場合のすべての可能な結果を特徴とする。この関数は数値を受け取り、2 つの結果を生成する。1 つは否定され、もう 1 つは変更されない。つまり、>>=を使ってこのリストを関数に渡すと、すべての数値が否定され、また変更されない。ラムダからの x は、与えられたリストのすべての値を受け取る。
これがどのように実現されるかは、実装を追えばわかる。まず、リスト[3,4,5]から始める。その上でラムダをマップすると、次のようになる:

```
1. [[3,-3],[4,-4],[5,-5]]
```

The lambda is applied to every element and we get a list of lists. Finally, we just flatten the list and voila! We've applied a non-deterministic function to a non-deterministic value!

Non-determinism also includes support for failure. The empty list `[]` is pretty much the equivalent of `Nothing`, because it signifies the absence of a result. That's why failing is just defined as the empty list. The error message gets thrown away. Let's play around with lists that fail:

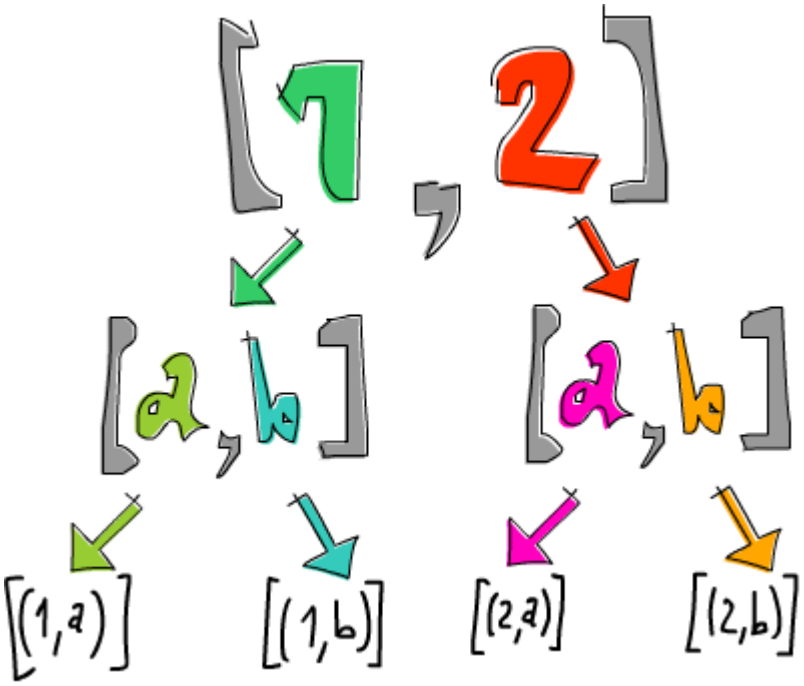
ラムダがすべての要素に適用され、リストのリストが得られる。最後に、リストを平坦化すれば出来上がり！これで、非決定論的な値に非決定論的な関数を適用したことになる！
非決定性には失敗のサポートも含まれる。空リスト[]は Nothing とほぼ等価で、結果がないことを意味するからだ。そのため、failure は単に空リストとして定義される。エラーメッセージは捨てられる。失敗するリストで遊んでみよう:

```
ghci> [] >>= ¥x -> ["bad","mad","rad"]
[]
ghci> [1,2,3] >>= ¥x -> []
[]
```

In the first line, an empty list is fed into the lambda. Because the list has no elements, none of them can be passed to the function and so the result is an empty list. This is similar to feeding `Nothing` to a function. In the second line, each element gets passed to the function, but the element is ignored and the function just returns an empty list. Because the function fails for every element that goes in it, the result is a failure.
Just like with `Maybe` values, we can chain several lists with `>>=`, propagating the non-determinism:

最初の行では、ラムダに空のリストが入力されている。リストには要素がないので、関数に渡すことができず、結果は空リストになる。これは、関数に Nothing を渡すのと似ている。2 行目では、各要素が関数に渡されるが、その要素は無視され、関数は空リストを返すだけだ。関数に渡されるすべての要素に対して失敗するため、結果は失敗となります。
Maybe 値の場合と同じように、>>=で複数のリストを連結して、非決定性を伝播させることができる:

```
ghci> [1,2] >>= ¥n -> ['a','b'] >>= ¥ch -> return (n,ch)
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```



The list `[1,2]` gets bound to `n` and `['a','b']` gets bound to `ch`. Then, we do `return (n,ch)` (or `[(n,ch)]`), which means taking a pair of `(n,ch)` and putting it in a default minimal context. In this case, it's making the smallest possible list that still presents `(n,ch)` as the result and features as little non-determinism as possible. Its effect on the context is minimal. What we're saying here is this: for every element in `[1,2]`, go over every element in `['a','b']` and produce a tuple of one element from each list.
Generally speaking, because `return` takes a value and wraps it in a minimal context, it doesn't have any extra effect (like failing in `Maybe` or resulting in more non-determinism for lists) but it does present something as its result.

When you have non-deterministic values interacting, you can view their computation as a tree where every possible result in a list represents a separate branch.

Here's the previous expression rewritten in `do` notation:

リスト[1,2]は n に、['a','b']は ch にバインドされる。そして、(n,ch)(または[(n,ch)])を返す。これは、(n,ch)のペアを取り出し、デフォルトの最小コンテキストに置くことを意味する。この場合、(n,ch)を結果として提示し、非決定性を可能な限り抑えた、可能な限り小さなリストを作ることになる。コンテキストへの影響は最小限である。ここで言っているのは、[1,2]の各要素に対して、['a','b']の各要素を調べ、それぞれのリストから 1 つの要素のタプルを生成するということである。
一般的に言って、return は値を受け取り、それを最小限の文脈で包むので、余計な効果（Maybe で失敗するとか、リストの非決定性が増すとか）はないが、結果として何かを提示する。
非決定論的な値が相互作用している場合、その計算をツリーとして見ることができます。
先ほどの式を do 記法で書き直すとこうなる:

```
listOfTuples :: [(Int,Char)]
listOfTuples = do
    n <- [1,2]
    ch <- ['a','b']
    return (n,ch)
```

This makes it a bit more obvious that `n` takes on every value from `[1,2]` and `ch` takes on every value from `['a','b']`. Just like with `Maybe`, we're extracting the elements from the monadic values and treating them like normal values and `>>=` takes care of the context for us. The context in this case is non-determinism.
Using lists with `do` notation really reminds me of something we've seen before. Check out the following piece of code:

これにより、n が[1,2]からすべての値を取り、ch が['a','b']からすべての値を取ることが少し明白になる。Maybe の場合と同じように、我々はモナド値から要素を抽出して通常の値のように扱っており、>>=がコンテキストを処理してくれる。この場合のコンテキストとは、非決定性である。
do 記法でリストを使うと、以前にも見たことがあるような気がする。次のコードを見てほしい:

```
ghci> [ (n,ch) | n <- [1,2], ch <- ['a','b'] ]
[(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

Yes! List comprehensions! In our `do` notation example, `n` became every result from `[1,2]` and for every such result, `ch` was assigned a result from `['a','b']` and then the final line put `(n,ch)` into a default context (a singleton list) to present it as the result without introducing any additional non-determinism. In this list comprehension, the same thing happened, only we didn't have to write `return` at the end to present `(n,ch)` as the result because the output part of a list comprehension did that for us.
In fact, list comprehensions are just syntactic sugar for using lists as monads. In the end, list comprehensions and lists in `do` notation translate to using `>>=` to do computations that feature non-determinism.
List comprehensions allow us to filter our output. For instance, we can filter a list of numbers to search only for that numbers whose digits contain a `7`:

はい！リスト内包だ！do 記法の例では、n が[1,2]からのすべての結果になり、そのような結果ごとに ch が['a','b']からの結果に割り当てられ、最終行で(n,ch)がデフォルトのコンテキスト（シングルトンリスト）に置かれ、追加の非決定性を導入することなく結果として提示されました。このリスト内包でも同じことが起こりますが、リスト内包の出力部がそれをやってくれるので、(n,ch)を結果として提示するために最後に return を書く必要はありません。
実際、リスト内包はリストをモナドとして使うための構文上の砂糖に過ぎない。結局のところ、リスト内包と do 記法のリストは、非決定性を特徴とする計算を行うために>>=を使用することになる。

リスト内包を使うと、出力にフィルタをかけることができる。例えば、数字のリストをフィルタリングして、7 を含む数字だけを検索することができる:

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

We apply `show` to `x` to turn our number into a string and then we check if the character `'7'` is part of that string. Pretty clever. To see how filtering in list comprehensions translates to the list monad, we have to check out the `guard` function and the `MonadPlus` type class. The `MonadPlus` type class is for monads that can also act as monoids. Here's its definition:

x に show を適用して数値を文字列に変換し、文字'7'がその文字列の一部であるかどうかをチェックする。なかなか賢い。リスト内包のフィルタリングがリスト・モナドにどのように変換されるかを見るには、ガード関数と MonadPlus 型クラスをチェックする必要がある。MonadPlus 型クラスは、モノイドとしても機能するモナドのためのものだ。以下がその定義だ:

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

`mzero` is synonymous to `mempty` from the `Monoid` type class and `mplus` corresponds to `mappend`. Because lists are monoids as well as monads, they can be made an instance of this type class:

mzero は Monoid 型クラスの mempty と同義であり、mplus は mappend に相当する。リストはモノイドであると同時にモナドでもあるので、この型クラスのインスタンスにすることができる:

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

For lists `mzero` represents a non-deterministic computation that has no results at all — a failed computation. `mplus` joins two non-deterministic values into one. The `guard` function is defined like this:

mplus は 2 つの非決定論的な値を 1 つに結合する。ガード関数は次のように定義される:

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

It takes a boolean value and if it's `True`, takes a `()` and puts it in a minimal default context that still succeeds. Otherwise, it makes a failed monadic value. Here it is in action:

これはブール値を取り、それが True の場合は()を取り、それでも成功する最小限のデフォルト・コンテキストに置く。そうでなければ、失敗したモナド値を作る。これがその動作だ:

```
ghci> guard (5 > 2) :: Maybe ()
Just ()
ghci> guard (1 > 2) :: Maybe ()
Nothing
ghci> guard (5 > 2) :: [()]
[()]
```

```
ghci> guard (1 > 2) :: [()]
[]
```

Looks interesting, but how is it useful? In the list monad, we use it to filter out non-deterministic computations. Observe:

面白そうだが、どう役に立つのか？リスト・モナドでは、非決定論的な計算をフィルタリングするために使っている。見てみよう:

```
ghci> [1..50] >>= (¥x -> guard ('7' `elem` show x) >> return x)
[7,17,27,37,47]
```

The result here is the same as the result of our previous list comprehension. How does `guard` achieve this? Let's first see how `guard` functions in conjunction with `>>`:

ここでの結果は、前回のリスト理解の結果と同じである。guard はどのようにしてこれを実現しているのだろうか？まず、guard が>>とどのように連携して機能するかを見てみよう:

```
ghci> guard (5 > 2) >> return "cool" :: [String]
["cool"]
ghci> guard (1 > 2) >> return "cool" :: [String]
[]
```

If `guard` succeeds, the result contained within it is an empty tuple. So then, we use `>>` to ignore that empty tuple and present something else as the result. However, if `guard` fails, then so will the `return` later on, because feeding an empty list to a function with `>>=` always results in an empty list. A `guard` basically says: if this boolean is `False` then produce a failure right here, otherwise make a successful value that has a dummy result of `()` inside it. All this does is to allow the computation to continue.
Here's the previous example rewritten in `do` notation:

ガードが成功すると、その中に含まれる結果は空のタプルになる。そこで、>>を使ってその空のタプルを無視し、別のものを結果として提示する。というのも、空のリストを >>= で関数に渡すと、常に空のリストが返されるからです。ガードは基本的に、「もしこのブール値が False なら、ここで失敗を返す。これは計算を続行させるためのものである。
先ほどの例を do 記法で書き直すとこうなる:

```
sevensOnly :: [Int]
sevensOnly = do
    x <- [1..50]
    guard ('7' `elem` show x)
    return x
```

Had we forgotten to present `x` as the final result by using `return`, the resulting list would just be a list of empty tuples. Here's this again in the form of a list comprehension:

もし return を使って x を最終結果として提示するのを忘れていたら、結果のリストは空のタプルのリストになっていただろう。リスト内包の形でもう一度説明しよう:

```
ghci> [ x | x <- [1..50], '7' `elem` show x ]
[7,17,27,37,47]
```

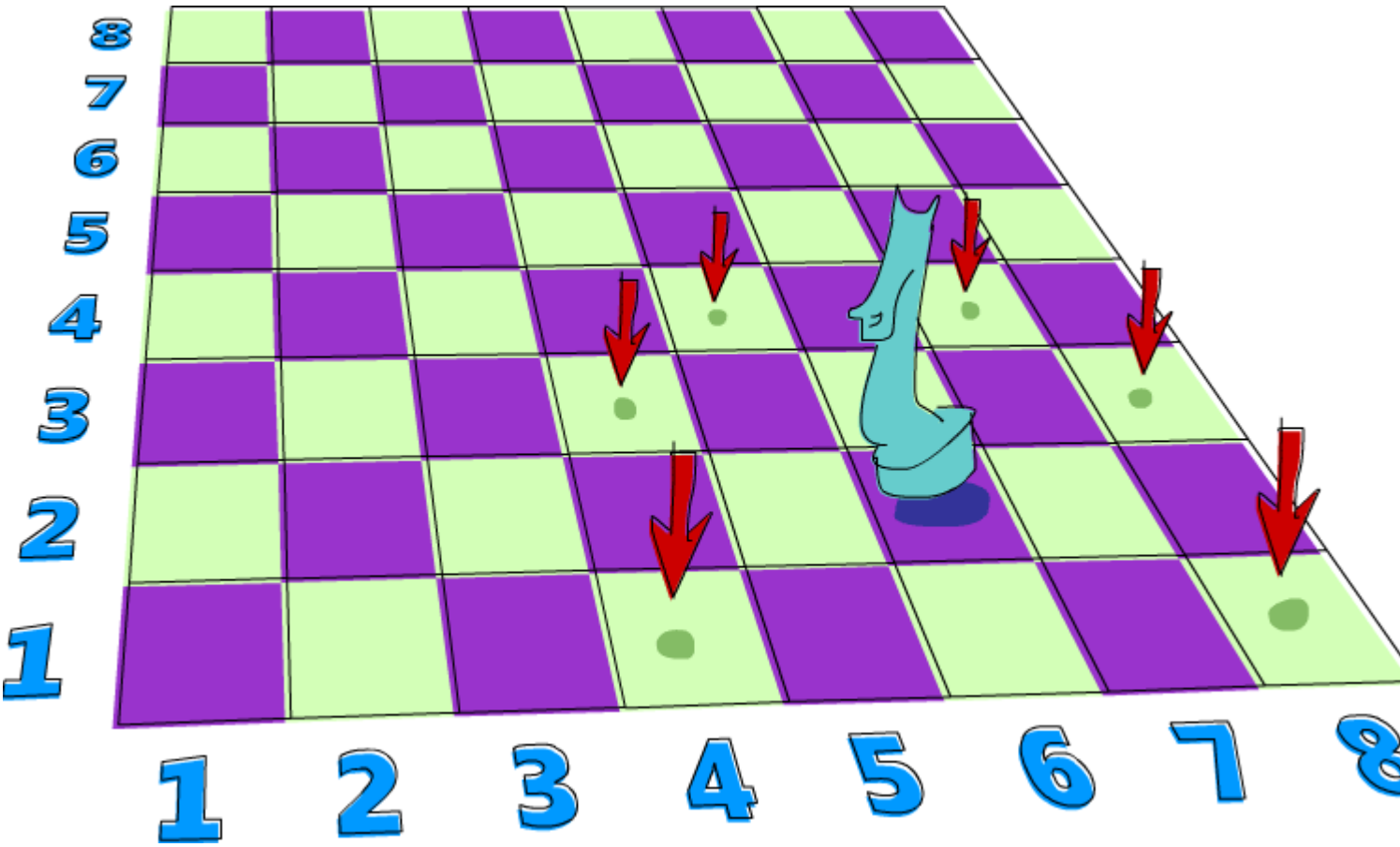
So filtering in list comprehensions is the same as using `guard`.

つまり、リスト内包でのフィルタリングはガードを使うのと同じことだ。

A knight's quest

Here's a problem that really lends itself to being solved with non-determinism. Say you have a chess board and only one knight piece on it. We want to find out if the knight can reach a certain position in three moves. We'll just use a pair of numbers to represent the knight's position on the chess board. The first number will determine the column he's in and the second number will determine the row.

非決定性で解くのに適した問題がある。チェス盤があって、そこにナイトの駒が 1 つだけあるとする。そのナイトが 3 手である位置に到達できるかどうかを調べたい。チェス盤上のナイトの位置を表すために、2 つの数字を使う。最初の数字が彼のいる列を、2 番目の数字が行を決める。



Let's make a type synonym for the knight's current position on the chess board:

チェス盤上のナイトの現在の位置を表す同義語を作ろう:

```
1. type KnightPos = (Int,Int)
```

So let's say that the knight starts at `(6,2)`. Can he get to `(6,1)` in exactly three moves? Let's see. If we start off at `(6,2)` what's the best move to make next? I know, how about all of them! We have non-determinism at our

disposal, so instead of picking one move, let's just pick all of them at once. Here's a function that takes the knight's position and returns all of its next moves:

では、ナイトが(6,2)からスタートしたとしよう。果たして 3 手で(6,1)に到達できるだろうか？見てみよう。もし(6,2)から始めるとしたら、次に打つべき最善手は？そうだな、全部でどうだろう！せっかく非決定性があるのだから、1 手だけ選ぶのではなく、一度に全部選んでしまおう。ここに、ナイトの位置を受け取り、次の手をすべて返す関数がある：

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = do
    (c',r') <- [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
               ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
               ]
    guard (c' `elem` [1..8] && r' `elem` [1..8])
    return (c',r')
```

The knight can always take one step horizontally or vertically and two steps horizontally or vertically but its movement has to be both horizontal and vertical. **`(c',r')`** takes on every value from the list of movements and then **`guard`** makes sure that the new move, **`(c',r')`** is still on the board. If it it's not, it produces an empty list, which causes a failure and **`return (c',r')`** isn't carried out for that position.
This function can also be written without the use of lists as a monad, but we did it here just for kicks. Here is the same function done with **`filter`**:

ナイトは常に水平または垂直に 1 歩、水平または垂直に 2 歩動くことができるが、その動きは水平と垂直の両方でなければならない。(c',r')は移動のリストからすべての値を取り、新しい移動、(c',r')がまだボード上にあることを確認する。もしそうでなければ、空のリストが生成され、失敗となり、(c',r')はその位置では実行されない。
この関数は、モナドとしてリストを使わなくても書くことができるが、ここではちょっとやってみた。同じ関数を filter で書いてみた：

```
moveKnight :: KnightPos -> [KnightPos]
moveKnight (c,r) = filter onBoard
    [(c+2,r-1),(c+2,r+1),(c-2,r-1),(c-2,r+1)
    ,(c+1,r-2),(c+1,r+2),(c-1,r-2),(c-1,r+2)
    ]
    where onBoard (c,r) = c `elem` [1..8] && r `elem` [1..8]
```

Both of these do the same thing, so pick one that you think looks nicer. Let's give it a whirl:

どちらもやることは同じなので、より素敵に見える方を選んでください。試してみましょう：

```
ghci> moveKnight (6,2)
[(8,1),(8,3),(4,1),(4,3),(7,4),(5,4)]
ghci> moveKnight (8,1)
[(6,2),(7,3)]
```

Works like a charm! We take one position and we just carry out all the possible moves at once, so to speak. So now that we have a non-deterministic next position, we just use `>>=` to feed it to `moveKnight`. Here's a function that takes a position and returns all the positions that you can reach from it in three moves:

魅力的に機能する！いわば、1 つのポジションをとって、可能なすべての手を一度に実行するだけだ。つまり、非決定的な次の位置が得られたので、それを>>=を使って moveKnight に与えるだけだ。あるポジションを取り、そこから 3 回の移動で到達できるすべてのポジションを返す関数を以下に示す：

```
in3 :: KnightPos -> [KnightPos]
in3 start = do
    first <- moveKnight start
    second <- moveKnight first
    moveKnight second
```

If you pass it `(6,2)`, the resulting list is quite big, because if there are several ways to reach some position in three moves, it crops up in the list several times. The above without `do` notation:

もし(6,2)を渡すと、出来上がったリストはかなり大きくなる。なぜなら、あるポジションに 3 回の移動で到達する方法がいくつかある場合、それが何度もリストに現れるからである。上記は do 記法なし：

```
in3 start = return start >>= moveKnight >>= moveKnight >>= moveKnight
```

Using `>>=` once gives us all possible moves from the start and then when we use `>>=` the second time, for every possible first move, every possible next move is computed, and the same goes for the last move.
Putting a value in a default context by applying `return` to it and then feeding it to a function with `>>=` is the same as just normally applying the function to that value, but we did it here anyway for style.
Now, let's make a function that takes two positions and tells us if you can get from one to the other in exactly three steps:

そして 2 回目に >>= を使うと、最初の手の可能性がすべて計算され、次の手の可能性もすべて計算される。

return を適用して値をデフォルト・コンテキストに置き、それを>>=で関数に渡すのは、普通にその値に関数を適用するのと同じだが、ここではスタイルのためにそうしている。

では、2 つの位置を取り、片方からもう片方へ正確に 3 ステップで移動できるかどうかを教えてくれる関数を作ってみよう：

```
canReachIn3 :: KnightPos -> KnightPos -> Bool
canReachIn3 start end = end `elem` in3 start
```

We generate all the possible positions in three steps and then we see if the position we're looking for is among them. So let's see if we can get from `(6,2)` to `(6,1)` in three moves:

3 つのステップで可能なポジションをすべて生成し、その中に探しているポジションがあるかどうかを確認するのだ。では、3 手で(6,2)から(6,1)に行けるか見てみよう：

```
ghci> (6,2) `canReachIn3` (6,1)
True
```

Yes! How about from `(6,2)` to `(7,3)`?

そうだ！(6,2)から(7,3)はどうだ？

```
ghci> (6,2) `canReachIn3` (7,3)
False
```

No! As an exercise, you can change this function so that when you can reach one position from the other, it tells you which moves to take. Later on, we'll see how to modify this function so that we also pass it the number of moves to take instead of that number being hardcoded like it is now.

いいえ！練習として、この関数を変更して、もう一方の位置からもう一方の位置に到達できたときに、どの手を取るべきかを教えてくれるようにすることができる。後ほど、この関数を変更して、現在のようにハードコードされた手数ではなく、取るべき手数も渡すようにします。

Monad laws

Just like applicative functors, and functors before them, monads come with a few laws that all monad instances must abide by. Just because something is made an instance of the `Monad` type class doesn't mean that it's a monad, it just means that it was made an instance of a type class. For a type to truly be a monad, the monad laws must hold for that type. These laws allow us to make reasonable assumptions about the type and its behavior.
Haskell allows any type to be an instance of any type class as long as the types check out. It can't check if the monad laws hold for a type though, so if we're making a new instance of the `Monad` type class, we have to be reasonably sure that all is well with the monad laws for that type. We can rely on the types that come with the standard library to satisfy the laws, but later when we go about making our own monads, we're going to have to manually check the if the laws hold. But don't worry, they're not complicated.

アプリケーティブ・ファンクターやそれ以前のファンクターと同じように、モナドにはすべてのモナドのインスタンスが守らなければならないいくつかの法則がある。何かがモナドの型クラスのインスタンスになったからといって、それがモナドであるということにはならない。ある型が本当にモナドであるためには、モナドの法則がその型に対して成立していなければならない。これらの法則によって、型とその振る舞いについて合理的な仮定をすることができる。
Haskell では、型がチェックアウトされる限り、どの型もどの型クラスのインスタンスにもなることができる。しかし、モナドの法則が型に対して成り立つかどうかをチェックすることはできないので、モナド型クラスの新しいインスタンスを作成する場合は、その型に対するモナドの法則がすべて正しいことを合理的に確認する必要がある。標準ライブラリに付属している型が法則を満たしていることを頼りにすることはできるが、後で独自のモナドを作るときには、法則が成立しているかどうかを手作業でチェックしなければならない。でも心配しないで。

Left identity

The first monad law states that if we take a value, put it in a default context with `return` and then feed it to a function by using `>>=`, it's the same as just taking the value and applying the function to it. To put it formally:

- `return x >>= f` is the same damn thing as `f x`

If you look at monadic values as values with a context and `return` as taking a value and putting it in a default minimal context that still presents that value as its result, it makes sense, because if that context is really minimal, feeding this monadic value to a function shouldn't be much different than just applying the function to the normal value, and indeed it isn't different at all.
For the `Maybe` monad `return` is defined as `Just`. The `Maybe` monad is all about possible failure, and if we have a value and want to put it in such a context, it makes sense that we treat it as a successful computation because, well, we know what the value is. Here's some `return` usage with `Maybe`:

最初のモナドの法則は、値を受け取り、return でデフォルト・コンテキストに入れ、>>=を使って関数に渡すと、単に値を受け取って関数を適用するのと同じになる、というものだ。形式的に言えば
- return x >>= f は f x と同じものである。
モナド値をコンテキストを持つ値と見なし、return を値を取り出し、その値を結果として提示するデフォルトの最小コンテキストに置くことと見なせば、それは理にかなっている。そのコンテキストが本当に最小であれば、このモナド値を関数に与えることは、通常の値に関数を適用することと大差ないはずであり、実際まったく変わらないからだ。
Maybe モナドの return は Just と定義されている。Maybe モナドは失敗の可能性を扱うものであり、ある値を持っていてそれをそのような文脈に置きたい場合、その値が何であるかわかっているのだから、それを計算の成功として扱うのは理にかなっている。以下は、Maybe を使った戻り値の使い方だ:

```
ghci> return 3 >>= (¥x -> Just (x+100000))
Just 100003
ghci> (¥x -> Just (x+100000)) 3
Just 100003
```

For the list monad `return` puts something in a singleton list. The `>>=` implementation for lists goes over all the values in the list and applies the function to them, but since there's only one value in a singleton list, it's the same as applying the function to that value:

リスト・モナドの return は、シングルトンリストに何かを入れる。リストに対する >>= の実装は、リスト内のすべての値を調べて関数を適用しますが、シングルトンリストの値は 1 つだけなので、その値に関数を適用するのと同じです:

```
ghci> return "WoM" >>= (¥x -> [x,x,x])
["WoM","WoM","WoM"]
ghci> (¥x -> [x,x,x]) "WoM"
["WoM","WoM","WoM"]
```

We said that for `IO`, using `return` makes an I/O action that has no side-effects but just presents a value as its result. So it makes sense that this law holds for `IO` as well.

IO の場合、return を使うと、副作用のない I/O アクションになるが、その結果として値を提示するだけである、と述べた。だから、IO でもこの法則が成り立つのは理にかなっている。

Right identity

The second law states that if we have a monadic value and we use `>>=` to feed it to `return`, the result is our original monadic value. Formally:

- `m >>= return` is no different than just `m`

This one might be a bit less obvious than the first one, but let's take a look at why it should hold. When we feed monadic values to functions by using `>>=`, those functions take normal values and return monadic ones. `return` is also one such function, if you consider its type. Like we said, `return` puts a value in a minimal context that still presents that value as its result. This means that, for instance, for `Maybe`, it doesn't introduce any failure and for lists, it doesn't introduce any extra non-determinism. Here's a test run for a few monads:

つ目の法則は、モナド値があり、それを >>= を使って return に送ると、結果は元のモナド値になるというものだ。形式的には
- m >>= return は単なる m と変わらない
これは最初のものより少しわかりにくいかもしれないが、なぜそうなるのかを見てみよう。モナド値を >>= を使って関数に与える場合、その関数は通常の値を取り、モナド値を返します。先ほども言ったように、return は値を最小限のコンテキストに置き、その結果としての値を提示する。つまり、例えば Maybe の場合は失敗をもたらさないし、リストの場合は余計な非決定性をもたらさない。いくつかのモナドをテストしてみましょう:

```
ghci> Just "move on up" >>= (¥x -> return x)
Just "move on up"
ghci> [1,2,3,4] >>= (¥x -> return x)
[1,2,3,4]
ghci> putStrLn "Wah!" >>= (¥x -> return x)
Wah!
```

If we take a closer look at the list example, the implementation for `>>=` is:

リストの例をよく見てみると、>>=の実装はこうなっている:

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

So when we feed `[1,2,3,4]` to `return`, first `return` gets mapped over `[1,2,3,4]`, resulting in `[[1],[2],[3],[4]]` and then this gets concatenated and we have our original list.
Left identity and right identity are basically laws that describe how `return` should behave. It's an important function for making normal values into monadic ones and it wouldn't be good if the monadic value that it produced did a lot of other stuff.

したがって、[1,2,3,4]を return に渡すと、まず return は[1,2,3,4]にマップされ、[[1],[2],[3],[4]]となり、これが連結されて元のリストになる。
左恒等式と右恒等式は基本的に、戻り値がどのように振る舞うべきかを記述する法則である。これは通常の値をモナド的なものにするための重要な関数であり、この関数が生成するモナド的な値が他の多くのことを行うのであれば、それは良いことではない。

Associativity

The final monad law says that when we have a chain of monadic function applications with `>>=`, it shouldn't matter how they're nested. Formally written:

- Doing `(m >>= f) >>= g` is just like doing `m >>= (¥x -> f x >>= g)`

Hmmm, now what's going on here? We have one monadic value, `m` and two monadic functions `f` and `g`. When we're doing `(m >>= f) >>= g`, we're feeding `m` to `f`, which results in a monadic value. Then, we feed that monadic value to `g`. In the expression `m >>= (¥x -> f x >>= g)`, we take a monadic value and we feed it to a function that feeds the result of `f x` to `g`. It's not easy to see how those two are equal, so let's take a look at an example that makes this equality a bit clearer.
Remember when we had our tightrope walker Pierre walk a rope while birds landed on his balancing pole? To simulate birds landing on his balancing pole, we made a chain of several functions that might produce failure:

最後のモナドの法則は、>>=を使ったモナド関数の連鎖があるとき、それらがどのように入れ子になっているかは問題ではない、というものだ。正式にはこう書く:

- (m>>=f)>>=g とすることは、m>>=(¥x -> f x >>=g)とすることと同じである。

うーん、どうしたものか。m >>= f）>>= g とするとき、m を f に与えている。m >>= (¥x -> f x >>= g)という式では、モナド値を受け取り、それを f x の結果を g に渡す関数に渡している。

綱渡り芸人のピエールに、鳥が彼のバランス・ポールに着地する間に綱を歩かせたのを覚えているだろうか？鳥が彼のバランス・ポールに着地するのをシミュレートするために、私たちは失敗を生むかもしれないいくつかの関数の連鎖を作った:

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

We started with `Just (0,0)` and then bound that value to the next monadic function, `landRight 2`. The result of that was another monadic value which got bound into the next monadic function, and so on. If we were to explicitly parenthesize this, we'd write:

まず(0,0)から始め、その値を次のモナド関数 landRight 2 にバインドした。その結果はまた別のモナド値で、次のモナド関数にバインドされる。これを明示的に括弧でくくると、こうなる:

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

But we can also write the routine like this:

しかし、このようなルーチンを書くこともできる:

```
return (0,0) >>= (¥x ->
landRight 2 x >>= (¥y ->
landLeft 2 y >>= (¥z ->
landRight 2 z)))
```

`return (0,0)` is the same as `Just (0,0)` and when we feed it to the lambda, the `x` becomes `(0,0)`. `landRight` takes a number of birds and a pole (a tuple of numbers) and that's what it gets passed. This results in a `Just (0,2)` and when we feed this to the next lambda, `y` is `(0,2)`. This goes on until the final bird landing produces a `Just (2,4)`, which is indeed the result of the whole expression.
So it doesn't matter how you nest feeding values to monadic functions, what matters is their meaning. Here's another way to look at this law: consider composing two functions, `f` and `g`. Composing two functions is implemented like so:

return(0,0)は Just(0,0)と同じで、これをラムダに渡すと x は(0,0)になる。landRight は鳥の数とポール（数字のタプル）を受け取り、それが渡される。この結果、ちょうど(0,2)となり、これを次のラムダに渡すと、y は(0,2)となる。これは、最後の鳥の着地が(2,4)を生成するまで続く。
つまり、モナド関数にどのように値をネストさせるかは問題ではなく、重要なのはその意味なのだ。この法則を見るもう一つの方法がある。2 つの関数 f と g を合成することを考えてみよう:

```
(.) :: (b -> c) -> (a -> b) -> (a -> c)
f . g = (¥x -> f (g x))
```

If the type of `g` is `a -> b` and the type of `f` is `b -> c`, we arrange them into a new function which has a type of `a -> c`, so that its parameter is passed between those functions. Now what if those two functions were monadic, that is, what if the values they returned were monadic values? If we had a function of type `a -> m b`, we couldn't just pass its result to a function of type `b -> m c`, because that function accepts a normal `b`, not a monadic one. We could however, use `>>=` to make that happen. So by using `>>=`, we can compose two monadic functions:

g の型が a→b、f の型が b→c の場合、それらを a→c の型を持つ新しい関数にアレンジし、そのパラメータがこれらの関数間で受け渡しされるようにする。さて、もしこの 2 つの関数がモナド型であったとしたら、つまり、関数が返す値がモナド値であったとしたらどうだろうか。a -> m b 型の関数があったとして、その結果を b -> m c 型の関数に渡すことはできない。しかし、>>=を使えばそれが可能になる。つまり、>>=を使うことで、2 つのモナド関数を合成することができるのだ:

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

So now we can compose two monadic functions:

これで 2 つのモナド関数を合成することができる:

```
ghci> let f x = [x,-x]
ghci> let g x = [x*3,x*2]
ghci> let h = f <=< g
ghci> h 3
[9,-9,6,-6]
```

Cool. So what does that have to do with the associativity law? Well, when we look at the law as a law of compositions, it states that `f <=< (g <=< h)` should be the same as `(f <=< g) <=< h`. This is just another way of saying that for monads, the nesting of operations shouldn't matter.

If we translate the first two laws to use `<=<`, then the left identity law states that for every monadic function `f`, `f <=< return` is the same as writing just `f` and the right identity law says that `return <=< f` is also no different from `f`.
This is very similar to how if `f` is a normal function, `(f . g) . h` is the same as `f . (g . h)`, `f . id` is always the same as `f` and `id . f` is also just `f`.
In this chapter, we took a look at the basics of monads and learned how the `Maybe` monad and the list monad work. In the next chapter, we'll take a look at a whole bunch of other cool monads and we'll also learn how to make our own.

クールだ。では、それが連想法則とどんな関係があるのだろう？まあ、この法則を合成の法則として見ると、f <=< (g <=< h)は(f <=< g) <=< h)と同じであるべきだということになる。
最初の 2 つの法則を<=<を使うように変換すると、左の恒等式はすべてのモナド関数 f について、f <=< return は f だけを書くのと同じであることを示し、右の恒等式は return <=< f も f と変わらないことを示す。
これは、f が通常の関数である場合、(f . g) . h は f . (g . h)と同じであり、f . id は常に f と同じであり、id . f もまた f と同じであることによく似ている。
この章では、モナドの基本を見て、Maybe モナドとリスト・モナドがどのように機能するかを学んだ。次の章では、他のクールなモナドをたくさん見て、独自のモナドを作る方法も学びます。