Chapter 1

Is Haskell a category?

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.

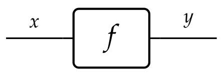
```
Prelude> de-space "Yay composition"
"Yaycomposition"
```

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.

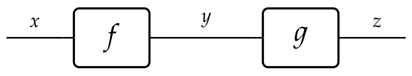
As coauthor-Bartosz once wrote: a category is an embarrassingly simple concept. A category is a bunch of objects, and some arrows that go between them. We assume nothing about what these objects or arrows are, all we have are names for them. We might call our objects very abstract names, let x, y, and 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 $f\colon x \to y$ and an arrow $g\colon y \to z$, we may compose them to get an arrow with source x and target z. We denote this arrow $f \fatsemi g\colon x \to z$; 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.

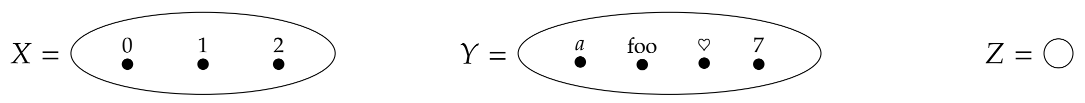
A category is a network of relationships, or slightly more precisely, a bunch of objects, some arrows that go between them, and a formula for composing arrows. A programming language generally has a bunch of types and some programs that go between them (ie. take input of one type, and turn it into output 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.

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.

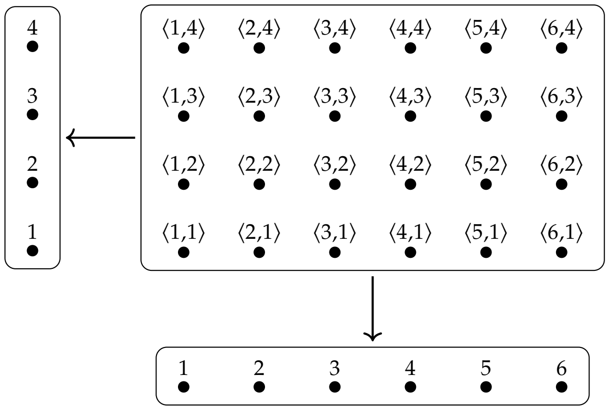
Exercise 1.4.

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.

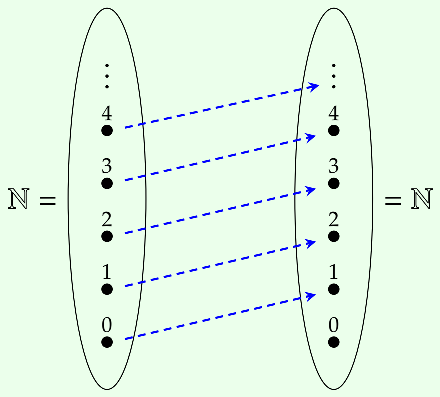
Definition 1.8 (Function)**.** Let X and Y be sets. A *function f from X to Y*, denoted $f : X \to Y$, is a subset of $f \subseteq X \times Y$ with the following properties.
1. For any $x \in X$ there is at least one $y \in Y$ for which $(x, y) \in f$.
2. For any $x \in X$ there is at most one $y \in Y$ 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 $(x, y) \in f$.

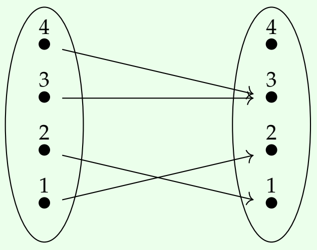
This is a rather abstract definition; perhaps some examples will help. One way of denoting a function $f\colon X \to Y$ is by drawing "maps-to" arrows $\mapsto$ that emanate from some particular $x \in X$ and point to some particular $y \in Y$. Every $x \in X$ gets exactly one arrow emanating from it,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

Example 1.9. The *successor* function $s\colon \mathbb{N} \to \mathbb{N}$ 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.

$\lozenge$

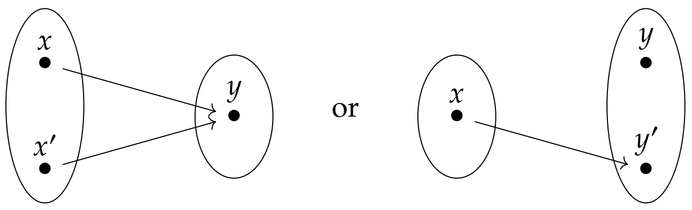
Exercise 1.12.

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? ◊

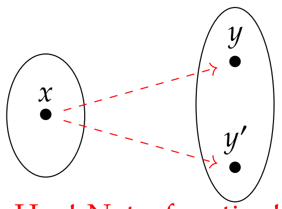
In general, for any $a, b \in \mathbb{N}$ there are b^a functions $\underline{a} \to \underline{b}$. You can check your answers above using this formula. The one case that you might be confused is when $a = b = 0$. In this case, a calculus teacher would say "the expression 0^0 is undefined", but we're 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:



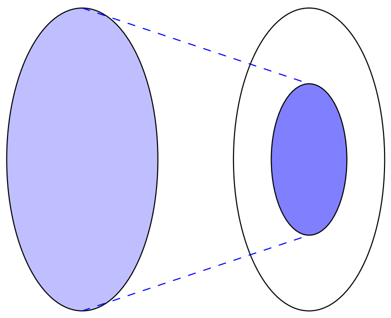
On the other hand, a function is forbidden from splitting a source element into multiple target elements.



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.

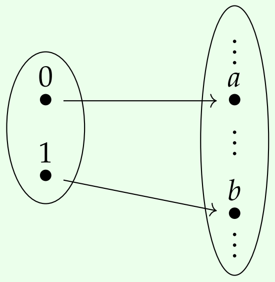
You may think of a function that maps many things to one as discarding some information. The function $\mathbb{N} \to \mathbb{B}$ that takes a natural number and returns `true` if it's even and `false` otherwise doesn't care about the precise value of a number, it only cares about it being even or odd.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

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

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.) ◊

Remark 1.17. 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, even most category theorists continue to work within set-theoretic foundations. 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.

1.3 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 1.18. Let $f \colon X \to Y$ and $g \colon Y \to Z$ be functions. Then their *composite*, denoted either $g \circ f$ or $f \mathbin{⨟} g$, is the function $X \to Z$ sending each $x \in X$ to $g(f(x)) \in Z$.

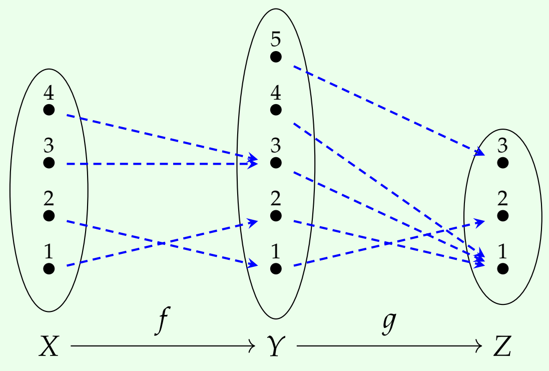
The above definition makes it look like $g \circ f$ is better notation, because $(g \circ f)(x) = g(f(x))$ looks than the backwards-seeming formula $(f \mathbin{⨟} g)(x) = g(f(x))$, which has a sort of switcheroo built in. But there are good reasons for using $f \mathbin{⨟} g$, e.g. this diagram

$$\begin{tikzcd}
 & Y \arrow[dr, "g"] & \\
X \arrow[ur, "f"] \arrow[rr, "f ⨟ g"'] & & Z
\end{tikzcd}$$

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 $x \mathbin{⨟} f$

$$\begin{tikzcd}
 & X \arrow[dr, "f"] & \\
1 \arrow[ur, "x"] \arrow[rr, "x ⨟ f"'] & & Y
\end{tikzcd}$$

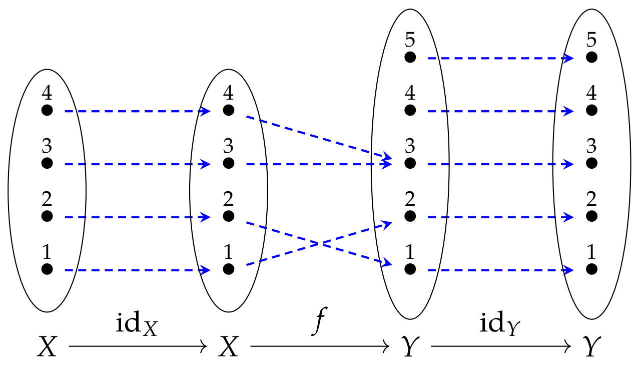
Example 1.19. A great way to visualize function composition is by path following.



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

$$\begin{tikzcd}
 & & e \fatsemi (f \fatsemi (g \fatsemi h)) \arrow[dll, dash, dotted] \arrow[drr, dash, dotted] & & \\
e \fatsemi ((f \fatsemi g) \fatsemi h) \arrow[dr, dash, dotted] & & & & (e \fatsemi f) \fatsemi (g \fatsemi h) \arrow[dl, dash, dotted] \\
 & (e \fatsemi (f \fatsemi g)) \fatsemi h \arrow[rr, dash, dotted] & & ((e \fatsemi f) \fatsemi g) \fatsemi h &
\end{tikzcd} \tag{1.20}$$

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

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:

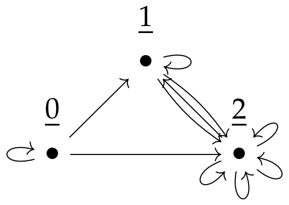
A category is an organized network of relationships.

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

[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 [**].

network of arrows pointing from one set to another.



(and if you toss in $\underline{3}$, you'll need to add $3^0+3^1+3^2+3^3+2^3+1^3+0^3 = 49$ more arrows with it! See https://oeis.org/A231344) (1.22)

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 1.23. 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 $f\colon c \to d$;
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 $f\colon b \to c$ and $g\colon c \to d$, a specified morphism $(f \fatsemi g)\colon b \to d$ called the *composite of* f *and* g, sometimes denoted $g \circ f$.

These constituents are subject to three constraints:

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

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

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

Example 1.24 (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 $\mathbf{Set}(A,B) := \{f\colon A \to B \mid f \text{ is a function}\}$. There's sets everywhere: objects are sets, for every two objects $\mathbf{Set}(A,B)$ is also a set. This situation where the collection of morphisms $A \to B$ itself forms an object is a distinctive feature of certain categories that we'll see a lot of later, called "closed categories." For example, just as there is a *set* of functions from one set to another, a student of linear algebra might know that there is a *vector space* of linear transformations from one vector space to another.

But you don't have to worry about closed categories for now; we'll get to it.

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

Exercise 1.25. 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 1.26. There is a single-object category **1** with no other arrows but the identity arrow.

$$\boxed{\begin{tikzcd} 1 \arrow[loop above, "\mathrm{id}_1"] \end{tikzcd}}$$

Example 1.27 (Discrete categories). You can also have a category with two objects 1 and 2 and two identity morphisms id_1 and id_2.

$$\mathbf{Disc}(2) = \boxed{\begin{tikzcd} 1 \arrow[loop above, "\mathrm{id}_1"] & 2 \arrow[loop above, "\mathrm{id}_2"] \end{tikzcd}}$$

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

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

In particular, the empty category is given by $\mathbf{0} = \mathbf{Disc}(\underline{0})$.

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 $ar\colon 1 \to 2$ to the two-object category.

Example 1.28. This tiny category is sometimes called the *walking arrow category* **2**.

$$\begin{tikzcd} \mathbf{2} := & \overset{1}{\bullet} \arrow[r, "f"] \arrow[loop left, "\mathrm{id}_1"] & \overset{2}{\bullet} \arrow[loop right, "\mathrm{id}_2"] \end{tikzcd}$$

Exercise 1.29. Show that the walking arrow category **2** satisfies all the laws of a category. ◊

Since an identity morphism id_a automatically has to be there for each object a, and since a composite morphism $g \circ f$ automatically has to be there for every pair of morphisms $f : a \to b$ and $g : b \to c$, we often leave these out of our pictures.

Example 1.30 (Not drawing all morphisms). In the picture $\boxed{\bullet \to \bullet \to \bullet}$, only two arrows are drawn, but there are implicitly six morphisms: three identities, the two drawn arrows, and their composite.

So with this new convention, we redraw the walking arrow category from Example 1.28 as

$$\begin{tikzcd} \mathbf{2} := & \overset{1}{\bullet} \arrow[r, "f"] & \overset{2}{\bullet} \end{tikzcd}$$

Example 1.31 (Ordinal categories). There is a progression of *ordinal categories* that look like this:

$$\mathbf{0}\qquad \boxed{}$$

$$\mathbf{1}\qquad \boxed{\overset{1}{\bullet}}$$

$$\begin{tikzcd} \mathbf{2} & \overset{1}{\bullet} \arrow[r, "f_{12}"] & \overset{2}{\bullet} \end{tikzcd}$$

$$\begin{tikzcd} \mathbf{3} & \overset{1}{\bullet} \arrow[r, "f_{12}"] & \overset{2}{\bullet} \arrow[r, "f_{23}"] & \overset{3}{\bullet} \end{tikzcd}$$

$$\cdots$$

Adherents of the religion of Twodeism—called Twoish people—believe that **2** is the categorical manifestation of the creator of all things. (Twoish people also worship the set $\underline{2}$, its "higher-categorical" analogues, and various products of these. Among other pastimes, they enjoy working in *cubical type theory*.)

Exercise 1.32. Being sure to take into account Example 1.30,

1. How many morphisms are there in **3**?
2. How many morphisms are there in **n**?
3. Are **0** and **Disc**(0) the same?
4. Are **1** and **Disc**(1) the same? ◊

Example 1.33 (The walking isomorphism). In the following category there are two objects and four morphisms:

$$\begin{tikzcd}
\overset{1}{\bullet} \arrow[r, bend left, "f"] & \overset{2}{\bullet} \arrow[l, bend left, "g"]
\end{tikzcd}$$

$$g \circ f = \mathrm{id}_1$$
$$f \circ g = \mathrm{id}_2$$

below $\circ$ right	id_1	id_2	f	g
id_1	id_1	Hey!	Hey!	g
id_2	Hey!	id_2	f	Hey!
f	f	Hey!	Hey!	id_2
g	Hey!	g	id_1	Hey!

To the left we see the drawing, with equations that tell us how morphisms compose. We see that f and g are inverses; each is an isomorphism in the sense of Definition 1.51. To the right, we see a table of all the composites in the category. Whenever two morphisms are not composable—the output object of one doesn't match the input type of the other—the table yells at you. In Haskell, the error message is something like "couldn't match types".

Exercise 1.34. Suppose that someone tells you that their category $\mathcal{C}$ has two objects c, d and two non-identity morphisms, $f : c \to d$ and $g : d \to c$, but no other morphisms. Does f have to be the inverse of g, i.e. is it forced by the category axioms that $g \circ f = \mathrm{id}_c$ and $f \circ g = \mathrm{id}_d$? ◊

Example 1.35 (Monoids). A monoid is like a video game controller: it's a set of "things you can do in sequence". More precisely, we have the following definition.

Definition 1.36. A *monoid* $(M, e, \diamond)$ consists of

1. a set M, called the *carrier set*;
2. an element $e \in M$, called the *unit*; and
3. a function $\diamond : M \times M \to M$, called the *operation*.

These are subject to two conditions:

a. for any $m \in M$, we have $e \diamond m = m$ and $m \diamond e = m$, and
b. for any $l, m, n \in M$, we have $(l \diamond m) \diamond n = l \diamond (m \diamond n)$.

For example, $(\mathbb{N}, 0, +)$ is called the *additive monoid of natural numbers*. The carrier is $\mathbb{N}$, the unit is 0, and the operation is $+$. In this monoid, you can "add numbers in sequence", e.g. $5 + 6 + 2 + 2$.

What's the point? It turns out that a monoid can always be regarded as a *category with one object*[a] If a category has one object, say

[a]The term monoid comes from mono+id, where *mono* means one and *id* means the Freudian self.

Example 1.37. We don't expect you to know Haskell at this point, but here's a preview of how one might define categories and explain to the compiler that monoids are categories:[a]

```
class Category obj mor | mor -> obj where
  dom :: mor -> obj
  cod :: mor -> obj
  idy :: obj -> mor
  cmp :: mor -> mor -> Maybe mor
```

The "Maybe" part is saying that two morphisms may not compose (e.g. $f\colon a \to b$ and $g\colon b' \to c$ only compose if $b = b'$). Note also that there are no laws—associative or unital—so the user of this class has to just certify that these laws really do hold in their specific case. Finally, the weird `| mor -> obj` thing at the top is called a "functional dependency" and is just there to soothe the compiler.

In Haskell, there is already a `monoid` class; what we call the unit it calls `mempty` (like monoid-empty), and it denotes the operation we write as $\diamond$ by `<>`. So now we explain to the compiler that a monoid is a category with one (denoted `()` in Haskell) object:

```
instance Monoid mor => (Category () mor) where
  dom _   = ()
  cod _   = ()
  idy _   = mempty
  cmp m n = Just (m <> n)
```

[a]The code above doesn't quite compile; you need to enable some language extensions first:
```
{-# language FlexibleInstances #-}
{-# language MultiParamTypeClasses #-}
{-# language FunctionalDependencies #-}
```

Exercise 1.38. Implement the category $\mathbf{2} = \boxed{\bullet \to \bullet}$ from Example 1.31 as an instance of the `Category` class from Example 1.37. $\lozenge$

Monoids can be regarded as categories with one object. They're categories that are tiny in terms of objects—just one! Not quite analogously, but similarly, we'll next discuss categories which are tiny in terms of morphisms.

Example 1.39 (Preorders). A *preorder* is a category such that, for every two objects a, b, there is at most one morphism $a \to b$. That is, there either is or is not a morphism from a to b, but there are never two morphisms a to b. If there is a morphism $a \to b$, we write $a \leq b$; if there is not a morphism $a \to b$, we don't.

For example, there is a preorder $\mathcal{P}$ whose objects are the positive integers $\mathrm{Ob}(\mathbf{P}) = \mathbb{N}_{\geq 1}$ and where

$$\mathcal{P}(a, b) := \{x \in \mathbb{N} \mid x * a = b\}$$

This is a preorder because either $\mathcal{P}(a, b)$ is empty (if b is not dividible by a) or contains exactly one element.

Exercise 1.40. Consider the category $\mathcal{P}$.

1. What is the identity on 12?
2. Show that if $x\colon a \to b$ and $y\colon b \to c$ are morphisms, then there is a morphism $y \circ x$ to serve as their composite.
3. Would it have worked just as well to take $\mathcal{P}$ to have all of $\mathbb{N}$ as objects, rather than just the positive integers? $\Diamond$

Example 1.41. In Haskell there are various type classes. A type class is basically a template for named functions of certain types: it's a bunch of key words—each referring to a function of some type—that any member of the class needs to give meanings for.

For example, the type class `Show` is the template {`show`}: any type `T` that wants to be in `Show` needs to say what the word `show` means to them; it needs to say what function `show:: T -> String` should mean. The type class `Eq` is the template for the words `==` and `/=`; any member `T` of that class needs to say what functions `==:: T -> Bool` and `/=:: T -> Bool` should mean.

There is a hierarchy of type classes in Haskell, and a hierarchy is another name for preorder; there is a preorder of type classes in Haskell. Here's a picture of part of it:

$$\begin{tikzcd}
\boxed{\text{Eq}} & \\
\boxed{\text{Ord}} \arrow[u] & \boxed{\text{Num}} \\
\boxed{\text{Real}} \arrow[u] & \boxed{\text{Fractional}} \arrow[ul] \arrow[u]
\end{tikzcd}$$

Our arrows point in the opposite direction from the way most people write it. Our arrows $A \to B$ point in the direction "if something is of type A then it is also of type B". If you prefer the other order, please apply $^{\text{op}}$, defined in Example 1.44, thus answering Exercise 1.45.

Example 1.42 (Free category on a graph). A graph consists of two sets V, E and two functions $\text{src}\colon E \to V$ and $\text{tgt}\colon E \to V$. The elements of V are called vertices, the elements of E are called edges, and for each edge $e \in E$, the vertex $\text{src}(e)$ is called its source and the vertex $\text{tgt}(e)$ is called its target.

One can depict a graph by drawing a dot on for each element $v \in V$ and an arrow for each element $e \in E$, making the arrow point from the source of e to the target of e:

$$\begin{tikzcd}
a \arrow[r, shift left, "1"] \arrow[r, shift right, "2"'] \arrow[d, shift left, "4"] \arrow[dr, "3"'] & b \arrow[loop right, "5"] \\
c \arrow[u, shift left, "6"] & d
\end{tikzcd}$$

Edge	src	tgt
1	a	b
2	a	b
3	b	b
4	a	c
5	b	b
6	c	a

Vertex
a
b
c
d

For any graph G as above, there is an associated category, called the *free category on* G, denoted $\mathbf{Free}(G)$. The objects of $\mathbf{Free}(G)$ are the vertices of G. The morphisms of $\mathbf{Free}(G)$ are the paths in G, i.e. the head-to-tail sequences of edges in G. For each vertex, the trivial path—no edges—counts as a morphism, namely the identity. You can compose paths (just stick them head-to-tail), and this composition is associative.

Exercise 1.43. Is the ordinal category $\mathbf{3}$ (see Example 1.31) the free category on a graph? If so, which graph; if not, why not? ◊

Example 1.44 (Opposite category). For any category $\mathcal{C}$, there is a category $\mathcal{C}^{\text{op}}$ defined by "turning all the arrows around". That is, the two categories have the same objects, and all the arrows simply point the other way:

$$\text{Ob}(\mathcal{C}^{\text{op}}) := \text{Ob}(\mathcal{C}) \qquad \text{and} \qquad \mathcal{C}^{\text{op}}(a,b) := \mathcal{C}(b,a).$$

Exercise 1.45. If $\mathbf{TC}$ is the category shown in Example 1.41, how would you draw $\mathbf{TC}^{\text{op}}$? ◊

Example 1.46 (Product category). Suppose $\mathcal{C}$ and $\mathcal{D}$ are categories. We can form a new category $\mathcal{C} \times \mathcal{D}$ as follows:

$$\text{Ob}(\mathcal{C} \times \mathcal{D}) := \text{Ob}(\mathcal{C}) \times \text{Ob}(\mathcal{D}) \qquad \text{and} \qquad (\mathcal{C} \times \mathcal{D})\big((c,d),(c',d')\big) := \mathcal{C}(c,c') \times \mathcal{D}(d,d').$$

For example, the very Twoish category $\mathbf{2} \times \mathbf{2}$ looks like this:

$$\begin{tikzcd}
\langle 1,1\rangle \arrow[r, "\langle \mathrm{id}_1, f\rangle"] \arrow[d, "\langle f, \mathrm{id}_1\rangle"'] \arrow[dr, "\langle f,f\rangle" description] & \langle 1,2\rangle \arrow[d, "\langle f, \mathrm{id}_2\rangle"] \\
\langle 2,1\rangle \arrow[r, "\langle \mathrm{id}_2, f\rangle"'] & \langle 2,2\rangle
\end{tikzcd}$$

$$\langle \mathrm{id}_2, f\rangle \circ \langle f, \mathrm{id}_1\rangle = \langle f, f\rangle$$
$$\langle f, \mathrm{id}_2\rangle \circ \langle \mathrm{id}_1, f\rangle = \langle f, f\rangle$$

Example 1.47 (Full subcategories). Let $\mathcal{C}$ be any category, and suppose you want "only some of the objects, but all the morphisms between them". That is, you start with a subset of the objects, say $D \subseteq \mathrm{Ob}(\mathcal{C})$, and you want the biggest subcategory of $\mathcal{C}$ containing just those objects; this is called the *full subcategory of* $\mathcal{C}$ *spanned by* D and we denote it $\mathcal{C}_{\mathrm{Ob}=D}$. It's defined by

$$\mathrm{Ob}(\mathcal{C}_{\mathrm{Ob}=D}) \coloneqq D \qquad \text{and} \qquad \mathcal{C}_{\mathrm{Ob}=D}(d_1, d_2) = \mathcal{C}(d_1, d_2).$$

For example, the category of finite sets is the full subcategory of **Set** spanned by the finite sets.

Exercise 1.48. Does the picture shown in Eq. (1.22) look like the full subcategory of **Set** spanned by $\{\underline{0}, \underline{1}, \underline{2}\} \subseteq \mathrm{Ob}(\mathbf{Set})$? Why or why not? ◊

Exercise 1.49. Make an instance of the class **`Category`** from Example 1.37 that defines the full subcategory of Hask spanned by two objects, **`Bool`** and **`Int`**. ◊

1.3.4 Generalized elements: a first peek at the Yoneda perspective

A category is a web of relationships. Let a be an object in a category $\mathcal{C}$. Given any object c in $\mathcal{C}$, we can ask what a looks like from the point of view of c. (We choose c for seer.) The answer to this is encoded by the homset $\mathcal{C}(c, a)$, which is the set of all morphisms from c to a.

Recall from Example 1.13 that elements of a set X are in one-to-one correspondence with functions $1 \to X$. Inspired by this, we often abuse our language to say that 'an element of X' *is* a function $x\colon 1 \to X$. More precisely, we could say that x is 'an element of shape 1'. We then generalize this notion to arrive at the following definition.

Definition 1.50. Let c be an object in a category $\mathcal{C}$. Given an object a, a *generalized element of a of shape c* is a morphism $e\colon c \to a$.

The reader might object: this is just another word for morphism! Nonetheless, we will find it useful to speak and think in these terms, and believe this is enough to justify making such a definition.

This allows us to put slightly more nuance in our mantra that a category is about relationships. Note that, almost by definition, a set is determined by its (generalized) elements (of shape 1). Not all categories have an object that can play the all-seeing role of 1. It is true, however, more democratically: an object in a category is determined by its generalized elements of all shapes.

A nuance is that, as always, we should take into account the role of morphisms. Suppose we have a morphism $f\colon a \to b$. We then may obtain sets of generalized elements $\mathcal{C}(c,a)$ and $\mathcal{C}(c,b)$. But more than this, we also obtain a *function*

$$\begin{aligned} - \mathbin{;} f\colon \mathcal{C}(c,a) &\longrightarrow \mathcal{C}(c,b); \\ x &\longmapsto x \mathbin{;} f. \end{aligned}$$

This function describes how f transforms generalized elements of a into generalized elements of b.

To say more precisely what we mean requires the notion of functor, which we shall get to in the next section. We will return to the Yoneda viewpoint in the next chapter, with a few more tools under our belt. First though, one more lesson in the philosophy of category theory.

Isomorphisms: when are two objects the same?

Let's think about the category of sets for a moment. Suppose we have the set $\underline{2} = \{0,1\}$ and the set $B = \{\text{apple}, \text{pear}\}$. Are they the same set? Of course not! One contains numbers, and the other (names of) fruits! And yet, they have something in common: they both have the same number of elements.

How do we express this in categorical terms? In a category, we don't have the ability to count the number of elements in an object – indeed, objects need not even have elements! We're only allowed to talk of objects, morphisms, identities, and composition. But this is enough to express a critically (and categorically) important notion of sameness: isomorphism.

Morphisms in **Set** are functions, and we can define a function $f\colon \underline{2} \to B$ that sends 0 to apple and 1 to pear. In the reverse direction, we can define a function $g\colon B \to \underline{2}$ sending apple to 0 and pear to 1. These two functions have the special property that, in either order, they compose to the identity: $f \mathbin{;} g = \mathrm{id}_{\underline{2}}$, $g \mathbin{;} f = \mathrm{id}_B$.

This means that we can map from $\underline{2}$ to B, and then B back to $\underline{2}$, and vice versa, without losing any information.

Definition 1.51. Let a and b be objects in a category $\mathcal{C}$. We say that a morphism $f\colon a \to b$ is an *isomorphism* if there exists $g\colon b \to a$ such that $f \mathbin{;} g = \mathrm{id}_a$ and $g \mathbin{;} f = \mathrm{id}_b$. We will call g the *inverse* of f, and sometimes use the notation f^{-1}.

If an isomorphism $f\colon a \to b$ exists, we say that the objects a and b are *isomorphic*.

We will spend a lot of time talking about isomorphisms in the category of sets, so they have a special name.

Definition 1.52. A *bijection* is a function that is an isomorphism in the category **Set**. If there is a bijection between sets X and Y, we say the elements of X and Y are in *one-to-one correspondence*.

Isomorphic objects are considered indistinguishable within category theory. One way to understand why is to remember that category theory is about relationships, and that isomorphic objects relate in the same way to other objects. Let's talk about this in terms of generalized elements.

Fix a shape c. Recall that a morphism $f\colon a \to b$ induces a function from the generalized elements of a to those of b. If f is an isomorphism, then this function is a bijection.

Proposition 1.53. Let $f\colon a \to b$ be an isomorphism. Then for all objects c, we have a bijection

$$\begin{tikzcd} \mathcal{C}(c,a) \arrow[r, shift left, "- \fatsemi f"] & \mathcal{C}(c,b) \arrow[l, shift left, "- \fatsemi f^{-1}"] \end{tikzcd}$$

To see this, consider the following diagram.

$$\begin{tikzcd} & c \arrow[dl, "h"'] \arrow[dr, "h'"] & \\ a \arrow[rr, bend left, "f"] \arrow[loop left, "\mathrm{id}_a"] & & b \arrow[ll, bend left, "f^{-1}"] \arrow[loop right, "\mathrm{id}_b"] \end{tikzcd}$$

Given a generalized element $h\colon c \to a$ of a, $- \fatsemi f$ sends it to $h' = h \fatsemi f$. Conversely, $- \fatsemi f^{-1}$ sends h' to $h' \fatsemi f^{-1} = h \fatsemi f \fatsemi f^{-1} = h$, since f and f^{-1} are inverses.

Exercise 1.54.

1. What is the composite $- \fatsemi f$ followed by $- \fatsemi f^{-1}$?
2. Prove Proposition 1.53.

◊

In other words, Proposition 1.53 that for each shape c, if a and b are isomorphic, their generalized elements are in one-to-one correspondence. In general, morphisms that are not isomorphisms may lose information: they need not induce a bijection on generalized elements. Using the intuition of Section 1.2.3, if f is not an isomorphism, then the induce function $- \fatsemi f$ it may collapse two generalized elements of a, or may not cover the entire set of generalized elements of b.

1.4 Thinking of Haskell as a category

Enough philosophy, let's get to some programming. In the introduction to this chapter, we spoke about how composition is at the core of programming, and saw how we can take two Haskell functions, [2] such as `concat` and `words`, and compose them using the `.` syntax to get a new function. Now equipped with a definition of category, we can see that these functions might be considered as morphisms in some category. But there are questions to be answered. First, what is a Haskell function? And second, if Haskell functions are the morphisms, what are the objects?

In **??** we saw that set-theoretically a function is just a special kind of relation between elements of two sets. In programming, a function is defined by a formula. Such formulas are constructed using rules, defined by the syntax of a language. This brings us to the lambda calculus.

1.4.1 The lambda calculus

Haskell's syntax is based on the lambda calculus. The lambda calculus comes to us from Alonzo Church's work in mathematical logic in the 1930s. It is a neat, compact language for writing down mathematical functions using two primitive notions: lambda abstraction and function application.

We begin with some variable symbols, like x, y, z, and so on. Sentences in the language of the lambda calculus are called *lambda terms*, and these variables are considered lambda terms themselves.

Given a lambda term A and a variable x, lambda abstraction creates a new lambda term $\lambda x.A$. This can be thought of as declaring that any instance of x in the lambda term A should be thought of as a variable; in other words, the new lambda term can be thought of as, in some some, a 'function' of x.

Given two lambda terms A and B, function application creates a new lambda term AB. If we are thinking of A as a function of some variable x, then we think of this as substituting in B wherever we see x. We also include parentheses in the language, to make the order of construction of a term explicit.

For example, here are some lambda terms that you might see:

$$x \qquad \lambda x.x \qquad \lambda x.xy \qquad \lambda x.xx \qquad \lambda x.(\lambda y.x)$$

The rules of the lambda calculus say that we should consider two lambda terms the same if we can turn one into another by this idea of function application. So, for example, we have

$$(\lambda x.x)y = y \qquad (\lambda x.x)(xy) = xy \qquad \text{and} \qquad (\lambda x.(xy)x)(zz) = ((zz)y)(zz).$$

[2]Note that we've defined the word function as a special sort of relationship between sets. Programming also often refers to functions, like `cat` or `words`. These two notions are indeed related, but distinct ideas, and care should be taken not to get them confused. If you're ever confused, it's a good idea to ask what sort of function is being referred to: mathematical or programming?

Notice that the lambda term $\lambda x.x$ behaves like an identity function: given some x, it just returns x. Haskell lets us almost directly employ this same notation; we would write it:

```
id = \x -> x
```

The Greek letter lambda is replaced, in ASCII, with a backslash \ presumably because it looks like the back part of a lambda λ.

Identity is part of the standard Haskell library called **Prelude**. If we allow ourselves to use some other functions from **Prelude**, we can also define functions such as:

```
        square = \x -> x^2
        implies = \x -> \y -> not y || x
```

Haskell note 1.55 (Pattern matching). There is an alternative, and by far more common, syntax for function definition, using Haskell's pattern matching features. In this syntax, we might write the above three functions as:

```
id x = x
square x = x^2
implies y x = not y || x
```

But keep in mind that this is just "syntactic sugar": the compiler converts the pattern matched syntax down to the more basic lambda syntax.

Exercise 1.56 (Church Booleans). Boolean logic may be encoded in the lambda calculus using the following definitions:

$$\begin{aligned}\text{True} &= \lambda x.(\lambda y.x)\\ \text{False} &= \lambda x.(\lambda y.y)\\ \text{AND} &= \lambda p.(\lambda q.(pq)p)\\ \text{OR} &= \lambda p.(\lambda q.(pp)q)\end{aligned}$$

Evaluate the lambda terms True AND False and False OR True. ◊

Exercise 1.57 (The Y combinator). The Y combinator is an iconic lambda term whose reduction does not terminate. It is defined as follows:

$$Y = \lambda f.\big((\lambda x.f(xx))(\lambda x.f(xx))\big)$$

Compute Yg. ◊

1.4.2 Types

Untyped lambda calculus is very expressive. In fact it's Turing complete, which means that any program that can be run on a Turing machine can be also expressed in lambda calculus. A Turing machine is an idealized computer that has access to an infinite tape from which it can read, and to which it can output data. And just like a Turing machine is totally impractical from the programmer's point of view, so is lambda calculus. For instance, it's true that you can encode Boolean logic and natural numbers using lambda expressions–this is called Church encoding–but working with this encoding is impossibly tedious and error prone, not to mention inefficient. And because everything is a function in lambda calculus, it's okay to apply a number to a function and get meaningless result. Imagine debugging programs written in lambda calculus! [3]

Adding types to lambda calculus means that functions can't be composed willy-nilly: the target type of one must match the source type of the next one. This immediately suggests a category in which objects are types and morphisms are functions.

More properly then, we should say Haskell's syntax is based on the *simply-typed lambda calculus*. Every Haskell term is required to have a type. If a term `x` has type `A`, we write:

```
x :: A
```

In fact, in GHCi, you can ask what a term's type is, using the command `:t` or `:type`. For example:

```
Prelude>:t "hi"
"hi" :: [Char]
```

Exercise 1.58. Fire up GHCi. What are the types of the following terms:

1. `42`
2. `"cat"`
3. `True`

◊

Exercise 1.59. Recall the Y combinator from Exercise 1.57. Try to assign the Y combinator a type. What goes wrong? ◊

In Haskell we have some built-in types, like `Integer`, which is an arbitrary precision integer, and `Int`, which is its fixed size companion (with present implementations, usually a 64-bit integer (that is, an integer from -2^{63} to $2^{63} - 1$)). In practice, a fixed-size representation is preferred for efficiency reasons, unless you are worried about

[3]There is also the issue of the Kleene-Rosser paradox (or its simplified version called the Curry's paradox), which shows that untyped lambda calculus is inconsistent; but a little inconsistency never stopped a programming language from being widely accepted.

overflow (e.g., when calculating large factorials). A lot of common types, rather than being built-in, are defined in the **Prelude**. Such types include strings **String** and Booleans, **Bool**. Much of this course is about how to construct new types from existing types, and categorical ideas like universal constructions help immensely.

Since every Haskell term is required to have a type, sometimes it's your responsibility to tell the Haskell compiler what type your term is. This is done by simply by declaring it, using the same syntax as above. So to tell Haskell compiler you have a variable `x` of type **A**, we write:

```
x :: A
```

Note that it's not necessary to declare the type of absolutely every line of code: the Haskell compiler has a powerful type inference system within it, and will be able to work with a bare minimum of type declarations.

Haskell note 1.60. In Haskell, the names of concrete types start with an upper case letter, e.g. `Int`. The names type variables, or type parameters, like `a` here, start with lowercase letters.

Notice that we defined the identity function `id` without giving a type signature. In fact our definition works for any arbitrary type. We call such a function a *polymorphic* function. We'll talk about polymorphism in more detail in **??**. For now, it's enough to know that a polymorphic function is defined for all types. (This is the troubling self-referential aspect of polymorphism: "all types" includes the type we are just defining. For instance, `id` can be instantiated for the type `a = b -> b` and so on.)

It's possible, although not necessary, to use the universal quantifier `forall` in the definition of *polymorphic functions*:

```
id :: forall a. a -> a
```

This latter syntax, however, requires the use of the language pragma **ExplicitForAll** (we'll explain this later).[4]

Remark 1.61. Types are very similar to sets, and it frequently will be useful to think of the type declaration `x :: A` as analogous to the set theoretic statement $x \in A$; ie. that x is an element of the set A. There are some key differences though, which the category theoretic perspective helps us keep straight. The main difference is that in the world of sets, sets have elements. So given a set A, it's a fair question to ask what its elements are – that's exactly what A is, a bag of dots. Types and terms are the other way around: terms have types. So you can always ask what the type of a term is, but you can't ask for all the terms of a given type.

[4]Note that the period after the `forall` clause is just a separator and has nothing to do with function composition, which we'll meet shortly.

1.4.3 Haskell functions

In category theory, we write $f : a \to b$ to mean f is a morphism from a to b. In other words, we are implicitly working in some category $\mathcal{C}$, and f is an element of the homset:

$$f \in \mathcal{C}(a, b)$$

In Haskell, we can also declare that terms are (Haskell) functions, using an almost identical syntax. Given two Haskell terms `A` and `B`, there is a *type of functions from* `A` *to* `B` `A -> B` whose terms are (Haskell) functions accepting an `A` and producing a `B`. So for example, we have

```
>:t not
not :: Bool -> Bool
```

We call the type `A -> B` the *type of functions*. If you're worried that we are mixing types, which are objects in our category, with functions between them, which are morphisms, this will become clear when we talk about exponentials.

When defining a mathematical function, it's necessary to first specify the domain and codomain. For example, we might want to define a function f that squares a natural number, so we write $f : \mathbb{N} \to \mathbb{N}$. We might then define the function itself: $f(n) = n^2$. Similarly, when defining a Haskell function, it is customary to first declare its type; this is often referred to as its *type signature*. So, for example, to define the function `square`, we might first give the type signature

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

A type signature must then be accompanied by an *implementation*. We have to tell the computer how to evaluate a function. We have to write some code that will be executed when the function is called; for example

```
square x = x^2
```

Remark 1.62 (A note on Haskell functions vs mathematical functions). "Functions" in programming are not just simple functions between sets. A "function" may loop forever, be it because it enters an infinite loop or because of recursion. So maybe we don't need recursion? Maybe we could define a programming language in which the execution of every function is guaranteed to terminate? One way to do this would be to ban recursion altogether: no function could call itself (or, more precisely, the call graph cannot contain cycles). Unfortunately, most algorithms and data structures used in programming are defined recursively (or, equivalently, using looping constructs).

In computer science we say that a language without recursion or some kind of looping construct is not Turing complete. We could ask if it's possible to somehow limit recursion and allow only recursive definitions that are bound to terminate. Often it's possible to show that a recursive step reduces the problem to a simpler one, until it reaches some kind of a bottom (e.g., counting down from n to zero is bound to terminate for any n). Unfortunately, it's also known that termination is undecidable. There is no algorithm that could decide whether an arbitrary program will terminate or not.

For these reasons, general-purpose programming languages allow unlimited recursion and non-termination. There are ways of dealing with this problem within set theory. Recursion can be described in domain theory, which is based on partially ordered sets. Later, we'll talk about polymorphism, which also stretches the boundaries of set theory by introducing self-referential types. Again, there are ways of dealing with it in set theory, but they are not easy. Category theory provides a much simpler high-level setting for interpreting programs.

1.4.4 Composing functions

What do we do with functions? Compose them! As function composition is such a basic operation, in Haskell with give it almost the simplest name: `.`. Function composition in Haskell is written in *application order*. This means that the composite of a function `f:: a -> b` and a function `g:: b -> c` is `g . f :: a -> c`.

Haskell note 1.63. Note that we've written composition as an *infix* operator. This is an operator of two arguments that is written between the two arguments. Another example is addition: we write `4 + 5` to add two numbers.

To use an infix operator as an *outfix* operator – that is, a binary operator that is written before its arguments – we place it in parentheses. So `4 + 5` may also be written `(+) 4 5`.

Composition is itself a Haskell function. What is its type signature? The Haskell definition takes advantage of *currying*. We'll talk much more about this in Chapter 3, but in this case, it's a trick that allows us to consider a function of two arguments as a function of the first argument that returns a function of the second argument. This trick gives the type signature:

```
(.) :: (b -> c) -> ((a -> b) -> (a -> c))
```

The function type symbol `->` by default associates to the right, so this is equal to the type

```
(.) :: (b -> c) -> (a -> b) -> a -> c
```

Note that the rest of the parentheses are essential.

Here's the implementation. Given two functions `f :: b -> c` and `g :: a -> b`, we produce a third function defined as a lambda. The only thing we can do with the functions is to apply them to arguments, and that's what we do:

```
(.) f g = \x -> f (g x)
```

The result of calling `g` with the argument `x` is being passed to `f`. Notice the parentheses around `g x`. Without them, `f g x` would be parsed as: `f` is a function of two arguments being called with `g` and `x`. This is because function application is left associative. All this requires some getting used to, but the compiler will flag most inconsistencies.

Composition is just a function with a funny name, `(.)`. You can form a function name using symbols, rather than more traditional strings of letters and digits, by putting parentheses around them. You can then use these symbols, without parentheses, in infix notation. So, in particular, the above can be rewritten:

```
f . g = \x -> f (g x)
```

or, using the syntactic sugar for function definition, simply as:

```
(f . g) x = f (g x)
```

Composition, just like identity before, is a fully polymorphic function, as witnessed by lowercase type arguments. It could be written as:

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

Exercise 1.64. Suppose $f\colon \texttt{Int} \to \texttt{Int}$ sends an integer to its square, $f(x) := x^2$ and that $g\colon \texttt{Int} \to \texttt{Int}$ sends an integer to its successor, $g(x) := x + 1$.

1. Write f and g in Haskell, including their type signature and their implementation.
2. Let $h := f \circ g$. What is $h(2)$?
3. Let $i := f \mathbin{;} g$. What is $i(2)$? ◊

Defining new morphisms by composing existing ones is used in Haskell in point-free style of programming. Here's an example, similar to the one in the introduction.

Example 1.65. Let's call the function `words` with a string `"Hello world"`. Here's how it's done:

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

The result is a comma-separated list of words with spaces removed.

Now let's call the function `length` to calculate the length of the list of strings from the previous call:[a]

```
Prelude> length ["Hello","world"]
2
```

We can compose the two functions using the composition operator, which is just a period "." between the two functions. Let's apply it to the original string:

```
Prelude> (length . words) "Hello world"
2
```

[a]Note that writing "concat it" also works in GHCI.

Haskell note 1.66 (Binding and parentheses). Notice the use of parentheses. Without them, the line:

```
length . words "Hello world"
```

would result in an error, because function application binds stronger than the composition operator. In effect, this would be equivalent to:

```
length . (words "Hello world")
```

The compiler would complain with a somewhat cryptic error message. This is because the composition operator expects a function as a second argument, and what we are passing it is a list of strings.

Function application is the most common operation in Haskell, so its syntax is reduced to the absolute minimum. In most cases a space between two symbols is enough.

We can define new functions by composing existing functions:

```
Prelude> let wordCount = length . words
Prelude> wordCount "Hello Haskell!"
2
```

Haskell note 1.67 (let). In the interactive environment, all definitions must be prefixed with `let`. In a standalone file, `let` is only used in local definitions, inside a function body. The difference in syntax is there because all input and output must be executed in the `IO` monad (which we'll discuss later). The command line interface must therefore use monadic syntax, which is normally seen in `do` blocks.

Exercise 1.68. Recall the notion of an isomorphism from Definition 1.51. Here's an example of an isomorphism in Haskell (don't worry about the syntax, it will become clear later) between two types `Coin` and `Bool`. It consists of two function, one being the inverse of the other.

```
data Coin = Head | Tails

decide :: Coin -> Bool
decide Tail = False
decide Head = True

unDecide :: Bool -> Coin
unDecide True = Head
unDecide False = Tail
```

You can easily convince yourself by case analysis that `unDecide` is the inverse of `decide`, but this fact is not expressible in Haskell. There are programming languages like Idris or Agda, which use dependent type systems, and which treat proofs of equality as first order constructs that can be passed between functions or stored in data structures.

Identity is also an example of an isomorphism. Here's an example of a non-identity isomorphism that maps Boolean to Boolean by inverting the meaning of **`True`** and **`False`**:

```
not :: Bool -> Bool
not True = False
not False = True
```

Explain why these two functions are isomorphisms, and why **`Coin`** and **`Bool`** are isomorphic. ◊