8

Functoriality

Now that you know what a functor is, and have seen a few examples, let's see how we can build larger functors from smaller ones. In particular it's interesting to see which type constructors (which correspond to mappings between objects in a category) can be extended to functors (which include mappings between morphisms).

8.1 Bifunctors

Since functors are morphisms in **Cat** (the category of categories), a lot of intuitions about morphisms — and functions in particular — apply to functors as well. For instance, just like you can have a function of two arguments, you can have a functor of two arguments, or a *bifunctor*. On objects, a bifunctor maps every pair of objects, one from category **C**, and one from category **D**, to an object in category **E**. Notice that this is

8

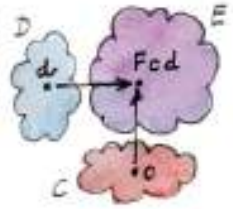
Functoriality

さて，ファンクターとは何か，そしていくつかの例を見てきたところで，より小さなファンクターからより大きなファンクターを作る方法を見てみよう．特に，（カテゴリ内のオブジェクト間のマッピングに対応する）型構成子を，（モルヒズム間のマッピングを含む）ファンクタに拡張できるかを見るのは興味深いことです．

8.1 Bifunctors

ファンクタは□□（カテゴリのカテゴリ）における形態素なので、形態素に関する多くの直感、特に関数はファンクタにも当てはまります。例えば、2つの引数の関数があるように、2つの引数のファンクタ、つまりバイファンクタがあります。オブジェクトの場合、バイファンクタは、カテゴリ□とカテゴリ□のオブジェクトのすべてのペアをカテゴリ□のオブジェクトにマッピングします。これは次のようなものであることに注意してください。

just saying that it's a mapping from a *Cartesian product* of categories $\mathbf{C} \times \mathbf{D}$ to $\mathbf{E}$.



That's pretty straightforward. But functoriality means that a bifunctor has to map morphisms as well. This time, though, it must map a pair of morphisms, one from $\mathbf{C}$ and one from $\mathbf{D}$, to a morphism in $\mathbf{E}$.

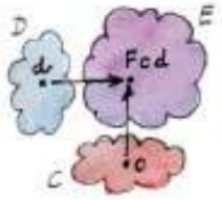
Again, a pair of morphisms is just a single morphism in the product category $\mathbf{C} \times \mathbf{D}$ to $\mathbf{E}$. We define a morphism in a Cartesian product of categories as a pair of morphisms which goes from one pair of objects to another pair of objects. These pairs of morphisms can be composed in the obvious way:

$$(f, g) \circ (f', g') = (f \circ f', g \circ g')$$

The composition is associative and it has an identity — a pair of identity morphisms $(\mathbf{id}, \mathbf{id})$. So a Cartesian product of categories is indeed a category.

An easier way to think about bifunctors would be to consider them functors in each argument separately. So instead of translating functorial laws — associativity and identity preservation — from functors to bifunctors, it would be enough to check them separately for each argument. However, in general, separate functoriality is not enough to prove joint functoriality. Categories in which joint functoriality fails are called *premonoidal*.

は、カテゴリ □ × Lu_1D403 のデカルト積から □ への写像であると言っているに過ぎないことに注意してください。



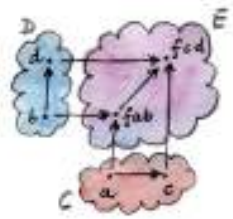
これはとてもわかりやすいですね。しかし、functorialityとは、bifunctorはmorphismsも写像しなければならないことを意味します。しかし、今回は、 Lu_1D402 と Lu_1D404 の形態素のペアを □の形態素に写像しなければならない。ここでも、モルヒズムのペアは、□×□から□への積カテゴリーにおける単一のモルヒズムに過ぎない。カテゴリーのデカルト積におけるモルヒズムを、ある対のオブジェクトから別の対のオブジェクトに向かうモルヒズムのペアと定義する。これらのモルヒズムの対は明白な方法で合成することができる。

$$(f, g) \circ (f', g') = (f \circ f', g \circ g')$$

合成は連想的であり、恒等式 - 恒等式モルヒズムのペア（id, id）を持つ。だから、カテゴリのデカルト積は確かにカテゴリである。二分器を考える簡単な方法は、各引数でファンクタを別々に考えることでしょう。つまり、ファンクタの法則である連想性、同一性保持をファンクタからバイファンクタに翻訳するのではなく、各引数で別々に確認すればよいことになる。しかし、一般に、別々の functoriality だけでは、joint functoriality を証明することはできません。合同 functoriality が失敗するカテゴリを premonoidal と呼ぶ。

Let's define a bifunctor in Haskell. In this case all three categories are the same: the category of Haskell types. A bifunctor is a type constructor that takes two type arguments. Here's the definition of the `Bifunctor` typeclass taken directly from the library `Control.Bifunctor`:

```
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
    bimap g h = first g . second h
    first :: (a -> c) -> f a b -> f c b
    first g = bimap g id
    second :: (b -> d) -> f a b -> f a d
    second = bimap id
```

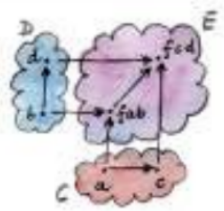


bimap

The type variable `f` represents the bifunctor. You can see that in all type signatures it's always applied to two type arguments. The first type signature defines `bimap`: a mapping of two functions at once. The result is a lifted function, `(f a b -> f c d)`, operating on types generated by the bifunctor's type constructor. There is a default implementation of `bimap` in terms of `first` and `second`. (As mentioned before, this doesn't always work, because the two maps may not commute, that is `first g . second h` may not be the same as `second h . first g`.)

Haskellでbifunctorを定義してみよう。この場合、3つのカテゴリはすべて同じで、Haskellの型のカテゴリである。バイファンクションは2つの型引数をとる型コンストラクタです。以下は、ライブラリ Control から直接引用した Bifunctor 型クラスの定義である。Bifunctor：

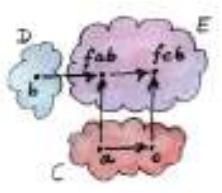
```
class Bifunctor f where
    bimap :: (a -> c) -> (b -> d) -> f a b -> f c d
    bimap g h = first g . second h
    first :: (a -> c) -> f a b -> f c b
    first g = bimap g id
    second :: (b -> d) -> f a b -> f a d
    second = bimap id
```



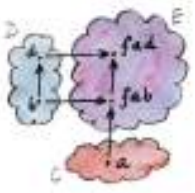
bimap

型変数 f はバイファンクタを表している。すべての型シグネチャで、常に2つの型引数に適用されているのがわかる。最初の型署名はbimap : 2つの関数を一度にマッピングすることを定義している。その結果は，バイファンクタの型コンストラクタで生成された型に対して動作する，（f a b -> f c d）という浮動小数点型関数になります．bimapのデフォルトの実装は，firstとsecondの2つです．（前に述べたように，2つの写像は通約しないことがあるので，これは必ずしも うまくいきません．つまり，first g . second hはsecond h . first gと同じではないかもしれません．）

The two other type signatures, `first` and `second`, are the two `fmaps` witnessing the functoriality of `f` in the first and the second argument, respectively.



first



second

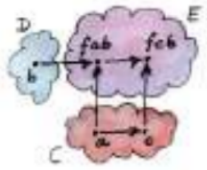
The typeclass definition provides default implementations for both of them in terms of `bimap`.

When declaring an instance of `Bifunctor`, you have a choice of either implementing `bimap` and accepting the defaults for `first` and `second`, or implementing both `first` and `second` and accepting the default for `bimap` (of course, you may implement all three of them, but then it's up to you to make sure they are related to each other in this manner).

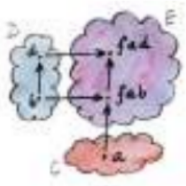
8.2 Product and Coproduct Bifunctors

An important example of a bifunctor is the categorical product — a product of two objects that is defined by a universal construction. If the product exists for any pair of objects, the mapping from those objects to the product is bifunctorial. This is true in general, and in Haskell in particular. Here's the `Bifunctor` instance for a pair constructor — the simplest product type:

他の二つの型署名、`first` と `second` は、それぞれ第一引数と第二引数における `f` の `functoriality` を目撃する二つの `fmap` です。



first



second

型クラスの定義では，`bimap` の観点から，これら2つのデフォルトの実装を提供する．`Bifunctor` のインスタンスを宣言するとき，`bimap` を実装して，第一引数と 第二引数のデフォルトを受け入れるか，第一引数と第二引数の両方を実装して， `bimap` のデフォルトを受け入れるかを選択できる（もちろん，この3つをすべて実装してもよいが，その場合は，このように互いに関連付けるかはあなた次第である）．

8.2 プロダクトとコプロダクト・ビファンクター

二分器の重要な例として、カテゴリカル積-普遍的な構成によって定義される2つの対象 の積-があります。この積が任意のオブジェクトのペアに存在する場合、それらのオブジェクトから積への写像は二分法である。これは一般的に言えることで、特にHaskellではそうです。以下は、ペア構成子（最も単純な積の型）のための二分法インスタンスです。

```
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

There isn't much choice: `bimap` simply applies the first function to the first component, and the second function to the second component of a pair. The code pretty much writes itself, given the types:

```
bimap :: (a -> c) -> (b -> d) -> (a, b) -> (c, d)
```

The action of the bifunctor here is to make pairs of types, for instance:

```
(,) a b = (a, b)
```

By duality, a coproduct, if it's defined for every pair of objects in a category, is also a bifunctor. In Haskell, this is exemplified by the `Either` type constructor being an instance of `Bifunctor`:

```
instance Bifunctor Either where
    bimap f _ (Left x) = Left (f x)
    bimap _ g (Right y) = Right (g y)
```

This code also writes itself.

Now, remember when we talked about monoidal categories? A monoidal category defines a binary operator acting on objects, together with a unit object. I mentioned that **Set** is a monoidal category with respect to Cartesian product, with the singleton set as a unit. And it's also a monoidal category with respect to disjoint union, with the empty set as a unit. What I haven't mentioned is that one of the requirements for a monoidal category is that the binary operator be a bifunctor. This is a very important requirement — we want the monoidal product to be


```
instance Bifunctor (,) where
    bimap f g (x, y) = (f x, g y)
```

bimapは単純に最初の関数を第一成分に、第二関数を第二成分に適用します。型が与えられれば、コードはほとんどそれ自体で書けます。

```
bimap :: (a -> c) -> (b -> d) -> (a, b) -> (c, d)
```

ここでのバイファンクタの動作は、例えば、型のペアを作ることです。

```
(,) a b = (a, b)
```

双対性によって、コプロダクトは、それがカテゴリのオブジェクトのすべてのペアのために定義されている場合、また、二分木である。Haskellでは、Either型コンストラクタがBifunctorのインスタンスであることが例証されています。

```
instance Bifunctor Either where
    bimap f _ (Left x) = Left (f x)
    バイマップ _ g ( Right y) = Right (g y)
```

このコードも自分で書きます。さて、モノイダルカテゴリの話をしたのを覚えていますか？モノイダルカテゴリは、オブジェクトに作用する二項演算子と、単位オブジェクトを定義するものです。□□は単項集合を単位とするデカルト積に関するモノイダルカテゴリであることを述べました。また、空集合を単位とする不連続和に関する単項類でもある。まだ触れていませんが、単項式カテゴリの要件の1つは、二項演算子が二分木であることです。これは非常に重要な要件で、モノイダル積は以下のようにしたい。

compatible with the structure of the category, which is defined by morphisms. We are now one step closer to the full definition of a monoidal category (we still need to learn about naturality, before we can get there).

8.3 Functorial Algebraic Data Types

We've seen several examples of parameterized data types that turned out to be functors — we were able to define `fmap` for them. Complex data types are constructed from simpler data types. In particular, algebraic data types (ADTs) are created using sums and products. We have just seen that sums and products are functorial. We also know that functors compose. So if we can show that the basic building blocks of ADTs are functorial, we'll know that parameterized ADTs are functorial too.

So what are the building blocks of parameterized algebraic data types? First, there are the items that have no dependency on the type parameter of the functor, like `Nothing` in `Maybe`, or `Nil` in `List`. They are equivalent to the `Const` functor. Remember, the `Const` functor ignores its type parameter (really, the *second* type parameter, which is the one of interest to us, the first one being kept constant).

Then there are the elements that simply encapsulate the type parameter itself, like `Just` in `Maybe`. They are equivalent to the identity functor. I mentioned the identity functor previously, as the identity morphism in *Cat*, but didn't give its definition in Haskell. Here it is:

```
data Identity a = Identity a
```


これは非常に重要な要件で、モノイダル積がモルヒズムによって定義されるカテゴリの構造と互換性があることを望んでいるのです。これで、モノイダルカテゴリーの完全な定義に一歩近づいた（そこにたどり着く前に、まだ、自然性について学ぶ必要がある）。

8.3 関数型代数的データ型

これまで，パラメータ化されたデータ型がファンクタであることが判明した例をいくつか見てきました．複雑なデータ型は，より単純なデータ型から構成されます．特に、代数的データ型（adts）は、和と積を使って作られる。和と積がファンクタであることは先程見たとおりである。また、ファンクタは合成することも知っている。つまり、adtsの基本構成要素がfunctionalであることを示せば、パラメータ化されたadtsもfunctionalであることがわかるはずだ。では、パラメタ化された代数的データ型の構成要素は何だろうか？まず，MaybeのNothingやListのNilのように，ファンクタの型パラメータに依存しない項目があります．これらはConstファンクタと等価です．Constファンクタはその型パラメータを無視します（本当は2番目の型パラメータが重要で，1番目の型パラメータは一定に保たれます）．それから，MaybeのJustのように型パラメータそのものを単純にカプセル化する要素もあります． これらはidentityファンクタと同等です．以前、Catの identity morphism として identity functor に触れたが、Haskell での定義はしていなかった。ここでは、それを紹介する。

```
data Identity a = Identity a
```

```
instance Functor Identity where
    fmap f (Identity x) = Identity (f x)
```

You can think of `Identity` as the simplest possible container that always stores just one (immutable) value of type `a`.

Everything else in algebraic data structures is constructed from these two primitives using products and sums.

With this new knowledge, let's have a fresh look at the `Maybe` type constructor:

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

It's a sum of two types, and we now know that the sum is functorial. The first part, `Nothing` can be represented as a `Const ()` acting on `a` (the first type parameter of `Const` is set to unit — later we'll see more interesting uses of `Const`). The second part is just a different name for the identity functor. We could have defined `Maybe`, up to isomorphism, as:

```
type Maybe a = Either (Const () a) (Identity a)
```

So `Maybe` is the composition of the bifunctor `Either` with two functors, `Const ()` and `Identity`. (`Const` is really a bifunctor, but here we always use it partially applied.)

We've already seen that a composition of functors is a functor — we can easily convince ourselves that the same is true of bifunctors. All we need is to figure out how a composition of a bifunctor with two functors works on morphisms. Given two morphisms, we simply lift one with one functor and the other with the other functor. We then lift the resulting pair of lifted morphisms with the bifunctor.


```
instance Functor Identity where
    fmap f ( アイデンティティ x) = アイデンティティ (f x)
```

Identityは、a型の（不変の）値を常に1つだけ格納する最も単純な容器と考えることができる。 代数データ構造における他のすべては、積と和を使用してこの2つのプリミティブから構築される。この新しい知識をもとに、Maybe型コンストラクタを改めて見てみましょう。

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

これは2つの型の和であり、和はファンクショナルであることがわかりました。最初の部分、Nothing は a に作用する Const () として表現できます（Const の最初の型パラメータは unit に設定されます - 後で Const のより興味深い使い方を見ることになります）。2番目の部分は、単にIDファンクタの別名である。Maybeは同型までなら、次のように定義できる。

```
type Maybe a = Either ( Const () a) ( アイデンティティ a)
```

つまり，Maybeは2分器Eitherと2つのファンクタConst ()およびIdentityの合成です． Constは本当は2分器ですが，ここでは常に部分適用して使います）ファンクタの合成がファンクタであることはすでに見てきましたので，2分器についても同じことが言えると簡単に納得できるでしょう．あとは、2つのファンクタの合成がモルヒズムに対してどのように働くかを理解すればよい。2つのモルヒズムが与えられたら、一方を一方のファンクタで持ち上げ、他方をもう一方のファンクタで持ち上げるだけである。そして、その結果得られたモルヒズムのペアを二分法で持ち上げます。

We can express this composition in Haskell. Let's define a data type that is parameterized by a bifunctor `bf` (it's a type variable that is a type constructor that takes two types as arguments), two functors `fu` and `gu` (type constructors that take one type variable each), and two regular types `a` and `b`. We apply `fu` to `a` and `gu` to `b`, and then apply `bf` to the resulting two types:

```
newtype BiComp bf fu gu a b = BiComp (bf (fu a) (gu b))
```

That's the composition on objects, or types. Notice how in Haskell we apply type constructors to types, just like we apply functions to arguments. The syntax is the same.

If you're getting a little lost, try applying `BiComp` to `Either`, `Const ()`, `Identity`, `a`, and `b`, in this order. You will recover our bare-bone version of `Maybe b` (`a` is ignored).

The new data type `BiComp` is a bifunctor in `a` and `b`, but only if `bf` is itself a `Bifunctor` and `fu` and `gu` are `Functors`. The compiler must know that there will be a definition of `bimap` available for `bf`, and definitions of `fmap` for `fu` and `gu`. In Haskell, this is expressed as a precondition in the instance declaration: a set of class constraints followed by a double arrow:

```
instance (Bifunctor bf, Functor fu, Functor gu) =>
  Bifunctor (BiComp bf fu gu) where
    bimap f1 f2 (BiComp x) = BiComp ((bimap (fmap f1) (fmap
    ↪  f2)) x)
```

The implementation of `bimap` for `BiComp` is given in terms of `bimap` for `bf` and the two `fmaps` for `fu` and `gu`. The compiler automatically infers all the types and picks the correct overloaded functions whenever `BiComp` is used.

この合成をHaskellで表現することができる。二分器bf（二つの型を引数に取る型構成子で型変数です）、二つのファンクタfuとgu（それぞれ一つの型変数を取る型構成子）、二つの正規型a、bでパラメータ化されたデータ型を定義してみましょう．a に fu を，b に gu を適用し，できた2つの型に bf を適用する．

```
newtype BiComp bf fu gu a b = BiComp (bf (fu a) (gu b))
```

これがオブジェクト、つまり型に対する合成です。Haskellでは、関数を引数に適用するのと同じように、型構成子を型に適用していることに注意してください。構文は同じです。もし少し迷ったら、BiCompをEither、Const ()、Identity、a、bの順に適用してみてください。そうすると、素っ気ないMaybe b（aは無視される）が復元される。新しいデータ型BiCompは、bfがそれ自体Bifunctorであり、fuとguがFunctor sである場合にのみ、aとbでbifunctorとなる。コンパイラは、bfに対してbimapの定義があり、fuとguに対してfmapの定義があることを知っている必要がある。Haskellでは、これはインスタンス宣言の前提条件として表現される。つまり、クラス制約のセットと二重矢印が続く。

```
インスタンス ( Bifunctor bf, Functor fu, Functor gu) =>
  ビファンクタ ( BiComp bf fu gu) ここで
    bimap f1 f2 ( BiComp x) = BiComp ((bimap (fmap f1) (fmap
    ↪  f2)) x)
```

BiCompのbimapの実装は、bfのbimapと、fuとguの2つのfmapで示される。コンパイラは、BiCompが使われるたびに、自動的にすべての型を推論し、正しいオーバーロードされた関数を選択する。

The x in the definition of bimap has the type:

```
bf (fu a) (gu b)
```

which is quite a mouthful. The outer bimap breaks through the outer bf layer, and the two fmaps dig under fu and gu, respectively. If the types of f1 and f2 are:

```
f1 :: a -> a'
f2 :: b -> b'
```

then the final result is of the type bf (fu a') (gu b'):

```
bimap :: (fu a -> fu a') -> (gu b -> gu b')
  -> bf (fu a) (gu b) -> bf (fu a') (gu b')
```

If you like jigsaw puzzles, these kinds of type manipulations can provide hours of entertainment.

So it turns out that we didn't have to prove that Maybe was a functor — this fact followed from the way it was constructed as a sum of two functorial primitives.

A perceptive reader might ask the question: If the derivation of the Functor instance for algebraic data types is so mechanical, can't it be automated and performed by the compiler? Indeed, it can, and it is. You need to enable a particular Haskell extension by including this line at the top of your source file:

```
{-# LANGUAGE DeriveFunctor #-}
```

and then add deriving Functor to your data structure:

bimapの定義にあるxは型を持っている.

```
bf (fu a) (gu b)
```

となっており、かなり口惜しい。外側のbimapは外側のbf層を突破し、2つのfmap sはそれぞれfuとgu 、の下を掘る。もしf1とf2の型が

```
f1 :: a -> a'
f2 :: b -> b'
```

であれば、最終的な結果はbf（fu a'）（gu b'）という型になる。

```
bimap :: (fu a -> fu a') -> (gu b -> gu b')
  -> bf (fu a) (gu b) -> bf (fu a') (gu b')
```

ジグソーパズルが好きな人なら、このような型操作は何時間でも楽しめるでしょう。つまり、Maybeがファンクタであることを証明する必要はなく、2つのファンクタ・プリミティブの和として構成されていることから、この事実が導かれるのである。察しの良い読者はこう質問するかもしれない。もし、代数的なデータ型に対するファンクタのインスタンスの導出がそんなに機械的なら、コンパイラが自動化して実行することはできないのか？という質問を受けるかもしれない。実際、それは可能である。特定のHaskell拡張を有効にするには、ソースファイルの先頭に次の行を追加する必要があります。

```
{-# LANGUAGE DeriveFunctor #-}
```

そして、Functorの導出をデータ構造に追加してください。

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

and the corresponding `fmap` will be implemented for you.

The regularity of algebraic data structures makes it possible to derive instances not only of `Functor` but of several other type classes, including the `Eq` type class I mentioned before. There is also the option of teaching the compiler to derive instances of your own typeclasses, but that's a bit more advanced. The idea though is the same: You provide the behavior for the basic building blocks and sums and products, and let the compiler figure out the rest.

8.4 Functors in C++

If you are a C++ programmer, you obviously are on your own as far as implementing functors goes. However, you should be able to recognize some types of algebraic data structures in C++. If such a data structure is made into a generic template, you should be able to quickly implement `fmap` for it.

Let's have a look at a tree data structure, which we would define in Haskell as a recursive sum type:

```
data Tree a = Leaf a | Node (Tree a) (Tree a)
    deriving Functor
```

As I mentioned before, one way of implementing sum types in C++ is through class hierarchies. It would be natural, in an object-oriented language, to implement `fmap` as a virtual function of the base class `Functor` and then override it in all subclasses. Unfortunately this is impossible because `fmap` is a template, parameterized not only by the type of the object it's acting upon (the `this` pointer) but also by the return type of


```
data Maybe a = Nothing | Just a deriving Functor（派生ファンクタ
```

とすれば、対応するfmapが実装されます。代数的データ構造の規則性を利用して、Functorだけでなく、先に述べたEq型クラスなど、いくつかの型クラスのインスタンスを派生させることが可能です。また，コンパイラに自分自身の型クラスのインスタンスを導出させるという方法もあるが，これは少し高度である。基本的な構成要素や和や積の振る舞いを提供し、残りはコンパイラに任せるという考え方は同じです。

8.4 Functors in C++

もしあなたがC++プログラマーなら、ファンクタの実装に関しては明らかに自己責任で行うことになります。しかし、C++の代数的データ構造のいくつかのタイプを認識することはできるはずです。そのようなデータ構造が汎用テンプレート化されていれば、それに対する fmap の実装はすぐにできるはずだ。Haskellでは再帰的和型として定義される木構造を見てみよう。

```
data Tree a = Leaf a | Node ( Tree a) ( Tree a)
    deriving Functor
```

前に述べたように、C++で和型を実装する方法の1つは、クラス階層を利用することです。オブジェクト指向の言語では、fmapを基底クラスFunctorの仮想関数とし て実装し、すべてのサブクラスでそれをオーバーライドするのが自然でしょう。なぜなら，fmapはテンプレートであり，作用するオブジェクトの型(このポインタ)だけでなく，その戻り値の型によってもパラメータ化されるからです.

the function that's been applied to it. Virtual functions cannot be templatized in C++. We'll implement `fmap` as a generic free function, and we'll replace pattern matching with `dynamic_cast`.

The base class must define at least one virtual function in order to support dynamic casting, so we'll make the destructor virtual (which is a good idea in any case):

```
template<class T>
struct Tree {
    virtual ~Tree() {};
};
```

The `Leaf` is just an `Identity` functor in disguise:

```
template<class T>
struct Leaf : public Tree<T> {
    T _label;
    Leaf(T l) : _label(l) {}
};
```

The `Node` is a product type:

```
template<class T>
struct Node : public Tree<T> {
    Tree<T> * _left;
    Tree<T> * _right;
    Node(Tree<T> * l, Tree<T> * r) : _left(l), _right(r) {}
};
```

When implementing `fmap` we take advantage of dynamic dispatching on the type of the `Tree`. The `Leaf` case applies the `Identity` version of `fmap`, and the `Node` case is treated like a bifunctor composed with two copies of the `Tree` functor. As a C++ programmer, you're probably

の戻り値の型によってパラメータ化されます。C++では仮想関数をテンプレート化することはできません。fmapを汎用的なfree関数として実装し、パターンマッチングをdynamic_castに置き換えることにします．ダイナミックキャストをサポートするためには，ベースクラスは少なくとも1つの仮想関数を定義しなければならないので，デストラクタを仮想化します（これはいずれにせよ良いアイデアです）．

```
template<class T>
struct Tree {
    virtual ~Tree() {};
};
```

LeafはIdentityファンクタを偽装しているだけです．

```
template<class T>
struct Leaf : public Tree<T> {
    T _label;
    Leaf(T l) : _label(l) {}
};
```

The `Node` is a product type:

```
template<class T>
struct Node : public Tree<T> {
    Tree<T> * _left;
    Tree<T> * _right;
    Node(Tree<T> * l, Tree<T> * r) : _left(l), _right(r) {}
};
```

fmapを実装する際に、Treeの型に対する動的ディスパッチを利用する。Leafの場合はfmapのIdentityバージョンを適用し，Nodeの場合はTreeファンクタの2つのコピーで構成されるバイファンクタのように扱われます．C++プログラマであれば，おそらく次のようなことを考えるでしょう．

not used to analyzing code in these terms, but it's a good exercise in categorical thinking.

```
template<class A, class B>
Tree<B> * fmap(std::function<B(A)> f, Tree<A> * t) {
    Leaf<A> * pl = dynamic_cast <Leaf<A>*>(t);
    if (pl)
        return new Leaf<B>(f (pl->_label));
    Node<A> * pn = dynamic_cast<Node<A>*>(t);
    if (pn)
        return new Node<B>( fmap<A>(f, pn->_left)
                          , fmap<A>(f, pn->_right));
    return nullptr;
}
```

For simplicity, I decided to ignore memory and resource management issues, but in production code you would probably use smart pointers (unique or shared, depending on your policy).

Compare it with the Haskell implementation of `fmap`:

```
instance Functor Tree where
    fmap f (Leaf a) = Leaf (f a)
    fmap f (Node t t') = Node (fmap f t) (fmap f t')
```

This implementation can also be automatically derived by the compiler.

8.5 The Writer Functor

I promised that I would come back to the Kleisli category I described earlier. Morphisms in that category were represented as "embellished" functions returning the `Writer` data structure.

C++プログラマとしては、このような用語でコードを分析するのは慣れていないかもしれませんが、これはカテゴリ的な考え方を身につけるための良い練習になります。

```
template<class A, class B>
Tree<B> * fmap(std::function<B(A)> f, Tree<A> * t) {
    Leaf<A> * pl = dynamic_cast <Leaf<A>*>(t);
    if (pl)
        return new Leaf<B>(f (pl->_label));
    Node<A> * pn = dynamic_cast<Node<A>*>(t);
    if (pn)
        return new Node<B>( fmap<A>(f, pn->_left)
                          , fmap<A>(f, pn->_right));
    return nullptr;
}
```

簡単のために、メモリとリソース管理の問題は無視することにしました。しかし、実運用コードでは、おそらくスマートポインタ（ユニークまたは共有、あなたのポリシーに依存）を使用するでしょう。fmapのHaskellの実装と比較してみてください：

```
instance Functor Tree where
    fmap f (Leaf a) = Leaf (f a)
    fmap f ( Node t t') = Node (fmap f t) (fmap f t')
```

この実装はコンパイラによって自動的に導出されることもあります.

8.5 The Writer Functor

前に説明したクライスリーカテゴリに戻ると約束した。そのカテゴリのモルヒズムは、Writerデータ構造を返す「装飾された」関数として表現されました。

```
type Writer a = (a, String)
```

I said that the embellishment was somehow related to endofunctors. And, indeed, the `Writer` type constructor is functorial in a. We don't even have to implement `fmap` for it, because it's just a simple product type.

But what's the relation between a Kleisli category and a functor — in general? A Kleisli category, being a category, defines composition and identity. Let me remind you that the composition is given by the fish operator:

```
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
m1 >=> m2 = \x ->
    let (y, s1) = m1 x
        (z, s2) = m2 y
    in (z, s1 ++ s2)
```

and the identity morphism by a function called `return`:

```
return :: a -> Writer a
return x = (x, "")
```

It turns out that, if you look at the types of these two functions long enough (and I mean, *long* enough), you can find a way to combine them to produce a function with the right type signature to serve as `fmap`. Like this:

```
fmap f = id >=> (\x -> return (f x))
```

Here, the fish operator combines two functions: one of them is the familiar `id`, and the other is a lambda that applies `return` to the result


```
type Writer a = (a, String)
```

私は、この装飾はエンドファンクタと何らかの関係があると言いました。そして、実際、Writerの型コンストラクタは、関数型である。 単なる積型なので、fmapを実装する必要すらない。しかし、一般に、クライスリー・カテゴリーとファンクタの関係はどうなっているのだろう。クライスリー・カテゴリーは、カテゴリーである以上、構成と恒等式を定義する。合成は魚演算子によって与えられることを思い出してみよう。

```
(>=>) :: (a -> Writer b) -> (b -> Writer c) -> (a -> Writer c)
m1 >=> m2 = \x ->
    let (y, s1) = m1 x
        (z, s2) = m2 y
    in (z, s1 ++ s2)
```

という関数で与えられ、return という関数で同一性モルヒズムが与えられます。

```
return :: a -> Writer a
return x = (x, "")
```

この2つの関数の型を十分長く見ていれば（つまり、十分長く見ていれば）、それらを組み合わせて、fmapとして機能する正しい型シグネチャを持つ関数を生成する方法を見つけることができることがわかった。こんな風に。

```
fmap f = id >=> (\x -> return (f x))
```

ここで、fish演算子は2つの関数を組み合わせています。1つはおなじみのidで、もう1つはreturnを結果に適用するラムダです。

of acting with `f` on the lambda's argument. The hardest part to wrap your brain around is probably the use of `id`. Isn't the argument to the fish operator supposed to be a function that takes a "normal" type and returns an embellished type? Well, not really. Nobody says that `a` in `a -> Writer b` must be a "normal" type. It's a type variable, so it can be anything, in particular it can be an embellished type, like `Writer b`.

So `id` will take `Writer a` and turn it into `Writer a`. The fish operator will fish out the value of `a` and pass it as `x` to the lambda. There, `f` will turn it into a `b` and `return` will embellish it, making it `Writer b`. Putting it all together, we end up with a function that takes `Writer a` and returns `Writer b`, exactly what `fmap` is supposed to produce.

Notice that this argument is very general: you can replace `Writer` with any type constructor. As long as it supports a fish operator and `return`, you can define `fmap` as well. So the embellishment in the Kleisli category is always a functor. (Not every functor, though, gives rise to a Kleisli category.)

You might wonder if the `fmap` we have just defined is the same `fmap` the compiler would have derived for us with `deriving Functor`. Interestingly enough, it is. This is due to the way Haskell implements polymorphic functions. It's called *parametric polymorphism*, and it's a source of so called *theorems for free*. One of those theorems says that, if there is an implementation of `fmap` for a given type constructor, one that preserves identity, then it must be unique.

8.6 Covariant and Contravariant Functors

Now that we've reviewed the writer functor, let's go back to the reader functor. It was based on the partially applied function-arrow type constructor:

ラムダの引数にfを作用させた結果にreturnを適用したものです。一番頭を悩ませるのは、id の使い方でしょう。fish演算子の引数は、「通常の」型を受け取り、装飾された型を返す関数であるべきではないでしょうか？まあ、そうでもないんですけどね。a -> Writer b のaが「普通の」型でなければならないとは誰も言っていない。これは型変数ですから、どんなものでも構いませんし、特にWriter bのような装飾された型でも構いません。そこで、idはWriter aを受け取り、Writer aに変えます。 fish演算子はaの値を取り出し、それをxとしてラムダに渡します。そこで、fはそれをa bに変え、returnはそれを装飾してWriter bにする。このように、fmapはライターaを受け取りライターbを返すという、まさにfmapが生成するはずの関数を生成しているのだ。この引数は非常に一般的なもので、Writerを任意の型構成子で置き換えることができることに注意してください。魚演算子とreturnをサポートする限り、fmapも定義できる。つまり、クライスリーカテゴリーの装飾は常にファンクタである。(今定義したfmapが，コンパイラがderiving Functor .Fmapで定義したものと同じかどうか，疑問に思うかもしれません．興味深いことに、そうなのだ。これは，Haskellが多相関数を実装する方法によるものです．これはパラメトリックポリモーフィズムと呼ばれるもので、いわゆる定理を無料で提供してくれるものだ。その定理のひとつに、与えられた型構成子に対して、同一性を保持するfmapの実装があれば、それは一意でなければならないというものがあります。

8.6 共変量ファンクターと共変量ファンクター

ライターファンクタについて復習したところで、リーダーファンクタに戻りましょう。これは部分的に適用される関数-矢印型コンストラクタを基にしていました。

```
(->) r
```

We can rewrite it as a type synonym:

```
type Reader r a = r -> a
```

for which the `Functor` instance, as we've seen before, reads:

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

But just like the pair type constructor, or the `Either` type constructor, the function type constructor takes two type arguments. The pair and `Either` were functorial in both arguments — they were bifunctors. Is the function constructor a bifunctor too?

Let's try to make it functorial in the first argument. We'll start with a type synonym — it's just like the `Reader` but with the arguments flipped:

```
type Op r a = a -> r
```

This time we fix the return type, `r`, and vary the argument type, `a`. Let's see if we can somehow match the types in order to implement `fmap`, which would have the following type signature:

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

With just two functions taking `a` and returning, respectively, `b` and `r`, there is simply no way to build a function taking `b` and returning `r`! It would be different if we could somehow invert the first function, so that it took `b` and returned `a` instead. We can't invert an arbitrary function, but we can go to the opposite category.


```
(->) r
```

これを型シノニムとして書き直すことができます。

```
type Reader r a = r -> a
```

というファンクタのインスタンスを読み取ります。

```
instance Functor ( Reader r) ここで
    fmap f g = f . g
```

しかし、pair型コンストラクタやEither型コンストラクタと同じように、関数型コンストラクタは2つの型引数を取ります。pair や Either は両方の引数で functorial、つまり bifunctor でした。関数型コンストラクタもバイファンクタなのでしょうか？最初の引数でファンクショナルになるようにしてみましょう。まずは型同義語からです。これはReaderと同じですが、引数がひっくり返っています。

```
type Op r a = a -> r
```

今回は戻り値の型、r を固定し、引数の型、a を変えます。 fmap を実装するために、どうにかして型を一致させられるかどうか見てみましょう。

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

aを受け取ってbとrを返す2つの関数だけでは、bを受け取ってrを返す関数を作るのは無理です ！ もし、最初の関数を反転させて、bを受け取り、代わりにaを返すことができれば、話は別です。任意の関数を反転させることはできないが、逆のカテゴリーに行くことはできる。

A short recap: For every category $\mathbf{C}$ there is a dual category $\mathbf{C}^{op}$. It's a category with the same objects as $\mathbf{C}$, but with all the arrows reversed.

Consider a functor that goes between $\mathbf{C}^{op}$ and some other category $\mathbf{D}$:

$$F :: \mathbf{C}^{op} \to \mathbf{D}$$

Such a functor maps a morphism $f^{op} :: a \to b$ in $\mathbf{C}^{op}$ to the morphism $Ff^{op} :: Fa \to Fb$ in $\mathbf{D}$. But the morphism f^{op} secretly corresponds to some morphism $f :: b \to a$ in the original category $\mathbf{C}$. Notice the inversion.

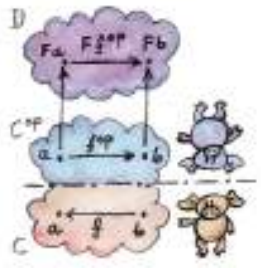
Now, F is a regular functor, but there is another mapping we can define based on F, which is not a functor — let's call it G. It's a mapping from $\mathbf{C}$ to $\mathbf{D}$. It maps objects the same way F does, but when it comes to mapping morphisms, it reverses them. It takes a morphism $f :: b \to a$ in $\mathbf{C}$, maps it first to the opposite morphism $f^{op} :: a \to b$ and then uses the functor F on it, to get $Ff^{op} :: F\,a \to F\,b$.

Considering that Fa is the same as Ga and Fb is the same as Gb, the whole trip can be described as: $Gf :: (b \to a) \to (Ga \to Gb)$ It's a "functor with a twist." A mapping of categories that inverts the direction of morphisms in this manner is called a *contravariant functor*. Notice that a contravariant functor is just a regular functor from the opposite category. The regular functors, by the way — the kind we've been studying thus far — are called *covariant* functors.

簡単に復習しておく。すべてのカテゴリ には、双対カテゴリ が存在します。これは、と同じオブジェクトを持つカテゴリですが、すべての矢印が逆になっています。と他のカテゴリの間を行き来するファンクタを考えてみましょう。

$$F :: \mathbf{C}^{op} \to \mathbf{D}$$

このようなファンクタは、　のモルヒズム　を　のモルヒズム→　に写すものです。しかし、モルヒズム は、元のカテゴリー　　における何らかのモルヒズム →　 Ll_lD44E に密かに対応している。反転に注目。さて、　Lu_lD439 は正規のファンクタですが、Lu_lD439 に基づいて定義できる、ファンクタではない別の写像があります--これを　Lu_lD43A と呼ぶことにしましょう。これは から への写像である。これは ŭ と同じようにオブジェクトを写像するが、形態素を写像するときには、それを逆にする。　の形態素 をとって、まず反対の形態素 に写像し、それにファンクタ を用いて、　→を得ます。は と同じ、は と同じと考えると、旅の全貌は次のように表現できる。）→（） ”ひねりのあるファンクタ ”である。このようにモルヒズムの向きを反転させるカテゴリの写像をcontravariant functorと呼ぶ。逆変換ファンクタは、反対側のカテゴリーからの正規ファンクタに過ぎないことに注意しよう。ちなみに、これまで勉強してきたような普通のファンクターは、共変ファンクターと呼ばれます。



Here's the typeclass defining a contravariant functor (really, a contravariant *endo*functor) in Haskell:

```
class Contravariant f where
    contramap :: (b -> a) -> (f a -> f b)
```

Our type constructor `Op` is an instance of it:

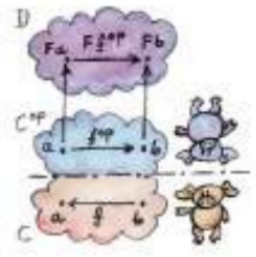
```
instance Contravariant (Op r) where
    -- (b -> a) -> Op r a -> Op r b
    contramap f g = g . f
```

Notice that the function `f` inserts itself *before* (that is, to the right of) the contents of `Op` — the function `g`.

The definition of `contramap` for `Op` may be made even terser, if you notice that it's just the function composition operator with the arguments flipped. There is a special function for flipping arguments, called `flip`:

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f y x = f x y
```

With it, we get:



Haskellでcontravariant functor（本当はcontravariant endofunctor）を定義する型クラスは次のとおり。

```
class Contravariant f where
    contramap :: (b -> a) -> (f a -> f b)
```

私たちの型構成子 `Op` はそのインスタンスである.

```
インスタンス Contravariant ( Op r) ここで
    -- (b -> a) -> Op r a -> Op r b
    contramap f g = g . f
```

関数fは、Opの内容である関数gの前に（つまり、右側に）挿入されることに注意してください。Opのcontramapの定義は、関数合成演算子の引数を反転させただけであることに気づけば、もっと簡単です。引数を反転させるための特別な関数として `flip` があります。

```
flip :: (a -> b -> c) -> (b -> a -> c)
flip f y x = f x y
```

With it, we get:

```
contramap = flip (.)
```

8.7 Profunctors

We've seen that the function-arrow operator is contravariant in its first argument and covariant in the second. Is there a name for such a beast? It turns out that, if the target category is **Set**, such a beast is called a *profunctor*. Because a contravariant functor is equivalent to a covariant functor from the opposite category, a profunctor is defined as:

$$\mathbf{C}^{op} \times \mathbf{D} \to \mathbf{Set}$$

Since, to first approximation, Haskell types are sets, we apply the name `Profunctor` to a type constructor `p` of two arguments, which is contrafunctorial in the first argument and functorial in the second. Here's the appropriate typeclass taken from the `Data.Profunctor` library:

```
class Profunctor p where
    dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
    dimap f g = lmap f . rmap g
    lmap :: (a -> b) -> p b c -> p a c
    lmap f = dimap f id
    rmap :: (b -> c) -> p a b -> p a c
    rmap = dimap id
```

All three functions come with default implementations. Just like with `Bifunctor`, when declaring an instance of `Profunctor`, you have a choice of either implementing `dimap` and accepting the defaults for `lmap` and `rmap`, or implementing both `lmap` and `rmap` and accepting the default for `dimap`.


```
contramap = flip (.)
```

8.7 Profunctors

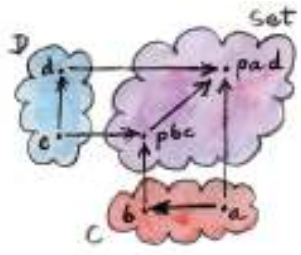
関数アロー演算子は第1引数でcontravariant、第2引数でcovariantであることを見てきました。このような獣の名前はあるのでしょうか？対象カテゴリが□　なら、そのような獣はプロファンクタと呼ばれることが判明している。逆変数ファンクタは反対のカテゴリの共変数ファンクタと等価なので、プロフ ァンクトルは次のように定義される。

$$\mathbf{C}^{op} \times \mathbf{D} \to \mathbf{Set}$$

Haskell の型は近似的に集合なので，2つの引数の型構成子 p に対して Profunctor という名前を適用し，最初の引数ではcontrafunctorial，2番目の引数では functorial になる．以下は，Data.Profunctor ライブラリから取り出した適切な型クラスである．

```
class Profunctor p where
    dimap :: (a -> b) -> (c -> d) -> p b c -> p a d
    dimap f g = lmap f . rmap g
    lmap :: (a -> b) -> p b c -> p a c
    lmap f = dimap f id
    rmap :: (b -> c) -> p a b -> p a c
    rmap = dimap id
```

3つの関数はすべてデフォルトで実装されています．Bifunctor と同じように、Profunctor のインスタンスを宣言するとき、 dimap を実装して lmap と rmap のデフォルトを受け入れるか、lmap と rmap の両方を実装して dimap のデフォルトを受け入れるかを選択することができま す。



dimap

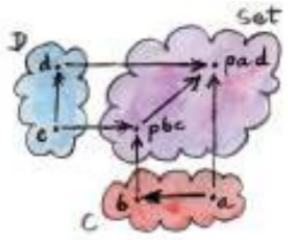
Now we can assert that the function-arrow operator is an instance of a `Profunctor`:

```
instance Profunctor (->) where
    dimap ab cd bc = cd . bc . ab
    lmap = flip (.)
    rmap = (.)
```

Profunctors have their application in the Haskell lens library. We'll see them again when we talk about ends and coends.

8.8 The Hom-Functor

The above examples are the reflection of a more general statement that the mapping that takes a pair of objects a and b and assigns to it the set of morphisms between them, the hom-set $\mathbf{C}(a, b)$, is a functor. It is a functor from the product category $\mathbf{C}^{op} \times \mathbf{C}$ to the category of sets, **Set**.



dimap

これで、function-arrow 演算子は Profunctor のインスタンスであることが証明されました。

```
instance Profunctor (->) where
    dimap ab cd bc = cd . bc . ab
    lmap = flip (.)
    rmap = (.)
```

プロファンクタはHaskellのレンズライブラリに応用されています。また、終端と終端について話すときにも出てくるでしょう。

8.8 The Hom-Functor

上の例は、オブジェクト　Ll_lD44E と □ のペアを取って、それらの間の形態素の集合である hom-set □(□) を代入するマッピングがファンクタである、というより一般的な記述の反映である。これは積のカテゴリー□ × □から集合のカテゴリー□□へのファンクタである。

Let's define its action on morphisms. A morphism in $\mathbf{C}^{op} \times \mathbf{C}$ is a pair of morphisms from $\mathbf{C}$:

$$f :: a' \to a$$
$$g :: b \to b'$$

The lifting of this pair must be a morphism (a function) from the set $\mathbf{C}(a, b)$ to the set $\mathbf{C}(a', b')$. Just pick any element h of $\mathbf{C}(a, b)$ (it's a morphism from a to b) and assign to it:

$$g \circ h \circ f$$

which is an element of $\mathbf{C}(a', b')$.

As you can see, the hom-functor is a special case of a profunctor.

8.9 Challenges

1. Show that the data type:

   ```
   data Pair a b = Pair a b
   ```

   is a bifunctor. For additional credit implement all three methods of `Bifunctor` and use equational reasoning to show that these definitions are compatible with the default implementations whenever they can be applied.
2. Show the isomorphism between the standard definition of `Maybe` and this desugaring:

   ```
   type Maybe' a = Either (Const () a) (Identity a)
   ```

   Hint: Define two mappings between the two implementations. For additional credit, show that they are the inverse of each other using equational reasoning.

モルヒズムに対する作用を定義しよう。□ × □ におけるモルヒズムは □ からのモルヒズムの組である。

$$f :: a' \to a$$
$$g :: b \to b'$$

このペアのリフトは集合 □(1D44E, □) から集合 □(1D44E , □) へのモーフィズム（関数）でなければなりません。 1D44E, □) の任意の要素 1D21 0E を選んで（ 1D44E から□へのモーフィズムだ）、それに代入すればいい。

$$g \circ h \circ f$$

これは （ 1D44E , □ ）の要素である。見ての通り、hom -functor は profunctor の特殊な場合です。

8.9 Challenges

1. Show that the data type:

   ```
   data Pair a b = Pair a b
   ```

   はバイファンクタです。追加単位としては、Bifunctorの3つのメソッドすべてを実装し、これらの定義が適用できるときはいつでもデフォルトの実装と互換性があることを等式推論で示しなさい。2. 2. Maybeの標準的な定義と、このデスガリングとの間の同型性を示せ。

   ```
   type Maybe' a = Either ( Const () a) ( Identity a)
   ```

   ヒント：2つの実装の間に2つのマッピングを定義しなさい。追加のクレジットとして、等式推論を用いて、それらが互いに逆であることを示せ。

3. Let's try another data structure. I call it a `PreList` because it's a precursor to a `List`. It replaces recursion with a type parameter `b`.

   ```
   data PreList a b = Nil | Cons a b
   ```

   You could recover our earlier definition of a `List` by recursively applying `PreList` to itself (we'll see how it's done when we talk about fixed points).
   Show that `PreList` is an instance of `Bifunctor`.
4. Show that the following data types define bifunctors in `a` and `b`:

   ```
   data K2 c a b = K2 c

   data Fst a b = Fst a

   data Snd a b = Snd b
   ```

   For additional credit, check your solutions against Conor McBride's paper Clowns to the Left of me, Jokers to the Right[1].
5. Define a bifunctor in a language other than Haskell. Implement `bimap` for a generic pair in that language.
6. Should `std::map` be considered a bifunctor or a profunctor in the two template arguments `Key` and `T`? How would you redesign this data type to make it so?

[1] http://strictlypositive.org/CJ.pdf

3. もう一つのデータ構造を試してみましょう。これはListの前駆体なので、PreListと呼んでいます。これは再帰を型パラメータbで置き換えたものです。

   ```
   data PreList a b = Nil | Cons a b
   ```

PreListを自分自身に再帰的に適用することで、先ほどのListの定義を復元することができます（固定点の話をするときに、どのように行うか見てみましょう）。PreListがBifunctorのインスタンスであることを示せ。4. 以下のデータ型がa,bでバイファンクタを定義することを示せ.

```
data K2 c a b = K2 c

data Fst a b = Fst a

data Snd a b = Snd b
```

また、Conor McBride の論文 Clowns to the Left of me, Jokers to the Right 1 と比較して、解答をチェックしてください。5. Haskell以外の言語でbifunctorを定義してください。その言語でbimapを実装し、一般的なペアを定義しなさい。6. 6. Std::map は Key と T の2つのテンプレート引数において、バイファンクタ とプロファンクタのどちらと考えるべきでしょうか？このデータ型をそうするために、どのように再設計しますか？

[1] http://strictlypositive.org/CJ.pdf