2

FROM SETS TO CATEGORIES

In this chapter we will see some more set-theoretic constructs, but we will also introduce their category-theoretic counterparts in an effort to gently introduce the concept of a category itself.

When we are finished with that, we will try, (and almost succeed) to define categories from scratch, without actually relying on set theory.

Products

In the previous chapter there were several places where needed a way to construct a set whose elements are *composite* of the elements of some other sets: when we discussed mathematical functions, we couldn't define $+$ and $-$ because we could only formulate functions that take one argument. Then, when we introduced the primitive types in programming languages, like `Char` and `Number`, we mentioned that most of the types that we actually use are *composite* types. So how do we construct those?

The simplest composite type, of the sets B, that contains b's and the set Y, that contains y's is the *Cartesian product* of B and Y, that is the set of *ordered pairs* that contain one element of the set Y and one element of the set B. Or formally speaking: $Y \times B = \{(y, b)\}$ where $y \in Y, b \in B$ ($\in$ means "is an element of").

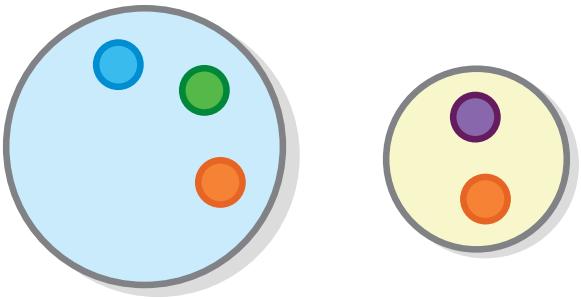

Figure 2-1: Product parts

It is denoted $B \times Y$ and it comes equipped with two functions for retrieving the b and the y from each (b, y).

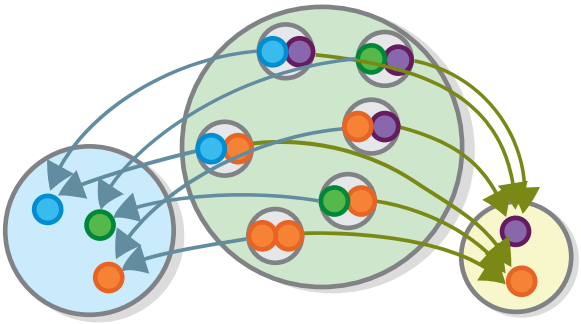

Figure 2-2: Product

Question: Why is this called a product? Hint: How many elements does it have?

{% if site.distribution == 'print'%}

Interlude - coordinate systems

The concept of the Cartesian product was first defined by the mathematician and philosopher René Descartes as a basis for the *Cartesian coordinate system*. Although it does not look like it, both concepts are named after him (or after the Latinized version of his name.)

You probably know how the Cartesian coordinate system works, but an equally interesting question, of which you probably haven't thought about, is how we can define it using sets and functions.

An Cartesian coordinate system consists of two perpendicular lines, situated on an *Euclidian plane* and some kind of mapping that resembles a function, connecting any point in these two lines to a number, representing the distance between the point that is being mapped and the lines' point of overlap (which is mapped to the number 0).

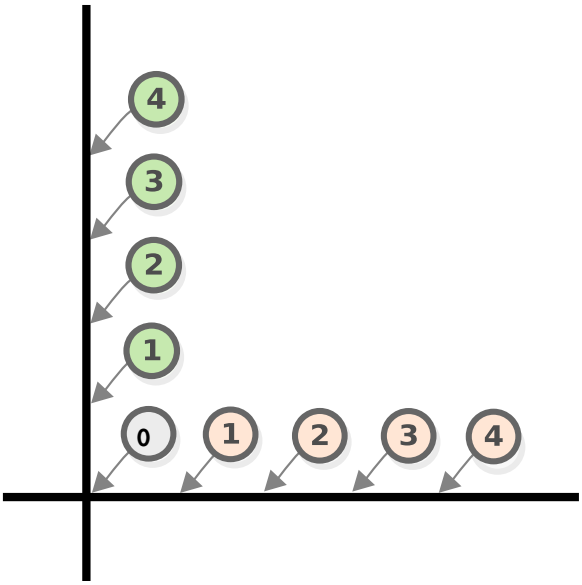

Figure 2-3: Cartesian coordinates

Using this construct (as well as the concept of a Cartesian product), we can describe not only the points on the lines, but any point on the Euclidian plane. We do that by measuring the distance between the point and those two lines.

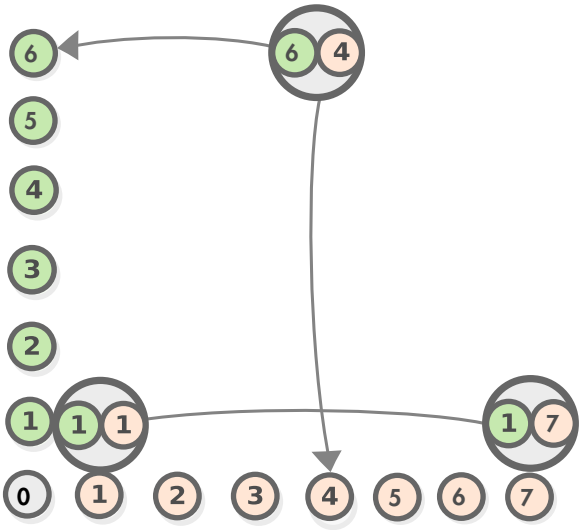

Figure 2-4: Cartesian coordinates

And since the point is the main primitive of Euclidian geometry, the coordinate system allows us to also describe all kinds of geometric figures such as this triangle (which is described using products of products.)

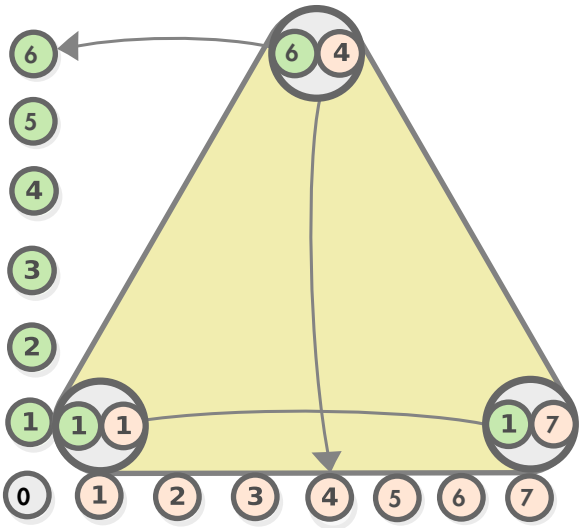

Figure 2-5: Cartesian coordinates

So we can say that the Cartesian coordinate system is some kind of function-like mapping between all kinds of sets of (products of) *products of numbers* and *geometric figures* that correspond to these numbers, using which we can derive some properties of the figures using the numbers (for example, using the products in the example below, we can compute that the triangle that they represent has base of 6 units and height of 5 units.

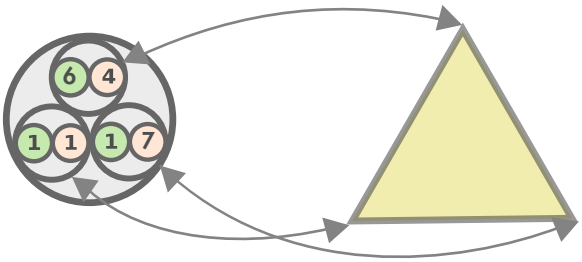

Figure 2-6: Cartesian coordinates

What's even more interesting is that this mapping is one-to-one, which makes the two realms *isomorphic* (traditionally we say that the point is *completely* described by the coordinates, which is the same thing.)

Our effort to represent Cartesian coordinates with sets is satisfactory, but incomplete, as we still don't know what these function-like things that connect points to numbers are - they make intuitive sense as functions, and that they exhibit all relevant properties (many-to-one mapping, or one-to-one in this case), however, we have only covered functions as mappings between sets and in this case, even if we can think of the coordinate system as a set (of points and figures), geometrical figures are definitely not a set, as it has a lot of additional things going on (or additional *structure*, as a category theorist would say.)

So defining this mapping formally, would require us to also formalize both geometry and algebra, and moreover to do so in a way in which they are compatible with one another. This is some of the ambitions of category theory and this is what we will attempt to do later in this book (even if not for this exact example.)

But before we continue with that, let's see some other neat uses of products.

{%endif%}

Products as Objects

In the previous chapter we established the correspondence of various concepts in programming languages and set theory sets resemble types, functions resemble methods/subroutines. This picture is made complete with products, that are like stripped-down *classes* (also called *records* or *structs*) - the sets that form the product correspond to the class's *properties* (also called *members*) and the functions for accessing them are like what programmers call *getter methods*.

The famous example of object-oriented programming of a `Person` class with `name` and `age` fields is nothing more than a product of the set of strings, and the sets of numbers. Objects with more than two values can be expressed as products the composites of which are themselves products.

Using Products to Define Numeric Operations

Products can also be used for expressing functions that take more than one argument. For example, "plus" and "minus", are functions from the set of products of two numbers to the set of numbers. (So, $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$.) Of course, we cannot draw the function itself, even partly, because it has too much arrows and would look messy.

Actually, here it is.

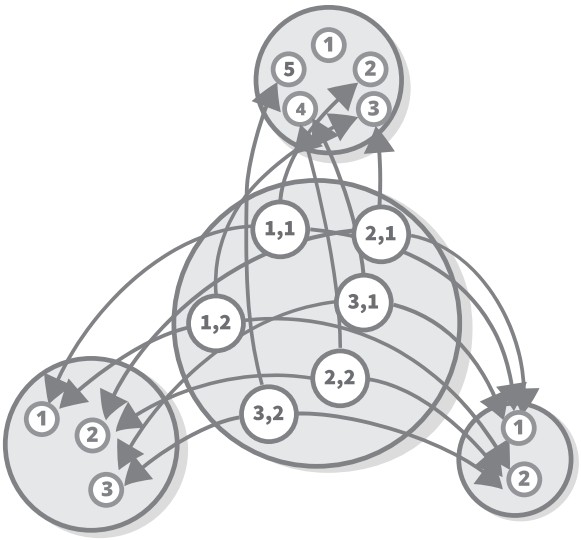

Figure 2-7: The plus function

Note that there are languages, such as the ones from the ML family, where the *pair* data structure (also called a *tuple*) is a first-level construct, and multi-argument functions are really implemented in this way.

Defining products in terms of sets

When we said that the product is a set of *ordered* pairs (formally speaking $A \times B \neq B \times A$). But we didn't define how ordered pairs formally. Note that the criteria for order prevents us from symbolizing the pair with just a set containing the two elements, as while some mathematical operations (such as addition) indeed don't care about order, others (such as subtraction) do. And in programming, we have the ability to assign names to each member of an object, which accomplishes the same purpose as ordering does for pairs.

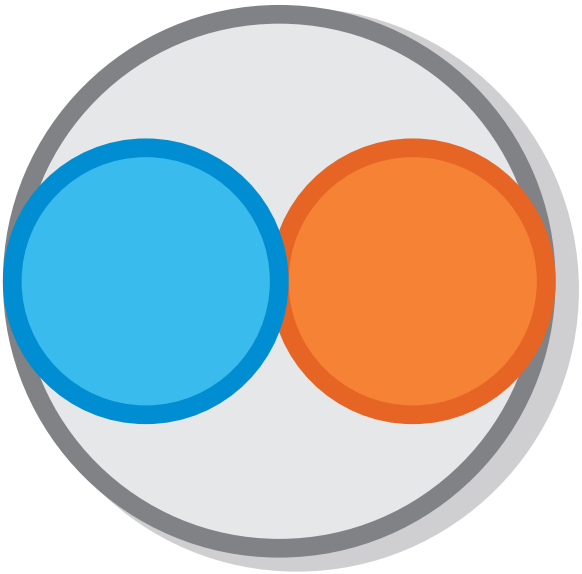

Figure 2-8: A pair

So does that mean that we have to define ordered pair as a "primitive" type, like we defined sets in order to use them? That's possible, but there is another approach if we can define a construct that is *isomorphic* to the ordered pair, using only sets, we can use that construct instead of them. And mathematicians had come up with multiple ingenious ways to do that. Here is the first one, which was discovered by Norbert Wiener in 1914. Note the smart use of the fact that the empty set is unique.

Figure 2-9: A pair, represented by sets

The next one was discovered in the same year by Felix Hausdorff. In order to use that one, we just have to define 1, and 2 first.

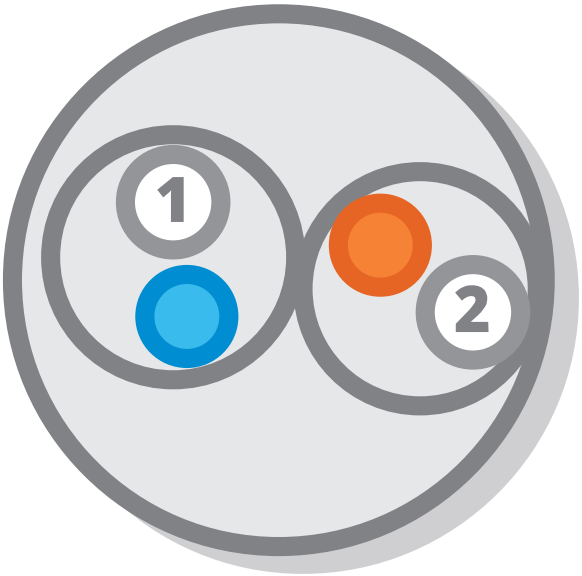

Figure 2-10: A pair, represented by sets

Discovered in 1921 Kazimierz Kuratowski, this one uses just the component of the pair.

Figure 2-11: A pair, represented by sets

Defining products in terms of functions

In the product definitions presented in the previous section worked by *zooming in* into the individual elements of the product and seeing what they can be made of. I call this the *low-level* approach. This time we will try to do the opposite - be as oblivious to the contents of our sets as possible i.e. instead of zooming in we will *zoom out*, and try to define the product in terms of functions and functional composition. Effectively we will be working at a *higher level* of abstraction.

How can we define products in terms of functions? Let's begin with an external diagram, showing the definition of the product.

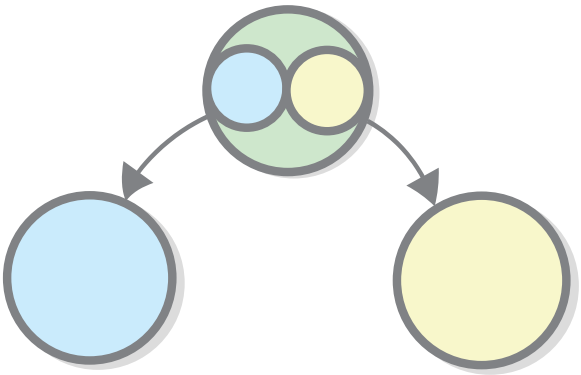

Figure 2-12: Product, external diagram

This diagram already contains the first piece of the puzzle: if we have a set G which is the product of sets Y and B, then we should also have functions which give us back the elements of the product, so $G \to Y$ and $G \to B$.

This definition is not complete, however, because the product of G and B is not the only set for which such functions can be defined. For example, a set of triples of $Y \times B \times R$ for any random element R also qualifies. And if there is a function from G to B then the set G itself meets our condition for being the product, because it is connected to B and to itself. And there can be many other such objects.

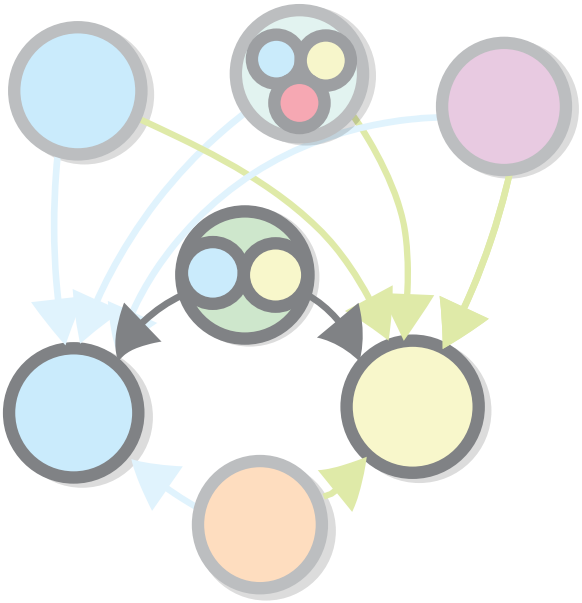

Figure 2-13: Product, external diagram

So how do we set apart the true product from all those "impostor" products? Simple - by using the observation that *they all can be converted to it*, This observation is true, because. The pair is nothing more than the sum of its elements. And you can always have a function that converts a more complex structure, to a simpler one (we saw an example of this when we covered the functions that convert subsets to their supersets).

More formally, if we suppose that there is a set I that can serve as an impostor product of sets B and Y i.e. that I is such that there exist two functions, which we will call $b : I \to B$ and $y : I \to Y$ that allow us to derive elements B and Y from it, then there must also exist a function with the type signature $I \to B \times Y$ that converts the impostor from the true product. We can be sure that this function exists because I (being an impostor) would contain some extra information other than the information contained in the true pair. So given we have functions $b : I \to B$ and $y : I \to Y$ that function would be $(i) \to b(i) \times y(i)$ for each element $i : I$.

Therefore, we can define the product of B and Y as a set that has functions for deriving B and Y, but, more importantly, all other sets that have such functions can be converted to it. The second requirement would mean that

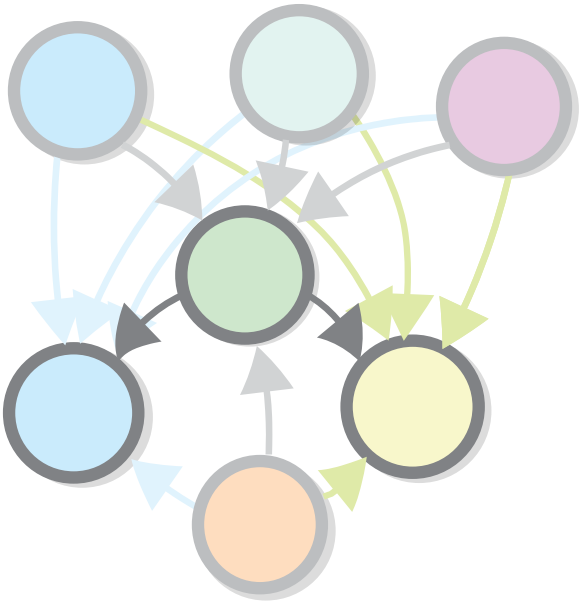

Figure 2-14: Product, external diagram

In category theory, this type of property that a given object might possess (participating in a structure such that all similar objects can be converted to/from it) is called a *universal property*. I don't want to go into more detail, as it is a bit early for that now (after all we haven't even defined what category theory is). One thing that I like to point out is that this definition (as, by the way, all the previous ones) does not rule out the sets which are isomorphic to the product - when we represents things using universal properties, an isomorphism is the same as equality.

Sums

We will now study a construct that is pretty similar to the product but at the same time is very different. Similar because, like the product, it is a relation between two sets which allows you to unite them into one, without erasing their structure. But different as it encodes a quite different type of relation - a product encodes an *and* relation between two sets, while the sum encodes an *or* relation.

A sum of two sets B and Y, denoted $B+Y$ is a set that contains *all elements from the first set combined with all elements from the second one*.

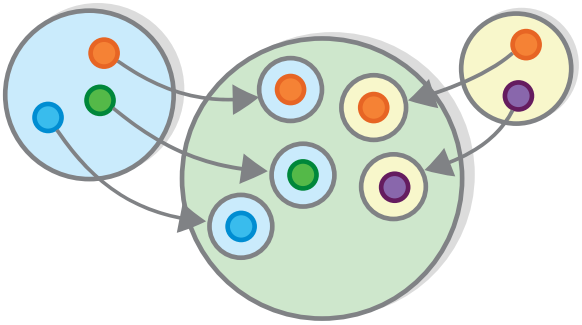

Figure 2-15: Sum or coproduct

We can immediately see the connection with the *or* logical structure: For example, because a parent is either a mother or a father of a child, the set of all parents is the sum of the set of mothers and the set of fathers, or $P = M + F$.

Defining Sums in Terms of Sets

As with the product, representing sums in terms of sets is not so straightforward. This time the complication comes from the fact that when a given object is an element of both sets, then it appears in the sum twice. This is why this type of sum of two sets is also called a *disjoint union*. Because of this, if two sets can have the same element as a member, then their sum will have that element twice which is not permitted, because a set cannot contain the same element twice. As with the product, the solution is to put some extra structure.

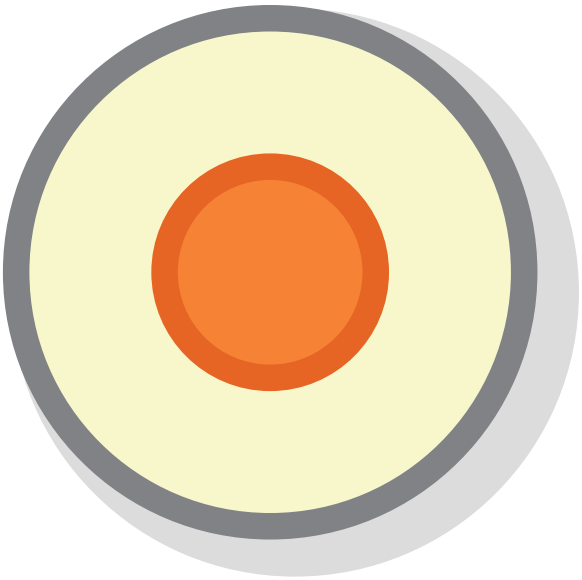

Figure 2-16: A member of a coproduct

And as with the product, there is a low-level way to express a sum using sets alone. Incidentally, we can use pairs.

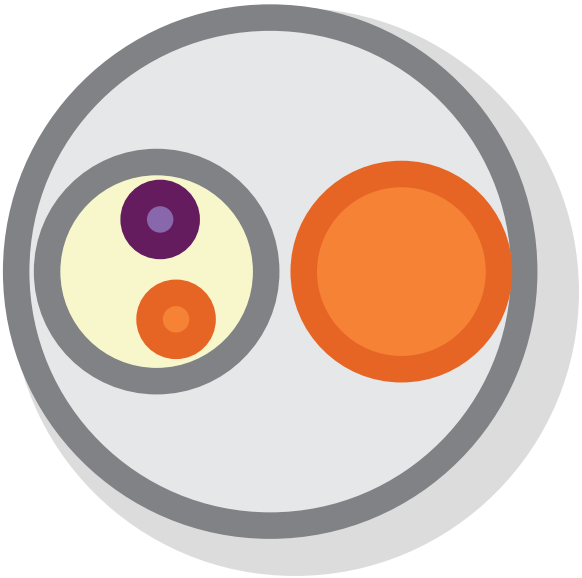

Figure 2-17: A member of a coproduct, examined

But again, this distinction is only relevant only when the two sets have common elements. If they don't then just uniting the two sets is sufficient to represent their sum.

Defining sums in terms of functions

As you might already suspect, the interesting part is expressing the sum of two sets using functions. To do that we have to go back to the conceptual part of the definition. We said that sums express an *or* relation between two things.

A property of every *or* relation is that if something is an A that something is also an $A \vee B$, and same for B (The $\vee$ symbol means *or* by the way). For example, if my hair is *brown*, then my hair is also *either blond or brown*. This is what *or* means, right? This property can be expressed as a function, two functions actually - one for each set that takes part in the sum relation (for example, if parents are either mothers or fathers, then there surely exist functions $mothers \to parents$ and $fathers \to parents$.)

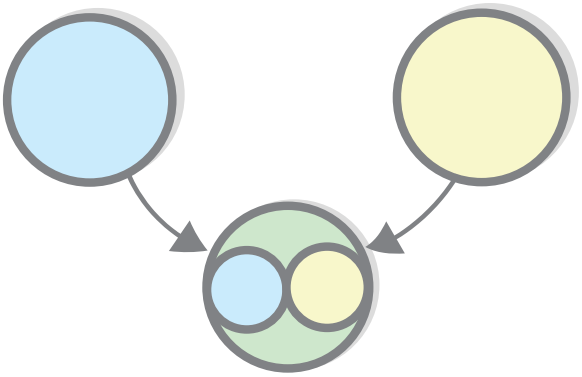

Figure 2-18: Coproduct, external diagram

As you might have already noticed, this definition is pretty similar to the definition of the product from the previous section. And the similarities don't end here. As with products, we have sets that can be thought of as *impostor* sums - ones for which these functions exist, but which also contain additional information.

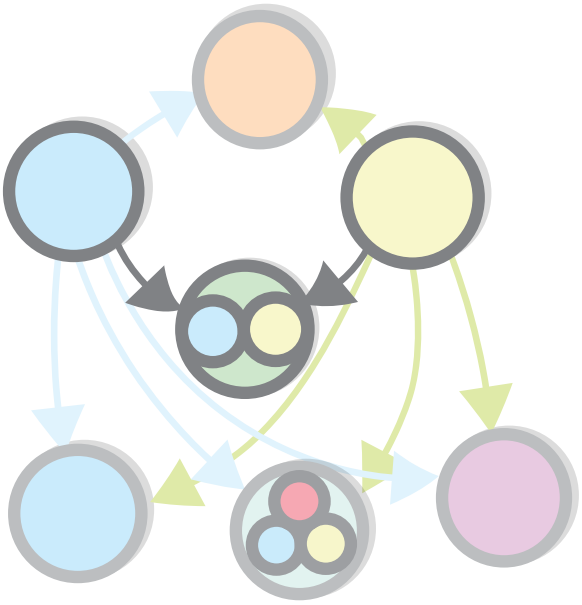

Figure 2-19: Coproduct, external diagram

All these sets express relationships which are more vague than the simple sum, and therefore given such a set (an "impostor" set as we called it earlier), there would exist a function that would distinguish it from the true sum. The only difference is that, unlike with the products, this time this function goes *from the sum* to the impostor.

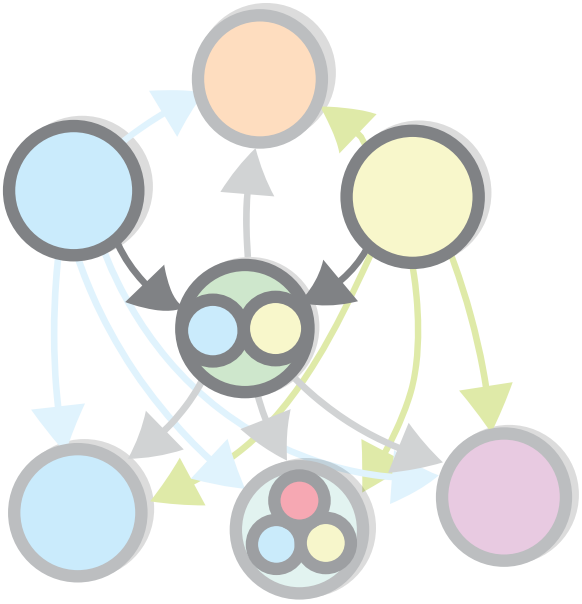

Figure 2-20: Coproduct, external diagram

Categorical Duality

The concepts of product and sum might already look similar in a way when we view them through their internal diagrams, but once we zoom out to the external view, and we draw the two concepts external diagrams, this similarity is quickly made precise.

I use "diagrams" in plural, but actually the two concepts are captured *by one and the same diagram*, the only difference between the two being that their arrows are flipped - many-to-one relationships become one-to-many and the other way around.

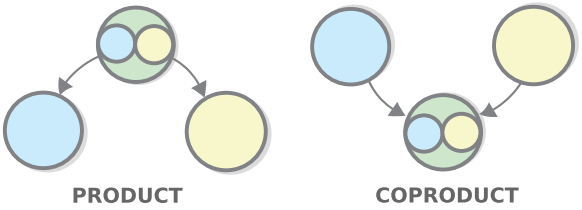

Figure 2-21: Coproduct and product

The universal properties that define the two construct are the same as well - if we have a sum $Y + B$, for each impostor sum, such as $Y + B + R$, there exist a trivial function $Y + B \to Y + B + R$.

And, if you remember, with product the arrows go the other way around - the equivalent example for product would be the function $Y ×B ×R Y ×B $

This fact uncovers a deep connection between the concepts of the *product* and *sum*, which is not otherwise apparent - they are each other's opposites - *product* is the opposite of *sum* and *sum* is the opposite of *product*.

In category theory, concepts that have such a relationship are said to be *dual* to each other. So the the concepts of *product* and *sum* are dual. This is why sum is known in a category-theoretic setting as *converse product*, or *coproduct* for short. This naming convention is used for all dual constructs in category theory.

{% if site.distribution == 'print'%}

Interlude - De Morgan duality

Now let's look at how the concepts of product and sum from the viewpoint of *logic*. We mentioned that:

- The *product* of two sets contains an element of the first one *and* one element of the second one.
- A *sum* of two sets contains an element of the first one *or* one element of the second one.

When we view those sets as propositions, we discover the concept of the *product* ($\times$) corresponds exactly to the *and* relation in logic (denoted $\wedge$.) From this viewpoint, the function $Y \times B \rightarrow Y$ can be viewed as instance of a logical rule of inference called *conjunction elimination* (also called *simplification*) stating that, $Y \wedge B \rightarrow Y$, for example, if your hair is partly blond and partly brown, then it is partly blond.

By the same token, the concept of a *sum* ($+$) corresponds the *or* relation in logic (denoted $\vee$.) From this viewpoint, the function $Y \rightarrow Y + B$ can be viewed as instance of a logical rule of inference called *disjunction introduction*, stating that, $Y \rightarrow Y \vee B$ for example, if your hair is blond, it is either blond or brown.

This means, among other things, that the concepts of *and* and *or* are also dual - an idea which was put forward in the 19th century by the mathematician Augustus De Morgan and is henceforth known as *De Morgan duality*, and which is a predecessor to the modern idea of duality in category theory, that we examined before.

This duality is subtly encoded in the logical symbols for *and* and *or* ($\wedge$ and $\vee$) - they are nothing but stylized-versions of the diagrams of products and coproducts.

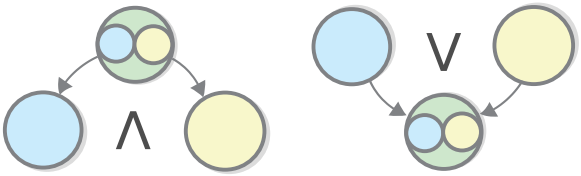

Figure 2-22: Coproduct and product

To understand, the connection, consider the two formulas which are most often associated with De Morgan which are known as De Morgan laws, although De Morgan didn't actually discover those (they were previously formulated, by William of Ockham (of "Ockham's razor" fame) among other people.

$\neg(A \wedge B) = \neg A \vee \neg B$

$\neg(A \vee B) = \neg A \wedge \neg B$

You can read the second formula as, for example, if my hair is not blond *or* brown, this means that my hair is not blond *and* my hair is not brown, and vice versa (the connection work both ways)

Now we will go through the formulas and we would try to show that they are actually a simple corollary of the duality between *and* and *or*

Let's say we want to find the statement that is opposite of "blond *or* brown".

$A \vee B$

The first thing we want to do is, to replace the statements that constitute it with their opposites, which would make the statement "not blond *or* not brown"

$\neg A \vee \neg B$

However, this statement is clearly not the opposite of "blond *or* brown"(saying that my hair is not blond *or* not brown does in fact allow it to be blond and also allows for it to be brown, it just doesn't allow it to be both of these things.)

So what have we missed? Simple - although we replaced the propositions that constitute our proposition with their opposites, we didn't replace the operator that connects them - it is still *or* for both propositions. So we must replace it with *or* converse. As we said earlier, and as you can see by analyzing this example, this operator is *and* So the formula becomes "not blond *and* not brown".

$\neg A \wedge \neg B$

Saying that this formula is the opposite or "blond and brown" - is the same thing as saying that it is equivalent to it's negation, which is precisely what the second De Morgan formula says.

$\neg(A \vee B) = \neg A \wedge \neg B$

And if we "flip" this whole formula (we can do that without changing the signs of the individual propositions, as it is valid for all propositions) we get the first formula.

$\neg(A \wedge B) = \neg A \vee \neg B$

This probably provokes a lot of questions, but I won't get into more detail here, as we have a whole chapter on logic. But before we get to it, we have to see what categories are.

{% endif %}

Category Theory - brief definition

Maybe it is about time to see what a category is. We will start with a short definition - a category consists of objects (an example of which are sets) and morphisms that go from one object to another (which can be viewed as func-

tions) and that should be composable. We can say a lot more about categories, and even present a formal definition, but for now it is sufficient for you to remember that sets are one example of a category and that categorical objects are like sets, except that we don't *see* their elements. Or to put it another way, category-theoretic notions are captured by the external diagrams, while strictly set-theoretic notions can be captured by internal ones.

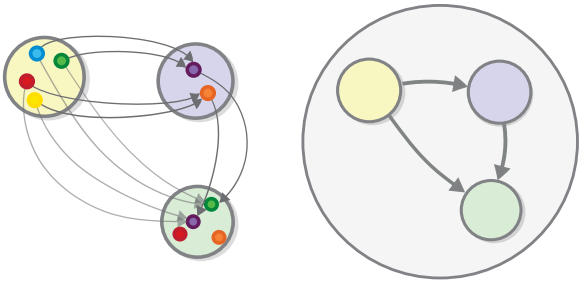

Figure 2-23: Category theory and set theory compared

When we are at the realm of sets we can view the set as a collection of individual elements. In category theory we don't have such notion, but we saw how taking this notion away allows us to define concepts such as the sum and product sets in a whole different and more general way.

Still why would we want to restrict ourselves from looking at the individual elements? It is because, in this way we can relate this viewpoint to objects other than sets. We already discussed one such object - types in programming languages. Remember that we said that programming types (classes) are somewhat similar to sets, and programming methods are somewhat similar to functions between sets, but they are not exactly identical? A formal connection between the two can be made via category theory.

Category Theory	Set theory	Programming Languages
Category	N/A	N/A
Objects and Morphisms	Sets and Functions	Classes and methods
N/A	Element	Object

Category theory allows us to see the big picture when it comes to sets and similar structures - looking at the table, we cannot help but notice the somehow weird, (but actually completely logical) symmetry (or perhaps "reverse symmetry") between the world as viewed through the lenses of set theory, and the way it is viewed through the lens of category theory:

Category Theory	Set theory
Category	**N/A**
Objects and Morphisms	Sets and functions
N/A	Element

By switching to external diagrams, we lose sight of the particular (the elements of our sets), but we have gained the ability to see the whole universe that we have been previously trapped in. Just as the whole realm of sets can be thought as one category, a programming language can also be thought as a category. The concept of a category allows us to find and analyze similarities between these and other structures.

NB: The word "Object" is used in both programming languages and in category theory, but for completely different things. The equivalent a categorical object is equivalent to a *type* or a *class* in programming language theory.

Sets VS Categories

One remark before we go: in the last paragraphs I sound as if I'm *comparing* categories and sets (and rooting for categories, in order to get more copies of my book sold) and I don't want you to get the wrong impression that the two concepts are somehow competing with one another. Perhaps that notion would be somewhat correct if category and set theory were meant to describe *concrete* phenomena, in the way that the theory of relativity and the theory of quantum mechanics in physics. Concrete theories are conceived mainly as *descriptions* of the world, and as such it makes sense for them to be connected to one another in some sort of hierarchy. Abstract theories, like category theory and set theory, on the other hand, are more like languages for expressing such descriptions - they still can be connected, and are connected in more than one way, but there is no inherent hierarchy between the two and therefore arguing over which of the two is more basic, or more general, is just a chicken-and-egg problem, as you would see in the next chapter.

Defining Categories (again)

All category theory books (including this one) starts by talking about set theory. However looking back I really don't know why that is the case - most books that focus around a given subject don't usually start off by introducing an *entirely different subject* before even starting to talk about the main one, even if the two subjects are so related.

Perhaps the set-first approach *is* the best way to introduce people to categories. Or perhaps using sets to introduce categories is just one of those things that people do because everyone else does it. But one thing is for certain - we don't need to study sets in order to understand categories. So now I would like to start over and talk about categories as a first concept. So pretend like it's a new book (I wonder if I can dedicate this to a different person.)

So. A category is a collection of objects (things) where the "things" can be anything you want. Consider, for example, these colorful gray balls:

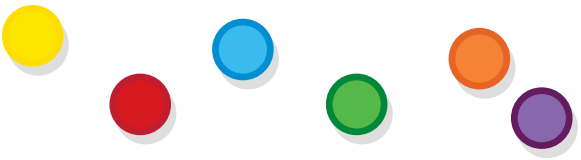

Figure 2-24: Balls

A category consists of a collection of objects as well as some arrows connecting some of them to one another. We call the arrows, *morphisms*.

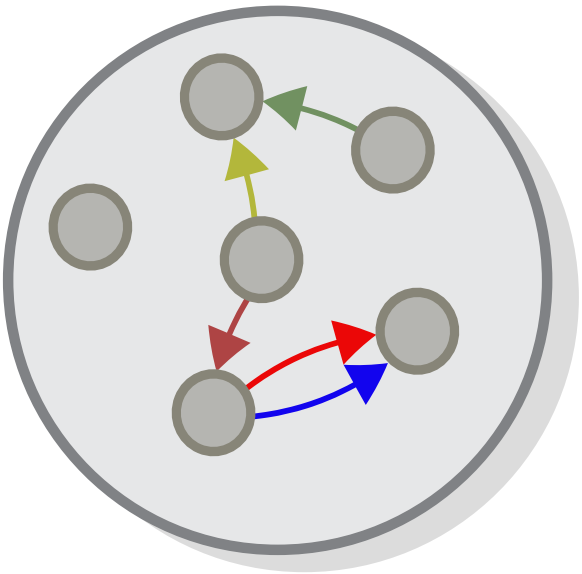

Figure 2-25: A category

Wait a minute - we said that all sets form a category, but at the same time any one set can be seen as a category on its own right (just one which has no morphisms). This is true and an example of a phenomenon that is very characteristic of category theory - one structure can be examined from many different angles and may play many different roles, often in a recursive fashion.

This particular analogy (a set as a category with no morphisms) is, however, not very useful. Not because it's in any way incorrect, but because category theory is *all about the morphisms*. If in set theory arrows are nothing but a connection between a source and a destination, in category theory it's the *objects* that are nothing but a source and destination for the arrows that connect them to other objects. This is why, in the diagram above, the arrows, and not the objects, are colored: if you ask me, the category of sets should really be called *the category of functions*.

Speaking of which, note that objects in a category can be connected by multiple arrows and that arrows having the same source and target sets does not in any way make them equivalent (it does not actually mean that they would produce the same value).

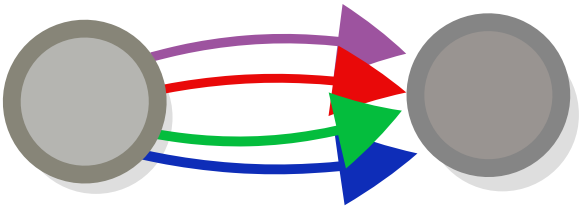

Figure 2-26: Two objects connected with multiple arrows

Why that is true is pretty obvious if we go back to set theory for a second. (OK, maybe we really *have* to do it from time to time.) There are, for example, an infinite number of functions that go from number to boolean, and the fact that they have the same input type and the same output type (or the same *type signature*, as we like to say) does not in any way make them equivalent to one another.

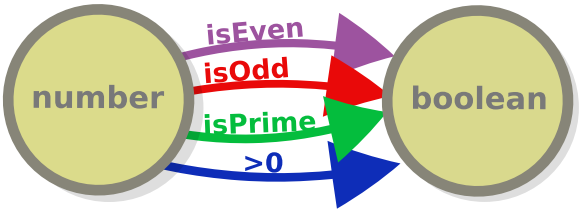

Figure 2-27: Two sets connected with multiple functions

There are some types of categories in which only one morphism between two objects is allowed (or one in each direction), but we will talk about them later.

Composition

One of the few or maybe even the only requirement for a structure to be called a category is that *two morphisms can make a third*, or in other words, that morphisms are *composable* - given two successive arrows with appropriate type signature, we can draw a third one that is equivalent to the consecutive application of the other two.

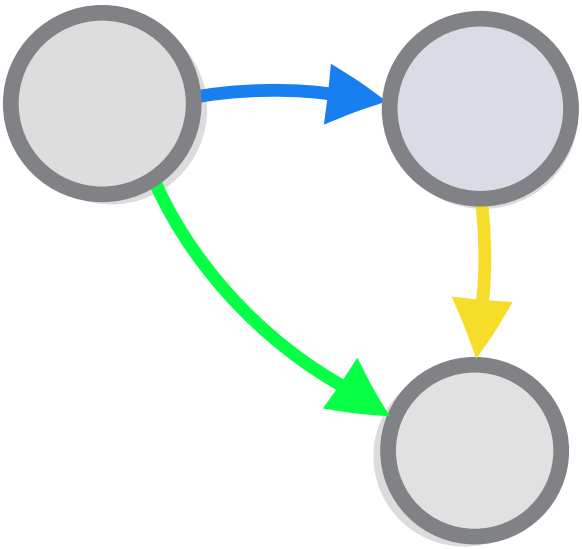

Figure 2-28: Composition of morphisms

Formally, this requirement says that there should exist an operation (denoted with the symbol $\bullet$) such that for each two functions $g : A \to B$ and $f : B \to C$, there exists exactly one function $(f \bullet g) : A \to C$. Again, note that this criteria is not met by just *any* morphism with this type signature. There is *exactly one* morphism that fits these criteria, and there may be some which don't.

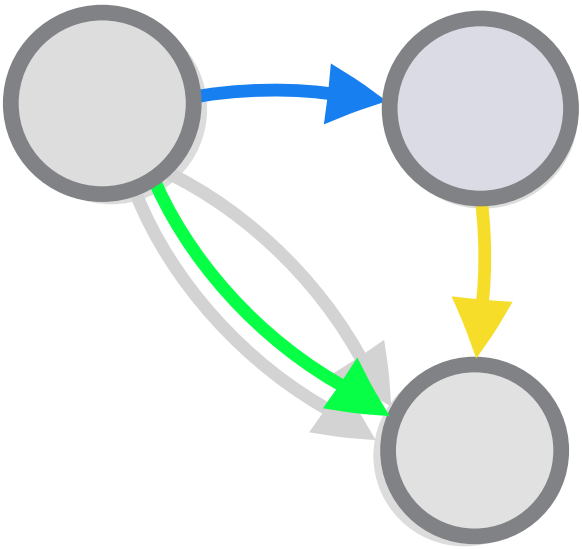

Figure 2-29: Composition of morphisms in the context of additional morphism

NB: Note (if you haven't already) that functional composition is written from right to left. e.g. applying g and then applying f is written $f \bullet g$ and not the other way around. (You can think of it as a shortcut to $f(g(a))$.)

Commuting diagrams

The diagram above, uses colors to illustrate the fact that the green morphism is equivalent to the other two (and not just some unrelated morphism), but in practice this notation is a little redundant - the only reason to draw diagrams in the first place is to represent paths that are equivalent to each other - all other paths just belong in different diagrams.

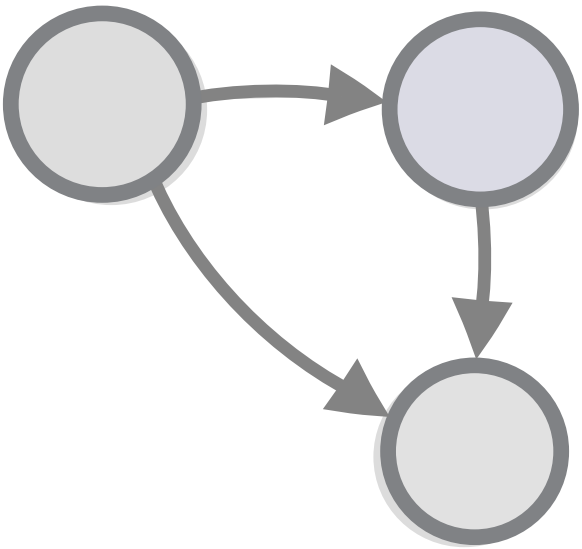

Figure 2-30: Composition of morphisms - a commuting diagram

Diagrams that are like that (ones in which any two paths between two objects are equivalent to one another) are called *commutative diagrams* (or diagrams that *commute*). All diagrams in this book (except the wrong ones) commute.

The law of associativity

Functional composition is special not only because you can take any two morphisms with appropriate signatures and make a third, but because you can do so indefinitely, i.e. given n successive arrows, each of which starts from the object that the other one finishes, we can draw one (exactly one) arrow that is equivalent to the consecutive application of all n arrows.

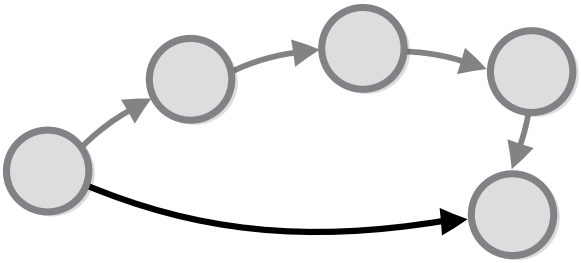

Figure 2-31: Composition of morphisms with many objects

But let's get back to the math. If we carefully review the definition above, we can see that it can be reduced to multiple applications of the following formula: given 4 objects and 3 morphisms between them $f\ g\ h$, combining h and g and then combining the end result with f should be the same as combining h to the result of g and f (or simply $(h \bullet g) \bullet f = h \bullet (g \bullet f)$).

This formula can be expressed using the following diagram, which would only commute if the formula is true (given that all our category-theoretic diagrams commute, we can say, in such cases, that the formula and the diagram are equivalent.)

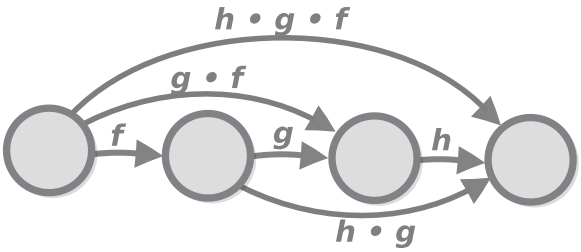

Figure 2-32: Composition of morphisms with many objects

This formula (and the diagram) is the definition of a property called *associativity*. Being associative is required for functional composition to really be called functional composition (and for a category to really be called category). It is also required for our diagrams to work, as diagrams can only represent associative structures (imagine if the diagram above does not commute - it would be super weird.)

Associativity is not just about diagrams. For example, when we express relations using formulas, associativity just means that brackets don't matter in our formulas (as evidenced by the definition $(h \bullet g) \bullet f = h \bullet (g \bullet f)$).

And it is not only about categories either, it is a property of many other operations on other types of objects as well e.g. if we look at numbers, we can see that the multiplication operation is associative e.g. $(1 \times 2) \times 3 = 1 \times (2 \times 3)$. While division is not $(1/2)/3 = 1/(2/3)$.

This approach (composing indefinitely many things) for building stuff is often used in programming. To see some examples, you don't need to look further than the way the pipe operator in Unix (|), which feeds the standard output of a program with the standard input of another program, is (ab)used. If you *want* to look further, note that there is a whole program-

ming paradigm based on functional composition, called "concatenative programming".

Identity

Before the standard Arabic numerals that we use today, there were Roman numbers. They were no good, for the simple reason that they lacked the concept of *zero* - a number that indicated the absence of number. Any number system that lacks this simple concept is extremely limited. It is the same in programming, where we have multiple values that indicate the absence of a value and it is the same in category theory - in order to be able to define more stuff using morphisms in category theory, we too would want to define zero, or what we call the "identity morphism" for each object. In short, this is a morphism, that doesn't do anything.

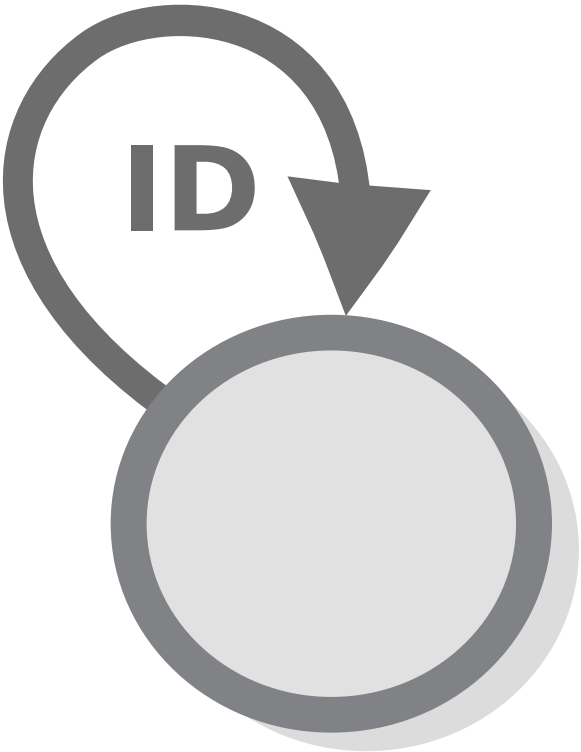

Figure 2-33: The identity morphism (but can also be any other morphism)

It's important to mark this morphism, because there can be (let's add the very important (and also very boring) reminder) many morphisms that go from one object to the same object, many of which actually do stuff. For example, mathematics deals with a multitude of functions that have the set of numbers as source and target, such as *negate*, *square*, *addone*, and are not at all the identity morphism.

Question: What is the identity morphism in the category of sets?

Isomorphism

Why do we need to define a morphism that does nothing? It's because morphisms are the basic building blocks of our language, and we need this one to be able to speak properly. For example, once we have the concept of identity morphism defined, we can have a category-theoretic definition of an *isomorphism* (which is important, because the concept of an isomorphism is very important for category theory). Like we said in the previous chapter, an isomorphism between two objects (A and B) consists of two morphisms - ($A \rightarrow B$. and $B \rightarrow A$) such that their compositions are equivalent to the identity functions of the respective objects. Formally, objects A and B are isomorphic if there exist morphisms $f : A \rightarrow B$ and $g : B \rightarrow A$ such that $f \bullet g = id_B$ and $g \bullet f = id_A$.

And here is the same thing expressed with a commuting diagram.

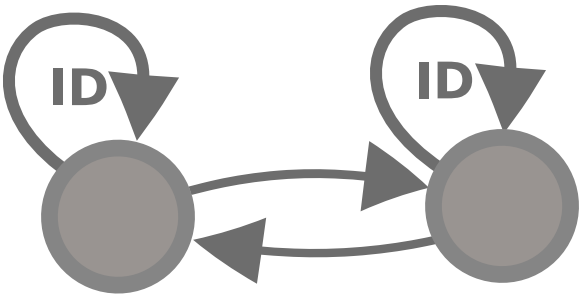

Figure 2-34: Isomorphism

Like the example with the law of associativity, the diagram expresses the same (simple) fact as the formula, namely that going from the one of objects (A and B) to the other one and then back again is the same as applying the identity morphism i.e. doing nothing.

A summary

For future reference, let's restate what a category is.

A category is a collection of *objects* (we can think of them as *points*) and *morphisms* (or *arrows*) that go from one object to another, where: 1. Each object has to have the identity morphism. 2. There should be a way to compose two morphisms with an appropriate type signature into a third one in a way that is *associative*.

This is it.

3

MONOIDS ETC

Since we are done with categories, let's look at some other structures that are also interesting - monoids. Like categories, monoids/groups are also abstract systems consisting of set of elements and operations for manipulating these elements, however the operations look different than the operations we have for categories. Let's see them.

What are monoids

Monoids are simpler than categories. A monoid is defined by a collection/set of elements (called the monoid's *underlying set*, together with an *monoid operation* - a rule for combining two elements that produces a third element one of the same kind.

Let's take our familiar colorful balls.

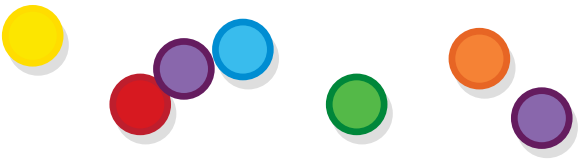

Figure 3-1: Balls

We can define a monoid based on this set by defining an operation for "combining" two balls into one. An example of such operation would be blending the colors of the balls, as if we are mixing paint.

Figure 3-2: An operation for combining balls

You can probably think of other ways to define such an operation. This will help you realize that there can be many ways to create a monoid from a given set of set elements i.e. the monoid is not the set itself, it is the set *together with the operation*.

Associativity

The monoid operation should, like functional composition, be *associative* i.e. applying it on the same number of elements in a different order should make no difference.

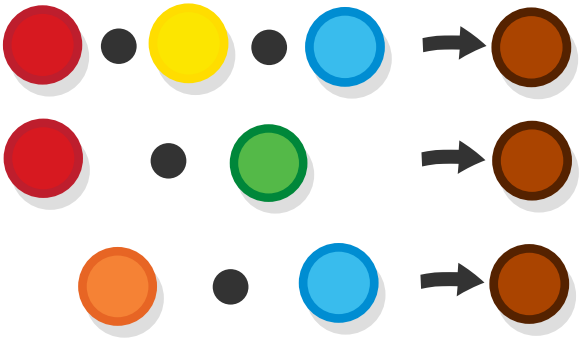

Figure 3-3: Associativity in the color mixing operation

When an operation is associative, this means we can use all kinds of algebraic operations to any sequence of terms (or in other words to apply equation reasoning), like for example we can replace any element with a set of elements from which it is composed of, or add a term that is present at both sides of an equation and retaining the equality of the existing terms.

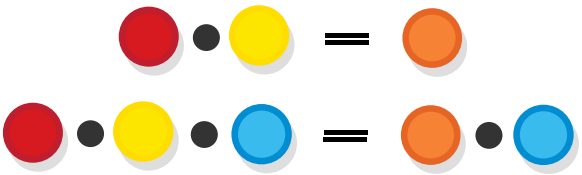

Figure 3-4: Associativity in the color mixing operation

The identity element

Actually, not any (associative) operation for combining elements makes the balls form a monoid (it makes them form a *semigroup*, which is also a thing, but that's a separate topic). To be a monoid, a set must feature what is called an *identity element* of a given operation, the concept of which you are already familiar from both sets and categories - it is an element that when combined with any other element gives back that same element (not the identity but the other one). Or simply $x \bullet i = x$ and $i \bullet x = x$ for any x.

In the case of our color-mixing monoid the identity element is the white ball (or perhaps a transparent one, if we have one).

Figure 3-5: The identity element of the color-mixing monoid

As you probably remember from the last chapter, functional composition is also associative and it also contains an identity element, so you might start suspecting that it forms a monoid in some way. This is indeed the case, but with one caveat.

Basic monoids

To keep the suspense, before we discuss the relationship between monoids and categories, we are going through see some simple examples of monoids.

Monoids from numbers

Mathematics is not only about numbers, however numbers do tend to pop up in most of its areas, and monoids are no exception. The set of natural numbers $\mathbb{N}$ forms a monoid when combined with the all too familiar operation of addition (or *under* addition as it is traditionally said.) This group is denoted $\langle \mathbb{N}, + \rangle$ (in general, all groups are denoted by specifying the set and the operation, enclosed in angle brackets.)

Figure 3-6: The monoid of numbers under addition

If you see a $1 + 1 = 2$ in your textbook you know you are either reading something very advanced, or very simple, although I am not really sure which of the two applies in the present case.

Anyways, the natural numbers also form a monoid under multiplication as well.

Figure 3-7: The monoid of numbers under multiplication

Question: Which are the identity elements of those monoids?

Task: Go through other mathematical operations and verify that they are monoidal.

Monoids from boolean algebra

Thinking about operations that we covered, we may remember the boolean operations *and* and *or*. Both of them form monoids, which operate on the set, consisting of just two values $\{True, False\}$.

Task: Prove that $\wedge$ is associative by expanding the formula $(A \wedge B) \wedge C = A \wedge (B \wedge C)$ with all possible values. Do the same for *or*.

Question: Which are the identity elements of the *and* and *or* operations?

Monoid operations in terms of set theory

We now know what the monoid operation is, and we even saw some simple examples. However, we never defined the monoid rule/operation formally i.e. using the language of set theory with which we defined everything else. Can we do that? Of course we can - everything can be defined in terms of sets.

We said that a monoid consists of two things a set (let's call it A) and a monoid operation that acts on that set. Since A is already defined in set theory (because it is just a set), all we have to do is define the monoid operation.

Defining the operation is not hard at all. Actually, we have already done it for the operation $+$ - in chapter 2, we said that *addition* can be represented in set theory as a function that accepts a product of two numbers and returns a number (formally $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$).

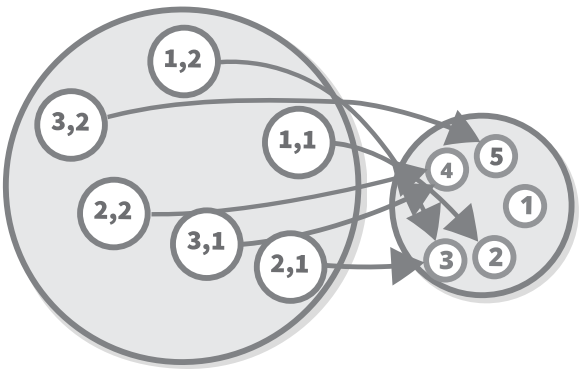

Figure 3-8: The plus operation as a function

Every other monoid operation can also be represented in the same way - as a function that takes a pair of elements from the monoid's set and returns one other monoid element.

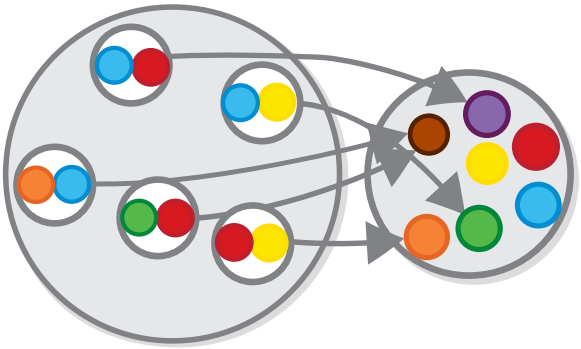

Figure 3-9: The color-mixing operation as a function

Formally, we can define a monoid from any set A, by defining an (associative) function with type signature $A \times A \rightarrow A$. That's it. Or to be precise, that is *one way* to define the monoid operation. And there is another way, which we will see next. Before that, let's examine some more categories.

Other monoid-like objects

Monoid operations obey two laws - they are *associative* and there is an *identity element*. In some cases we come across operations that also obey other laws that are also interesting. Imposing more (or less) rules to the way in which (elements) objects are combined results in the definition of other monoid-like structures.

Commutative (abelian) monoids

Looking at the monoid laws and the examples we gave so far, we observe that all of them obey one more rule (law) which we didn't specify - the order in which the operations are applied is irrelevant to the end result.

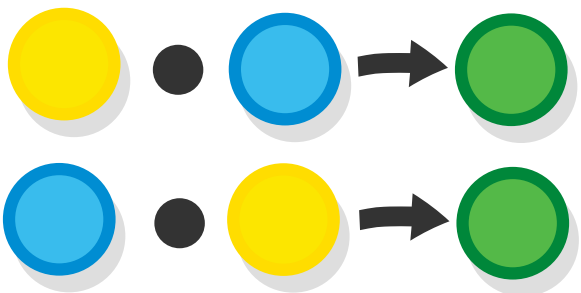

Figure 3-10: Commutative monoid operation

Such operations (ones for which combining a given set of elements yields the same result no matter which one is first and which one is second) are called *commutative* operations. Monoids with operations that are commutative are called *commutative monoids*.

As we said, addition is commutative as well - it does not matter whether if I have given you 1 apple and then 2 more, or if I have given you 2 first and then 1 more.

Figure 3-11: Commutative monoid operation

All monoids that we examined so far are also *commutative*. We will see some non-commutative ones later.

Groups

A group is a monoid such that for each of it's elements, there is another element which is the so called "inverse" of the first one where the element and its inverse cancel each other out when applied one after the other. Plain-English definitions like this make you appreciate mathematical formulas more - formally we say that for all elements x, there must exist x' such that $x \bullet x' = i$ (where i is the identity element).

If we view *monoids* as a means of modeling the effect of applying a set of (associative) actions, we use *groups* to model the effects of actions are also *reversible*.

A nice example of a monoid that we covered that is also a group is the set of integers under addition. The inverse of each number is its opposite number (positive numbers' inverse are negatives and vice versa). The above formula, then, becomes $x + (-x) = 0$

The study of groups is a field that is much bigger than the theory of monoids (and perhaps bigger than category theory itself). And one of its the biggest branches is the study of the "symmetry groups" which we will look into next.

Summary

But before that, just a quick note - the algebraic structures that we saw can be summarized based on the laws that define them in this table.

	Semigroups	Monoids
Associativity	X	X
Identity		X
Invertability		

And now for the symmetry groups.

Symmetry groups and group classifications

An interesting kinds of groups/monoids are the groups of *symmetries* of geometric figures. Given some geometric figure, a symmetry is an action after which the figure is not displaced (e.g. it can fit into the same mold that it fit before the action was applied).

We won't use the balls this time, because in terms of symmetries they have just one position and hence just one action - the identity action (which is it's own reverse, by the way). So let's take this triangle, which, for our purposes, is the same as any other triangle (we are not interested in the triangle itself, but in its rotations).

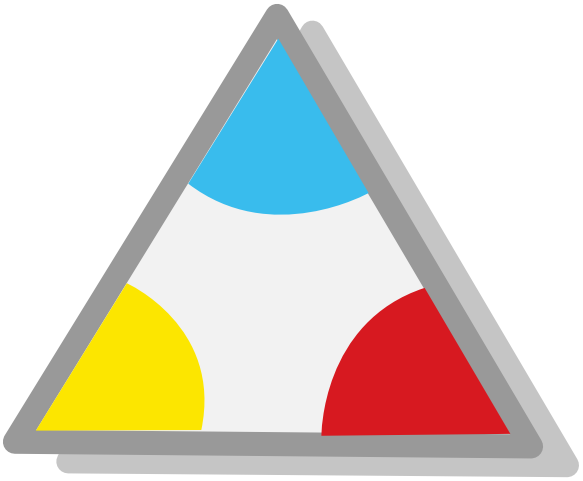

Figure 3-12: A triangle

Groups of rotations

Let's first review the group of ways in which we can rotate our triangle i.e. its *rotation group*. A geometric figure can be rotated without displacement in positions equal to the number of its sides, so for our triangle there are 3 positions.

Figure 3-13: The group of rotations in a triangle

Connecting the dots (or the triangles in this case) shows us that there are just two possible rotations that get us from any state of the triangle to any other one - a *120-degree rotation* (i.e. flipping the triangle one time) and a *240-degree rotation* (i.e. flipping it two times (or equivalently, flipping it once, but in the opposite direction)). Adding the identity action of 0-degree rotation makes up for 3 rotations (objects) in total.

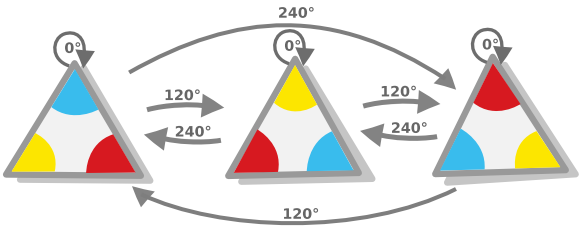

Figure 3-14: The group of rotations in a triangle

The rotations of a triangle form a monoid - the *rotations are objects* (of which the zero-degree rotation is the identity) and the monoid operation which combines two rotations into one is just the operation of performing the first rotation and then performing the second one.

NB: Note once again that the elements in the group are the *rotations*, not the triangles themselves, actually the group has nothing to do with triangles, as we shall see later.

Cyclic groups/monoids

The diagram that enumerates all the rotations of a more complex geometrical figure looks quite messy at first.

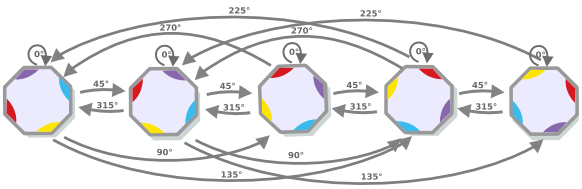

Figure 3-15: The group of rotations in a more complex figure

But it gets much simpler to grasp if we notice the following: although our group has many rotations, and there are more still for figures with more sides (if I am not mistaken, the number of rotations is equal to the number of the sides), *all those rotations can be reduced to the repetitive application of just one rotation*, (for example, the 120-degree rotation for triangles and the 45-degree rotation for octagons). Let's make up a symbol for this rotation.

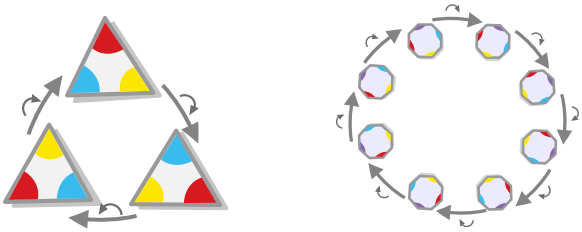

Figure 3-16: The group of rotations in a triangle

Symmetry groups that have such "main" rotation, and, in general, groups and monoids that have an object that is capable of generating all other ob-

jects by it's repeated application, are called *cyclic groups*. And such rotation are called the group's *generator*.

All rotation groups are cyclic groups. Another example of a cyclic groups is, yes, the natural numbers under addition. Here we can use $+1$ or -1 as generators.

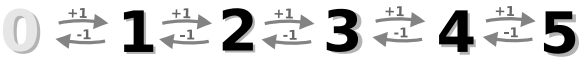

Figure 3-17: The group of numbers under addition

Wait, how can this be a cyclic group when there are no cycles? This is because the integers are an *infinite* cyclic group.

A number-based example of a finite cyclic group is the group of natural numbers under *modular arithmetic* (sometimes called "clock arithmetic"). Modular arithmetic's operation is based on a number called the modulus (let's take 12 for example). In it, each number is mapped to the *remainder of the integer addition of that number and the modulus*.

For example: $1 \pmod{12} = 1$ (because $1/12 = 0$ with 1 remainder) $2 \pmod{12} = 2$ etc.

But $13 \pmod{12} = 1$ (as $13/12 = 1$ with 1 remainder) $14 \pmod{12} = 2$, $15 \pmod{12} = 3$ etc.

In effect numbers "wrap around", forming a group with as many elements as it the modulus number. Like for example a group representation of modular arithmetic with modulus 3 has 3 elements.

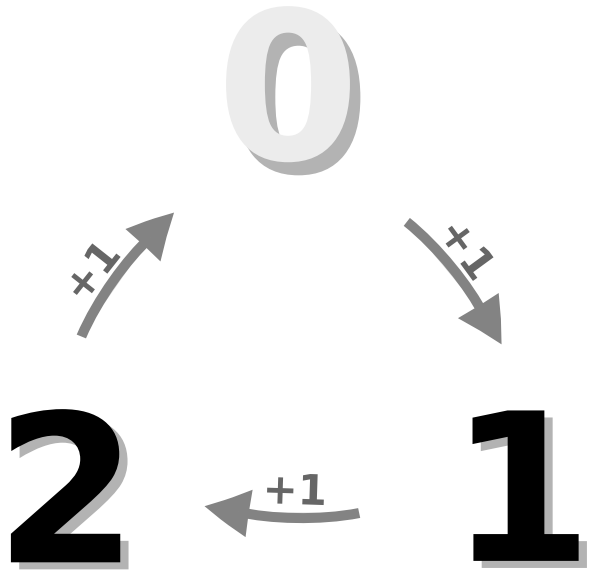

Figure 3-18: The group of numbers under addition

All cyclic groups that have the same number of elements (or that are of the *same order*) are isomorphic to each other (careful readers might notice that we haven't yet defined what a group isomorphisms are. Even more careful readers might already have an idea about what it is.)

For example, the group of rotations of the triangle is isomorphic to the natural numbers under the addition with modulo 3.

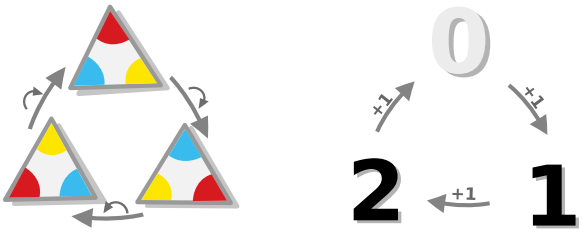

Figure 3-19: The group of numbers under addition

All cyclic groups are *commutative* (or "abelian" as they are also called).

Task: Show that there are no other groups with 3 objects, other than Z_3.

There are abelian groups that are not cyclic, but, as we shall see below, the concepts of cyclic groups and of abelian groups are deeply related.

Group isomorphisms

We already mentioned group isomorphisms, but we didn't define what they are. Let's do that now - an isomorphism between two groups is an isomorphism (f) between their respective sets of elements, such that for any a and b we have $f(a \bullet b) = f(a) \bullet f(b)$. Visually, the diagrams of isomorphic groups have the same structure.

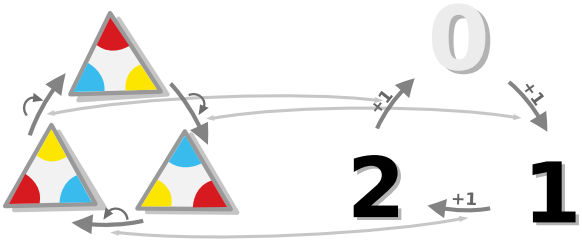

Figure 3-20: Group isomorphism between different representations of S3

As in category theory, in group theory isomorphic groups they considered instances of one and the same group. For example the one above is called Z_3.

Finite groups

Like with sets, the concept of an isomorphism in group theory allows us to identify common finite groups.

The smallest group is just the trivial group Z_1 that has just one element.

Figure 3-21: The smallest group

The smallest non-trivial group is the group Z_2 that has two elements.

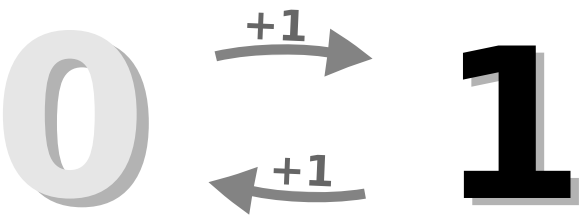

Figure 3-22: The smallest non-trivial group

Z_2 is also known as the *boolean group*, due to the fact that it is isomorphic to the *True*, *False* set.

Like Z_3, Z_1 and Z_2 are cyclic.

Group/monoid products

We already saw a lot of abelian groups that are also cyclic, but we didn't see any abelian groups that are not cyclic. So let's examine what those look like. This time, instead of looking into individual examples, we will present a general way for producing abelian non-cyclic groups from cyclic ones - it is by uniting them by using *group product*.

Given any two groups, we can combine them to create a third group, comprised of all possible pairs of elements from the two groups and of the sum of all their actions.

Let's see how the product looks like take the following two groups (which, having just two elements and one operation, are both isomorphic to $Z2$). To make it easier to imagine them, we can think of the first one as based on the vertical reflection of a figure and the second, just the horizontal reflection.

Figure 3-23: Two trivial groups

We get set of elements of the new group by taking *the Cartesian product* of the set of the elements of the first group and the set of the element of the second one.

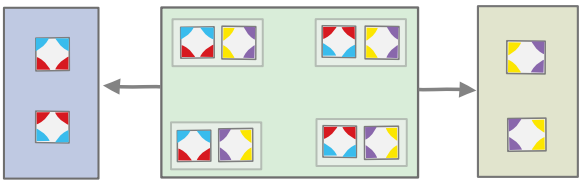

Figure 3-24: Two trivial groups

And the *actions* of a product group are comprised of the actions of the first group, combined with the actions of the second one, where each action is applied only on the element that is a member of its corresponding group, leaving the other element unchanged.

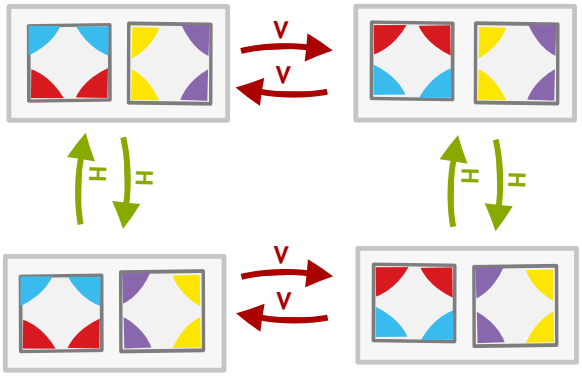

Figure 3-25: Klein four

The product of the two groups we presented is called the *Klein four-group* and it is the simplest *abelian non-cyclic* group.

Another way to present the Klein-four group is the *group of symmetries of a non-square rectangle*.

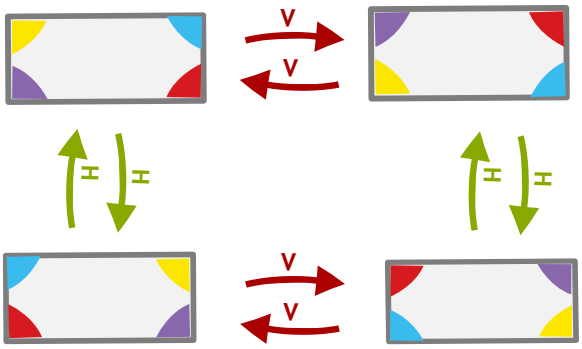

Figure 3-26: Klein four

Task: Show that the two representations are isomorphic.

Like all product groups, the Klein-four group is *non-cyclic* (because there are not one, but two generators) - vertical and horizontal spin. It is, however, still *abelian*, because the ordering of the actions still does not matter for the end results. Actually, the Klein-four group is the *smallest non-cyclic group*.

In fact, products groups (except the ones that feature the trivial group) are always *non-cyclic*, because even if the two groups that comprise the product it are cyclic, and have just 1 generator each, their product would have 2 generators.

Product groups are still abelian, provided that the groups that form them are abelian - we can see that this is true by noticing that, although the generators are more than one, each of them acts only on it's own part of the group, so they don't interfere with each other in any way.

Fundamental theorem of Finite Abelian groups

Products provide one way to create non-cyclic abelian groups - by creating a product of two or more cyclic groups. The fundamental theory of finite abelian groups is a result that tells us that *this is the only way* to produce non-cyclic abelian groups i.e.

> All abelian groups are either cyclic or products of cyclic groups.

We can use this law to gain intuitive understanding of the what abelian groups are, but also to test whether a given group can be broken down to a product of more elementary groups.

{% if site.distribution == 'print' %}

Color-mixing monoid as a product

To see how can we use this theorem, let's revisit our color mixing monoid that we saw earlier.

Figure 3-27: color-mixing group

As there doesn't exist a color that, when mixed with itself, can produce all other colors, the color-mixing monoid is *not cyclic*. However, the color mixing monoid *is abelian*. So according to the theorem of finite abelian groups (which is valid for monoids as well), the color-mixing monoid must be (isomorphic to) a product.

And it is not hard to find the monoids that form it - although there isn't one color that can produce all other colors, there are three colors that can do that - the prime colors. This observation leads us to the conclusion that the color-mixing monoid, can be represented as the product of three monoids, corresponding to the three primary colors.

Figure 3-28: color-mixing group as a product

You can think of each color monoid as a boolean monoid, having just two states (colored and not-colored).

Figure 3-29: Cyclic groups, forming the color-mixing group

Or alternatively, you can view it as having multiple states, representing the different levels of shading.

Figure 3-30: Color-shading cyclic group

In both cases the monoid would be cyclic.
{%endif%}

Groups/monoid of rotations and reflections

Now, let's finally examine a non-commutative group - the group of rotations *and reflections* of a given geometrical figure. It is the same as the last one, but here besides the rotation action that we already saw (and its composite actions), we have the action of flipping the figure vertically, an operation which results in its mirror image:

Figure 3-31: Reflection of a triangle

Those two operations and their composite results in a group called *Dih*3 that is not abelian (and is furthermore the *smallest* non-abelian group).

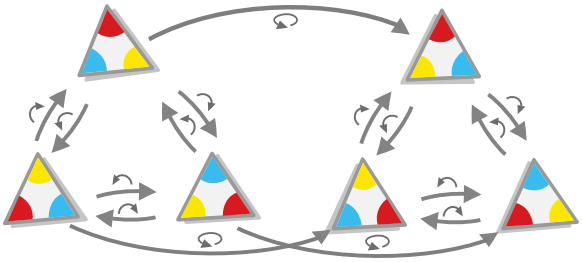

Figure 3-32: The group of rotations and reflections in a triangle

Task: Prove that this group is indeed not abelian.

Question: Besides having two main actions, what is the defining factor that makes this and any other group non-abelian?

Groups/monoids categorically

We began by defining a monoid as a set of composable *elements*. Then we saw that for some groups, like the groups of symmetries and rotations, those elements can be viewed as *actions*. And this is actually true for all other groups as well, e.g. the *redball* in our color-blending monoid can be seen as the action of *addingthecolorred* to the mix, the number 2 in the monoid of addition can be seen as the operation $+2$ etc. This observation leads to a categorical view of the theory of groups and monoids.

Currying

When we defined monoids, we saw that their operations are two-argument functions. And we introduced a way for representing such functions using set theory - by uniting the two arguments into one using products. i.e. we showed that a function that accepts two arguments (say A and B) and maps them into some result (C), can be thought as a mapping from the product of the sets of two arguments to the result. So $A \times B \to C$.

However, this is not the only way to represent multi-argument function set-theoretically - there is another, equally interesting way, that doesn't rely on any data structures, but only on functions: that way is to have a function that maps the first of the two arguments (i.e. from A) to *another function* that maps the second argument to the final result (i.e. $B \to C$.) So $A \to B \to C$.

The practice of transforming a function that takes a pair of objects to a function that takes just one object and returns a function that takes another

one is called *currying*. It is achieved by a higher-order function. Here is how such a function might be implemented.

```
const curry = <A, B, C> (f:(a:A, b:B) => C) => (a:A) => (b:B) => f(a, b)
```

And equally important is the opposite function, which maps a curried function to a multi-argument one, which is known as *uncurry*.

```
const uncurry = <A, B, C> (f:(a:A) => (b:B) => C) => (a:A, b:B ) => f(a)(b)
```

There is a lot to say about these two functions, starting from the fact that its existence gives rise to an interesting relationship between the concept of a *product* and the concept of a *morphism* in category theory, called the *adjunction*. But we will cover this later. For now we are interested in the fact the two function representations are isomorphic, formally $A \times B \to C \cong A \to B \to C$.

By the way, this isomorphism can be represented in terms of programming as well. It is equivalent to the statement that the following function always returns `true` for any arguments,

```
(...args) => uncurry(curry(f(...args)) === f(...args)
```

This is one part of the isomorphism, the other part is the equivalent function for curried functions.

Task: Write the other part of the isomorphism.

Monoid elements as functions/permutations

Let's take a step back and examine the groups/monoids that we covered so far in the light of what we learned. We started off by representing group operation as a function from pairs. For example, the operation of a symmetric group,(let's take Z_3 as an example) are actions that converts two rotations to another rotation.

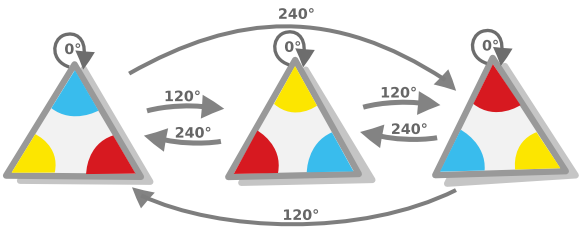

Figure 3-33: The group of rotations in a triangle - group notation

Using currying, we can represent the elements of a given group/monoid as functions by uniting them to the group operation, and the group operation itself - as functional composition. For example, the 3 elements of Z_3 can be seen as 3 bijective (invertable) functions from a set of 3 elements to itself (in group-theoretic context, these kinds of functions are called *permutations*, by the way.)

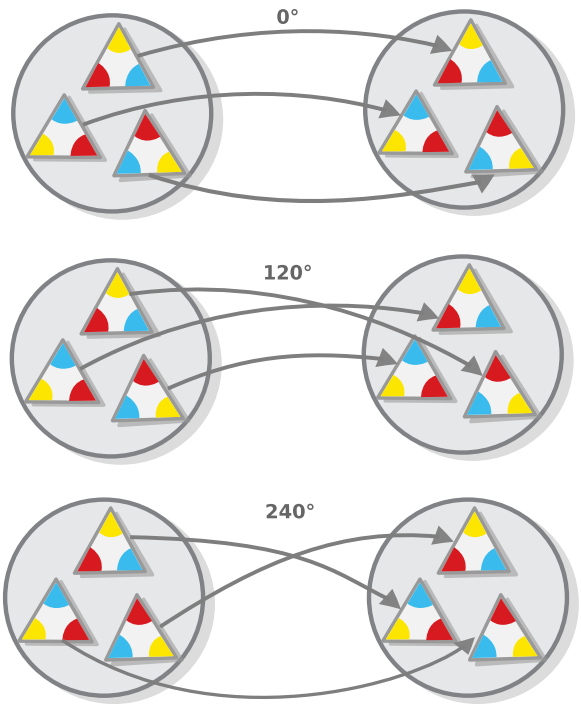

Figure 3-34: The group of rotations in a triangle - set notation

We can do the same for the addition monoid - numbers can be seen not as *quantities* (as in two apples, two oranges etc.), but as *operations*, (e.g. as the action of adding two to a given quantity).

Formally, the operation of the addition monoid, that we saw above has the following type signature.

$+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$

Because of the isomorphism we presented above, this function is equivalent to the following function.

$+ : \mathbb{Z} \to (\mathbb{Z} \to \mathbb{Z})$

When we apply an element of the monoid to that function (say 2), the result is the function $+2$ that adds 2 to a given number.

$+2 : \mathbb{Z} \to \mathbb{Z}$

And because the monoid operation is always a given in the context of a given monoid, we can view the element 2 and the function $+2$ as equivalent in the context of the monoid.

$2 \cong +2$

In other words, in addition to representing the monoid elements in the set as *objects* that are combined using a function, we can represent them as *functions* themselves.

Monoid operations as functional composition

The functions that represent the monoid elements have the same set as source and target, or same signature, as we say (formally, they are of the type

$A \rightarrow A$ for some A). Because of that, they all can be composed with one another, using *functional composition*, resulting in functions that *also has the same signature*.

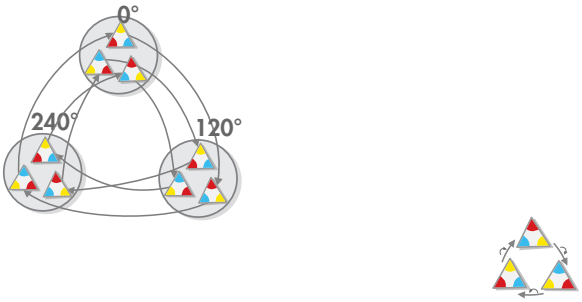

Figure 3-35: The group of rotations in a triangle - set notation

And the same is valid for the addition monoid - number functions can be combined using functional composition.

$+2 \bullet +3 \cong +5$

So, basically the functions that represent the elements of a monoid also form a monoid, under the operation of functional composition (and the functions that represent the elements that form a group also form a group).

Question: Which are the identity elements of function groups?

Task: Show that the functions representing inverse group elements are also inverse.

Cayley's theorem

We saw how using currying we can represent the elements of any group as permutations that, also form a monoid. Cayley's theorem tells us that those two groups are isomorphic:

> Any group is isomorphic to a permutation group.

Formally, if we use *Perm* to denote the permutation group then $Perm(A) \cong A$ for any A.

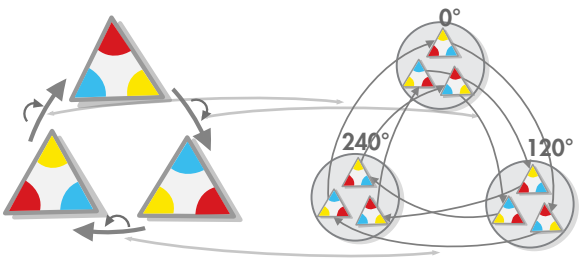

Figure 3-36: The group of rotations in a triangle - set notation and normal notation

Or in other words, representing the elements of a group as permutations actually yields a representation of the monoid itself (sometimes called it's standard representation.)

Cayley's theorem may not seem very impressive, but that only shows how influential it has been as a result.

{% if site.distribution == 'print' %}

Interlude: Symmetric groups

The most important thing that you have to know about the symmetric groups is that they are *not the same thing as symmetry groups*. Once we have that out of the way, we can understand what they actually are: given a natural number n, the symmetric group of n, denoted S_n (symmetric group of degree n) is the group of all possible permutations of a set with n elements. The number of the elements of such groups is equal to are $1 \times 2 \times 3 ... \times n$ or $n!$ (n-factorial.)

So, for example the group S_1 of permutations of the one-element set has just 1 element (because a 1-element set has no other functions to itself other than the identity function.

Figure 3-37: The S1 symmetric group

The group S_2, has $1 \times 2 = 2$ elements (by the way, the colors are there to give you some intuition as to why the number of permutations of a n-element set is $n!$.)

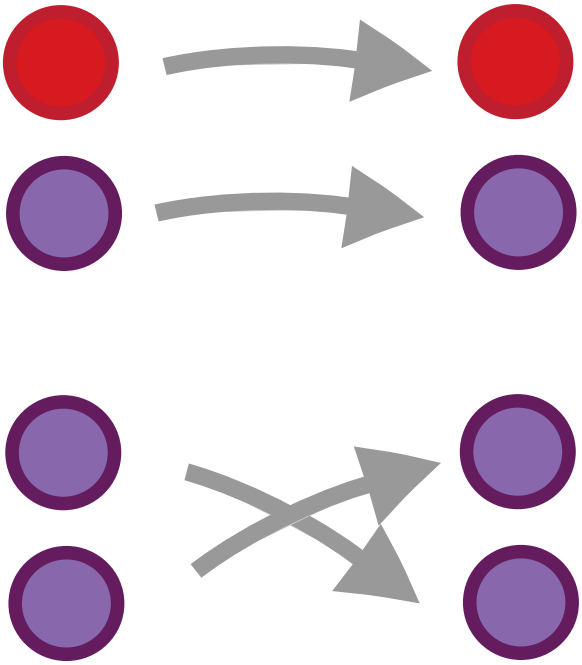

Figure 3-38: The S2 symmetric group

And with S_3 we are already feeling the power of exponential (and even faster than exponential!) growth of the factorial function - it has $1 \times 2 \times 3 = 6$ elements.

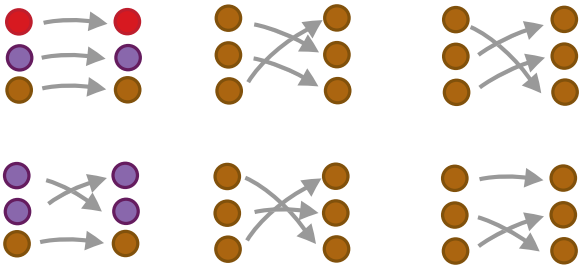

Figure 3-39: The S3 symmetric group

Each symmetric group S_n contains all groups of order n - this is so, because (as we saw in the prev section) every group with n elements is isomorphic to a set of permutations on the set of n elements and the group S_n contains all such permutations. In particular, S_1 is isomorphic to Z_1 (the *trivial group*), S_2 is isomorphic to Z_2 (the *boolean group*). And the top three permutations of S_3 are isomorphic to the group Z_3.

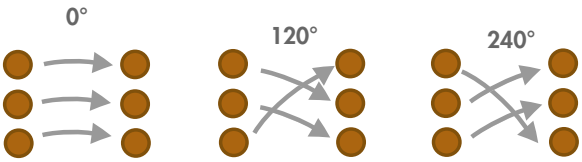

Figure 3-40: The S3 symmetric group

Based on this insight, can state Cayley's theorem in terms of symmetric groups in the following way:

> All groups are isomorphic to subgroups of symmetric groups.

Task: Show how the two are equivalent.

Fun fact: the study of group theory actually started by examining symmetric groups, so this theorem was actually a prerequisite for the emergence of the normal definition of groups that we all know and love (OK, at least *I* love it) - it provided a proof that the notion described by this definition is equivalent to the already existing notion of symmetric groups.

{% endif %}

Monoids as categories

We saw that converting the monoid's elements to actions/functions yields an accurate representation of the monoid in terms of sets and functions.

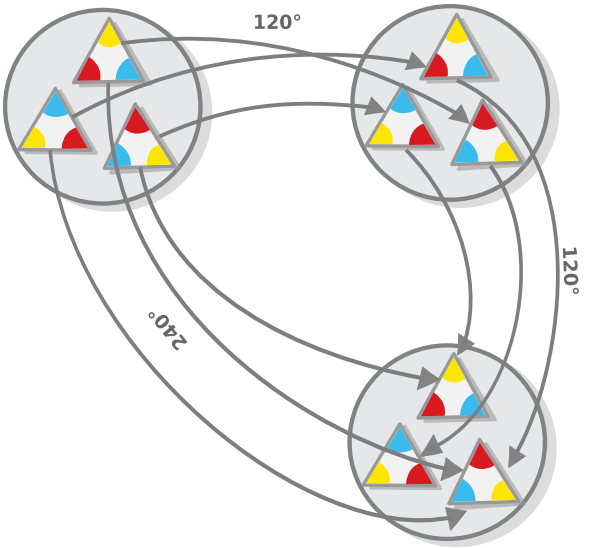

Figure 3-41: The group of rotations in a triangle - set notation and normal notation

However, it seems that the set part of the structure in this representation is kinda redundant - you have the same set everywhere - so, it would do it good if we can simplify it. And we can do that by depicting it as an external (categorical) diagram.

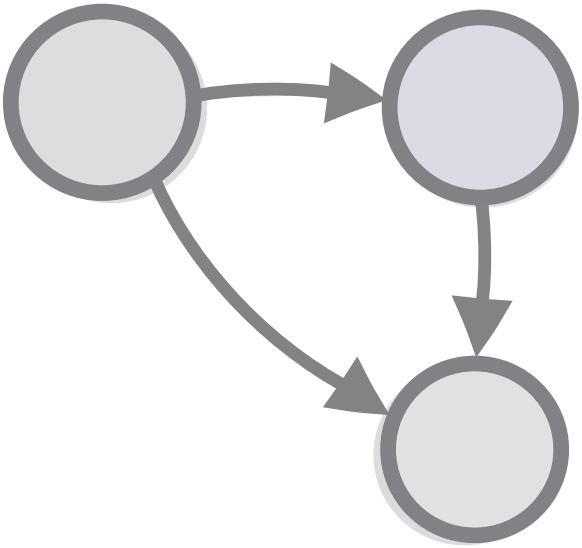

Figure 3-42: The group of rotations in a triangle - categorical notation

But wait, if the monoids' underlying *sets* correspond to *objects* in category theory, then the corresponding category would have just one object. And so the correct representation would involve just one point from which all arrows come and to which they go.

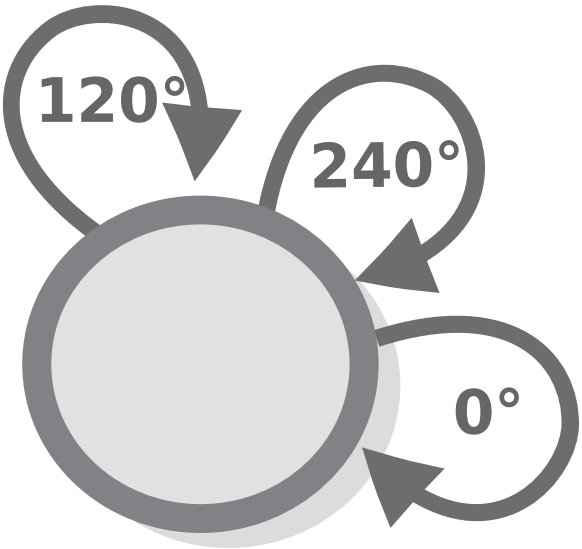

Figure 3-43: The group of rotations in a triangle - categorical notation

The only difference between different monoids would be the number of morphisms that they have and the relationship between them.

The intuition behind this representation from a category-theoretic standpoint is encompassed by the law of *closure* that monoid and group operations have and that categories lack - it is the law stating that applying the operation (functional composition) on any two objects always yields the same object, e.g. no matter how do you flip a triangle, you'd still get a triangle.

	Categories	Monoids
Associativity	X	X
Identity	X	X
Invertability		
Closure		X

When we view a monoid as a category, this law says that all morphisms in the category should be from one object to itself - a monoid, any monoid, can be seen as a *category with one object*.

Let's elaborate on this thought by reviewing the definition of a category from chapter 2.

> A category is a collection of *objects* (we can think of them as points) and *morphisms* (arrows) that go from one object to another, where: 1. Each object has to have the identity morphism. 2. There should be a way to compose two morphisms with an appropriate type signature into a third one in a way that is associative.

Aside from the little-confusing fact that *monoid objects are morphisms* when viewed categorically, this describes exactly what monoids are.

Categories have an identity morphism for each object, so for categories with just one object, there should also be exactly one identity morphism. And monoids do have an identity object, which when viewed categorically corresponds to that identity morphism.

Categories provide a way to compose two morphisms with an appropriate type signature, and for categories with one object this means that *all morphisms should be composable* with one another. And the monoid operation does exactly that - given any two objects (or two morphisms, if we use the categorical terminology), it creates a third.

Philosophically, defining a monoid as a one-object category means corresponds to the view of monoids as a model of how a set of (associative) actions that are performed on a given object alter it's state. Provided that the object's state is determined solely by the actions that are performed on it, we can leave it out of the equation and concentrate on how the actions are combined. And as per usual, the actions (and elements) can be anything, from mixing colors in paint, or adding a quantities to a given set of things etc.

Group/monoid presentations

When we view cyclic groups/monoids as categories, we would see that they correspond to categories that (besides having just one object) also have *just one morphism* (which, as we said, is called a *generator*) along with the morphisms that are created when this morphism is composed with itself. In fact the infinite cyclic monoid (which is isomorphic to the natural numbers), can be completely described by this simple definition.

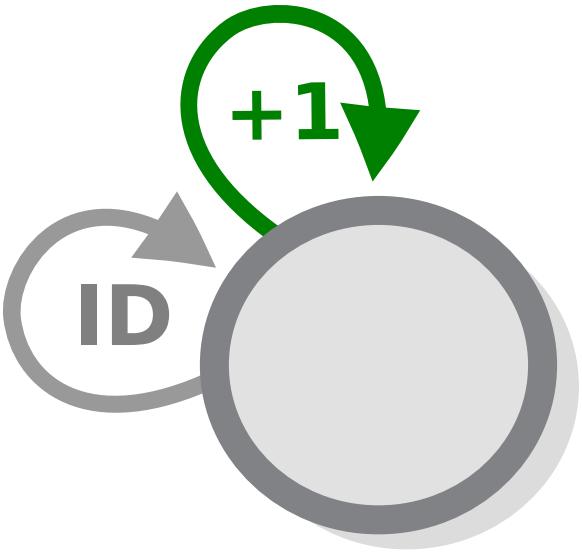

Figure 3-44: Presentation of an infinite cyclic monoid

This is so, because applying the generator again and again yields all elements of the infinite cyclic group. Specifically, if we view the generator as the action $+1$ then we get the natural numbers.

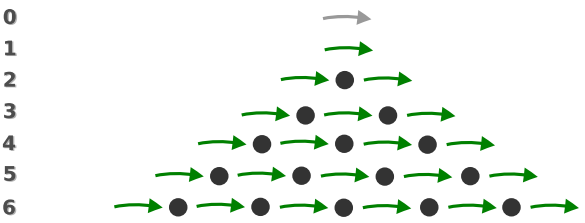

Figure 3-45: Presentation of an infinite cyclic monoid

Finite cyclic groups/monoids are the same, except that their definition contains an additional law, stating that that once you compose the generator with itself n number of times, you get identity morphism. For the cyclic group Z_3 (which can be visualized as the group of triangle rotations) this law states that composing the generator with itself 3 times yields the identity morphism.

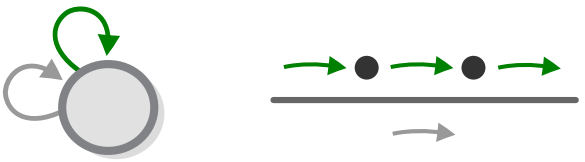

Figure 3-46: Presentation of a finite cyclic monoid

Composing the group generator with itself, and then applying the law, yields the three morphisms of Z_3.

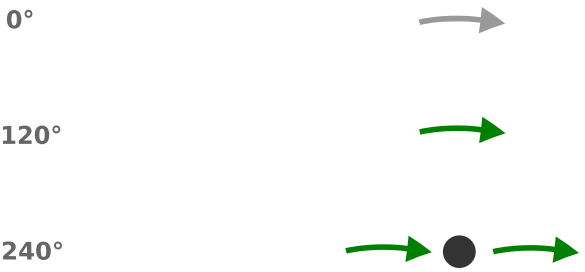

Figure 3-47: Presentation of a finite cyclic monoid

We can represent product groups this way too. Let's take Klein four-group as an example, The Klein four-group has two generators that it inherits from the groups that form it (which we viewed like vertical and horizontal rotation of a non-square rectangle) each of which comes with one law.

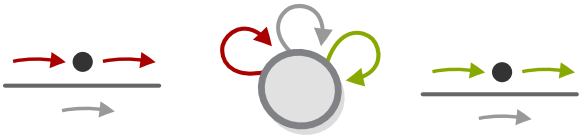

Figure 3-48: Presentation of Klein four

To make the representation complete, we add the law for combining the two generators.

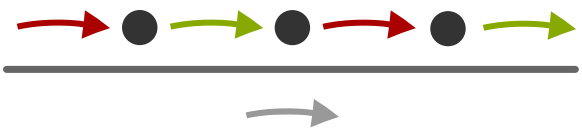

Figure 3-49: Presentation of Klein four - third law

And then, if we start applying the two generators and follow the laws, we get the four elements.

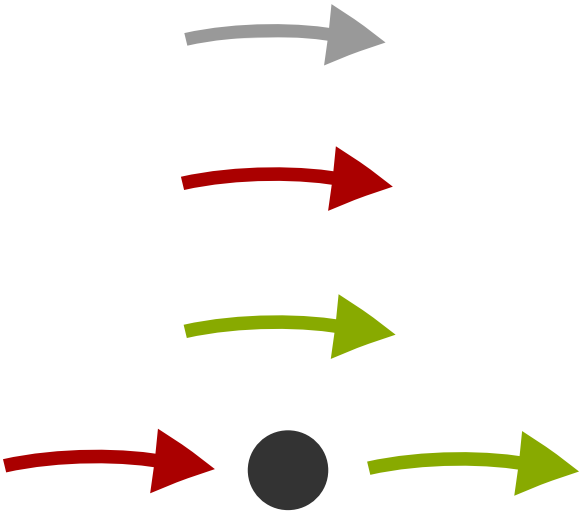

Figure 3-50: The elements of Klein four

The set of generators and laws that defines a given group is called the *presentation of a group*. Every group has a presentation.

Free monoids

We saw how picking a different selection of laws gives rise to different types of monoid. But what monoids would we get if we pick no laws at all? These monoids (we get a different one depending on the set we picked) are called a *free monoids* (the word "free" is used in the sense that once you have the set, you can upgrade it to a monoid for free (i.e. without having to define anything else.)

If you revisit the previous section you will notice that we already saw one such monoid. The free monoid with just one generator is isomorphic to the monoid of integers.

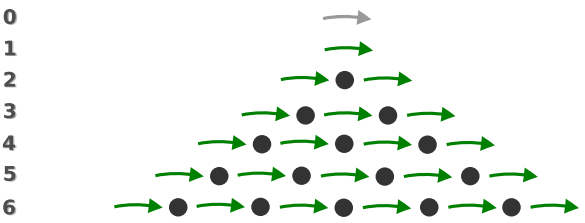

Figure 3-51: The free monoid with one generator

We can make a free monoid from the set of colorful balls - the monoid's elements would be sequences of all possible combinations of the balls.

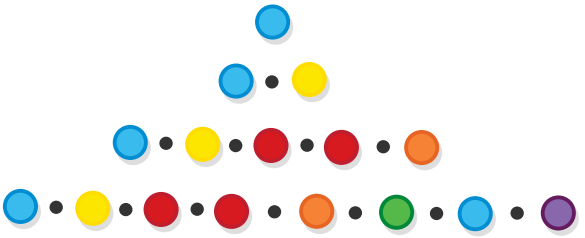

Figure 3-52: The free monoid with the set of balls as a generators

The free monoid is a special one - each element of the free monoid over a given set, can be converted to a corresponding element in any any other monoid that uses the same set of generators by just applying the monoid's laws. For example, here is how the elements above would look like if we apply the laws of the color-mixing monoid.

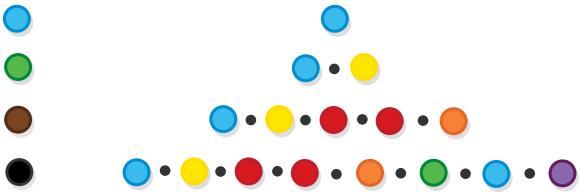

Figure 3-53: Converting the elements of the free monoid to the elements of the color-mixing monoid

Task: Write up the laws of the color-mixing monoid.

If we put out programmers' hat, we will see that the type of the free monoid under the set of generators T (which we can denote as `FreeMonoid<T>`) is isomorphic to the type `List<T>` (or `Array<T>`, if you prefer) and that the intuition behind the special property that we described above is actually very simple: keeping objects in a list allows you to convert them to any other structure i.e. when we want to perform some manipulation on a bunch of objects, but we don't know exactly what this manipulation is, we just keep a list of those objects until it's time to do it.

While the intuition behind free monoids seems simple enough, the formal definition is not our cup of tea... yet, simply because we have to cover more stuff.

We understand that, being the most general of all monoids for a given set of generators, a free monoid can be converted to all of them. i.e. there exist a function from it to all of them. But what kind of function would that be?

Plus, what about other monoids that also can have this property (that they are connected to any other monoids) Simple - they are the free monoid with some extra structure, so we can filter them out by using an universal property.

But, what would this property be, (and what the hell are universal properties anyways?), tune in after a few chapters to find out.