Chapter 1

Categories, functors, natural transformations (DRAFT)

1.1 Programming: the art of composition

What is programming? In some sense, programming is just about giving instructions to a computer, a strange, very literal beast with whom communication takes some art. But this alone does not explain why more and more people are programming, and why so many (perhaps including you, dear reader) are interested in learning to program better. Programming is about using the immense power of a computer to solve problems. Programming, and computers, allow us to solve big problems, such as forecasting the weather, controlling a lunar landing, or instantaneously sending a photo to your mom on the other side of the planet.

How do we write programs to solve these big problems? We decompose the big problems into smaller ones. And if they are still too big, we decompose them again, and again, until we are left writing the very simple functions that come at the base of a programming language, such as concatenating two lists (or even modifying a register). Then, by solving these small problems and composing the solutions, we arrive at a solution to the larger problem.

To take a very simple example, suppose we wanted to take a sentence, and remove all the spaces. This capability is not provided to us in the base library of a language like Haskell. Luckily, we can perform the task by composition.

Our first ingredient will be the function `words`, which takes a sentence (encoded as a string) and turns it into a comma-separated list of words. For example, here's what happens if we call it on the sentence `"Hello world"`:

```
Prelude> words "Hello world"
["Hello","world"]
```

Our second ingredient is the function `concat`, which takes a list of strings and concatenates them to return a single string. For example, we might run:

```
Prelude> concat ["I","like","cats"]
"Ilikecats"
```

Composition in Haskell is denoted by a period "." between two functions. We can define new functions by composing existing functions:

```
Prelude> let de-space = concat . words
```

This produces a solution to our problem.

```
Prelude> de-space "Yay composition"
"Yaycomposition"
```

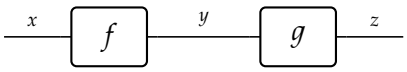
Given that composition is such a fundamental part of programming, and problem solving in general, it would be nice to have a science devoted to understanding the essence of composition. In fact, we have one: it's known as category theory.

As coauthor-Bartosz once wrote: a category is an embarrassingly simple concept. A category is a bunch of objects, and some arrows that go between them. We assume nothing about what these objects or arrows are, all we have are names for them. We might call our objects very abstract names, let x, y, and z, or more evocative ones, like 42 or `True` or `String`. Our arrows have a source and a target; for example, we might have an arrow called f, with source x and target y. This could be depicted as an arrow:

$$x \xrightarrow{f} y$$

Or as a box called f, that accepts an 'x' and outputs a 'y':

What is important, is that in a category we can compose. Given an arrow $f : x \to y$ and an arrow $g : y \to z$, we may compose them to get an arrow with source x and target z. We denote this arrow $f \fatsemi g : x \to z$; we might also draw it as piping together two boxes:



This should remind you of our de-spacing example above, stripped down to its bare essence.

Categories are about arrows from one object to another, and how to compose them; they emphasize the viewpoint that arrows, the way objects relate to each other, are important. In fact, in a beautiful example of categorical thinking, mathematicians discovered categories not on their own, and not even by thinking about the way they relate to each other, but by thinking about how their *relationships* relate to each other.

Relationships between categories are known as *functors*, and relationships between functors are known as *natural transformations*. In this chapter, we'll formally introduce

categories, functors, and natural transformations, together with a bunch of useful examples to help you think about them.

These three structures will form the cornerstones of our toolbox for thinking about programming. Through this chapter we'll introduce the programming language Haskell, and hint at how these categorical structures find manifestations in type constructors, polymorphic functions, and many other ways.

A category is a network of relationships, or slightly more precisely, a bunch of objects, some arrows that go between them, and a formula for composing arrows. A programming language generally has a bunch of types and some programs that go between them (ie. take input of one type, and turn it into output of another). The guiding principle of this book, is that if you think of your (ideal) programming language like a category, good programs will result.

So let's go forward, and learn about categories. But first, it will be helpful to mention a few things about another fundamental notion: sets.

Category theory has a love/hate relationship with set theory. Secretly, category theorists dream of overthrowing the rule of set theory as the foundation of mathematics (more recently, homotopy type theorists have restarted this quest). But so far there is no formulation of category theory that would not require sets at its basis. You'll see that we might skirt the issue by saying that category is a "bunch" of objects, because they don't really have to form a set-theoretical set, but then we can't say the same about morphisms between objects. They do form sets. Every time we draw a commuting diagram, we are asserting that two (or more) elements in some set of morphisms are equal. Granted, there are ways of postponing this fallback to sets by introducing enriched categories, but even these eventually force you to draw diagrams that, at some level, assert the equality of set elements. The bottom line is that we have to have some knowledge of set theory before we tackle category theory. We can think of sets as these primordial, structureless categories, but in order to build more interesting structures we have to start from the structureless.

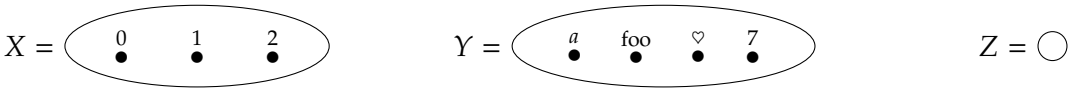
There is another incentive to studying set theory, that is that sets themselves form a category, which is a very fertile source of examples and intuitions. On the one hand, we wouldn't like you to think of morphisms in an arbitrary category as being some kind of functions. On the other hand, we can define and deeply understand many properties of sets and functions that can be generalized to categories.

What an empty set is everybody knows, and there's nothing wrong in imagining that an initial object in a category is "sort of" like an empty set. At the very least, it might help in quickly rejecting some wrong ideas that we might form about initial objects. We can quickly test them on empty sets. Programmers know a lot about testing, so this idea that the category of sets is a great testing platform should sound really attractive.

1.2 Two fundamental ideas: sets and functions

1.2.1 What is a set?

A set, in this book, is a bag of dots.



This set X has three *elements*, the dots. We could write it in text form as $X = \{0, 1, 2\}$; when we write $1 \in X$ it means "1 is an element of X". The set Z has no elements; it's called the *empty set*. The number of elements of a set X is called its *cardinality* because cardinals were the first birds to recognize the importance of this concept. We denote the cardinality of X as $|X|$. Note that cardinalities of infinite sets may involve very large numbers indeed.

Example 1.1. Here are some sets you'll see repeated throughout the book.

Name	**Symbol**	**Elements between braces**
The natural numbers	$\mathbb{N}$	$\{0, 1, 2, 3, \ldots, 42^{2048}{+}17, \ldots\}$
The nth ordinal	$\underline{n}$	$\{1, \ldots, n\}$
The empty set	$\varnothing$	$\{\}$
The integers	$\mathbb{Z}$	$\{\ldots, -59, -58, \ldots -1, 0, 1, 2, \ldots\}$
The booleans	$\mathbb{B}$	$\{\texttt{true}, \texttt{false}\}$

Exercise 1.2.

1. What is the cardinality $|\mathbb{B}|$ of the booleans?
2. What is the cardinality $|\underline{n}|$ of the nth ordinal?
3. Write $\underline{1}$ explicitly as elements between braces.
4. Is there a difference between $\underline{0}$ and $\varnothing$? ◊

Definition 1.3. Given a set X, a *subset* of it is another set Y such that every element of Y is an element of X. We write $Y \subseteq X$, a kind of curved version of a less-than-or-equal-to symbol.

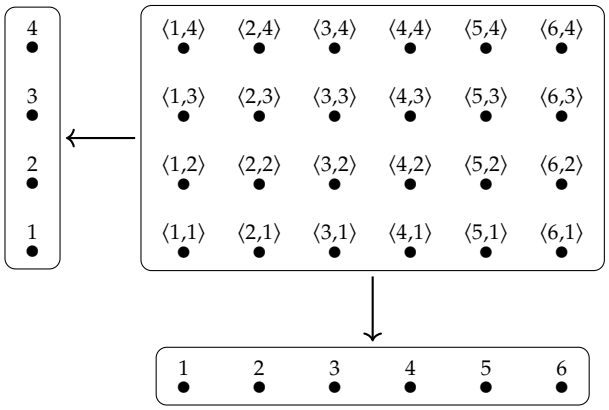
Exercise 1.4.

1. Suppose that a set X has finitely many elements and Y is a subset. Is it true that the cardinality of Y is necessarily less-than-or-equal-to the cardinality of X? That is, does $Y \subseteq X$ imply $|Y| \leq |X|$?
2. Suppose now that Y and X are arbitrary sets, but that $|Y| \leq |X|$. Does this imply

$Y \subseteq X$? If so, explain why; if not, give a counterexample. ◊

Definition 1.5. Given a set X and a set Y, their *(cartesian) product* is the set $X \times Y$ that has pairs $\langle x, y \rangle$ as elements, where $x \in X$ and $y \in Y$.

One should picture the product $X \times Y$ as a grid of dots. Here is a picture of $\underline{6} \times \underline{4}$ and its projections:



The name *product* is nice because the cardinality of the product is the product of the cardinalities: $|X \times Y| = |X| \times |Y|$. For example $|\underline{6} \times \underline{4}| = 24$, the product of 6 and 4.

Exercise 1.6. We said that the cardinality of the product $X \times Y$ is the product of $|X|$ and $|Y|$. Does that work even when X is empty? Explain why or why not. ◊

One can take the product of any two sets, even infinite sets, e.g. $\mathbb{N} \times \mathbb{Z}$ or $\mathbb{N} \times \underline{4}$.

Exercise 1.7.

1. Name three elements of $\mathbb{N} \times \underline{4}$.
2. Name three subsets of $\mathbb{N} \times \underline{4}$. ◊

1.2.2 Functions

A function is a machine that turns input values into output values. It's *total* and *deterministic*, meaning that every input results in at least one and at most one—i.e. exactly one—output. If you put in 5 today, you'll get an answer, and you'll get exactly the same answer as if you put in 5 tomorrow.

Definition 1.8 (Function)**.** Let X and Y be sets. A *function f from X to Y*, denoted $f : X \to Y$, is a subset of $f \subseteq X \times Y$ with the following properties.

1. For any $x \in X$ there is at least one $y \in Y$ for which $(x, y) \in f$.
2. For any $x \in X$ there is at most one $y \in Y$ for which $(x, y) \in f$.

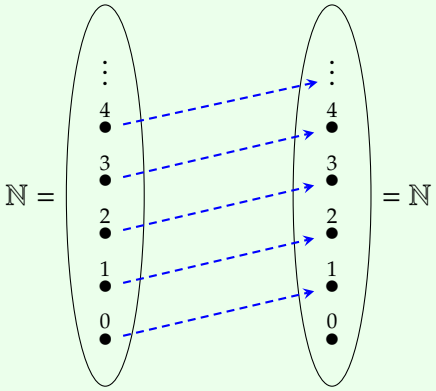
If f satisfies the first property we say it is *total*, and if it satisfies the second property

we say it is *deterministic*.

If f is a function (satisfying both), then we write $f(x)$ or $f\ x$ to denote the unique y such that $(x, y) \in f$.

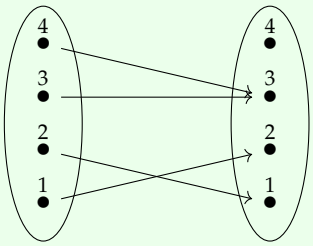
This is a rather abstract definition; perhaps some examples will help. One way of denoting a function $f : X \to Y$ is by drawing "maps-to" arrows $\mapsto$ that emanate from some particular $x \in X$ and point to some particular $y \in Y$. Every $x \in X$ gets exactly one arrow emanating from it, but no such rule for y's.

Example 1.9. The *successor* function $s : \mathbb{N} \to \mathbb{N}$ sends $n \mapsto n+1$. For example $s(52) = 53$. Here's a picture:



Every natural number n input is sent to exactly one output, namely $s(n) = n + 1$.

Example 1.10. Here's a picture of a function $\underline{4} \to \underline{4}$:



Exercise 1.11.

1. Suppose someone says "$n \mapsto n-1$ is also a function $\mathbb{N} \to \mathbb{N}$. Are they right?
2. Suppose someone says "$n \mapsto 42$ is also a function $\mathbb{N} \to \mathbb{N}$. Are they right?
3. Draw arrows from elements in $\underline{3}$ to elements in $\underline{4}$ in a way that's not total.
4. Draw arrows from elements in $\underline{3}$ to elements in $\underline{4}$ in a way that's not deterministic.

◊

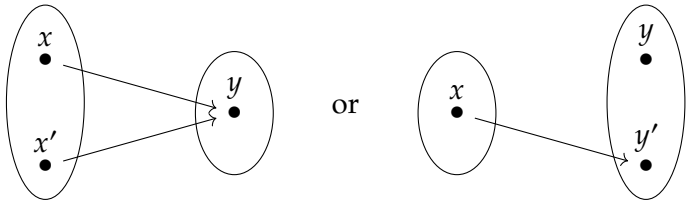
Exercise 1.12.

1. How many functions $\underline{3} \to \underline{2}$ are there? Write them all.
2. How many functions $\underline{1} \to \underline{7}$ are there? Write them all.
3. How many functions $\underline{3} \to \underline{3}$ are there?
4. How many functions $\underline{0} \to \underline{7}$ are there?
5. How many functions $\underline{0} \to \underline{0}$ are there?
6. How many functions $\underline{7} \to \underline{0}$ are there? ◊

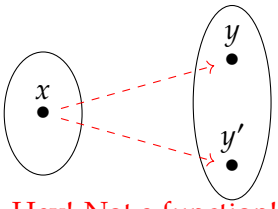
In general, for any $a, b \in \mathbb{N}$ there are b^a functions $\underline{a} \to \underline{b}$. You can check your answers above using this formula. The one case that you might be confused is when $a = b = 0$. In this case, a calculus teacher would say "the expression 0^0 is undefined", but we're not in calculus class. It may be true that as a functions of real numbers, there is no smooth way to define 0^0, but for natural numbers, the formula "count the number of functions $a \to b$" works so well, that we define $0^0 = 1$.

1.2.3 Some intuitions about functions

A lot of intuitions about functions translate into category theory. A function is allowed to collapse multiple elements from the source into one element of the target or to miss elements of the target:



On the other hand, a function is forbidden from splitting a source element into multiple target elements.

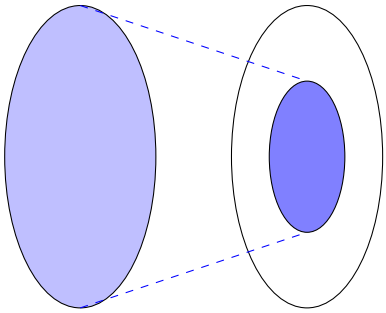


Hey! Not a function!

There is another source of asymmetry: functions are defined for all elements in the source set. This condition is not symmetric: not every element in the target set has to be covered. The subset of elements in the target that are in a functional relation with the source is called the *image* of a function:

$$\operatorname{im} f = \{y \in Y \mid \exists x \in X.\, f(x) = y\}$$

"The image of f is the set of y's in Y such that there exists an x in X where $f(x) = y$."



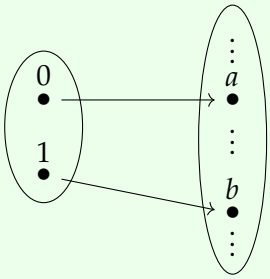
The directionality of functions is reflected in the notation we are using: we represent functions as arrows going from source to target, from *domain* to *codomain*. This directionality makes them interesting.

You may think of a function that maps many things to one as discarding some information. The function $\mathbb{N} \to \mathbb{B}$ that takes a natural number and returns `true` if it's even and `false` otherwise doesn't care about the precise value of a number, it only cares about it being even or odd. It *abstracts* some important piece of information by discarding the details it considers inessential.

You may think of a function that doesn't cover the whole codomain as *embedding* its source in a larger environment. It's creating a model of its source in a larger context, especially when it additionally collapses it by discarding some details. A helpful intuition is to think of the source set as defining a shape that is projected into a bigger set and forms a pattern there. Compared to category theory, set theory offers a very limited choice of bare-bones shapes.

Example 1.13. A singleton set is the simplest non-trivial shape. A function from a singleton picks a single element from the target set. There are as many distinct functions from a singleton to a non-empty set as there are elements in that set. In fact we may identify elements of a set with functions from the singleton set. We'll see this idea used in category theory to define *global elements*.

Example 1.14. A two-element set can be used to pick pairs of elements in the target set. It embodies the idea of a pair.



As an example, consider a function from a two-element set to a set of musical notes. You may say that it embodies the idea of a musical interval.

Exercise 1.15. Let N be the set of musical notes, or perhaps the keys on a standard piano. Person A says "a musical interval is a subset $I \subseteq N$ such that I has two elements. Person B says "no, a musical interval is a function $i\colon \underline{2} \to N$, from a two element set to N. They prepare to fight bitterly, but a peacemaker comes by and says "you're both saying the same thing!" Are they? ◊

Exercise 1.16. How would you describe functions from an empty set to another set A. What do they model? (This is more of a Zen meditation than an exercise. There are no right or wrong answers.) ◊

Abstraction and modeling are the two major tools that help us understand the world.

1.3 Categories

In this section we'll define categories, and give a library of useful examples. Instead of beginning directly with a definition, we'll motivate the definition by discussing the ur-example: the category of sets and functions.

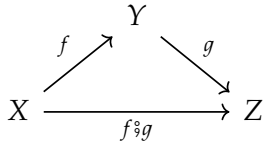
1.3.1 The category of sets

The identity function on a set X is the function $\mathrm{id}_X\colon X \to X$ given by $\mathrm{id}_X(x) = x$. It does nothing. This might seem like a very boring thing, but it's like 0: adding it does nothing, but that makes it quite central. For example, 0 is what defines the relationship between 6 and -6: they add to 0.

Just like 0 as a number really becomes useful when you know how to combine numbers using +, the identity function really becomes useful when you know how to combine functions using "composition".

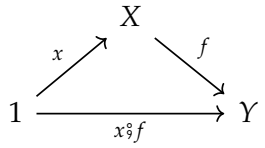
Definition 1.17. Let $f\colon X \to Y$ and $g\colon Y \to Z$ be functions. Then their *composite*, denoted either $g \circ f$ or $f \fatsemi g$, is the function $X \to Z$ sending each $x \in X$ to $g(f(x)) \in Z$.

The above definition makes it look like $g \circ f$ is better notation, because $(g \circ f)(x) = g(f(x))$ looks than the backwards-seeming formula $(f \fatsemi g)(x) = g(f(x))$, which has a sort of switcheroo built in. But there are good reasons for using $f \fatsemi g$, e.g. this diagram

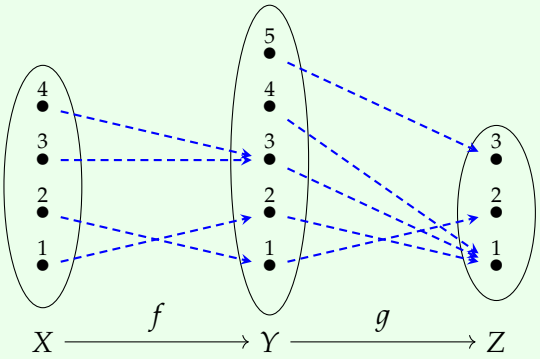


Another is that—as we saw in **??**—an element of X is just a function $\underline{1} \to X$. From this point of view, function application is just composition. That is, if $f\colon X \to Y$ is a

function then what's normally denoted $f(x)$ could instead be denoted $x \fatsemi f$



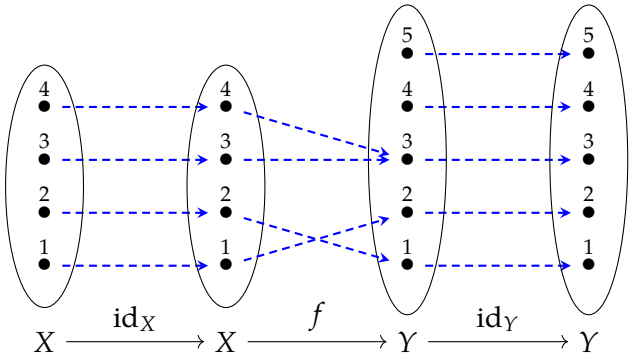
Example 1.18. A great way to visualize function composition is by path following.



Following the paths from X to Z, we see that $g(f(3) = 1$ and $g(f(2)) = 2$.

Now the above backwards-seeming formula becomes $x \fatsemi (f \fatsemi g) = (x \fatsemi f) \fatsemi g$, and the switcheroo is gone. That is, we see that the preference for the $\circ$ notation is built in to the $f(x)$ notation, which is already backwards. In the $\fatsemi$ direction, you start with an x, then you apply f, then you apply g, etc.

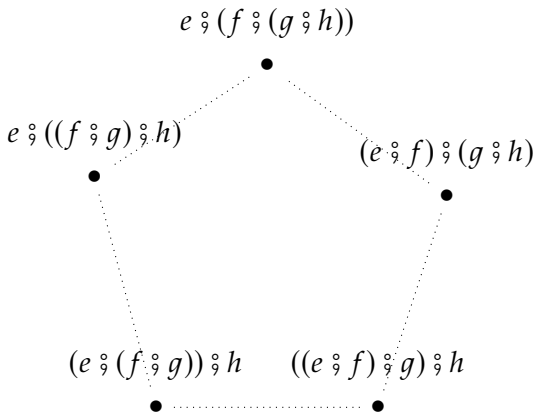
Suppose that $f : X \to Y$ is a function. Then if we compose it with either (or both!) of the identity functions, $\mathrm{id}_X : X \to X$ or $\mathrm{id}_Y : Y \to Y$, the result is again f.



Composing with the identity doesn't do anything.

The second property we want to highlight is that you can compose multiple functions at once, not just two. That is, if you have a string of n functions $X_0 \xrightarrow{f_1} X_1 \xrightarrow{f_2} \cdots \xrightarrow{f_n} X_n$, you can collapse it into one function by composing two-at-a-time in many

different ways. This is denoted mathematically using parentheses. For example we can compose this string of functions $V \xrightarrow{e} W \xrightarrow{f} X \xrightarrow{g} Y \xrightarrow{h} Z$ as any of the five ways represented in the pentagon below:



(1.19)

It turns out that all these different ways to collapse four functions give the same answer. You could write it simply $e \fatsemi f \fatsemi g \fatsemi h$ and forget the parentheses all together.

A better word than "collapse" is *associate*: we're associating the functions in different ways. The *associative law* says that $(f \fatsemi g) \fatsemi h) = f \fatsemi (g \fatsemi h)$.

When composing functions, how you parenthesize doesn't matter: you'll get the same answer no matter what.

Exercise 1.20. Consider the pentagon (sometimes called the *associahedron*) in Eq. (1.19).

1. Show that each of the five dotted edges corresponds to an instance of the associative law in action.
2. Are there any other places where we could do an instance of the associative law that isn't drawn as a dotted edge?

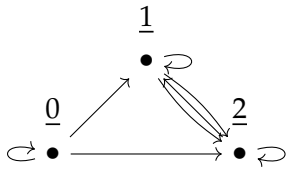
◊

1.3.2 Definition of category

We begin with a slogan:

A category is an organized network of relationships.

The prototypical category is **Set**, the category of sets.[1] The objects of study in **Set** are, well, sets. The relationships of study in **Set** are the functions. These form a vast network of arrows pointing from one set to another.



(and if you toss 3 into the picture, you'll need to add 44 more arrows with it!)

But **Set** not just any old network: it's organized in the sense that we know how to compose the functions. This imposes a tight constraint: if pretty much any function was somehow left out of the network, it would cause a huge catastrophe of missing composites.

Let's see the precise definition.

Definition 1.21. A *category* $\mathcal{C}$ consists of four constituents:

1. a set $\mathrm{Ob}(\mathcal{C})$, elements of which are called *objects of* $\mathcal{C}$;
2. for every pair of objects $c, d \in \mathrm{Ob}(\mathcal{C})$ a set $\mathcal{C}(c,d)$, elements of which are called *morphisms from c to d* and often denoted $f : c \to d$;
3. for every object c, a specified morphism $\mathrm{id}_c \in \mathcal{C}(c,c)$ called the *identity morphism for c*; and
4. for every three objects b, c, d and morphisms $f : b \to c$ and $g : c \to d$, a specified morphism $(f \fatsemi g) : b \to d$ called the *composite of f and g*, sometimes denoted $g \circ f$.

These constituents are subject to three constraints:

Left unital: for any $f : c \to d$, the equation $\mathrm{id}_c \fatsemi f = f$ holds

Right unital: for any $f : c \to d$, the equation $f \fatsemi \mathrm{id}_d = f$ holds

Associative: for any $f_1 : c_1 \to c_2$, $f_2 : c_2 \to c_3$, and $f_3 : c_3 \to c_4$, the equation $(f_1 \fatsemi f_2) \fatsemi f_3 = f_1 \fatsemi (f_2 \fatsemi f_3)$ holds

Example 1.22 (The category of sets). The category of sets, denoted **Set** has all the sets[a] as its objects the sets. Given two sets $A, B \in \mathrm{Ob}(\mathbf{Set})$, we have $\mathbf{Set}(A,B) \coloneqq \{f : A \to B \mid f \text{ is a function}\}$. There's sets everywhere: objects are sets, for every two objects $\mathbf{Set}(A,B)$ is also a set. This use of sets as both objects and morphisms is a distinctive feature of certain categories that we'll see a lot of later, called "closed categories." For

[1] Actually, **Set** is the category of *small sets*, meaning sets all of whose elements come from some huge pre-chosen universe $\mathbb{U}$. The axiom of *Grothendieck universes*, which says that there's an infinite ascending hierarchy of these $\mathbb{U}$'s uses some pretty heavy set theory to make sure you don't have to worry about weird paradoxes that come about when one allows themselves to speak of the infamous 'set of all sets.' For a given universe $\mathbb{U}$ the purportedly 'small sets' can include uncountably infinite sets, as long as they're in $\mathbb{U}$. This way, rather than talking about the set of all sets, we talk about the (larger) set of all (small) sets, and everything is just a set, albeit in different universes.

The point is, you don't need to worry about all these "size issues"; just focus on the category theory for now. If you want to get deep into the technical set-theoretic issues, see [**].

now, just roll with it.

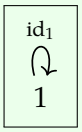
[a] In some universe $\mathbb{U}$; don't worry about this for now.

Exercise 1.23. Let's return to a remark we made earlier about the category **Set**. Is there any single morphism you can take out of it, such that—without taking away any more morphisms—the remaining collection of objects and morphisms is still a category? ◊

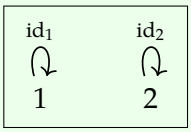
1.3.3 Some elementary examples

Even though the notion of category is somehow modeled on **Set**, there are tons of categories that don't really resemble it at all! Some of the most important categories are the little ones. It's like how the numbers 0, 1, and 2 are perhaps even more important than the number 9^9.

Example 1.24. There is a single-object category **1** with no other arrows but the identity arrow.



Example 1.25 (Discrete categories). You can also have a category **2** with two objects 1 and 2 and two identity morphisms id_1 and id_2.



In fact, for every set S, there's an associated category $\mathbf{Disc}(\mathbf{S})$, called the *discrete category on* S. The objects of $\mathbf{Disc}(\mathbf{S})$ are the elements of S, i.e. $\mathrm{Ob}(\mathbf{Disc}(\mathbf{S})) = S$, and the morphisms are just the identities:

$$\mathbf{Disc}(\mathbf{S})(s, s') = \begin{cases} \{\mathrm{id}_s\} & \text{if } s = s' \\ \varnothing & \text{if } s \neq s' \end{cases}$$

So far we haven't gained any advantage over sets. But in a category we not only have objects; we may have arrows between them. This is when interesting structures arise. For instance, we can add a morphism $ar : 1 \to 2$ to the two-object category.

Example 1.26. This tiny category is sometimes called the *interval* category **I**.

$$\mathrm{id}_1 \circlearrowright 1 \xrightarrow[ar]{} 2 \circlearrowright \mathrm{id}_2$$

Exercise 1.27. Show that the interval category **I** satisfies all the laws of a category. ◊

Example 1.28. $\mathrm{id}_1 \circlearrowright 1 \underset{f^{-1}}{\overset{f}{\rightleftarrows}} 2 \circlearrowright \mathrm{id}_2$

Exercise 1.29. Show that a category with only two nontrivial arrows going in the opposite direction satisfies all the laws of a category. In particular, show that the two morphisms must be the inverse of each other. ◊