… whenever new abstract objects are constructed in a specified way out of given ones, it is advisable to regard the construction of the corresponding induced mappings on these new objects as an integral part of their definition. [1]

A **functor** is a morphism of categories

Definition 0.1. A **covariant functor** $F : \mathcal{C} \to \mathcal{D}$ between categories $\mathcal{C}$ and $\mathcal{D}$ consists of the **data**

- an object $Fc \in \mathcal{D}$ for every object $c \in \mathcal{C}$[1]
- a morphism $Ff : Fc \to Fc'$ for every morphism $f : c \to c'$ in $\mathcal{C}$[2]

$$\begin{tikzcd}
\mathcal{C} \arrow[d, "F"'] & c \arrow[r, "f"] \arrow[d, "F"'] & c' \arrow[d, "F"] \\
\mathcal{D} & Fc \arrow[r, "Ff"'] & Fc'
\end{tikzcd}$$

which satisfy the **functoriality axioms**

- for any composable morphisms f, g in $\mathcal{C}$

$$Fg \circ Ff = F(g \circ f)$$

- for each object $c \in \mathcal{C}$

$$F\mathbf{1}_c = \mathbf{1}_{Fc}$$

See [2] Ch.1.3, [3] 1.3, [4] Ch.7 and 8.

Definition 0.2. A **contravariant functor** between categories $\mathcal{C}$ and $\mathcal{D}$ is a functor $F : \mathcal{C}^{op} \to \mathcal{D}$ that is

- an object $Fc \in \mathcal{D}$ for every object $c \in \mathcal{C}$

[1] this in general is not a function, the categories might not be small
[2] there exists a *function* $F_{c,c'} : \mathcal{C}(c, c') \to \mathcal{D}(Fc, Fc')$ for every pair of objects $c, c' \in \mathcal{C}$ mapping hom-sets

- a morphism $Ff : Fc' \to Fc$ for every morphism $f : c \to c'$ in $\mathcal{C}$[3]

$$\begin{tikzcd}
\mathcal{C}^{op} \arrow[d, "F"'] & c \arrow[r, "f"] \arrow[d, "F"'] & c' \arrow[d, "F"] \\
\mathcal{D} & Fc & Fc' \arrow[l, "Ff"]
\end{tikzcd}$$

which satisfy the **functoriality axioms**

- for any composable morphisms f, g in $\mathcal{C}$

$$Ff \circ Fg = F(g \circ f)$$

- for each object $c \in \mathcal{C}$

$$F\mathbf{1}_c = \mathbf{1}_{Fc}$$

See also Contravariant functors are Weird.

Definition 0.3. A **bifunctor** is a functor

$$B : \mathcal{C} \times \mathcal{D} \to \mathcal{E}$$

from a product category

$$\begin{tikzcd}
\mathcal{C} \times \mathcal{D} \arrow[d, "B"] & (c,d) \arrow[r, "(f,g)"] \arrow[d, "B"'] & (c',d') \arrow[d, "B"] \\
\mathcal{E} & B(c,d) \arrow[r, "B(f,g)"'] & B(c',d')
\end{tikzcd}$$

Definition 0.4. A functor $F : \mathcal{C} \to \mathcal{D}$ is

- **faithful** iff the map $\mathcal{C}(c, c') \to \mathcal{D}(Fc, Fc')$ is injective
- **full** iff the map $\mathcal{C}(c, c') \to \mathcal{D}(Fc, Fc')$ is surjective

for any $c, c' \in \mathcal{C}$.

[3]here we represent a morphism $f : c' \to c$ in $\mathcal{C}^{op}$ as the corresponding morphism $f : c \to c'$ in $\mathcal{C}$

Proposition 0.1. *Functors preserve isomorphisms.*

Proof. Consider a functor F and an isomorphism $f : c \to c'$ ($f \circ f^{-1} = \mathbf{1}_{c'}$, $f^{-1} \circ f = \mathbf{1}_c$) by functoriality we have

$$Ff \circ Ff^{-1} = F(f \circ f^{-1}) = F(\mathbf{1}_{c'}) = \mathbf{1}_{Fc'}$$
$$Ff^{-1} \circ Ff = F(f^{-1}f) = F(\mathbf{1}_c) = \mathbf{1}_{Fc}$$

thus Ff is an isomorphism.
□

Note that $\mathcal{D}$ may be richer in morphisms than $\mathcal{C}$ so a functor F may create morphisms there. That is even if does not exists a $f : x \to y$ in $\mathcal{C}$ there may exist a $Fx \to Fy$ in $\mathcal{D}$. F cannot delete morphisms though.

We say that F **preserves** morphisms but does not **reflect** them.

Functors formalize the the idea of **pattern recognition**.
The idea is to

1. define a structure: the **pattern**
2. find its instances inside another structure: the **target**

Since categories formalize structure then both the pattern and the target can be formalized as categories.

A **match** of the pattern into the target is a correspondence between the objects and the morphisms of the pattern and the objects and the morphisms of the target.
Moreover the correspondences must preserve the category structure (it must preserve composition).

Another way to look at $F : \mathcal{C} \to \mathcal{D}$ is that it is a *picture* of $\mathcal{C}$ (the **model**) into $\mathcal{D}$.

This naturally leads to the definition of the functor as *structure preserving* morphism between categories.

Another way to look at (endo-)functors is as **containers** (look especially at the `List` and `Maybe` examples). To apply a function to a container is to apply it to the content. The `Reader` is a also container: think of the function as a container indexed by the function argument[4].

But we said that objects in a category have no internal structure. Functors hide one type inside another without forgetting about it, it means that it must someway encode information about the content with *morphisms*. Note that the functor does not allow you to retrieve the content but it allows you to operate on it.

Contravariant (endo-)functors can be seen as **negative containers** (look at `Op r` example). They do not contain an item, they require an item to work: they contain an anti-item.

See also

- Category Theory 6.1: Functors
- Category Theory 6.2: Functors in programming
- Category Theory 7.1: Functoriality, bifunctors
- Functors (examples)

[1] S. M. Lane, *Categories for the Working Mathematician* (Springer New York, 1971).
[2] E. Riehl, *Category Theory in Context* (Dover, 2015).
[3] P. Perrone, *Notes on Category Theory with Examples from Basic Mathematics*, (2019).
[4] B. Milewski, *Category Theory for Programmers* (2019).

[4] in Haskell there's not really a distinction between data and function