7

Functors

At the risk of sounding like a broken record, I will say this about functors: A functor is a very simple but powerful idea. Category theory is just full of those simple but powerful ideas. A functor is a mapping between categories. Given two categories, **C** and **D**, a functor F maps objects in **C** to objects in **D** — it's a function on objects. If a is an object in **C**, we'll write its image in **D** as Fa (no parentheses). But a category is not just objects — it's objects and morphisms that connect them. A functor also maps morphisms — it's a function on morphisms. But it doesn't map morphisms willy-nilly — it preserves connections. So if a morphism f in **C** connects object a to object b,

$$f :: a \to b$$

the image of f in **D**, Ff, will connect the image of a to the image of b:

$$Ff :: Fa \to Fb$$

7

Functors

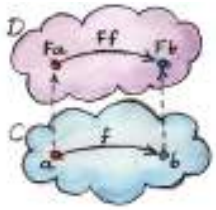
さて、ここでファンクタについて一言。ファンクターは非常に単純でありながら強力なアイデアです。カテゴリー理論には，このような単純だが強力な考え方がたくさんある．ファンクタはカテゴリ間の写像である．2つのカテゴリ□と□があるとき、ファンクタ□は□のオブジェクトに写像します - これはオブジェクトに対する関数です。 Ll_1D44E が Lu_1D402 のオブジェクトなら、その □ におけるイメージを □ Ll_1D44E と書く（括弧はつけない）。しかし、カテゴリは単なるオブジェクトではなく、オブジェクトとそれらをつなぐモルヒズムである。ファンクタはモルヒズムを写すものであり、モルヒズムに対する関数である。しかし、ファンクタはモルヒズムを無闇に写すわけではなく、接続を保存するのです。つまり、もし □ のモルヒズム □ がオブジェクト □ に接続するならば、それは

$$f :: a \to b$$

の □ の像、□ は □ の像を □ の像につなげます。

$$Ff :: Fa \to Fb$$

(This is a mixture of mathematical and Haskell notation that hopefully makes sense by now. I won't use parentheses when applying functors to objects or morphisms.)

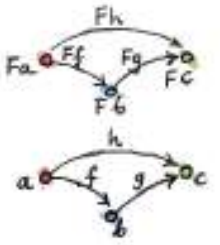


As you can see, a functor preserves the structure of a category: what's connected in one category will be connected in the other category. But there's something more to the structure of a category: there's also the composition of morphisms. If h is a composition of f and g:

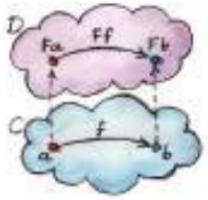
$$h = g.f$$

we want its image under F to be a composition of the images of f and g:

$$Fh = Fg \,.\, Ff$$



(これは数学とHaskellの表記が混ざったものですが、もう意味がわかるといいですね。ファンクタをオブジェクトやモルヒズムに適用するときには括弧を使わないことにします)。

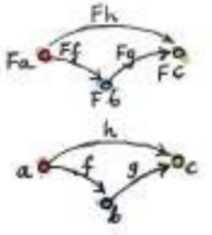


見ての通り，ファンクタはカテゴリの構造を保持します：あるカテゴリで接続されているものは，他のカテゴリでも接続されます．しかし、カテゴリの構造にはもっと別のものがあります。モルヒズムの合成もあるのです。 Ll_210E が ▯ と ▯ の合成である場合。

$$h = g.f$$

は、 Lu_1D439 の下のその像が ▯ と ▯ の像の合成であってほしいのです。

$$Fh = Fg \,.\, Ff$$



Finally, we want all identity morphisms in **C** to be mapped to identity morphisms in **D**:

$$F\mathbf{id}_a = \mathbf{id}_{Fa}$$

Here, $\mathbf{id}_a$ is the identity at the object a, and $\mathbf{id}_{Fa}$ the identity at Fa. Note that these conditions make functors much more restrictive than regular functions. Functors must preserve the structure of a category. If you picture a category as a collection of objects held together by a network of morphisms, a functor is not allowed to introduce any tears into this fabric. It may smash objects together, it may glue multiple morphisms into one, but it may never break things apart. This no-tearing constraint is similar to the continuity condition you might know from calculus. In this sense functors are "continuous" (although there exists an even more restrictive notion of continuity for functors). Just like functions, functors may do both collapsing and embedding. The embedding aspect is more prominent when the source category is much smaller than the target category. In the extreme, the source can be the trivial singleton category — a category with one object and one morphism (the identity). A functor from the singleton category to any other category simply selects an object in that category. This is fully analogous to the property of morphisms from singleton sets selecting elements in target sets. The maximally collapsing functor is called the constant functor Δ_c. It maps every object in the source category to one selected object c in the target category. It also maps every morphism in the source category to the identity morphism $\mathbf{id}_c$. It acts like a black hole, compacting everything into one singularity. We'll see more of this functor when we discuss limits and colimits.

最後に、□のすべての恒等モルヒズムは□の恒等モルヒズムに写像されることを望む。

$$F\mathbf{id}_a = \mathbf{id}_{Fa}$$

ここで、id　Ll_lD44E はオブジェクト　Ll_lD44E での恒等式、id □ は　Lu_lD44E での恒等式である。これらの条件により、ファンクタは通常の関数よりもはるかに制限されることに注意。ファンクタはカテゴリの構造を保持しなければならない．もし，カテゴリを形態素のネットワークによって保持されたオブジェクトの集合体と見なすなら，ファンクタはこの布にいかなる裂け目も入れることはできないのです．ファンクタはオブジェクト同士をくっつけたり、複数のモルヒズムを1つにくっつけたりすることはできますが、決して物事をバラバラにしてしまうことはできません。この「破れない」という制約は、微積分でおなじみの「連続性」の条件に似ている。この意味で，ファンクタは「連続」である（ただし，ファンクタにはより限定的な連続性の概念が存在する）．関数と同じように、ファンクタも折りたたみと埋め込みの両方を行うことができる。埋め込みは，元カテゴリが対象カテゴリよりずっと小さい場合に顕著になる．極端な話、ソースは些細なシングルトン・カテゴリー、つまり、1つのオブジェクトと1つのモルヒズム（同一性）を持つカテゴリーであってもいいのです。シングルトン・カテゴリから他のカテゴリへのファンクタは、単にそのカテゴリ内のオブジェクトを選択する。これは、シングルトン集合からのモルヒズムが対象集合の要素を選択する性質と完全に類似している。最大公約数的なファンクターは、定数ファンクターΔ □と呼ばれる。これは、元カテゴリーのすべてのオブジェクトを、ターゲットカテゴリーで選択された1つのオブジェクトŐに写像する。また、元カテゴリーのすべてのモルヒズムを恒等モルヒズムid Őに写像する。これはブラックホールのように作用し、すべてを一つの特異点に圧縮する。このファンクタについては、極限と反極限について議論するときに、もっと詳しく見ることにしよう。

7.1 Functors in Programming

Let's get down to earth and talk about programming. We have our category of types and functions. We can talk about functors that map this category into itself — such functors are called endofunctors. So what's an endofunctor in the category of types? First of all, it maps types to types. We've seen examples of such mappings, maybe without realizing that they were just that. I'm talking about definitions of types that were parameterized by other types. Let's see a few examples.

7.1.1 The Maybe Functor

The definition of `Maybe` is a mapping from type `a` to type `Maybe a`:

```
data Maybe a = Nothing | Just a
```

Here's an important subtlety: `Maybe` itself is not a type, it's a *type constructor*. You have to give it a type argument, like `Int` or `Bool`, in order to turn it into a type. `Maybe` without any argument represents a function on types. But can we turn `Maybe` into a functor? (From now on, when I speak of functors in the context of programming, I will almost always mean endofunctors.) A functor is not only a mapping of objects (here, types) but also a mapping of morphisms (here, functions). For any function from `a` to `b`:

```
f :: a -> b
```

we would like to produce a function from `Maybe a` to `Maybe b`. To define such a function, we'll have two cases to consider, corresponding to the two constructors of `Maybe`. The `Nothing` case is simple: we'll just return `Nothing` back. And if the argument is `Just`, we'll apply the function `f` to its contents. So the image of `f` under `Maybe` is the function:

7.1 Functors in Programming

さて、本題に入り、プログラミングの話をしましょう。型と関数のカテゴリがある。このカテゴリを自分自身に写すファンクタについて話すことができます。このようなファンクタはエンドファンクタと呼ばれます。では、型のカテゴリーにおけるエンドファンクタとは何でしょうか？まず、型から型への写像です。このような写像の例を見たことがあると思いますが、もしかしたら、そのような写像であることに気づいていないかもしれません。つまり、他の型によってパラメータ化された型の定義についてです。いくつかの例を見てみましょう。

7.1.1 The Maybe Functor

Maybeの定義はタイプaからタイプMaybe aへのマッピングである：

```
data Maybe a = Nothing | Just a
```

ここで重要な微妙な点として、Maybe 自身は型ではなく、型コンストラクタであることが挙げられます。型に変換するためには、Int や Bool のような型引数を与えなければならない。引数のないMaybeは、型に対する関数を表している。しかし、Maybeをファンクタにすることはできるだろうか？(今後、プログラミングの文脈でファンクターといえば、ほとんどエンドファンクターを指すことになるだろう)。ファンクタはオブジェクト(ここでは型)の写像であるだけでなく，モルヒズム(ここでは関数)の写像でもある．aからbへの任意の関数に対して．

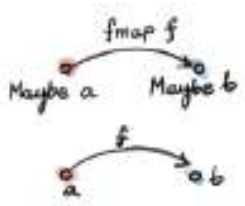
```
f :: a -> b
```

という関数に対して、Maybe a から Maybe b への関数を作りたい。このような関数を定義するために、Maybe の2つのコンストラクタに対応する2つのケースを考える必要がある。 Nothing のケースは単純で、Nothing を返すだけである。そして、引数が Just の場合は、その中身に関数 f を適用する。つまり、Maybeの下のfのイメージは関数になる。

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

(By the way, in Haskell you can use apostrophes in variables names, which is very handy in cases like these.) In Haskell, we implement the morphism-mapping part of a functor as a higher order function called `fmap`. In the case of `Maybe`, it has the following signature:

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



We often say that `fmap` *lifts* a function. The lifted function acts on `Maybe` values. As usual, because of currying, this signature may be interpreted in two ways: as a function of one argument — which itself is a function `(a -> b)` — returning a function `(Maybe a -> Maybe b)`; or as a function of two arguments returning `Maybe b`:

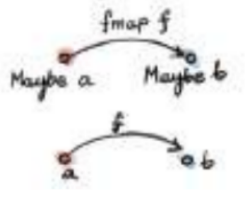
```
fmap :: (a -> b) -> Maybe a -> Maybe b
```

Based on our previous discussion, this is how we implement `fmap` for `Maybe`:


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

（ちなみに、Haskellでは変数名にアポストロフィを使うことができるので、こういう場合はとても便利です）Haskellでは、ファンクタのモルヒズム・マッピング部分をfmapという高次関数で実装する。Maybeの場合、それは次のようなシグネチャを持っています。

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



我々はしばしば `fmap` が関数を持ち上げると言う。リフトされた関数はMaybeの値に対して作用する。いつものように、currying のために、このシグネチャは2つの方法で解釈されるかもしれない: 1つの引数の関数として - それ自身は関数 `(a -> b)` - 関数 `(Maybe a -> Maybe b)` を返す ; または2つの引数の関数として `Maybe b` を返す :

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

これまでの議論を踏まえて、`Maybe` に対する `fmap` の実装は次のようになります。

```haskell
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
```

To show that the type constructor `Maybe` together with the function `fmap` form a functor, we have to prove that `fmap` preserves identity and composition. These are called "the functor laws," but they simply ensure the preservation of the structure of the category.

7.1.2 Equational Reasoning

To prove the functor laws, I will use *equational reasoning*, which is a common proof technique in Haskell. It takes advantage of the fact that Haskell functions are defined as equalities: the left hand side equals the right hand side. You can always substitute one for another, possibly renaming variables to avoid name conflicts. Think of this as either inlining a function, or the other way around, refactoring an expression into a function. Let's take the identity function as an example:

```haskell
id x = x
```

If you see, for instance, `id y` in some expression, you can replace it with `y` (inlining). Further, if you see `id` applied to an expression, say `id (y + 2)`, you can replace it with the expression itself `(y + 2)`. And this substitution works both ways: you can replace any expression `e` with `id e` (refactoring). If a function is defined by pattern matching, you can use each sub-definition independently. For instance, given the above definition of `fmap` you can replace `fmap f Nothing` with `Nothing`, or the other way around. Let's see how this works in practice. Let's start with the preservation of identity:


```haskell
fmap _ Nothing = Nothing
fmap f (Just x) = Just (f x)
```

型構成子 `Maybe` と関数 `fmap` がファンクタを構成していることを示すには、`fmap` が同一性と合成を保存することを証明しなければならない。これらは「ファンクタの法則」と呼ばれるが、単にカテゴリの構造の保存を保証するものである。

7.1.2 Equational Reasoning

ファンクタ法則を証明するために、Haskellでよく使われる証明技法である等式推論を使うことにする。これは、`Haskell` の関数が等式で定義されることを利用したもので、左辺は右辺と等しい。また、名前の衝突を避けるために変数の名前を変更することもできます。これは関数のインライン化、またはその逆で式を関数にリファクタリングしていると考えてください。例えば、identity関数を例にとって考えてみましょう。

```haskell
id x = x
```

例えば、ある式の中にid yというのがあったら、それをyに置き換える（インライン化）ことができる。さらに、ある式に `id` が適用されている場合、たとえば `id (y + 2)` とすれば、式そのもの `(y + 2)` に置き換えることができます。また、この置き換えは双方向に作用し、任意の式eを `id e`に置き換えることができる（リファクタリング）。関数がパターンマッチで定義されている場合，各サブ定義を独立して使用することができます．たとえば、上のfmapの定義があれば、`fmap f Nothing`をNothingに置き換えることができますし、その逆も可能です。これが実際にどのように働くか見てみよう。まず、同一性の保存から始めましょう。

```
fmap id = id
```

There are two cases to consider: `Nothing` and `Just`. Here's the first case (I'm using Haskell pseudo-code to transform the left hand side to the right hand side):

```
  fmap id Nothing
= { definition of fmap }
  Nothing
= { definition of id }
  id Nothing
```

Notice that in the last step I used the definition of `id` backwards. I replaced the expression `Nothing` with `id Nothing`. In practice, you carry out such proofs by "burning the candle at both ends," until you hit the same expression in the middle — here it was `Nothing`. The second case is also easy:

```
  fmap id (Just x)
= { definition of fmap }
  Just (id x)
= { definition of id }
  Just x
= { definition of id }
  id (Just x)
```

Now, lets show that `fmap` preserves composition:

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

First the `Nothing` case:


```
fmap id = id
```

考えるべきは2つのケースである。`Nothing` と `Just` です。以下は最初のケースです（左辺を右辺に変換するためにHaskellの擬似コードを使っています）。

```
  fmap id Nothing
= { definition of fmap }
  Nothing
= { definition of id }
  id Nothing
```

最後のステップで、私はidの定義を逆に使ったことに注意してください。Nothingという式をid Nothingに置き換えたのです。実際には、このような証明は「両端のろうそくを燃やす」ようにして、真ん中の同じ式にぶつかるまで実行します（ここではNothingでした）。2番目のケースも簡単だ。

```
  fmap id (Just x)
= { definition of fmap }
  Just (id x)
= { definition of id }
  Just x
= { definition of id }
  id (Just x)
```

さて、fmapが合成を維持することを示しましょう。

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

First the `Nothing` case:

```
  fmap (g . f) Nothing
= { definition of fmap }
  Nothing
= { definition of fmap }
  fmap g Nothing
= { definition of fmap }
  fmap g (fmap f Nothing)
```

And then the `Just` case:

```
  fmap (g . f) (Just x)
= { definition of fmap }
  Just ((g . f) x)
= { definition of composition }
  Just (g (f x))
= { definition of fmap }
  fmap g (Just (f x))
= { definition of fmap }
  fmap g (fmap f (Just x))
= { definition of composition }
  (fmap g . fmap f) (Just x)
```

It's worth stressing that equational reasoning doesn't work for C++ style "functions" with side effects. Consider this code:

```
int square(int x) {
    return x * x;
}

int counter() {
    static int c = 0;
    return c++;
}
```



```
  fmap (g . f) Nothing
= { definition of fmap }
  Nothing
= { definition of fmap }
  fmap g Nothing
= { definition of fmap }
  fmap g (fmap f Nothing)
```

And then the `Just` case:

```
  fmap (g . f) (Just x)
= { definition of fmap }
  Just ((g . f) x)
= { definition of composition }
  Just (g (f x))
= { definition of fmap }
  fmap g (Just (f x))
= { definition of fmap }
  fmap g (fmap f (Just x))
= { definition of composition }
  (fmap g . fmap f) (Just x)
```


等式推論は、副作用のあるC++形式の「関数」には使えないことを強調しておく。次のようなコードを考えてみよう。

```
int square(int x) {
    return x * x;
}

int counter() {
    static int c = 0;
    return c++;
}
```

```
double y = square(counter());
```

Using equational reasoning, you would be able to inline `square` to get:

```
double y = counter() * counter();
```

This is definitely not a valid transformation, and it will not produce the same result. Despite that, the C++ compiler will try to use equational reasoning if you implement `square` as a macro, with disastrous results.

7.1.3 Optional

Functors are easily expressed in Haskell, but they can be defined in any language that supports generic programming and higher-order functions. Let's consider the C++ analog of `Maybe`, the template type `optional`. Here's a sketch of the implementation (the actual implementation is much more complex, dealing with various ways the argument may be passed, with copy semantics, and with the resource management issues characteristic of C++):

```
template<class T>
class optional {
    bool _isValid; // the tag
    T _v;
public:
    optional()    : _isValid(false) {}        // Nothing
    optional(T x) : _isValid(true) , _v(x) {} // Just
    bool isValid() const { return _isValid; }
    T val() const { return _v; } };
```

This template provides one part of the definition of a functor: the mapping of types. It maps any type `T` to a new type `optional<T>`. Let's define its action on functions:


```
double y = square(counter());
```

等式推論を使うと、二乗をインライン化することができます。

```
double y = counter() * counter();
```

これは間違いなく有効な変換ではありませんし、同じ結果を得ることはできません。にもかかわらず、C++ コンパイラは `square` をマクロとして実装すると等式推論を使おうとし、悲惨な結果になります。

7.1.3 Optional

ファンクタは Haskell では簡単に表現できますが、ジェネリックプログラミングと高階関数をサポートするどのような言語でも定義することができます。ここでは、C++のMaybeの類似品であるテンプレート型optionalを考えてみましょう。実際の実装はもっと複雑で、引数を渡すさまざまな方法、コピーセマンティクス、C++特有のリソース管理の問題などを扱っています）。

```
template<class T>
class optional {
    bool _isValid; // the tag
    T _v;
public:
    optional()    : _isValid(false) {}        // Nothing
    optional(T x) : _isValid(true) , _v(x) {} // Just
    bool isValid() const { return _isValid; }.
    T val() const { return _v; } };
```

このテンプレートは，ファンクタの定義の一部である，型のマッピングを提供し ます．これは，任意の型Tを新しい型optional<T>に写像します．では，関数に対する作用を定義してみましょう．

```
template<class A, class B>
std::function<optional<B>(optional<A>)>
fmap(std::function<B(A)> f) {
    return [f](optional<A> opt) {
        if (!opt.isValid())
            return optional<B>{};
        else
            return optional<B>{ f(opt.val()) };
    };
}
```

This is a higher order function, taking a function as an argument and returning a function. Here's the uncurried version of it:

```
template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
    if (!opt.isValid())
        return optional<B>{};
    else
        return optional<B>{ f(opt.val()) };
}
```

There is also an option of making `fmap` a template method of `optional`. This embarrassment of choices makes abstracting the functor pattern in C++ a problem. Should functor be an interface to inherit from (unfortunately, you can't have template virtual functions)? Should it be a curried or an uncurried free template function? Can the C++ compiler correctly infer the missing types, or should they be specified explicitly? Consider a situation where the input function `f` takes an `int` to a `bool`. How will the compiler figure out the type of `g`:


```
template<class A, class B>
std :: function < optional < B > (optional < A > ) >.
fmap(std::function<B(A)> f) {
    return [f](optional<A> opt) {
        if (!opt.isValid())
            return optional<B>{};
        else
            return optional<B>{ f(opt.val()) };
    };
}
```

これは、関数を引数として受け取り、関数を返すという高次の関数です。以下は、その非循環バージョンです。

```
template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
    if (!opt.isValid())
        return optional<B>{};
    else
        return optional<B>{ f(opt.val()) };
}
```

fmapをoptionalのテンプレートメソッドにするオプションもあります。この選択肢の多さが、C++におけるファンクタパターンの抽象化を悩ませている。ファンクタは継承するインタフェースであるべきなのか（残念ながらテンプレート仮想関数は持てない）？ファンクタは，カールされたテンプレート関数であるべきか，カールされていない自由なテンプレート関数であるべきか？C++コンパイラは足りない型を正しく推論できるでしょうか、それとも明示的に指定すべきでしょうか？入力関数 f が int を bool に変換する場合を考えてみましょう。コンパイラはどのようにして g の型が分かるのでしょうか？

```
auto g = fmap(f);
```

especially if, in the future, there are multiple functors overloading `fmap`? (We'll see more functors soon.)

7.1.4 Typeclasses

So how does Haskell deal with abstracting the functor? It uses the typeclass mechanism. A typeclass defines a family of types that support a common interface. For instance, the class of objects that support equality is defined as follows:

```
class Eq a where
    (==) :: a -> a -> Bool
```

This definition states that type `a` is of the class `Eq` if it supports the operator `(==)` that takes two arguments of type `a` and returns a `Bool`. If you want to tell Haskell that a particular type is `Eq`, you have to declare it an *instance* of this class and provide the implementation of `(==)`. For example, given the definition of a 2D `Point` (a product type of two `Floats`):

```
data Point = Pt Float Float
```

you can define the equality of points:

```
instance Eq Point where
    (Pt x y) == (Pt x' y') = x == x' && y == y'
```

Here I used the operator `(==)` (the one I'm defining) in the infix position between the two patterns `(Pt x y)` and `(Pt x' y')`. The body of the function follows the single equal sign. Once `Point` is declared


```
auto g = fmap(f);
```

特に、将来、fmapをオーバーロードするファンクタが複数出てきた場合、コンパイラはどのようにgの型を把握するのでしょうか？（もうすぐ、もっと多くのファンクタを見ることになるでしょう）。

7.1.4 Typeclasses

では、Haskell はファンクタの抽象化をどのように行っているのだろうか。それは、型クラスのメカニズムを使う。型クラスは、共通のインタフェースをサポートする型のファミリーを定義する。たとえば，等式をサポートするオブジェクトのクラスは次のように定義される．

```
class Eq a where
    (==) :: a -> a -> Bool
```

この定義によると、型 a は、型 a の 2 つの引数を取り Bool を返す演算子（==）をサポートする場合、Eq クラスであることが示されています ..... 。もし、ある特定の型がEqであることをHaskellに伝えたい場合は、その型をこのクラスのインスタンスとして宣言し、(==）の実装を提供しなければなりません。例えば，2次元の点（2つのFloat型の積）の定義があったとします．

```
data Point = Pt Float Float
```

とすると，点の等価性を定義することができます．

```
instance Eq Point where
    (Pt x y) == (Pt x' y') = x == x' && y == y'
```

ここでは、2つのパターン（Pt x y）と（Pt x’ y’）の間の infix 位置に演算子（==）（私が定義しているもの）を使っています．関数本体は、1つの等号に続いています。Point が宣言されると

an instance of Eq, you can directly compare points for equality. Notice that, unlike in C++ or Java, you don't have to specify the Eq class (or interface) when defining Point — you can do it later in client code. Typeclasses are also Haskell's only mechanism for overloading functions (and operators). We will need that for overloading fmap for different functors. There is one complication, though: a functor is not defined as a type but as a mapping of types, a type constructor. We need a typeclass that's not a family of types, as was the case with Eq, but a family of type constructors. Fortunately a Haskell typeclass works with type constructors as well as with types. So here's the definition of the Functor class:

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

It stipulates that f is a Functor if there exists a function fmap with the specified type signature. The lowercase f is a type variable, similar to type variables a and b. The compiler, however, is able to deduce that it represents a type constructor rather than a type by looking at its usage: acting on other types, as in f a and f b. Accordingly, when declaring an instance of Functor, you have to give it a type constructor, as is the case with Maybe:

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

By the way, the Functor class, as well as its instance definitions for a lot of simple data types, including Maybe, are part of the standard Prelude library.

が Eq のインスタンスとして宣言されると、ポイントを直接比較して等価性を確認することができます。C++やJavaと違って、Pointを定義するときにEqクラス（またはインターフェース）を指定する必要はなく、クライアントコードの中で後で指定することができることに注意してください。型クラスはまた、関数（および演算子）をオーバーロードするための Haskell の唯一のメカニズムでもあります。異なるファンクタに対してfmapをオーバーロードするために、この機構が必要になります。ファンクタは型としてではなく，型のマッピング，つまり型構成子として定義される．Eq のように型のファミリーではなく，型コンストラクタのファミリーである型クラスが必要だ．幸い Haskell の型クラスは型と同様に型コンストラクタでも機能する。そこで，Functor クラスの定義を以下に示します．

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

これは，指定された型シグネチャを持つ関数 fmap が存在すれば， f はファンクタであると規定しています．小文字のfは型変数で，型変数aやbと同じようなものです．しかし，コンパイラは，f a や f b のように他の型に作用するという使用法を見れば，それが型ではなく型コンストラクタを表していると推論することができる．したがって、Functor のインスタンスを宣言するときには、Maybe と同じように型コンストラクタを与えなければなりません。

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

ちなみに、Functor クラスと、Maybe を含む多くの単純なデータ型のインスタンス定義は、標準的な Prelude ライブラリの一部です。

7.1.5 Functor in C++

Can we try the same approach in C++? A type constructor corresponds to a template class, like `optional`, so by analogy, we would parameterize `fmap` with a *template template parameter* `F`. This is the syntax for it:

```
template<template<class> F, class A, class B>
F<B> fmap(std::function<B(A)>, F<A>);
```

We would like to be able to specialize this template for different functors. Unfortunately, there is a prohibition against partial specialization of template functions in C++. You can't write:

```
template<class A, class B>
optional<B> fmap<optional>(std::function<B(A)> f, optional<A> opt)
```

Instead, we have to fall back on function overloading, which brings us back to the original definition of the uncurried `fmap`:

```
template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
    if (!opt.isValid())
        return optional<B>{};
    else
        return optional<B>{ f(opt.val()) };
}
```

This definition works, but only because the second argument of `fmap` selects the overload. It totally ignores the more generic definition of `fmap`.

7.1.5 Functor in C++

C++ でも同じようなことができるでしょうか？型コンストラクタは optional のようなテンプレート・クラスに対応しているので、類似の方法で fmap をテンプレート・パラメータ F でパラメータ化することになります。これはそのための構文です。

```
template<template<class > F, class A , class B>.
F<B> fmap(std::function<B(A)>, F<A>);
```

このテンプレートを異なるファンクタのために特化することができればと思います．残念ながら，C++ではテンプレート関数の部分的な特殊化は禁止されています．書けないのです。

```
template<class A, class B>
オプショナル < B > fmap < オプショナル > (std :: function < B(A) > f, オプショナル < A > opt)
```

そこで、関数のオーバーロードに頼るしかないのですが、そこで、元のfmapの定義に 戻ることになります。

```
template<class A, class B>
optional<B> fmap(std::function<B(A)> f, optional<A> opt) {
    if (!opt.isValid())
        return optional<B>{};
    else
        return optional<B>{ f(opt.val()) };
}
```

この定義はうまくいくが、それはfmapの第2引数がオーバーロードを選択するからである。これは、より一般的なfmapの定義を完全に無視しています。

7.1.6 The List Functor

To get some intuition as to the role of functors in programming, we need to look at more examples. Any type that is parameterized by another type is a candidate for a functor. Generic containers are parameterized by the type of the elements they store, so let's look at a very simple container, the list:

```
data List a = Nil | Cons a (List a)
```

We have the type constructor `List`, which is a mapping from any type `a` to the type `List a`. To show that `List` is a functor we have to define the lifting of functions: Given a function `a -> b` define a function `List a -> List b`:

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

A function acting on `List a` must consider two cases corresponding to the two list constructors. The `Nil` case is trivial — just return `Nil` — there isn't much you can do with an empty list. The `Cons` case is a bit tricky, because it involves recursion. So let's step back for a moment and consider what we are trying to do. We have a list of `a`, a function `f` that turns `a` to `b`, and we want to generate a list of `b`. The obvious thing is to use `f` to turn each element of the list from `a` to `b`. How do we do this in practice, given that a (non-empty) list is defined as the `Cons` of a head and a tail? We apply `f` to the head and apply the lifted (`fmap`ped) `f` to the tail. This is a recursive definition, because we are defining lifted `f` in terms of lifted `f`:

```
fmap f (Cons x t) = Cons (f x) (fmap f t)
```


7.1.6 The List Functor

プログラミングにおけるファンクタの役割について直感的に理解するために、 もっと多くの例を見てみる必要があります。他の型によってパラメタライズされる型はすべてファンクタの候補です．一般的なコンテナは、格納する要素の型によってパラメータ化されるので、 非常に単純なコンテナであるリストを見てみましょう。

```
data List a = Nil | Cons a ( List a)
```

リストがファンクタであることを示すには、関数のリフティングを定義する必要があります。関数 `a -> b` が与えられたとき，関数 `List a -> List b` を定義します．

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

リスト `a` に作用する関数は、2つのリストコンストラクタに対応する2つのケースを考慮しなければなりません。`Nil`の場合は些細なことで、ただ`Nil`を返せばよいのです。`Cons`の場合は再帰を含むので少し厄介です。では、一歩下がって、私たちが何をしようとしているのかを考えてみましょう。aのリストがあり、aをbに変換する関数fがあり、bのリストを生成したい。当然ながら、fを使ってリストの各要素をaからbに変換することになります。空でない）リストは、先頭と末尾の`Cons`として定義されることを考えると、実際にはどうすればよいのでしょうか。頭部にfを適用し、尾部には`fmap f`のfを適用するのです。これは再帰的な定義です。なぜなら、`lifted f`を`lifted f`の観点から定義しているからです。

```
fmap f ( Cons x t) = Cons (f x) (fmap f t)
```

Notice that, on the right hand side, `fmap f` is applied to a list that's shorter than the list for which we are defining it — it's applied to its tail. We recurse towards shorter and shorter lists, so we are bound to eventually reach the empty list, or `Nil`. But as we've decided earlier, `fmap f` acting on `Nil` returns `Nil`, thus terminating the recursion. To get the final result, we combine the new head `(f x)` with the new tail `(fmap f t)` using the `Cons` constructor. Putting it all together, here's the instance declaration for the list functor:

```
instance Functor List where
    fmap _ Nil = Nil
    fmap f (Cons x t) = Cons (f x) (fmap f t)
```

If you are more comfortable with C++, consider the case of a `std::vector`, which could be considered the most generic C++ container. The implementation of `fmap` for `std::vector` is just a thin encapsulation of `std::transform`:

```
template<class A, class B>
std::vector<B> fmap(std::function<B(A)> f, std::vector<A> v) {
    std::vector<B> w;
    std::transform( std::begin(v)
                  , std::end(v)
                  , std::back_inserter(w)
                  , f);
    return w;
}
```

We can use it, for instance, to square the elements of a sequence of numbers:

右側で、fmap fは定義されたリストより短いリストに適用されていることに注意してください。どんどん短いリストに向かって再帰していくので、最終的には空のリスト、つまりNilに到達するはずです。しかし、先ほど決めたように、Nilに対してfmap fを実行するとNilが返されるので、再帰は終了します。最終的な結果を得るには、Consコンストラクタを使って、新しい先頭（f x）と新しい末尾（fmap f t）を結合します。以上をまとめると、リストファンクタのインスタンス宣言は次のようになります。

```
instance Functor List where
    fmap _ Nil = Nil
    fmap f ( Cons x t) = Cons (f x) (fmap f t)
```

C++に慣れている人は，C++の最も一般的なコンテナであるstd::vectorの場合 を考えてみましょう．std::vector の fmap の実装は std::transform を薄くカプセル化したものに過ぎません．

```
template<class A, class B>
std::vector<B> fmap(std::function<B(A)> f, std::vector<A> v) {
    std::vector<B> w;
    std::transform( std::begin(v)
                  , std::end(v)
                  , std::back_inserter(w)
                  , f);
    return w;
}
```

これを利用して，例えば，数列の要素を2乗することができます．

```
std::vector<int> v{ 1, 2, 3, 4 };
auto w = fmap([](int i) { return i*i; }, v);
std::copy( std::begin(w)
         , std::end(w)
         , std::ostream_iterator(std::cout, ", "));
```

Most C++ containers are functors by virtue of implementing iterators that can be passed to `std::transform`, which is the more primitive cousin of `fmap`. Unfortunately, the simplicity of a functor is lost under the usual clutter of iterators and temporaries (see the implementation of `fmap` above). I'm happy to say that the new proposed C++ range library makes the functorial nature of ranges much more pronounced.

7.1.7 The Reader Functor

Now that you might have developed some intuitions — for instance, functors being some kind of containers — let me show you an example which at first sight looks very different. Consider a mapping of type `a` to the type of a function returning `a`. We haven't really talked about function types in depth — the full categorical treatment is coming — but we have some understanding of those as programmers. In Haskell, a function type is constructed using the arrow type constructor `(->)` which takes two types: the argument type and the result type. You've already seen it in infix form, `a -> b`, but it can equally well be used in prefix form, when parenthesized:

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

Just like with regular functions, type functions of more than one argument can be partially applied. So when we provide just one type argument to the arrow, it still expects another one. That's why:


```
std::vector<int> v{ 1, 2, 3, 4 };
auto w = fmap([](int i) { return i*i; }, v);
std::copy( std::begin(w)
         , std::end(w)
         , std::ostream_iterator(std::cout, ", "));
```


ほとんどの C++ コンテナは，fmap のより原始的ないとこである std::transform に渡すことができるイテレータを実装しているため，ファンクタとなっています．残念ながら，イテレータやテンポラリのせいで，ファンクタのシンプルさは失われています（上のfmapの実装を参照してください）．新しいC++レンジライブラリの提案によって，レンジのファンクタとしての性質がより顕著になったことは嬉しい限りです．

7.1.7 The Reader Functor

さて，みなさんは，ファンクタがある種のコンテナであるという直観をお持ちかもしれま せんが，一見するとまったく違うように見える例をお見せしましょう．a型を返す関数の型へのa型の写像を考えてみよう。 関数の型については、まだ深く話していない--完全なカテゴリカルな扱いはこれからだが--が、プログラマとしてはある程度理解している。Haskellでは、関数型は矢印型コンストラクタ(->)を使って構築されます。すでに見たように、a -> b のような infix 形式の関数ですが、括弧でくくると prefix 形式でも同じように使えます。


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


通常の関数と同じように、2つ以上の引数を持つ型関数は部分的に適用することができます。ですから、矢印に1つの型引数だけを与えても、矢印はもう1つの型引数を予期しています。そういうわけです。

```
(->) a
```

is a type constructor. It needs one more type `b` to produce a complete type `a -> b`. As it stands, it defines a whole family of type constructors parameterized by `a`. Let's see if this is also a family of functors. Dealing with two type parameters can get a bit confusing, so let's do some renaming. Let's call the argument type `r` and the result type `a`, in line with our previous functor definitions. So our type constructor takes any type `a` and maps it into the type `r -> a`. To show that it's a functor, we want to lift a function `a -> b` to a function that takes `r -> a` and returns `r -> b`. These are the types that are formed using the type constructor `(->) r` acting on, respectively, `a` and `b`. Here's the type signature of `fmap` applied to this case:

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

We have to solve the following puzzle: given a function `f :: a -> b` and a function `g :: r -> a`, create a function `r -> b`. There is only one way we can compose the two functions, and the result is exactly what we need. So here's the implementation of our `fmap`:

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

It just works! If you like terse notation, this definition can be reduced further by noticing that composition can be rewritten in prefix form:

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

and the arguments can be omitted to yield a direct equality of two functions:


```
(->) a
```

は型コンストラクタです。これは完全な型 `a -> b` を生成するためにもう1つの型 `b` を必要とします。このままでは，`a` をパラメータとする型コンストラクタのファミリ全体が定義されてしまいます． これがファンクタのファミリでもあるかどうか見てみましょう．2つの型パラメータを扱うと少し混乱するので、名前を変えてみましょう。前のファンクタの定義と同じように、引数を `r` 型、結果を `a` 型と呼ぶことにしましょう。この型コンストラクタは任意の型 `a` を受け取り、それを `r -> a` 型に写像します。 これがファンクタであることを示すために、関数 `a -> b` を、`r -> a` を受け取り `r -> b` を返す関数にリフトしたいと思います。これらは，それぞれ `a` と `b` に作用する型構成子 `(->) r` を用いて形成される型です．この場合に適用されるfmapの型シグネチャを以下に示します．

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

関数`f :: a -> b`と関数`g :: r -> a`が与えられたとき，関数`r -> b`を作るというパズルを解かなければなりません．この2つの関数を合成する方法は1つしかなく、その結果はまさに私たちが必要としているものです。以下は、このfmapの実装です。

```
instance ファンクタ (( -> ) r) where
    fmap f g = f . g
```

これでうまくいきました。もし、あなたが簡潔な記法を好むなら、この定義は、合成が接頭辞の形で書き換えられることに注意することで、さらに減らすことができます。

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

また、引数を省略することで、2つの関数を直接等価にすることができます。

```
fmap = (.)
```

This combination of the type constructor `(->) r` with the above implementation of `fmap` is called the reader functor.

7.2 Functors as Containers

We've seen some examples of functors in programming languages that define general-purpose containers, or at least objects that contain some value of the type they are parameterized over. The reader functor seems to be an outlier, because we don't think of functions as data. But we've seen that pure functions can be memoized, and function execution can be turned into table lookup. Tables are data. Conversely, because of Haskell's laziness, a traditional container, like a list, may actually be implemented as a function. Consider, for instance, an infinite list of natural numbers, which can be compactly defined as:

```
nats :: [Integer]
nats = [1..]
```

In the first line, a pair of square brackets is Haskell's built-in type constructor for lists. In the second line, square brackets are used to create a list literal. Obviously, an infinite list like this cannot be stored in memory. The compiler implements it as a function that generates `Integers` on demand. Haskell effectively blurs the distinction between data and code. A list could be considered a function, and a function could be considered a table that maps arguments to results. The latter can even be practical if the domain of the function is finite and not too large. It would not be practical, however, to implement `strlen` as table lookup, because there are infinitely many different strings. As programmers, we


```
fmap = (.)
```

この型構成子(->)rと上記のfmapの実装の組み合わせをリーダファンクタと呼びます。

7.2 Functors as Containers

これまで、プログラミング言語におけるファンクタの例として、汎用的なコンテナ、 あるいは、少なくともパラメータ化された型の値を含むオブジェクトを定義する ものをいくつか見てきました。私たちは関数をデータとして考えていないので、リーダファンクタは異端児のように 思われます。しかし、純粋な関数がメモ化され、関数の実行がテーブル検索に変わることを見てきた。テーブルはデータなのだ。逆に、Haskellの怠惰な性格から、リストのような伝統的なコンテナは、実は関数として実装されることがあります。例えば、自然数の無限リストを考えてみよう。これはコンパクトに次のように定義できる。

```
nats :: [Integer]
nats = [1..]
```

最初の行の角括弧のペアはリストに対するHaskellの組み込み型コンストラクタです。2行目の角括弧はリストリテラルを作成するために使用されます。もちろん、このような無限リストをメモリに格納することはできない。コンパイラは、必要に応じてInteger sを生成する関数として実装しています。Haskellはデータとコードの区別を効果的に曖昧にします。リストは関数とみなすことができるし、関数は引数と結果を対応付ける表とみなすこともできる。後者は、関数のドメインが有限で、大きすぎなければ、実用的でさえある。しかし、文字列は無限にあるので、strlenを表引きで実装するのは実用的ではないでしょう。プログラマとして、私たちは

don't like infinities, but in category theory you learn to eat infinities for breakfast. Whether it's a set of all strings or a collection of all possible states of the Universe, past, present, and future — we can deal with it! So I like to think of the functor object (an object of the type generated by an endofunctor) as containing a value or values of the type over which it is parameterized, even if these values are not physically present there. One example of a functor is a C++ `std::future`, which may at some point contain a value, but it's not guaranteed it will; and if you want to access it, you may block waiting for another thread to finish execution. Another example is a Haskell `IO` object, which may contain user input, or the future versions of our Universe with "Hello World!" displayed on the monitor. According to this interpretation, a functor object is something that may contain a value or values of the type it's parameterized upon. Or it may contain a recipe for generating those values. We are not at all concerned about being able to access the values — that's totally optional, and outside of the scope of the functor. All we are interested in is to be able to manipulate those values using functions. If the values can be accessed, then we should be able to see the results of this manipulation. If they can't, then all we care about is that the manipulations compose correctly and that the manipulation with an identity function doesn't change anything. Just to show you how much we don't care about being able to access the values inside a functor object, here's a type constructor that ignores completely its argument `a`:

```
data Const c a = Const c
```

The `Const` type constructor takes two types, `c` and `a`. Just like we did with the arrow constructor, we are going to partially apply it to create a functor. The data constructor (also called `Const`) takes just one value

しかし、カテゴリ理論では、無限を朝食として食べることを学びます。すべての文字列の集合であろうと、宇宙の過去、現在、未来のすべての可能な状態の集合であろうと、それを扱うことができるのです。だから私は、ファンクタオブジェクト（エンドファンクタが生成する型のオブジェクト）には、たとえ物理的にそこに値が存在しなくても、パラメータ化されている型の値（あるいは値）が含まれていると考えたい。ファンクタの一例はC++の`std::future`で，これはある時点で値を含むかも知れませんが，保証はされておらず，アクセスしようとすると，他のスレッドの実行終了を待つためにブロックされるかも知れません．他の例としては、HaskellのIOオブジェクトがあり、そこにはユーザの入力や、モニターに「Hello World！」と表示された我々の宇宙の未来版があるかもしれません。この解釈では、ファンクタオブジェクトは、パラメータ化された型の値や値を含む可能性があるものです。あるいは、それらの値を生成するためのレシピを含むかもしれません。私たちは、値にアクセスできるかどうかには全く関心がありません - それは完全にオプションであり、ファンクタの範囲外です。私たちが関心を持つのは、関数を使ってそれらの値を操作できることです。もし値にアクセスできるのであれば、その操作の結果を見ることができるはずです。もしアクセスできないなら、私たちが気にするのは、操作が正しく合成されることと、恒等式関数を使った操作が何も変えないことだ。ファンクタオブジェクトの中の値にアクセスできることがどれだけ重要でないかを示すために、引数aを完全に無視する型コンストラクタを以下に示します。

```
data Const c a = Const c
```

Const型コンストラクタはcとaの2つの型を取ります。 矢印のコンストラクタでやったのと同じように、これを部分的に適用してファンクタを作ることにします。データコンストラクタ（Constとも呼ばれます）は、1つの値だけを取ります。

of type c. It has no dependence on a. The type of fmap for this type constructor is:

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

Because the functor ignores its type argument, the implementation of fmap is free to ignore its function argument — the function has nothing to act upon:

```
instance Functor (Const c) where
    fmap _ (Const v) = Const v
```

This might be a little clearer in C++ (I never thought I would utter those words!), where there is a stronger distinction between type arguments — which are compile-time — and values, which are run-time:

```
template<class C, class A>
struct Const {
    Const(C v) : _v(v) {}
    C _v;
};
```

The C++ implementation of fmap also ignores the function argument and essentially re-casts the Const argument without changing its value:

```
template<class C, class A, class B>
Const<C, B> fmap(std::function<B(A)> f, Const<C, A> c) {
    return Const<C, B>{c._v};
}
```

Despite its weirdness, the Const functor plays an important role in many constructions. In category theory, it's a special case of the Δ_c functor I mentioned earlier — the endo-functor case of a black hole. We'll be seeing more of it in the future.

を取ります。この型コンストラクタの fmap の型は次のとおりです。

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

ファンクタは型の引数を無視するので，fmapの実装は関数の引数を自由に無視することができます．

```
instance ファンクタ ( Const c) where
    fmap _ (Const v) = Const v
```

C++では、型引数（コンパイル時）と値（実行時）の区別がもっとはっきりしてい ますから、この点はもう少しはっきりしているかもしれません。

```
template<class C, class A>
struct Const {
    Const(C v) : _v(v) {}
    C _v;
};
```

C++のfmapの実装では、関数引数も無視され、基本的にConst引数の値を変えずに再キャストされます。

```
template<class C , class A , class B>.
Const<C, B> fmap(std::function<B(A)> f, Const<C, A> c) {
    return Const<C, B>{c._v};
}
```

その奇妙さにもかかわらず，Constファンクタは多くの構文で重要な役割を果た しています．カテゴリ理論で言えば、先ほどのΔ ŭファンクタの特殊なケース、つまりブラックホールのエンドファンクタのケースです。今後、もっと出てくるでしょう。

7.3 Functor Composition

It's not hard to convince yourself that functors between categories compose, just like functions between sets compose. A composition of two functors, when acting on objects, is just the composition of their respective object mappings; and similarly when acting on morphisms. After jumping through two functors, identity morphisms end up as identity morphisms, and compositions of morphisms finish up as compositions of morphisms. There's really nothing much to it. In particular, it's easy to compose endofunctors. Remember the function `maybeTail`? I'll rewrite it using Haskell's built in implementation of lists:

```
maybeTail :: [a] -> Maybe [a]
maybeTail [] = Nothing
maybeTail (x:xs) = Just xs
```

(The empty list constructor that we used to call `Nil` is replaced with the empty pair of square brackets `[]`. The `Cons` constructor is replaced with the infix operator `:` (colon).) The result of `maybeTail` is of a type that's a composition of two functors, `Maybe` and `[]`, acting on `a`. Each of these functors is equipped with its own version of `fmap`, but what if we want to apply some function `f` to the contents of the composite: a `Maybe` list? We have to break through two layers of functors. We can use `fmap` to break through the outer `Maybe`. But we can't just send `f` inside `Maybe` because `f` doesn't work on lists. We have to send `(fmap f)` to operate on the inner list. For instance, let's see how we can square the elements of a `Maybe` list of integers:

7.3 Functor Composition

集合間の関数が合成されるのと同じように、カテゴリ間のファンクタも合成されること を納得するのは難しいことではありません。2つのファンクタの合成は、オブジェクトに作用する場合は、それぞれのオブジェクトの写像の合成に過ぎず、モルヒズムに作用する場合も同様である。2つのファンクタを飛び越えた後、恒等モルヒズムは恒等モルヒズムのままで終わり、モルヒズムの合成はモルヒズムの合成のまま終わります。これだけなら大したことはない。特に、エンドファンクタの合成は簡単である。関数maybeTailを覚えていますか？これをHaskell の組み込みのリストの実装を使って書き直してみる。

```
maybeTail :: [a] -> Maybe [a]
maybeTail [] = Nothing
maybeTail (x:xs) = Just xs
```

(Nil を呼んでいた空のリストコンストラクタは空の角括弧 [] で置き換えます。Cons コンストラクタは infix 演算子 :（コロン）に置き換えられます). maybeTailの結果は、aに作用する2つのファンクタ、Maybeと[]の合成型です。 これらのファンクタはそれぞれ独自のバージョンのfmapを備えていますが、合成型であるMaybeリストの内容に何らかの関数fを適用したい場合はどうすればよいでしょうか。ファンクタの2つのレイヤーを突破しなければならないのです。fmap を使って外側の Maybe を突破することはできる。 しかし、f はリストでは動作しないので、Maybe の内部に f を送るだけではいけない。内側のリストに対して操作するために、(fmap f) を送らなければならない。例えば、整数の Maybe リストの要素をどのように二乗するか見てみましょう。

```
square x = x * x

mis :: Maybe [Int]
mis = Just [1, 2, 3]

mis2 = fmap (fmap square) mis
```

The compiler, after analyzing the types, will figure out that, for the outer `fmap`, it should use the implementation from the `Maybe` instance, and for the inner one, the list functor implementation. It may not be immediately obvious that the above code may be rewritten as:

```
mis2 = (fmap . fmap) square mis
```

But remember that `fmap` may be considered a function of just one argument:

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

In our case, the second `fmap` in `(fmap . fmap)` takes as its argument:

```
square :: Int -> Int
```

and returns a function of the type:

```
[Int] -> [Int]
```

The first `fmap` then takes that function and returns a function:

```
Maybe [Int] -> Maybe [Int]
```



```
square x = x * x

mis :: Maybe [Int]
mis = Just [1, 2, 3]

mis2 = fmap (fmap square) mis
```

コンパイラは型を分析した後、外側のfmapにはMaybeインスタンスの実装を、内側のものにはリストファンクタの実装を使うべきであると判断します。上記のコードが次のように書き換えられることは、すぐにはわからないかもしれません。

```
mis2 = (fmap . fmap) square mis
```

しかし，fmapは1つの引数だけの関数と考えることができることを思い出してほしい．

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

この場合，(fmap . fmap)の2番目のfmapは引数として受け取ります．

```
square :: Int -> Int
```

という型の関数を返します。

```
[Int] -> [Int]
```

そして、最初のfmapはその関数を引数にとり、関数を返します。

```
Maybe [Int] -> Maybe [Int]
```

Finally, that function is applied to `mis`. So the composition of two functors is a functor whose `fmap` is the composition of the corresponding `fmaps`. Going back to category theory: It's pretty obvious that functor composition is associative (the mapping of objects is associative, and the mapping of morphisms is associative). And there is also a trivial identity functor in every category: it maps every object to itself, and every morphism to itself. So functors have all the same properties as morphisms in some category. But what category would that be? It would have to be a category in which objects are categories and morphisms are functors. It's a category of categories. But a category of *all* categories would have to include itself, and we would get into the same kinds of paradoxes that made the set of all sets impossible. There is, however, a category of all *small* categories called **Cat** (which is big, so it can't be a member of itself). A small category is one in which objects form a set, as opposed to something larger than a set. Mind you, in category theory, even an infinite uncountable set is considered "small." I thought I'd mention these things because I find it pretty amazing that we can recognize the same structures repeating themselves at many levels of abstraction. We'll see later that functors form categories as well.

7.4 Challenges

1. Can we turn the `Maybe` type constructor into a functor by defining:

   ```
   fmap _ _ = Nothing
   ```

   which ignores both of its arguments? (Hint: Check the functor laws.)
2. Prove functor laws for the reader functor. Hint: it's really simple.

最後に，その関数が `mis` に適用されます．というわけで，2つのファンクタの合成は，対応するfmap sの合成をfmapとするファンクタとなる．カテゴリ理論に戻ると，ファンクタの合成が連想的であることは明らかだ（対象の写像は連想的，モーフィズムの写像は連想的）．そして、すべてのカテゴリーには、すべてのオブジェクトをそれ自身に、すべてのモルヒズムをそれ自身に写像する、つまらない恒等ファンクターが存在します。つまり、ファンクタはあるカテゴリにおいてモルヒズムと同じ性質をすべて持っている。しかし、それはどのようなカテゴリーなのだろうか？それは，オブジェクトがカテゴリーで，モルヒズムがファンクターであるようなカテゴリーでなければならないだろう．それはカテゴリのカテゴリである。しかし，すべてのカテゴリのカテゴリはそれ自身を含まなければならず，すべての集合の集合を不可能にしたのと同じ種類のパラドックスに陥ってしまう．しかし、□□というすべての小カテゴリーのカテゴリーがある（これは大きいので、それ自体のメンバーにはなり得ない）。小カテゴリとは、集合より大きなものに対して、オブジェクトが集合を形成するものである。ちなみに、カテゴリー理論では、無限大の数えられない集合も「小さい」とみなされます。私がこのようなことを申し上げたのは、同じ構造が多くの抽象的なレベルで繰り返されることを認識できることが、とても素晴らしいと思ったからです。ファンクタがカテゴリを形成することも、後で見てみましょう。

7.4 Challenges

1. Maybe 型コンストラクタをファンクタにするために、次のような定義をすることができますか？
   ing:

   ```
   fmap _ _ = Nothing
   ```

   によって、その引数を両方とも無視するファンクタに変えることができますか？(ヒント：ファンクタの法則を調べてみてください。) 2. Reader ファンクタに対するファンクタの法則を証明してください。ヒント：とても簡単です。

3. Implement the reader functor in your second favorite language (the first being Haskell, of course).
4. Prove the functor laws for the list functor. Assume that the laws are true for the tail part of the list you're applying it to (in other words, use *induction*).

3. reader functorを、あなたの2番目に好きな言語（1番目はもちろんHaskell）で 実装してください。4. リストファンクタのファンクタ法則を証明しなさい。適用するリストの末尾部分について、法則が真であると仮定する（言い換えれば、 帰納法を使う）。