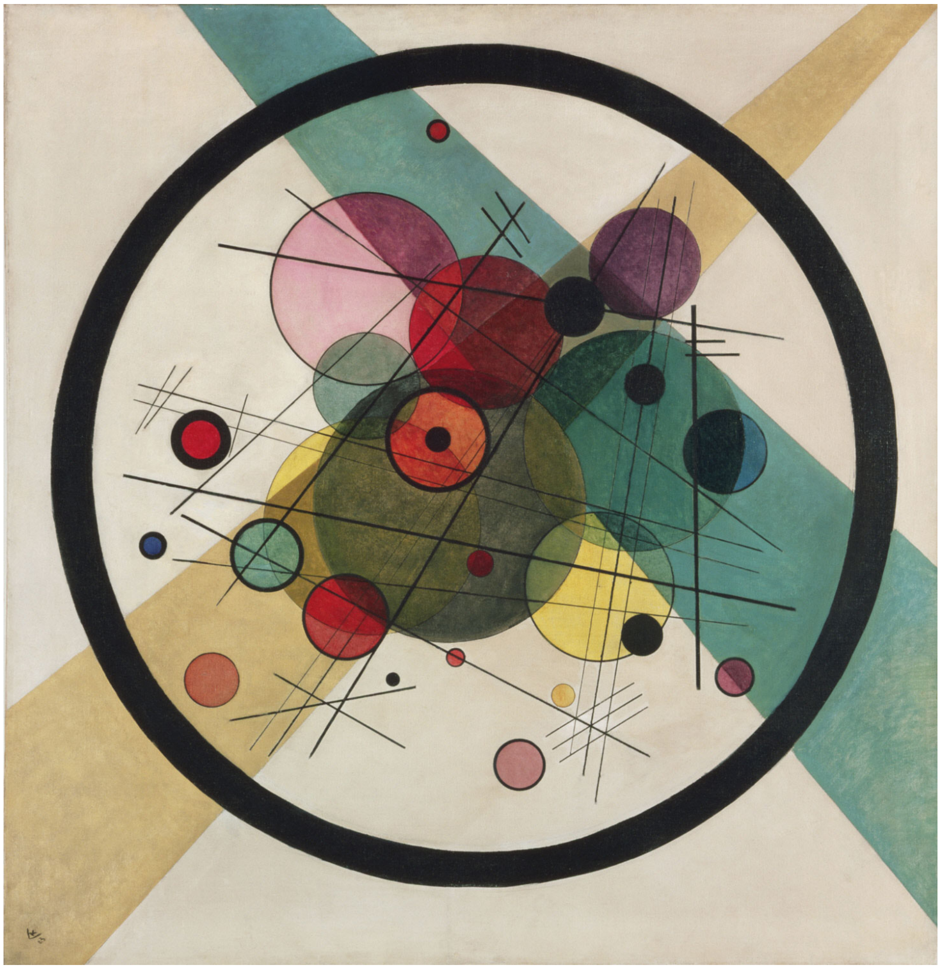

The Limits of Logic

Jeffrey Sanford Russell

University of Southern California

Last revised August 21, 2019

Image: Wassily Kandinsky, *Circles in a Circle*, 1923

Contents

Preface

The Big Picture

Between roughly 1870 and 1940 a group of people studying philosophy and mathematics—as well as fields that hadn't yet emerged as disciplines with their own names and course catalogues, like linguistics and computer science—made some of the most important and beautiful discoveries in the history of human inquiry. Through these remarkable discoveries, finite beings began to understand the limits on finite beings in new ways: limits on what can be counted, described, calculated, or proved. Even more remarkably, we began to precisely understand what is *beyond* those limits: we now can give precise, well-understood, and rigorously demonstrated general principles and specific examples of the infinite and uncountable, the indescribable, the uncomputable, and the unprovable.

Here are some of the highlights. These are facts which we will be examining in detail throughout this text.

The Uncountable. There are infinities of different sizes. (Indeed, infinitely many different sizes!) (This is called "Cantor's Theorem.")

The Inexpressible. For any precise language, there are properties that it cannot precisely express. These include the property of being a true sentence in that language. (This is called "Tarski's Theorem.")

The Undecidable. There are questions that cannot be answered in general by any systematic method. These "undecidable" questions include the question of which general systematic methods will eventually succeed. (This is called "Turing's Theorem".) They also include the question of which arguments are logically valid. (This is called "Church's Theorem.")

The Unprovable. For any reasonable logical system, there are facts that can be formally described, but cannot be formally proved. (This is called "Gödel's First Incompleteness Theorem.") Thus there can be no elegant "theory of

> everything": no reasonably simple, consistent principles can settle the answer to every question. Furthermore, among the facts that a logical theory cannot prove is the fact that its own theorizing is logically consistent. (This is called "Gödel's Second Incompleteness Theorem.")

Later on we will state each of these facts more exactly, and we will work through detailed arguments for each one of them. We'll work up to this slowly, starting by developing some basic skills and concepts for reasoning carefully about the relationship between language and the world. But let's start with a more informal look.

The arguments for each of these facts rely on the same basic idea. The starting place is the *Liar Paradox*. Let L be the following sentence:

```
This sentence is not true.
```

Apparently, what L says is just that L is not true. So presumably, if L is true, then what L says is so—that is, L is not true. And presumably, if L is *not* true, then what L says is *not* so—that is, L is true. But that means we have a contradiction either way. This is very puzzling.

People have known about this puzzle since the ancient Greeks. For a long time they thought it was just a brain-teaser—not especially important. Maybe there is something wrong with expressions like `this sentence`, but you might reasonably think that the responsible policy is just to avoid using self-referential sentences like L for official purposes, and move on to other more serious questions. But it turns out that this policy is not as easy to carry out as it seems—in a sense, self-reference is inevitable. And taking this old puzzle seriously turns out to lead us to some very important discoveries.

Here is another trick—a variation on the Liar Paradox. The word `red` applies to red things.

> For any thing x, the word `red` applies to x if and only if x is red.

Similarly, the word `short` applies to short things. In general:

> For any thing x, the word `short` applies to x if and only if x is short.

Some of the things that `short` applies to are short *words*. For example, the word `red` is short. So the word `short` applies to `red`. The word `short` is also short, so `short` applies to `short`. So we can say that `short` is *self-applying*. In contrast, `long` is not a long word, so `long` is *non-self-applying*.

Now let’s look at how things go for *this* new word we have introduced: `non-self-applying` .

> The word `non-self-applying` applies to `long`

> The word `non-self-applying` does not apply to `short`

Just like with `red` and `short` , we’d like to say in general,

> For any thing x, the word `non-self-applying` applies to x if and only if x is non-self-applying.

In other words:

> For any thing x, the word `non-self-applying` applies to x if and only if x does not apply to x.

This principle about `non-self-applying` seems very reasonable. But it leads to disaster! What happens if we plug in the word `non-self-applying` itself? Then we get:

> `non-self-applying` applies to `non-self-applying` if and only if `non-self-applying` does not apply to `non-self-applying` .

And this is logically contradictory. If `non-self-applying` *does* self-apply, then it doesn’t, and if `non-self-applying` *does not* self-apply, then it does!

In this version of the puzzle we never used self-referential words like `this sentence` . Instead of self-reference, we used *self-application*. We fed a word to itself, and this landed us in serious trouble.

Feeding something to itself like this is the devious trick behind all of the main facts we are going to study. For example, Cantor showed that if infinite sets were all the same size, then in a sense you could “feed a set to itself” and arrive at a contradiction. Roughly, you get to inexpressible properties by feeding *descriptions* to themselves, you get to undecidable questions by feeding *programs* to themselves, and you get to unprovable statements by feeding *proofs* to themselves. (This trick is sometimes called “diagonalization,” for reasons we will talk about in Section 1.5.)

Let’s look at how this idea works for Gödel’s Theorem about unprovability. Waving our hands a little, we can sketch an argument for a simplified version of this theorem. Suppose that we have some logical theory which is both reasonably powerful and also reasonably simple. We can show that either this theory *does* prove something false, or else it *doesn’t* prove something true.

The description `shorter than 100 symbols` is itself shorter than 100 symbols. In a reasonably powerful logical theory, we can also *prove* this statement.

```
”shorter than 100 symbols” is shorter than 100 symbols
```

Call this statement the *self-application* of the description `shorter than 100 symbols` . So to put it another way, the description `shorter than 100 symbols` *has a provable self-application*. In contrast, the following statement is false:

```
”does not contain a vowel” does not contain a vowel
```

So if our theory doesn't prove false things, then we *can't* prove that statement. In other words, the description `does not contain a vowel` *has an unprovable self-application.*

Now, it turns out that in any logical theory which is both reasonably powerful and reasonably simple, we can write down *this* description, too:

```
has an unprovable self-application
```

Let's call this description H. The self-application of H is the statement

```
”has an unprovable self-application” has an unprovable self-application
```

Let's call this statement G. This is another equivalent way of restating G:

```
The self-application of H is not provable
```

But we just said that G *is* the self-application of H! So in fact, what we just said is equivalent to

```
G is not provable
```

To sum up, the statement G is *equivalent* to the statement `G is not provable` . And furthermore, if we're careful, we can *prove* this equivalence in the logical theory we started out with. Notice that G is very similar to the Liar sentence L, except that now we are talking about what is provable instead of what is true.

Finally, we ask, can we prove G in our logical theory? If we can, then using the equivalence, we can also prove `G is not provable` —which means we can prove something false. But also, if we *can't* prove ***G***, then `G is not provable` is *true*. Since this statement is equivalent to ***G***, that means that ***G*** is also true—which means we *can't* prove something *true*. So either there is a false statement that we can prove, or else there is a true statement that we can't prove.

This very brief overview of Gödel's Theorem leaves a lot out, and it might seem mysterious and suspicious at this point. To do everything properly, without so much hand-waving, we'll have to start by carefully investigating what sets, numbers, descriptions, programs, and proofs are like. This will take quite a bit of work. (We'll get to the official version of this theorem in Section 8.5) But there is a big reward. We finite beings can use these tools to explore the infinite world: to count, describe, calculate, and reason. If we understand how these tools work—using precise language and careful reasoning to learn *about* language and reasoning themselves—then we can also understand their limits, and what is beyond them.

About This Text

Here are three ways in which I have aimed to make this text distinctive.

Philosophical

While I hope that students in other neighboring fields (like linguistics, computer science, and mathematics) will also find it helpful, this book is primarily aimed at people who are interested in philosophy, particularly advanced undergradutes and beginning graduate students. The results in this course—the Theorems named after Cantor, Tarski, Turing, Church, and Gödel—are not just bits of abstract mathematics: they are *philosophical* discoveries. Of course, they are also central to other disciplines besides philosophy. They are also especially rigorous and well-established and require a bit of technical skill to understand. But it would be a shame if we philosophers lose track of this part of our intellectual heritage for these reasons. Forgetting about discoveries like these leads people to sad thoughts like "philosophy makes no progress."

What's more, these discoveries are not just part of the philosophy of *mathematics*. These Theorems are central facts in the philosophy of language and epistemology. They also have important connections to metaphysics, philosophy of mind, decision theory, and many other topics. But the historical presentation of these ideas, which most texts faithfully transcribe, unfortunately obscures some of this. You might come away from many courses thinking that (for example) Gödel's

First Incompleteness Theorem is a parochial brainteaser about "formal theories of arithmetic"—a taste for which not that many of us acquire.

In this course, what takes center stage is not arithmetic but *language*. Languages used by finite beings (whether natural or artificial) typically consist of expressions that are straightforwardly represented as finite strings of discrete symbols. Thus, rather than theories of arithmetic, we will think a lot about theories of these *strings*. (Of course, we will still occasionally need to reason about numbers, so they are not entirely absent.) This shift in focus will make some of our results look unfamiliar to those who are already initiated. (For example, theories of arithmetic take a back seat to the *minimal theory of strings* S.) But the shift from theorems about numbers to theorems about strings is usually pretty straightforward, technically, and the string-centered approach is conceptually simpler. We can almost entirely dispense with one conceptual hurdle from the historical approach: the technique of Gödel-numbering. (This is still discussed in Section 6.9, since while it is dispensable, of course it is still *interesting* that theories of strings can be interpreted in a simple theory of arithmetic.)

I have also departed from historical presentations in other ways. In the 1930's three different equivalent definitions of *computability* were proposed: Gödel's general recursive functions, Church's untyped lambda-calculus, and Turing's machines. Those who are familiar with this history might be surprised to find that this text does not include any of these three topics. Nowadays the idea of a universal formal system for representing algorithms is very familiar—not under any of these three guises, but rather under the guise of a *programming language*. So in this text we will study an elementary fragment of a modern programming language. (We use Python, because it has especially tidy syntax, but pretty much any modern language has an equivalent fragment.) Besides using more widely familiar concepts, another nice pedagogical advantage of this approach is that we can use the very same techniques to study the semantics of first-order logic and the semantics of programs. This makes the parallels between first-order *definable* sets and effectively *decidable* sets more straightforward. (Of course, this model of computation is also equivalent to Gödel, Church, and Turing's versions, so there is no real change in the content of the theorems we prove.)

This text does not itself provide much detailed discussion of the philosophical questions that arise from these results, though I attempt to gesture at interesting connections along the way. But I don't think it's as if the Theorems and their proofs are a hard kernel of "mathematics" surrounded by a fuzzy penumbra of "philosophy." The Theorems and their proofs *are* philosophical theses and arguments, themselves—theses and arguments displaying a distinctive degree of precision, distinctively intricate reasoning, and displaying all of their premises with a distinctive degree of clarity. But these are virtues to which we can aspire with all philosophical argumentation. (It should go without saying: not the *only* such virtues!)

Accessible

This book presupposes that its readers have taken one previous course in formal logic which goes as far as first-order predicate logic—ideally one that at least mentioned models and assignment functions, but this much isn't absolutely essential. It does not presuppose *any* experience with mathematics, or mathematical logic. In particular, this text is not meant to presuppose any experience with reading or writing rigorous arguments ("informal proofs"). Rather, this text aims to teach those skills, alongside the technical and philosophical content.

Because of this, we start things off at a slow, gentle pace, building up technical tools from their foundations, introducing each new assumption as it arises. For students with a bit more technical experience, it may be reasonable to skim over the first three chapters pretty quickly. (But make sure not to skip Section 1.5, which introduces one of the key ideas!)

I have also chosen not to use the Greek alphabet (which involves an unnecessary extra deciphering step for students without the benefit of a classical education), and I have stated things in words rather than just symbols as much as seems practical. (I take the latter to be good practice even for experts.)

Skills-based

As far as I know, there is only one way for humans to learn this kind of material: by doing it. When it comes to a technical argument, just reading it or hearing it explained isn't usually enough to really understand it at more than a superficial level. You have to work it out yourself. You have to see how each step follows from the previous; you have to get a feel for which parts of a proof are important, and which are routine. You have to develop useful intuitions that give you a sense of which results are going to work out in the end. False proofs should smell fishy. An argument that seems like just a mass of details, one after another, is an argument that you don't fully understand yet.

Logic is often taught as a mass of details, one after another. (I've certainly been guilty of teaching it this way—there are a lot of details, after all.) Our hope is to get past that, to understand the important and beautiful parts. But we can't do this (at least, not very well) by just ignoring the details—rather, we have to get good enough at dealing with details that they become automatic and recede into the background. The way to do this is practice.

This text is intended to provide lots of practice, by providing lots of exercises. In the end, the exercises add up to proofs of the central Theorems (Cantor, Tarski, Church, Turing, Gödel). Generally, whenever I provide a proof myself, it's for one of two reasons: either (a) to provide examples of an important style of reasoning for what comes later, or (b) to save students from especially tedious or fiddly bits of the argument. I've tried to teach all of the main ideas through exercises, to allow students to learn things by doing them, rather than just being told.

When I teach this course, I use two different teaching modes. The first is a standard instructor-led lecture, which I use mainly to present new concepts, work through definitions, and do example exercises. The second mode is student-led, in which students present their own solutions to exercises, discuss any questions that come up about them, and collectively fix any problems. (I've found the logistics work best if students sign up online for specific exercises before class. For a somewhat larger class, you can give points just for volunteering, and choose which volunteer actually presents by lottery or some other system. For a very large class, you'll probably need to try something else.) I roughly alternate sessions between the two modes: in a course that meets twice a week, we'll have one "lecture day" and one "problem day." (I take over a bit more of the time at the points in the course that have a lot of new concepts: especially Chapter 1, Chapter 2, and Chapter 7.)

Be warned, this format takes a lot of class time. If you want to cover the material more quickly, in order to get to some more advanced topics, you could present more of the exercise solutions as part of a traditional lecture.

The starred sections can be skipped without losing the main thread. Some of them go into background issues in more detail, and others are more advanced topics.

Acknowledgments

This text has been a long time in the making, and I have benefited from a lot of help from a lot people. Thanks are due to all of the students at USC on whom I inflicted early drafts of this text. The first draft, which consisted of my teaching notes for PHIL 450 in the Fall of 2014, was particularly rough going, and I am very grateful for those students' patience.

I also owe special thanks to Cian Dorr, who "alpha tested" this text with his class at NYU when it was still in a pretty rough form, in the Spring of 2017. He wrote me

many long emails that semester full of detailed ideas for improving things, many of which I have incorporated.

Chapter 1

Sets and Functions

> All things are woven together, and they make a sacred pattern … For there is one universe made out of all things …
>
> Marcus Aurelius (121–180 CE), *Meditations* VII.9

Sometimes we reason about particular things one at a time; but it's often useful to reason about a whole collection of things taken together, all at once. We don't just look at what individual physical particles are like, but what the universe of *all* such particles is like; not just what individual sentences are like, but what a whole *language* is like. So it's generally useful to have a theory of such "totalities"—a theory of sets, and of how sets can be related to one another.

Our first job is to get familiar with some basic techniques for working with sets. These techniques are grounded in certain basic assumptions, or "axioms," about what sets there are and what they are like. We'll practice using these axioms to carefully show other simple facts about sets.[1]

[1] These "axioms" don't make up a standard *axiomatization* of set theory. First, they are redundant: with a bit of trickery, some of the axioms—like the "Axiom of Pairs"—can be derived from other axioms and definitions. Second, they are not complete enough for some purposes. You can find some further details about how these axioms are related to one another, and what has been left out, in Section 1.6. But these issues don't really matter for the main purposes of this course.

1.1 Sets

A **set** is a collection of **elements**.

1.1.1 Notation
Typically we'll use capital letters as labels for sets, and lowercase letters as labels for their elements. The notation $a \in A$ means that a is an element of A. Sometimes we'll describe a set by listing all of its elements. For example, the set

$$\{\text{Silver Lake, Echo Park}\}$$

has two members, both of which are neighborhoods in Los Angeles. The set

$$\{0, \quad 1+1, \quad 2+3, \quad 3-1\}$$

has three members (even though the list we've written out has four terms in it)—because $1 + 1$ and $3 - 1$ are the very same thing, the number two. (A set doesn't contain anything "more than once".) In general, it's good to remember that just because we're using two different *labels*, it doesn't follow that they are labels for two different *things*.

In general, if we say

$$A = \{a_1, a_2, \dots, a_n\}$$

then this means that $a_1, \dots, a_n$ are all of the elements of A.

(We'll also introduce a different "curly bracket" notation for sets in a moment.)

1.1.2 Definition
If A and B are sets, then A is a **subset** of B iff every element of A is an element of B. This is also written $A \subseteq B$ for short. We say A is a **proper subset** of B iff A is a subset of B, but not the same set as B.

(Just in case you haven't seen this abbreviation: "iff" means "if and only if". That is, "*blah* iff *zoom*" means the same thing as "if *blah*, then *zoom*, and if *zoom*, then *blah*".)

1.1.3 Example

(a) For any set A, A is a subset of A. (That is, $A \subseteq A$. We say $\subseteq$ is **reflexive**.)

(b) For any sets A, B, and C, if A is a subset of B, and B is a subset of C, then A is a subset of C. (That is, if $A \subseteq B$ and $B \subseteq C$ then $A \subseteq C$. We say $\subseteq$ is **transitive**.)

(c) $A = B$ iff A and B have exactly the same elements.

Proof

(a) Let A be a set. We want to show that $A \subseteq A$, which means that every element of A is an element of A. This is obvious: that is, for any $a \in A$, obviously $a \in A$. So we're done.

(b) Let A, B, and C be sets, and suppose that $A \subseteq B$ and $B \subseteq C$. We want to show that $A \subseteq C$. So suppose that a is any element of A; we want to show that $a \in C$. Since $A \subseteq B$, and we are supposing $a \in A$, it follows that $a \in B$. Then, since $B \subseteq C$, it follows that $a \in C$. So every element of A is an element of C, which means that $A \subseteq C$. □

To know what a set is, you just have to know what elements it has. No two *different* sets have the very same elements.

1.1.4 Technique (Proving sets are equal)

Say we have a set A and a set B, and we want to know whether they are the *same* set. (Remember—just because we are using different *labels*, it doesn't follow that they are labels for different *things*.) We can do this in two steps.

1. Show that every element of A is an element of B. That is, we come up with a subproof which begins "Let a be any element of A", and which ends "Therefore, a is an element of B".

2. Show that every element of B is an element of A. That is, we come up with a subproof which begins "Let b be any element of B", and which ends "Therefore, b is an element of A".

1.1.5 Example

If A and B are sets, then their **intersection** is a set $A \cap B$ whose elements are the things which are elements of *both* A and B. That is, to be explicit:

> For any object x, $x \in A \cap B$ iff $x \in A$ *and* $x \in B$.

A **union** of A and B is a set $A \cup B$ whose elements are the things which are elements of either one of the sets A or B (or maybe both). That is, to be explicit:

For any object x, $x \in A \cup B$ iff $x \in A$ *or* $x \in B$.

Show the following fact:

$$A \cap (A \cup B) = A$$

Proof
In order to show that the set $A \cap (A \cup B)$ is the same set as A, we need to show two things:

1. Every element of $A \cap (A \cup B)$ is an element of A.

   Let a be any element of $A \cap (A \cup B)$. By the definition of the intersection, this means that a is an element of A *and* a is an element of $A \cup B$. So obviously $a \in A$, and we're done.

2. Every element of A is an element of $A \cap (A \cup B)$.

   Let a be any element of A. To show that $a \in A \cap (A \cup B)$, we need to show that $a \in A$ and $a \in A \cup B$. The first part ($a \in A$) we already know. For the second part ($a \in A \cup B$), we need to show that $a \in A$ *or* $a \in B$. But since $a \in A$, that means the first case holds, and we're done. □

1.1.6 Exercise

Using the definitions from Example 1.1.5, show the following facts:

(a) If $A \subseteq B$, then $A \cup B = B$.
(b) If $A \cup B = B$, then $A \subseteq B$.
(c) For any set C, $C \subseteq A$ and $C \subseteq B$ iff $C \subseteq A \cap B$.

The reason that Technique 1.1.4 works is the following fundamental general principle about sets—which we call an *axiom*.

1.1.7 Axiom of Extensionality

If A is a subset of B and B is a subset of A, then A and B are the very same set. That is, $A = B$.

In other words, if A and B have the exact same elements—that is, every element of A is an element of B, and also every element of B is an element of A—then A and B are exactly the same set. Again, what this axiom tells us, practically, is that if we A and B are sets, and want to show that $A = B$, then we should first show that $A \subseteq B$, and next show that $B \subseteq A$.

The proof of Example 1.1.5 implicitly relies on the Axiom of Extensionality. So does any proof that uses Technique 1.1.4 to prove sets are equal.

1.1.8 Example

Suppose A is the set $\{1, 2, 3, 4, 5\}$. It's often useful to "separate out" some of the elements of this set into another set—such as the set containing just the *odd* elements of A, which is $\{1, 3, 5\}$. We can label this set

$$\{a \in A \mid a \text{ is an odd number}\}$$

Similarly,

$$\{a \in A \mid a \text{ is prime}\} = \{2, 3, 5\}$$

And the set

$$\{a \in A \mid a \text{ is greater than } 10\}$$

is the *empty* set, since no elements of A are greater than 10.

1.1.9 Example

For any sets A and B, there is a **set difference** of A and B, containing just those elements of A which are not in B. This is denoted $A - B$. The existence of this set follows from Separation. The difference of A and B is the set

$$\{a \in A \mid a \text{ is not an element of } B\}$$

Or more briefly:

$$A - B = \{a \in A \mid a \notin B\}$$

1.1.10 Technique (Defining a Subset)

Suppose we have a set A, and we want to come up with an example of a subset of A. We can do this using our special curly brace notation. We can write down:

Let $B = \{a \in A \mid \rule{1.5cm}{0.4pt}\}$.

Then we want to fill in the blank with some property. This should be a property that the elements of the subset we want have in common, and which no other elements of A have. Once we have written this down, we know that the elements of B are exactly the things which are elements of A that have the special property we wrote in the blank. That is, we know that every element of B is an element of A that has the special property. We also know that any element of A that has the special property is an element of B.

Again, the reason this technique works is because of another general principle about sets.

1.1.11 Axiom of Separation

Let F be any property. For any set A, there is a set B such that, for any thing a: a is an element of B iff a is an element of A and $F(a)$. The special notation we use for this is

$$B = \{a \in A \mid F(a)\}$$

There is something tricky about the Axiom of Separation. I have stated it in terms of *properties*. But this raises lots of philosophical questions. What are properties? Are there any such things? What are they like? This also raises technical questions: what exactly counts as a legitimate application of the Axiom of Separation?

There are standard alternative ways of stating the Axiom of Separation that either avoid or answer these philosophical questions (though there is more than one way to do it, and occasionally they give different answers to the question of "what counts"). But these alternative ways of making the Axiom of Separation more precise rely on concepts and tools that come much later in this text. For practical purposes, we can go ahead and use this intuitive version for now, with our intuitive sense of "properties", and postpone the difficulties until we're better equipped to handle them. Still, we can look ahead a bit to at least understand the general idea.

One way of restating the Axiom of Separation is as an *axiom schema* (which we will discuss in Section 5.4). Instead of a single axiom, what we really have are infinitely many different axioms: every single way of replacing $F(a)$ with some precise statement about a gives you a different axiom. One of these axioms says:

> For any set A, there is a set B such that, for any thing a: a is an element of B iff a is an element of A and a **is a non-empty set**.

Another axiom says:

> For any set A, there is a set B such that, for any thing a: a is an element of B iff a is an element of A and a **is an odd number**.

And so on. Every way of filling in the blank gives you another axiom. These are called *instances* of the Axiom Schema of Separation. But to make this idea totally precise, we would need to be more precise about what counts as an *instance*. What things can you write down in the blank and get a legitimate axiom? You can get paradoxes if you're not careful about this. To state the rules carefully, we'll need a theory of "things you can write down"—that is, a theory of *linguistic expressions*. We'll be working on that soon (particularly in Section 2.5, Section 3.2, and Chapter 5).

A second way of restating the Axiom of Separation uses what's called *second-order logic* (Chapter 9). Second-order logic lets us "generalize in predicate position." Putting it roughly, instead of just saying "anything", we can also say "anyhow." For any *way for a thing to be*, there is a subset of A that includes just the elements of A that are that way.

But we are getting way ahead of ourselves. For now, while we should note that there are philosophical and technical subtleties here, we should be able to get by using our intuitive understanding of properties.

1.1.12 Exercise
Let U be a set, and let A and B be subsets of U. Use "separation notation" (Technique 1.1.10) to show that there is a set which is a union of A and B, and that there is a set which is an intersection of A and B. (Use the definitions of "union" and "intersection" in Example 1.1.5.)

Here's another basic principle about sets.

1.1.13 Empty Set Axiom
There is a set with no elements. This is called the **empty set**. It is labeled $\varnothing$ or $\{\}$.

(In fact, we could equivalently have used the simpler axiom that *there is a set*. Then we could use Separation to conclude that there is a set with no elements: the set $\{a \in A \mid a \neq a\}$.)

1.1.14 Example
There is a *unique* set with no elements. (This justifies us in calling it "*the* empty set" rather than "*an* empty set".)

Proof
What we need to show is that if A and A' are *both* empty sets—that is, if A and A' each have no elements—then $A = A'$. We show this in two steps.

1. Every element of A is an element of A'.

   This is true because A has no elements! A counterexample would have to be an element of A which is *not* an element of A', and clearly there are no such things.

2. Every element of A' is an element of A.

   This works just the same way, this time because A' has no elements.

So $A = A'$ (by Extensionality). □

1.1.15 Technique (Existence and Uniqueness)
When we need to show that there is *exactly one* F, or (in other words) that *there is a unique* F, it's usually helpful to break this up into two steps.

1. **Existence.** We show that there is *at least one* F.
2. **Uniqueness.** We show that there is *at most one* F.

The Uniqueness part means that for any x and y which are both F's, x and y are the very same thing. So to prove there is at most one F, this is a good strategy. *Suppose* that x is F and y is F; then *show* that $x = y$.

1.2 Functions

Every building in Los Angeles has an address: a certain sequence of numbers and letters that labels that building, like `3709 Trousdale Parkway`. To keep track of the relationship between buildings and addresses, we can consider an **address function**, which we'll call "address". For each building b, address b is its address. Functions are useful throughout logic, because we are often interested in relationships like this one: for example, the relationship between things in the world and the words that we use to label them.

Here's another example: for every number, there is another number which immediately follows it. Zero is followed by one, one by two, and so on. We can represent this relationship between numbers using a function, which is called the **successor function**, and which we'll call suc for short. For each number n, there is a number suc n which is one more than n.

In general, suppose that A and B are sets. A **function from A to B** assigns an element of B to each element of A. For every element $a \in A$, there is some element of B which is the result of **applying** f to a. This is labeled $f\,a$. So for every $a \in A$, $f\,a \in B$.

(Some people write function application with lots of extra parentheses, always writing $f(a)$ rather than $f\,a$. But I won't do that unless things would be unclear otherwise.)

1.2.1 Notation

The notation $f : A \to B$ means that f is a function from A to B. We call A the **domain** of f, and B the **codomain** of f.

For example, the domain of the address function is the set of Los Angeles buildings, and its codomain is the set of all strings of symbols. The domain and the codomain of the successor function suc are both the set of natural numbers $\{0, 1, 2, \dots\}$.

(Sometimes functions are defined to be certain special sets—for instance, as sets of ordered pairs—see Section 1.6 for some discussion. But we won't bother with that for now. We'll just treat functions as another basic kind of thing alongside sets.)

We should distinguish the codomain from another thing. Not every string of symbols is the address of some building: there is no building in Los Angeles with the address `00000 Main Stresjkkj` . So there are elements of the codomain of the address function—which I said was the set of all strings of symbols—which are not actually "hit" by the function. We say that the string `00000 Main Stresjkkj` is not in the *range* of the address function.

1.2.2 Definition

The **range** of a function $f : A \to B$ is the set of elements of B that are assigned by f to some element of A. That is,

$$\text{range}\, f = \{b \in B \mid \text{for some } a \in A, f\,a = b\}$$

(What is the range of the successor function?)

We have three different ways of coming up with functions: the *list* method, the *rule* method, and the *relation* method.

1.2.3 Example

Let A be the set $\{1, 2, 3\}$ and let B be the set of cities in California. Then we can define a function $f : A \to B$ by specifying the value of f for each element of A. For instance, we could say

$$f(1) = \text{Los Angeles}$$
$$f(2) = \text{San Diego}$$
$$f(3) = \text{San Jose}$$

This definition of f uses the *list* method.

We can also use some handy alternative notation for this kind of function definition:

$$[\quad 1 \mapsto \text{Los Angeles}, \quad 2 \mapsto \text{San Diego}, \quad 3 \mapsto \text{San Jose} \quad]$$

You can read this notation as saying "the function that takes 1 to Los Angeles, 2 to San Diego, and 3 to San Jose."

Another way we could define a function $g : A \to B$ is using a *rule*:

$$g(n) = \text{the } n\text{th largest city in California (in 2015)} \quad \text{for every } n \in \{1, 2, 3\}$$

1.2.4 Technique (Coming up with a function using an explicit list of values)
If a set A is finite, we can define a function $f : A \to B$ by listing out its values one by one.

1.2.5 Technique (Coming up with a function using a rule)
This is the most common way of coming up with a function $f : A \to B$: we precisely describe what "output" value f should have for each "input" value a in A. For example, we can define the function from numbers to numbers that takes each number to the number six more than it:

$$f\,n = n + 6 \quad \text{for each number } n$$

In general, to come up with a function from A to B, first choose a name, like "f" (though we would might want to use something more descriptive, like "address" or "suc" or "greatest-prime-divisor"). Then write down

$$f(a) = \underline{\qquad} \quad \text{for every } a \in A$$

Fill in the blank with a description of an element of B, where this description can depend on a. For example "$a + 6$", "a's great-grandfather", or "the first letter of a" are all fine descriptions (though which of them make sense will depend on what kind of thing a is—that is, on what the elements of A are).

The third way to come up with a function is the trickiest. Instead of giving an explicit *rule* for what the value $f\,a$ should be, we can describe a general *relationship* that each input should have to their output. Here are some examples.

- Every building in Los Angeles has an address: that is, for every building b in Los Angeles, there is a string of symbols which is an address for b. So there is a *function*, address, from buildings in Los Angeles to strings of symbols such that, for each building b, address b is an address for b.
- Every number has a successor: that is, for every number n, there is some number that comes immediately after n. So there is a *function*, suc, from numbers to numbers such that, for each n, suc n comes immediately after n.

- For every number n, there is some number which is two more than n. So there is a function f from numbers to numbers such that, for every number n, $f n$ is two more than n.

- For every building b in Los Angeles, there is a person within one mile of b. So there is a function g from buildings to people such that, for each building b, gb is a person within one mile of b.

- For every non-empty set of numbers A, there is some number n which is an element of A. So there is a function h from non-empty sets of numbers to numbers such that, for each non-empty set of numbers A, $hA \in A$.

Notice that the first three examples, each "input" in the domain stands in the relationship to *just one* object in the codomain. A building only has one address, and a number only has one successor, and likewise only one number which is two more than it. But the last two examples aren't like that. In the last two examples, we didn't *uniquely* describe the output of the function for each input. But we did describe what relationship it should stand in to the input, and that is enough.

1.2.6 Technique (Coming up with a function using a relation)

Say we want to come up with a function from A to B. We can do this by describing a relationship that should hold between the inputs and outputs of the function. First, choose a name for the function, like f. Then write down this:

> For each $a \in A$ and $b \in B$, if $f a = b$, then _______________.

Fill in the blank with some relationship that should hold between a and b. You're not quite done: in order to conclude that there is a function like this, you first need to *show* this:

> For each $a \in A$, *there is some* $b \in B$ such that _______________.

Here you fill in the blank with the *same* relation that you wrote down in the first blank. Once you have shown this, then you're good to go: you can conclude that *there is a function* that satisfies the relation you wrote down.

Notice that there's another way of writing down the key property of your function f, which is a little more concise. Instead of saying

> For each $a \in A$ and $b \in B$, if $f a = b$, then $F(a, b)$.

(where $F(a, b)$ is the relationship between a and b that you want) you could write this equivalent thing instead:

For each $a \in A$, $F(a, f\,a)$

This is just another more concise way of saying that the relation F holds between each input a and its corresponding output $f\,a$. (Of course, what you actually write down depends on which relation F stands for.)

The "rule" method for defining a function is really just a more specific version of the "relation" method. If we define the function f using a rule

$f(a) = \_\_\_\_\_$ for each $a \in A$

Then we can think of this as defining f using a *relation*, like this:

For each each $a \in A$ and $b \in B$, if $f(a) = b$ then $b = \_\_\_\_\_$.

Here we fill in the blank with the very same description that we used for our rule.

Furthermore, the "list" method for defining functions is really just a more specific version of the "rule" method. For example, we can think of our function

$$[\quad 1 \mapsto \text{Los Angeles}, \quad 2 \mapsto \text{San Diego}, \quad 3 \mapsto \text{San Jose} \quad]$$

as defined by the three-part rule

$$f(n) = \begin{cases} \text{Los Angeles} & \text{if } n = 1 \\ \text{San Diego} & \text{if } n = 2 \\ \text{San Jose} & \text{if } n = 3 \end{cases}$$

So in fact, all three of these techniques for coming up with functions are versions of the "relation" method, and they are all based on the same fundamental principle. We can state the basic general principle like this.

1.2.7 Axiom of Choice
Let F be a relation. Let A and B be sets. If for every $a \in A$ there is some $b \in B$ such that $F(a, b)$, then there is a function $f : A \to B$ such that for every $a \in A$, $F(a, f\,a)$.

This principle is also tricky in the same way as the Axiom of Separation. I stated it in terms of a *relation* F. But again, it would be hard work to give a theory of relations, in addition to our theory of sets and functions. Again, one way to make this principle precise is to think of it as an axiom *schema*: we get a different axiom for each way of replacing $F(a,b)$ with a precise description of a relationship between a and b. To make this fully precise, we would have to spell out the rules for what statements about a and b you are allowed to "plug in" for $F(a,b)$. We can do that using ideas that come later in the text. But in practice, we can get by with common sense.

As it happens, Choice is more controversial than the rest of standard set theory, for a couple of reasons. First, Choice has some very surprising consequences when it comes to infinite sets. One famous example is the Banach-Tarski Theorem: you can use Choice to prove that a unit sphere can be divided into four pieces that can be rigidly rearranged to form *two* unit spheres, each exactly like the original. Second (and relatedly), unlike the other standard axioms of set theory, Choice is *non-constructive*. Choice tells us that there are functions that we have no way of describing uniquely. This challenges the philosophical idea that mathematical objects are things that we mentally "construct" in some sense.

These controversies are entirely about the kind of applications of Choice where we know that for every a there is *at least one b* such that $F(a,b)$, but we *don't* know that for every a there is *exactly one b* such that $F(a,b)$. For most of what we do in this text, we could get by with a restricted "Axiom of Function Existence," which only applies in the "exactly one" case. But a few things in this text really do rely on the full power of "at least one" Choice. In this text, I don't bother to carefully distinguish the things that rely on full-fledged Choice from the things that don't.

Now let's turn to one more basic technique for working with functions.

1.2.8 Example

Recall the functions f and g from Example 1.2.3, which take the numbers in $\{1,2,3\}$ to cities in California. As it happens, though we used different definitions, f and g are the very same function. The largest city in California is Los Angeles, the second largest is San Diego, and the third largest is San Jose. The functions f and g have the same "output" for every "input." Furthermore, there is no more to a function than what "output" it has for each "input"—it doesn't matter how this relationship is *described*. So f and g are the same function.

1.2.9 Technique (Proving Functions are Equal)

If f and g are functions from A to B, and we want to show that $f = g$—that is, we want to show that f and g are the same function. (Again, remember that just because we are using two different *names* for the function, that doesn't mean f and g are different *functions*!) To show this, we need to show that f and g have the same "output" for each possible "input". The proof usually goes like this:

> Let a be any element A. [Fill in reasoning.] So $fa = ga$. Therefore, $f = g$.

Once again, the reason this strategy works is because of a fundamental general principle about functions—another axiom.

1.2.10 Axiom of Function Extensionality

For any functions $f : A \to B$ and $g : A \to B$, if $f a = g a$ for every $a \in A$, then $f = g$.

Extensionality and Choice work together to tell us what functions are like. Choice guarantees that there are *enough* functions, and Extensionality guarantees that there are *not too many* functions.

1.2.11 Exercise

Suppose $\mathbb{2}$ is a set with exactly two elements, which we'll call True and False. We can think of functions to $\mathbb{2}$ as "tests", which say True for things that pass the test and False for the rest.

If X is a subset of A, we can define a function from A to $\mathbb{2}$ which we call the **characteristic function** of X, or char $X : A \to \mathbb{2}$ for short. Intuitively, this is the function that says whether something is an element of X. For every $a \in A$,

$$(\text{char } X)a = \begin{cases} \text{True} & \text{if } a \in X \\ \text{False} & \text{otherwise} \end{cases}$$

We can also go the other way around. If $f : A \to \mathbb{2}$ is a function, we can define a subset of A that includes just the things that pass the f-test. This is called the **kernel** of f, or ker f.

$$\ker f = \{a \in A \mid f a = \text{True}\}$$

(a) Show that for any set $X \subseteq A$,

$$\ker(\text{char } X) = X$$

(b) Show that for any function $f : A \to \mathbb{2}$,

$$\text{char}(\ker f) = f$$

Here's a special feature of the address function: no two buildings have exactly the same address. (Maybe this isn't quite true, since there can be more than one building on the same lot. But let's ignore this complication.) In other words, for any two different buildings b and b', address b and address b' are two different strings. Or to put that the other way around, for any buildings b and b', if address $b =$ address b', then $b = b'$. A function like this is called *one-to-one*: it never takes two or more inputs to one output.

On the other hand, as we noted earlier, there are many different strings of symbols which are not addresses of any building at all, like `alfkj/404.html`. We say that this function is not *onto*: its range does not completely "cover" the set of strings of symbols.

1.2.12 Definition

(a) A function $f : A \to B$ is **one-to-one** (or *injective*) iff for each $a, a' \in A$, if $fa = fa'$ then $a = a'$.

(b) A function $f : A \to B$ is **onto** (or *surjective*) iff for each $b \in B$ there is some $a \in A$ such that $fa = b$

(c) A function $f : A \to B$ is a **one-to-one correspondence** (or *bijective*) iff it is both one-to-one and onto.

Here's another way of putting this. The elements of the *domain* of a function are its "possible inputs," and the elements of the *codomain* of a function are its "possible outputs." Each possible input results in some possible output. For a *one-to-one* function, each possible output is the result of *at most one* possible input. For an *onto* function, each possible output is the result of *at least one* possible input. Thus, for a *one-to-one correspondence*, each possible output is the result of *exactly one* possible input.

1.2.13 Exercise

Give an example (other than the address function) of a function which is …

(a) One-to-one but not onto.

(b) Onto but not one-to-one.

(c) One-to-one and onto.

1.2.14 Exercise

(a) For any function $f : A \to B$, f is onto iff the range of f is B.

(b) For any sets A and B, there is a one-to-one function $f : A \to B$ iff there is a one-to-one correspondence from A to some subset of B.

1.2.15 Example
If $f : A \to B$ and $g : B \to C$ are each one-to-one, then there is a one-to-one function from A to C.

Proof
Let $f : A \to B$ and $g : B \to C$ be one-to-one functions. We can use these functions to define a function $h : A \to C$, like this:

$$h(a) = g(fa) \quad \text{for every } a \in A$$

Intuitively, we are chaining the two functions together: first apply f, then apply g. We need to check that the resulting function h is one-to-one. Let $a, a' \in A$, and suppose that $ha = ha'$. That is,

$$g(fa) = g(fa')$$

Since g is one-to-one, this tells us that

$$fa = fa'$$

Then since f is one-to-one, this tells us that

$$a = a'$$

This shows that for any a and a' in A, if $ha = ha'$, then $a = a'$, which means that h is one-to-one. □

1.2.16 Exercise

(a) If $f : A \to B$ and $g : B \to C$ are each onto, then there is an onto function from A to C.

(b) If $f : A \to B$ and $g : B \to C$ are each one-to-one correspondences, then there is a one-to-one correspondence from A to C.

1.2.17 Exercise
If B is a non-empty subset of C, and there is an onto function from A to C, then there is an onto function from A to B.

1.2.18 Definition

Let $f : A \to B$ and $g : B \to A$ be functions. We say g and f are **inverses** iff we have both

$$\begin{aligned} g(f(a)) &= a \qquad \text{for every } a \in A \\ f(g(b)) &= b \qquad \text{for every } b \in B \end{aligned}$$

That is, the two functions f and g undo each other.

1.2.19 Proposition

Every one-to-one correspondence has an inverse.

Proof

Suppose $f : A \to B$ is a one-to-one correspondence. That is, f is one-to-one and onto. Since f is onto, we know

> For each $b \in B$, there is some $a \in A$ such that $fa = b$.

Then by Choice, that means there is a *function* $g : B \to A$ such that,

$$f(gb) = b \quad \text{for every } b \in B$$

That tells us half of what we need to prove that g is an inverse of f. For the other half, suppose $a \in A$. Then, by the way we chose g, we have

$$f(g(fa)) = fa$$

Since f is one-to-one, this implies that $g(fa) = a$. So g is an inverse of f. □

1.2.20 Exercise

Every function that has an inverse is a one-to-one correspondence. That is, if $f : A \to B$ has an inverse $g : B \to A$, then f is a one-to-one correspondence.

The previous two facts, taken together, tell us that a function is a one-to-one correspondence *if and only if* it has an inverse. That means that, from now on, you can think of "function with an inverse" as if it was an alternative *definition* of "one-to-one correspondence," right there alongside the official definition "function which is one-to-one and onto." When you are doing any exercise involving one-to-one correspondences, you can unpack that in either of these two equivalent ways. You should go straight to whichever version makes the exercise easier to solve.

1.2.21 Exercise

For any sets A and B, if there is a one-to-one correspondence from A to B, then there is also a one-to-one correspondence from B to A.

It will sometimes be helpful to have two slightly more general versions of this fact about inverses to work with. If a function has an inverse, it is one-to-one and onto. But we can also split this up into a part for one-to-one functions, and a part for onto functions.

The first idea is about one-to-one functions. If we want to show that a certain function picks out a *unique* representative for each element of a set A, we can do this by showing how to "undo" the function: we can define a function that takes each representative to the specific thing in A that it represents.

1.2.22 Definition

A **partial function** from A to B is a function whose domain is a subset of A, and whose codomain is B.

We sometimes call a function from A to B a **total function** in order to emphasize that it is not merely a partial function.

If f is a partial function from A to B then we say f is **defined** for a iff a is in the domain of f. (We also say "$f a$ is defined" or "f has a value for a".)

1.2.23 Definition

Let $f : A \to B$ be a function, and let g be a partial function from B to A. We call g a **partial inverse** of f iff

$$g(f(a)) = a \quad \text{for every } a \in A$$

Intuitively, g "undoes" the function f.

1.2.24 Example

A function f is one-to-one iff f has a partial inverse.

Proof

Suppose $f : A \to B$ is one-to-one. Then f also puts A in one-to-correspondence with the range of f, which is a subset of B. So there is an inverse function $g : B' \to A$ such that

$$g(f(a)) = a \quad \text{for every } a \in A$$

That is, g is a partial inverse of f.

For the other direction, suppose g is a partial inverse of f, and let $a, a' \in A$, and suppose $f(a) = f(a')$. In that case,

$$a = g(f(a)) = g(f(a')) = a'$$

So f is one-to-one. □

The second idea is about *onto* functions. If a function is onto, then it is a way

1.2.25 Example
A partial function g from B to A is onto iff there is some function $f : A \to B$ such that g is a partial inverse of f.

Proof
The proof is again very short and simple, but rather abstract.

For the left-to-right direction, suppose g is a partial function from B to A which is onto. We want to show that we can "flip g around," taking things in A back to things in B that g takes to them. Since g is onto, we know:

> For each $a \in A$ there is some $b \in B$ such that $g(b) = a$.

Then using Choice, we know also know there is a *function* $f : A \to B$ such that

$$g(f(a)) = a \quad \text{for every } a \in A$$

That is, g is a partial inverse of f.

For the other direction, suppose that g is a partial inverse of f. To show g is onto, we need to show that for each $a \in A$ there is some $b \in B$ such that $gb = a$. In fact, this is given to us by f: for each $a \in A$, we know fa is an element of B such that $g(fa) = a$. □

1.2.26 Exercise
The following are equivalent:

(a) There is a one-to-one correspondence between A and some subset of B.
(b) There is a one-to-one function from A to B.
(c) There is an onto partial function from B to A.

1.2.27 Exercise

`TODO. Add an exercise showing that a function is one-to-one, using a partial inverse.`

1.3 Ordered Pairs

Sometimes we want to work with functions with more than one input (or more than one output). For example, addition takes *two* numbers m and n and spits out a single number $m + n$. One way to approach this would be to work out a whole separate theory of “multiple-input functions” in addition to the “single-input functions”—but that would end up repeating lots of work. A nicer way to do it is to think of a function that takes *two* numbers as its input as really being a function that takes one thing, a *pair* of numbers, as its input. That is, addition is a function from pairs of numbers to numbers.

An *ordered pair* (a, b) is something whose *first element* is a, and whose *second element* is b. Unlike a set, the elements of a pair are *ordered* (as the name suggests). The ordered pair $(1, 2)$ is different from the ordered pair $(2, 1)$. In contrast, the *set* $\{1, 2\}$ is the very same thing as the set $\{2, 1\}$, because they have the same elements.

1.3.1 Axiom of Pairs

For any sets A and B, there is a set $A \times B$ whose elements are called **ordered pairs.** Each ordered pair in $A \times B$ has a **first element**, which is an element of A, and a **second element**, which is an element of B. For any $a \in A$ and $b \in B$, there is exactly one ordered pair whose first element is a, and whose second element is b. This pair is labeled (a, b).

1.3.2 Exercise

Let $\mathbb{0}$, $\mathbb{1}$, and $\mathbb{2}$ be sets with 0, 1, and 2 elements, respectively. How many elements do the following sets have? Explain your answers.

(a) $\mathbb{1} \times \mathbb{2}$

(b) $\mathbb{2} \times \mathbb{0}$

(c) $(\mathbb{2} \times \mathbb{2}) \times \mathbb{2}$

1.3.3 Exercise
Show that for any sets A and B, there is a one-to-one correspondence between $A \times B$ and $B \times A$.

1.3.4 Exercise
For any set A, the **diagonal** of $A \times A$ is the set of all ordered pairs of the form (a, a). That is, it's the set

$$\{(a_1, a_2) \in A \times A \mid a_1 = a_2\}$$

Show that for any set A, there is a one-to-one correspondence between A and the diagonal of $A \times A$.

1.3.5 Exercise
For any sets A, B, and C, there is a one-to-one correspondence between $A \times B) \times C$ and $A \times (B \times C)$.

1.3.6 Exercise
For any function $f : A \to B$, there is a set of ordered pairs in $A \times B$ called the **graph** of f: this is the set of pairs

$$\{(a, b) \mid f\,a = b\}$$

Suppose that X is a subset of $A \times B$. Say that X is **functional** iff, for each $a \in A$, there is exactly one $b \in B$ such that (a, b) is in X. Show that X is functional iff X is the graph of some function from A to B.

1.4 Higher-Order Sets and Functions

Sets and functions become more powerful when we start to consider sets of *sets*, and sets of functions, and functions whose inputs and outputs are themselves sets or functions. These are "higher-order" sets and functions. Let's start with a simple example.

1.4.1 Axiom of Power Sets
For any set A there is set of all subsets of A. This is called the **power set** of A, or PA for short. In other words, for every B,

$$B \in PA \quad \text{iff} \quad B \subseteq A$$

1.4.2 Example

The power set of $\{0, 1\}$ is the four-membered set

$$\{\{\}, \{0\}, \{1\}, \{0, 1\}\}$$

1.4.3 Exercise

(a) For any set A, there is a one-to-one function from A to PA.

(b) For any non-empty set A, there is an onto function from PA to A.

Similarly, it is often useful to consider *sets of functions*.

1.4.4 Axiom of Functions

For any sets A and B, there is a set containing every function from A to B. This set is labeled B^A, or $A \to B$.

1.4.5 Exercise

For each function $f : A \to B$, the *range* of f is a subset of B: the set of elements of B which are equal to $f\,a$ for some $a \in A$ (Definition 1.2.2). In other words, for each function $f \in B^A$, there is a set

$$\text{range}\, f \in PB$$

So this defines a higher-order *function*

$$\text{range} : B^A \to PB$$

Is the range function one-to-one? Is it onto? Justify your answers.

1.4.6 Exercise

Suppose A and B are sets. If A is not empty, then there is a one-to-one function from B to B^A.

1.4.7 Exercise

In Exercise 1.2.11 we defined the *characteristic function* for a subset $X \subseteq A$ to be a certain function char $X : A \to \mathbb{2}$. So this defines a higher-order function:

$$\text{char} : PA \to \mathbb{2}^A$$

Show that this function is a one-to-one correspondence.

1.4.8 Example
For any set A, there is a one-to-one correspondence from $A^{\mathbb{2}}$ (the set of functions from a two-element set to A) to $A \times A$ (the set of ordered pairs of elements of A).

Proof
Let's call the two elements of $\mathbb{2}$ "1" and "2". The rough idea is that being given a value for 1 and a value for 2 amounts to the same thing as being given two values, in order, which amounts to the same as being given an ordered pair of values.

First we will define a function from $A^{\mathbb{2}}$ to $A \times A$, and then we will show that it is one-to-one and onto. We can define a function f as follows:

$$fh = (h1, h2) \quad \text{for each function } h : \mathbb{2} \to A$$

To show that f is one-to-one, suppose that h and h' are each functions from $\mathbb{2}$ to A, and $fh = fh'$. That is, $(h1, h2) = (h'1, h'2)$. That means that these ordered pairs have the same first element and the same second element, so $h1 = h'1$ and $h2 = h'2$. Since 1 and 2 are the only elements of $\mathbb{2}$, this shows that h and h' have the same output for every input. So by Function Extensionality, $h = h'$. So f is one-to-one.

To show that f is onto, suppose that (a, a') is any element of $A \times A$. We want to show that there is some element h in $A^{\mathbb{2}}$ such that $fh = (a, a')$. And there is: we can let

$$h = [1 \mapsto a, \quad 2 \mapsto a']$$

Then $fh = (h1, h2) = (a, a')$, which is what we wanted. □

1.4.9 Exercise
Let $\mathbb{1}$ be a set with exactly one element. For any set A, there is a one-to-one correspondence from A to $A^{\mathbb{1}}$.

1.4.10 Exercise
Let A, B, and C be any sets. There is a one-to-one correspondence between $C^{A \times B}$ (the set of two-place functions from $A \times B$ to C) and $(C^B)^A$ (the set of *higher-order* functions from A to functions from B to C).

(Applying this one-to-one correspondence is called "currying" a function, or sometimes “Schönfinkelizing” it, after the two people who independently discovered it, Curry and Schönfinkel.)

1.5 The Uncollectable

It is very useful to collect together some things into a single thing—a set. This move from *many things* to a *single set* is very natural, and you might think that it just amounts to an entirely innocent redescription. What difference could there be between there being some people, and there being a *set* of people?

But things are not so straightforward. Not every way of picking out *some things* picks out *a set.* Some things cannot be “gathered together” in a single set. This is weird! It might not be feasible to bring some things into the same *room*, because there are too many of them or they are too big or too far apart. But putting some elements into a set isn’t like putting them in a room. For example, there is a set containing the Moon, the Eiffel Tower, every finite string of letters which does *not* appear in any book in the Library of Congress, and the number seven. If there are sets as gerrymandered and arbitrary as this, it is hard to imagine what could possibly prevent some things from all going together in a set. We are faced with a very puzzling and counterintuitive result.

The proof that some things don’t form a set is very short, but it uses a clever trick. As we have discussed, sets can be elements of “higher-order” sets. Can a set be an element of *itself*? We have not said anything about this one way or the other. Normally this is ruled out by an extra axiom, called the Axiom of Foundation, which says (in part) that set membership can never go around in a circle. We won’t need the Axiom of Foundation for the purposes of this text. But the clever trick uses the idea of trying to “feed a set to itself.”

1.5.1 Exercise (Russell’s Paradox)
Call X a **self-member** iff X is an element of X. Call X a **non-self-member** iff X is not a self-member. Show that there is no set that contains all and only the non-self-members.

(Note that the Axiom of Foundation would imply that *all* sets are non-self-members. But we don’t have to use this axiom to do this problem.)

1.5.2 Exercise
(a) There is no set of all sets.

(b) There is no *universal set* containing everything there is.

These limits on set-formation are the reason why we stated the Axiom of Separation in the careful way that we did. The Axiom of Separation lets us *cut down* a set using some description of its elements. For example, if we start with a set of all the numbers, then we can cut it down to a set of all the odd numbers. But Separation doesn't let us *build up* a set from some arbitrary description. For example, we can't write down

$$\{x \mid x \notin x\}$$

and expect this to pick out a set—in fact, there is no such set, because of Russell's Paradox. Instead, if we start with a set A, we *can* write down

$$\{x \in A \mid x \notin x\}$$

By Separation, this will give us a set—the set of all elements of A which are not self-members. There is nothing paradoxical about this set. (In fact, assuming the Axiom of Foundation, which implies there are no self-members, this will just be the same set A that we started with.)

Russell's paradox kind of feels like a party trick. But it is really important, for two different reasons. First, it is foundationally important: the surprising fact that not just any things form a set, and in particular that there is no universal set, is crucial to understanding how set theory works at all. It makes a big technical difference to how we state our basic axioms, and a big philosophical difference to how we think about what sets are. Second, the devious trick we used in Russell's paradox is important—feeding sets to themselves in order to get a contradiction. This *self-application* trick (which is often called "diagonalization") will be used over and over again in this text: it shows up along the way to almost all of the central theorems in this course, about what is uncountable, inexpressible, undecidable, and unprovable. So it's worth going slowly to make sure we really understand what is going on here. While we're at it, we can also work out a more general lesson.

Let's consider an analogous puzzle.[2] A certain small English community in 1918 consists only of clean-shaven men (a Cambridge college, say). One of these clean-shaven men is a barber. Someone tells us:

> The barber shaves all the men in the community who do not shave themselves, and only them.

But this can't be true! Does the barber shave himself? If so, then he shaves someone who shaves himself, which contradicts what we are told. If not, then he fails to shave

[2] Russell (n.d., 101).

someone who does not shave himself, which again contradicts what we are told. So this situation is impossible (and not just because of the implausibility of Oxbridge social structure): *nobody* shaves just those people who do not shave themselves.

This was just the same reasoning as Russell's paradox, except instead of considering the relation "x is a an element of y", we considered the relation "x is shaved by y." Looking at it this way makes it clear that nothing depends on *which* relation we were talking about—there is a lesson here which is not specifically about either elements or shaving.

We can also redescribe things in terms of *functions,* instead of relations. Let A be the set of men in that community, and let f be the function that takes each person a to the set of people that a shaves. (It's fine if f takes some people to the empty set.) Call this "the shaving function." The shaving function is a function $f : A \to PA$, such that, for any people a_1 and a_2 in A,

$$a_2 \in f a_1 \quad \text{iff} \quad a_1 \text{ shaves } a_2$$

Now we can restate the lesson of the "barber paradox" in terms of the shaving function. What it told us is that there is a certain set of men X (namely, the set of men who do not shave themselves) such that nobody shaves just the men in X. Anybody who did that would be like the paradoxical barber, who shaves himself if and only if he doesn't shave himself, which is impossible. In short, there is no $a \in A$ such that $f a = X$. That means this set X is not in the *range* of the shaving function f. So what we have proved is that the shaving function is *not onto.*

This lesson has nothing special to do with *shaving.* Our reasoning applies equally well to *any* set A, and *any* function $f : A \to PA$. This general fact turns out to be very important.

1.5.3 Exercise (Cantor's Theorem)
For any set A, there is no onto function from A to PA.

Hint. Use the same reasoning as the barber paradox to show that for any function $f : A \to PA$, there is a set $X \subseteq A$ which is not in the range of f.

Notice that we can also use Cantor's Theorem to give an alternative proof that there is no set of all sets. Suppose there were a set S that contained every set. Then in particular every *subset* of S would also be in S. That means we would have $PS \subseteq S$. But then we could define an onto function from S to PS: for example, we could take each $X \in PS$ to itself, and if there are any sets left over we could take them to the empty set. This would contradict Cantor's Theorem.

In a moment we'll come back to why Cantor's Theorem matters. First, though, let's look at the proof idea—the self-application trick—from a couple other perspectives.

Here is a way of picturing the self-application trick that gives it the name "diagonalization." Take a simple example where the elements of the set A are just Al, Bea, and Cece. Then consider any function f from elements of A to subsets of A. Here is an example of a function like this:

$$\begin{array}{r l l l l}
\text{Al} \mapsto \{ & \text{Al}, & \text{Bea} & & \} \\
\text{Bea} \mapsto \{ & & & & \} \\
\text{Cece} \mapsto \{ & \text{Al}, & & \text{Cece} & \}
\end{array}$$

(The unusual spacing is just to keep the same elements visually lined up for each set.) To show that f is not onto, we need a rule for finding a subset of A which is not in the range of f. This rule should work not just for this example, but for any choice of a set A and a function $f : A \to PA$.

We can represent this function in a slightly different way. As we showed in Exercise 1.2.11, we can describe a subset $X \subseteq A$ by answering a series of True-or-False questions: for each element of A, we just need to know whether or not it is in X. So we can draw a picture of this particular function f by listing the answers to these questions:

	Al	Bea	Cece
Al ↦	True	True	False
Bea ↦	False	False	False
Cece ↦	True	False	True

As before, the rows of this diagram correspond to the "inputs" to the function f, which are just the elements $a \in A$. The list of Trues and Falses in row a correspond to the "output" set $f a$, represented by its characteristic function. Row Al, column Cece says False, because Cece is not an element of $f(\text{Al})$. Row Cece, column Al says True, because Al is an element of $f(\text{Cece})$. In general, at any row a_1 and column a_2, the table says True if $a_2 \in f a_1$, and it says False if $a_2 \notin f a_1$. (Make sure you see why this table matches the definition of f given above.)

We are trying to come up with a recipe that, given any table like this, gives us a set X that is not represented by any row of the table. To do that, it's enough to guarantee that for each row, X disagrees with the table for at least one True-or-False question in that row. That is, for each row a_1, there is some element $a_2 \in A$ such that either a_2 is in $f a_1$ but not in X, or else a_2 is in X but not in $f a_1$.

Here's a trick that accomplishes this: we can work our way down the *diagonal* of the table: row Al, column Al; row Bea, column Bea; row Cece, column Cece. If we make sure that our set X doesn't match any of these, then X is different from every row of the table in at least one place. That is, we can make sure that X disagrees with the Al-row about Al, and disagrees with the Bea-row about Bea, and disagrees with the Cece-row about Cece. Since the diagonal says "True, False, True", we just want to flip this and say "False, True, False". That is, we can let X be the set which does not contain Al, which does contain Bea, and which does not contain Cece—that is, in this case it's the set $\{\text{Bea}\}$.

In general, the diagonal of this table tells us, for each $a \in A$, whether a is an element of $f\,a$. The trick is to consider the set that "flips" the diagonal, by including a iff a is *not* an element of $f\,a$. This brings us back to the very same Russell-style set we considered before: the set $\{a \in A \mid a \notin f\,a\}$, the set of elements which are not among the things "shaved by" themselves.

Here's one more perspective on Russell and Cantor's arguments, which connects them to an idea we will explore more deeply later (Section 6.6). The Liar Paradox is about the sentence

```
This sentence is not true.
```

Call this sentence L. Since apparently what L says is just that L is not true, it seems that

$$L \text{ is true} \quad \text{iff} \quad L \text{ is not true} \tag{*}$$

Is L true? If so, then by (*) L is not true, which is a contradiction. Alternatively, if L is not true, then by (*) L is true, so again we have a contradiction. Since we have a contradiction either way, we have a paradox.

Russell's Paradox and Cantor's Theorem have the same structure as a variant of the Liar Paradox, called "Grelling's paradox." (In particular, unlike the original Liar, this variant does not depend on a self-referential sentence.) Adjectives, like `short` , `interesting` , or `simple` , are words that can be truly applied to some things but not others. (Let's ignore the problems of vagueness for now, and pretend that all of these adjectives are perfectly precise.) The *extension* of an adjective is the set of things that it truly applies to. So if A is the set of adjectives and D is some set of things, then the *extension function* for D is a function $f : A \to PD$ that takes each adjective $a \in A$ to the set of things in D that a truly applies to.

Words are themselves among the things that adjectives can apply to: for instance, `red` is a short word, so the adjective `short` applies to `red` . Furthermore, `short` is a short word, so `short` applies to itself; `long` is not a long word, so `long` does not apply to itself. Some adjectives self-apply, and others don't. Now consider *the set of all adjectives which do not self-apply.* Is there any adjective which has this set as its extension? Suppose there were such an adjective: in particular, suppose that the adjective `non-self-applying` applies just to those adjectives which do not self-apply. For all adjectives a:

$$\texttt{non-self-applying} \text{ applies to } a \quad \text{iff} \quad a \text{ does not apply to } a \qquad (**)$$

Does `non-self-applying` self-apply? If so, then it applies to some adjective which self-applies—namely `non-self-applying` itself—contradicting the assumption. If not, then it fails to apply to some adjective which does not self-apply—again, `non-self-applying` itself—again contradicting the assumption.

(This reasoning is just like the "barber paradox"; but unlike the "barber paradox", this case seems genuinely paradoxical, like the Liar: after all, `non-self-applying` is an expression we can understand, that applies to some adjectives, like `short` and not others, like `long` . So what else could its extension be, if not the set of adjectives which do not self-apply? This is a hard philosophical problem.)

Let f be the extension function for A, which takes each adjective in A to the set of adjectives that a truly applies to. So the set of adjectives that do not self-apply is the set

$$X = \{a \in A \mid a \notin fa\}$$

The reasoning we just went through shows that there is no adjective a such that $fa = X$. Once again, f is not onto. Not every set of things is the extension of some adjective. This is a simple version of a kind of reasoning that will be very important later on when we consider limits on what can be expressed in language and what can be proved.

Another reason that Cantor's Theorem is important is that it underlies a very useful technique called "counting arguments." We will return to this in Chapter 4, but for now let's look at a few examples of the basic idea. Intuitively, Cantor's Theorem can tell us when two sets are mismatched in a way that prevents one set from "covering" or "representing" all of another.

1.5.4 Example (Undefinable Sets)

Let S be a set of *strings of symbols*. Let A be a set of *descriptions*, and suppose that each description is a string. (That is, $A \subseteq S$.) Suppose that for any pair of a description $d \in A$ and a string $s \in S$, either d is *true of* s or else it is not. A set of strings X is *definable* iff there is some description $d \in A$ such that d is true of each string $s \in X$ and d is not true of any string $s \notin X$. Otherwise, X is *undefinable*. Show that there exists at least one undefinable set of strings.

Proof
The basic idea is that if every set was definable, then there would be an onto function from strings to sets of strings. Consider the function that takes each description to the set of strings it is true of. That is, for each description $a \in A$, let

$$f a = \{s \in S \mid a \text{ is true of } s\}$$

Suppose for contradiction there are no undefinable sets of strings: that is, every set of strings is definable. This means that $f : A \to PS$ is onto. But A is a subset of S, so PA is a subset of PS. So Exercise 1.2.17 tells us that there is an onto function from A to PA. This contradicts Cantor's Theorem. □

1.5.5 Exercise (Undecidable Sets)
Let S be a set of *strings*. Let P be a set of *programs.* Each program is a string in S. The set of strings S also includes two different strings `True` and `False`.

We can *run* a program with any given *input* string, and the program may or may not *output* a result string. That is, we have a three-place relation *running program A with input string s has output string t.*

A set of strings X is *decidable* iff there is some program A such that both of the following hold:

(a) For each string $s \in X$, running program A with input string s has output string `True`.
(b) For each string $s \notin X$, running program A with input string s has output string `False`.

If there is no program A like this, then X is *undecidable*. Show that there is at least one undecidable set of strings.

Hint. Consider the function that takes each program A to the set of all strings s such that running A with input s has output `True`.

1.5.6 Exercise (Kaplan's Paradox)
Let P be a set of *propositions*, and let W be a set of *possible worlds*. We'll consider two relations between propositions and possible worlds. First, a proposition can be *true at* a possible world. Second, a proposition p can be the only proposition that anyone believes at w; in this case we say that *w singles out p*.

We'll make two assumptions about these relations. First, for any set X of possible worlds, there is some proposition p_X which is true at each possible world in X, and which is not true at any possible world which is not in X. Second, no world singles out more than one proposition.

Given these assumptions, show that there is at least one proposition which is not singled out by any possible world. In other words, some proposition cannot possibly be uniquely believed.

Hint. Consider the function that takes each world w that singles out some proposition p to the set of worlds at which p is true.

1.6 Simplifications of Set Theory*

UNDER CONSTRUCTION.

We have introduced many different principles about sets as "axioms". But these principles are not all *independent* of one another. In fact, we can prove some of these principles from others. This allows us to reduce the number of assumptions that our reasoning relies on.

A closely related point is that we have treated several different kinds of objects as "sui generis": sets, ordered pairs, and functions were each introduced separately, and each as a kind of thing to be understood on its own terms. But in fact, there are ways of "constructing" some of these things from others. This allows us to simplify our abstract *ontology*.

One tricky point is that there is more than one way to do this—and the different ways of doing it provide us *different* pictures of our primitive ontology and basic assumptions. So if we are taking seriously the question of which of these kinds of objects (sets, or functions, or pairs) is *fundamental*, and which of these principles about them is really a fundamental *axiom*, then we have many different choices available. It isn't obvious how we would choose between them.

There is one choice of axioms which at least has the weight of historical tradition behind it. This axiomatization is called "Zermelo-Fraenkel Set Theory with Choice", or **ZFC**, after two of its main discoverers (Ernst Zermelo and Abraham Fraenkel) and one of its main distinctive axioms (the Axiom of Choice). I'll briefly sketch here how this goes and how it can be used to derive the other axioms I've mentioned in this chapter. (For now, though, I'll be setting aside the distinctive issues arising for *infinite* sets. We'll discuss this in the next chapter.) This way of presenting set theory is so common that it is what many people mean by "set theory" or "standard set theory". But after that, I'll also say a little about a different axiomatization of set theory, called the "Elementary Theory of the Category of Sets" or **ETCS**, which was developed more recently.

ZFC uses only one primitive kind of object, which is a *set*, and the basic relation of being an *element* of a set.

(One tricky point worth noticing is that ZFC is standardly written in a way that presupposes that *everything* is a set. For instance, the standard way of writing the Axiom of Exensionality says "For any x and y, if x and y have exactly the same elements, then $x = y$." But suppose that *I* am not a set, and so I have no elements. Then this version of the Axiom of Extensionality implies that I am identical to the empty set, since we both have exactly the same elements—none at all. The same would go for you, or Jupiter, or anything else that has no elements. There is a standard way of fixing this up, and it is called ZFCU, where the U stands for "urelements": things which are not sets, but are elements of sets. I won't be fussy about the distinction, and in this section I'll keep calling this theory "ZFC", even though that isn't quite historically accurate.)

ZFC has five axioms that we have already discussed, one we will discuss in the next chapter (the Axiom of Infinity) and three additional axioms that we won't need to use in this course. Here are the four familiar axioms:

Empty Set Axiom. There is a set with no elements.

Axiom of Extensionality. For any sets A and B, if every element of A is an element of B, and every element of B is an element of A, then A and B are the very same set.

Axiom of Separation. For any set A, there is a set whose elements are just those elements a of A such that $F(a)$.

(As we noted earlier, this is a *schematic* axiom: $F(a)$ can be replaced with any precise description of a.)

Axiom of Power Sets. For any set A, there is a set of all subsets of A.

Axiom of Choice. Let A and B be sets. Suppose that for each element $a \in A$, there is some element $b \in B$ such that $F(a, b)$. Then there is a function $f : A \to B$ such that, for each $a \in A$, $F(a, f a)$.

(This is also schematic: $F(a, b)$ can be replaced by any precise description of a and b.)

But there is something important to notice about the last one here, the Axiom of Choice. This is an axiom about *functions*. But functions are not a basic concept in ZFC. So in order to make sense of the Axiom of Choice, we have to say what "function $f : A \to B$" *means* (as well as "$f\,a$"). The standard way to do this uses the idea from Exercise 1.3.6: every function can be represented by a *graph*, which is a functional set of ordered pairs. In ZFC, we simply define the word "function" to *mean* "functional set of ordered pairs." In other words, ZFC uses this definition:

1.6.1 Definition

A **function** from A to B is a set of ordered pairs $f \subseteq A \times B$ such that for each $a \in A$ there is exactly one $b \in B$ such that $(a, b) \in f$. For $a \in A$, we let $f\,a$ stand for the unique $b \in B$ such that $(a, b) \in f$.

This pushes the problem back a bit. But note also that *ordered pair* is not a basic concept in ZFC. So we also have to say what "$A \times B$" and "(a, b)" are supposed to mean in this definition. The standard way to do this uses a clever trick. We can use unordered *sets* to represent ordered pairs. Of course, we can't just represent (a, b) with the set $\{a, b\}$. If we did that, then (a, b) and (b, a) would be represented by the very same set, which isn't what we want. Here's the trick: we can instead represent the ordered pair (a, b) with the set $X = \{\{a\}, \{a, b\}\}$. The two elements of the pair, a and b, are guaranteed to play different "roles" within X (unless $a = b$). The set X has just one element Y that is itself a set with only one element; the unique element of Y is the first element of the pair, a. If X has an element Z which has two elements, then just one element of Z is different from a, and this is the second element of the pair, b. But X might not have any element with two elements: in this case, X represents the pair (a, a).

1.6.2 Definition

For any a and b, let the **ordered pair** (a, b) be the set $\{\{a\}, \{a, b\}\}$.

The reasoning above shows that each ordered pair has a unique first element, and a unique second element: that is, for any ordered pairs (a, b) and (a', b'), if $(a, b) = (a', b')$, then $a = a'$ and $b = b'$. We can also prove that for any sets A and B, there is a set containing all ordered pairs (a, b) such that $a \in A$ and $b \in B$; but this actually relies on some of the other axioms of ZFC we haven't introduced yet. Once we prove that, this justifies using the notation $A \times B$ to denote this set of pairs.

So this shows that we can define ordered pairs and functions just in terms of sets and elements. Note that if we use these definitions, we don't have to take the Axiom of Pairs, the Axiom of Functions, or Function Extensionality as extra *axioms*. In fact, we can use the definitions (and the other axioms we just listed, plus one more below) to prove these facts as *theorems*. For example, there is a set of all functions from A to B, because there is a set of all functional subsets of $A \times B$: this follows from the Axiom of Power Sets and the Axiom of Separation. So there is something nice and economical about this approach. By using the right definitions, we have cut down both the how many undefined primitive concepts we are taking for granted, and also how many unproved basic assumptions we are taking for granted.

But this approach raises some hard philosophical questions. Is this really what a function *is*—a set of ordered pairs? If so, why think that a function $f : A \to B$ is a subset of $A \times B$, rather than a subset of $B \times A$? And similarly, is an ordered pair really just a set? If so, why think it's the set we described above, rather than some other set that could do the same job? These definitions look *arbitrary*.

These are philosophically important questions: we'd like to understand the nature of abstract objects, what things like functions and ordered pairs really are. But they aren't *technically* important questions. For the purposes of proving the theorems that come later, all we really care about is whether there is *something or other* that plays the role of functions, and something or other that plays the role of pairs. What we care about is whether there is *some* way of understanding "function" (and "$f\,a$") such that principles like Function Extensionality, the Axiom of Functions, and the Axiom of Choice come out true. The ZFC definitions are good enough for this. It doesn't really matter for the theorems if there is *more than one* way of understanding these principles that makes them come out true.

This kind of issue comes up over and over. What are numbers really? What are sequences, or strings, or sentences, or programs, or proofs? It's not clear how to answer these questions. But very often, for our purposes it's enough to find *something or other* that has the right structural features to play the *role* of numbers, sequences, strings, and so on.

ZFC also has three extra axioms we don't really need in this course—they guarantee "wide enough" and "deep enough" sets, and that sets have a nice hierarchical structure: if you take elements of elements of elements … you always eventually reach a bottom level of things without any more elements (which are either the empty set or urelements).

Axiom of Union. For any set X, there is a set $\bigcup X$ such that, for any a, $a \in \bigcup X$ iff there is some $A \in X$ such that $a \in A$.

Axiom of Replacement. Let A be a set, and suppose that for each element $a \in A$, there is exactly one b such that $F(a, b)$. Then there is a set B such that, for any b, $b \in B$ iff for some $a \in A$, $F(a, b)$.

Axiom of Foundation. For any non-empty set A, there is some element $a \in A$ such that a and A have no elements in common.

TODO. More discussion?

The final axiom is Infinity, which we will discuss in the next chapter.

TODO. Add a short overview of the ideas of ETCS.

1.7 Review

Key Techniques

- You can show that A and B are **the same set** in two steps:

  1. Show that every element of A is an element of B.
  2. Show that every element of B is an element of A.

- You can show that $f : A \to B$ and $g : A \to B$ are **the same function** by showing that $f a = g a$ for every element $a \in A$.

- You can **come up with a subset** of a set A by precisely stating a property of elements of A. By **Separation**, there is a subset $B \subseteq A$ whose elements are all and only the elements of A that have that property.

- You can **come up with a function** from A to B by precisely describing a relation between elements of A and elements of B such that each element of A stands in that relation to at least one element of B.

Key Concepts and Facts

- If A and B are **sets**, A is a **subset** of B ($A \subseteq B$) iff every **element** of A is an element of B.

- If f is a **function** from a set A to a set B (written $f : A \to B$) then f maps each element a in A to some element $f a$ in B.

- An **ordered pair** is something with a **first element** and a **second element.** For each $a \in A$ and $b \in B$, there is exactly one ordered pair in the set of ordered pairs $A \times B$ whose first element is a and whose second element is b. This ordered pair is labeled (a, b).
- A function $f : A \to B$ is **one-to-one** iff f maps *at most one* element of A to each element of B.
- A function $f : A \to B$ is **onto** iff f maps *at least one* element of A to each element of B.
- A function is a **one-to-one correspondence** iff it is both one-to-one and onto.
- The **power set** of A (called PA for short) is the set of all subsets of A.
- The **function set** B^A is the set of all functions from A to B.
- For any set A, there is a one-to-one correspondence between PA and $\mathbb{2}^A$. That is, we can put *subsets* of A in one-to-one correspondence with True-or-False "test functions" on A.
- **Russell's Paradox.** Some things don't form a set at all. In particular, there is no "universal set" containing every set.
- **Cantor's Theorem.** There is no onto funtion from a set A to its set of subsets PA.

Chapter 2

The Infinite

> Now it is the same thing to say this once and to keep saying it forever.
>
> Zeno (c. 490 – c. 430 BCE) as reported by Simplicius (c. 490 – c. 560 CE), ‘On Aristotle’s Physics’

In this chapter we’ll explore some important infinite sets—especially the set of *numbers* and the set of *strings.* Infinite set have some striking and counterintuitive properties. This can be delightful, if you have the taste for it, but you might worry that they are too far removed from practical experience to be important, and you might suspect that lessons we draw from infinity for our ordinary language and reasoning are insecure. Many philosophers and mathematicians have shared these worries and suspicions. But the infinite is very close to home.

We speak a language with finitely many words, and each sentence combines just finitely many of them. But it is possible to combine these words in ways no one else ever has in all of human history—and this will always be possible. Here’s a very simple example:

```
Snow is white.
Snow is white, and snow is white.
Snow is white, and snow is white, and snow is white.
```

We can go on this way indefinitely. It’s not as if there is some finite stopping point, beyond which one one would lapse into unintelligibility. So there are infinitely many such sentences. Each sentence in English is a finite thing. But *all* the English sentences taken together form an infinite set. It's plausible that we'll only ever get around to writing down some small finite subset of this vast variety, since it's plausible that humanity (or at least written English) will only exist for a finite amount of time. But to understand the structure of our language and thought in general, as a whole, we will need to confront the infinite.

Infinity shows up everywhere in logic. Our standard logical languages are like English: though each sentence is made up of finitely many symbols, there are infinitely many different sentences that allow us to express infinitely many different ideas. There are likewise infinitely many different formal proofs, infinitely many different counterexamples to invalid arguments, infinitely many different systematic procedures for answering questions, and so on.

In this chapter we will get acquainted with some basic tools for working with infinity, which we will use over and over again in the following chapters.

2.1 Numbers and Induction

We begin with the simplest infinite set.

The **natural numbers** are the "finite counting numbers" starting from zero: $0, 1, 2, \dots$ and so on. (In this text, by "number" I will always mean "natural number.") We'll use the symbol $\mathbb{N}$ as a label for the set of all natural numbers. Let's start with some basic observations.

The numbers have a starting place: **zero**. (Starting from zero instead of one turns out to be convenient in lots of ways. But it does introduce some potential confusion, since this means the first number is zero, the second is one, the third is two, and so on. This can be a source of "off-by-one bugs," so be careful.)

Every number is immediately followed by another bigger number. This is called its **successor**. The successor of n is $n+1$. But as it turns out, the notion of a successor is conceptually more basic than the notion of addition, so it will be helpful to give it its own special notation: we'll write $\operatorname{suc} n$ for the successor of the number n. (Some people use the notation n' or Sn instead.) In fact, the notion of successor is even conceptually more basic than the notion of *one*. We can define one as the successor of zero. (So defining $\operatorname{suc} n$ as $n + 1$ would be circular.)

For every number n, $\operatorname{suc} n$ is a number. This means we have a function $\operatorname{suc} : \mathbb{N} \to \mathbb{N}$. This is called the **successor function**.

2.1.1 Definition

The number one is the successor of zero, two is the successor of one; three is the successor of two; and so on.

$$
\begin{aligned}
1 &= \mathrm{suc}\,0 \\
2 &= \mathrm{suc}\,1 = \mathrm{suc}\,\mathrm{suc}\,0 \\
3 &= \mathrm{suc}\,2 = \mathrm{suc}\,\mathrm{suc}\,\mathrm{suc}\,0 \\
&\vdots
\end{aligned}
$$

By taking successors over and over again, we eventually reach every number. We also never double back on the same numbers over again: taking successors gives us a new, bigger number every time. Every number can be reached in just one way by starting from zero and taking successors. This means that if we keep going from one number to the next, we are never going to end up at a number we've already seen before. The successor function doesn't have any "loops"—it just goes on and on to ever-bigger numbers. You can't ever take a successor-step and end up back at zero. You also can't ever take a successor-step and end up at a number which was *already* a successor of some earlier number. We can sum up this "no looping" condition as follows:

2.1.2 Injective Property

(a) Zero is not a successor of any number;
(b) No two numbers have the same successor.

We can put this another way using the terminology of functions:

(a) Zero is not in the range of the successor function;
(b) The successor function is one-to-one.

Here is a concise way of representing the structure of numbers: they are generated by the following two rules.

$$
\frac{}{0 \text{ is a number}} \qquad \frac{n \text{ is a number}}{\mathrm{suc}\,n \text{ is a number}}
$$

Here's how to read this notation. Each rule says: if we have everything above the line, then we can also get the conclusion below the line. The zero rule has nothing above the line, because we can conclude that zero is a number without relying on any further assumptions. The successor rule says that, for any *n*, if *n* is a number, then suc *n* is also a number. Every number can be reached in exactly one way by repeatedly applying these rules. (In the case of numbers, this notation doesn't really make things any clearer than what we've already said. But when we consider more complicated structures later on, this notation for "formation rules" will become more useful.)

Every number can eventually be reached by starting with zero, and repeatedly taking successors. This is the basic idea behind a fundamental technique—one of the basic tools we will use over and over again—which is called **proof by induction**. Let's start with an example.

(Note that this mathematical use of the word "induction" is different from the traditional philosophical meaning of "induction", which is a way of gaining empirical knowledge by generalizing from past observations. The kind of induction we're talking about here—"mathematical induction"—is really a kind of *deduction*.)

2.1.3 Example

No number is its own successor. That is, there is no number n such that $\operatorname{suc} n = n$.

What we want to show is that every number n has a certain property: namely, the property that $\operatorname{suc} n \neq n$. Let's call a number *nice* iff it has this property: that is, a nice number is a number which is not its own successor. We want to show that every number is nice.

How can we prove this? It would be nice if we could do it in steps, by proving each of the following things:

$$\begin{gathered} 0 \text{ is nice} \\ 1 \text{ is nice} \\ 2 \text{ is nice} \\ \vdots \end{gathered}$$

The trouble is that this would require a proof with infinitely many steps, which we have no hope of finishing before the semester ends. But let's go ahead and try.

For the first step, we show that *zero* is nice: that is, $\operatorname{suc} 0 \neq 0$. This is guaranteed by the Injective Property, which says that zero is not the successor of any number—including zero itself.

Next we show that one is nice: that is, $\operatorname{suc} 1 \neq 1$. Remember that one is defined to be the successor of zero: so what we want to show is that $\operatorname{suc}(\operatorname{suc} 0) \neq \operatorname{suc} 0$. We just proved that $\operatorname{suc} 0 \neq 0$, so this follows from the fact that suc is one-to-one: $\operatorname{suc} 0$ and 0 are different numbers, so their successors are also different.

Next we show that two is nice: that is, $\operatorname{suc} 2 \neq 2$. Two is defined to be the successor of one, so what we want to show is that $\operatorname{suc}(\operatorname{suc} 1) \neq \operatorname{suc} 1$. We just proved that $\operatorname{suc} 1 \neq 1$, so again this follows from the fact that suc is one-to-one.

But now we notice a pattern: the proof that two is nice worked exactly the same way as the proof that one is nice. And we could keep going in the same way: the fact $\operatorname{suc} 2 \neq 2$ also implies that $\operatorname{suc} 3 \neq 3$, and this then implies that $\operatorname{suc} 4 \neq 4$, and so on. In other words, it looks like we can use the same pattern of reasoning to show each of the following things:

$$\begin{aligned}
&\text{if 0 is nice then suc 0 is nice}\\
&\text{if 1 is nice then suc 1 is nice}\\
&\text{if 2 is nice then suc 2 is nice}\\
&\vdots
\end{aligned}$$

Furthermore, we can prove each of these things in the very same way. So what we can do is write down a *general* argument that covers all of these cases at once. That is, instead of proving these conditionals one by one, we can prove the following *general* fact:

> For every number n, if n is nice, then $\operatorname{suc} n$ is nice.

Here is the proof. Let n be a number, and suppose that n is nice: that is, $\operatorname{suc} n \neq n$. We want to show that $\operatorname{suc} n$ is nice. That is, we want to show:

$$\operatorname{suc}(\operatorname{suc} n) \neq \operatorname{suc} n$$

The Injective Property says that the successor function is one-to-one. So our assumption that $\operatorname{suc} n$ and n are different numbers implies that they also have different successors. This is exactly what we wanted to show: $\operatorname{suc} n$ is nice.

To sum up, we showed two things.

1. 0 is nice.
2. For every number n, if n is nice, then $\operatorname{suc} n$ is nice.

Together, these two steps guarantee that *every* number is nice. Why does this follow? Well, in the first step, we showed that zero is nice.

$$\text{0 is nice}$$

Then the second step tells us in particular:

$$\text{If 0 is nice, then 1 is nice}$$

Thus

$$1 \text{ is nice}$$

The second step *also* tells us:

$$\text{If 1 is nice, then 2 is nice}$$

Thus

$$2 \text{ is nice}$$

And the second step *also* tells us:

$$\text{If 2 is nice, then 3 is nice.}$$

Thus

$$3 \text{ is nice}$$

And obviously we can keep going, using the same two steps to show that 4 is nice, and 5 is nice, and so on. By taking successors over and over, eventually we reach every number. So by applying our second step, "if n is nice, then $\operatorname{suc} n$ is nice", over and over again to larger and larger numbers n, eventually we can show that *any* given number is nice. So every number is nice.

That is the key idea of proof by induction. In order to prove *infinitely* many things, it is enough to prove just two things. The first thing is called the *base case:* 0 is nice. The second thing is called the *inductive step*: every nice number is followed by another nice number.

Now that we've figured it out, let's write out the proof in a more concise, official way, which shows the standard structure of a proof by induction.

Proof
Call a number n *nice* iff $\operatorname{suc} n \neq n$. We will prove by induction that for every number is nice.

Base case. By the Injective Property, 0 is not a successor, so $\operatorname{suc} 0 \neq 0$. That is, 0 is nice.

Inductive Step. We will show that for any number n, if n is nice then $\operatorname{suc} n$ is nice. Let n be any number, and suppose n is nice: that is, $\operatorname{suc} n \neq n$. By the Injective Property, the successor function is one-to-one, so this implies $\operatorname{suc}(\operatorname{suc} n) \neq \operatorname{suc} n$. This means that $\operatorname{suc} n$ nice. □

Let's do another example.

2.1.4 Example

Every number is either zero or a successor. That is, for any number n, either $n = 0$ or else there is some number m such that $n = \operatorname{suc} m$.

Proof

We'll prove this by induction as well. We want to show that every number n has a certain property: the property of either being zero, or else being the successor of some number. For short, let's say a number n is *good* iff either $n = 0$ or else there is some number m such that $n = \operatorname{suc} m$. We want to show that every number is good. Again, we can do this in two steps.

For the *base case*, we'll show that zero is good. That is, either $0 = 0$ or else 0 is a successor. Obviously the first case is true.

For the *inductive step*, we'll show that goodness is inherited by successors: for any number n, if n is good, then the successor of n is also good. That is, we assume that n is either zero or a successor, and we want to show that $\operatorname{suc} n$ is either zero or a successor. Again, this is obvious, because obviously $\operatorname{suc} n$ is the successor of some number (namely n). □

Like before, the two parts of this proof together guarantee that every number is good. The base case tells us that zero is good. The inductive step tells us that, if zero is good, so is one. The inductive step also tells us that if one is good, so is two. And it tells us that if two is good, so is three. And going on this way, eventually we can reach any number, and show that it is good. So every number is good.

2.1.5 Technique (Proof by Induction)

We use proof by induction when we are trying to show that every number has a certain property. To do a proof by induction, start by clearly identifying the property.

> We want to show that for every number n, _______________.

Fill in the blank with some statement about n.

Once you've identified the key property, a proof by induction has two parts. The first step is to show that zero has the property. This step is called the **base case**. It is usually the easiest part of the proof. (But not always!)

The second step is to prove a certain universal conditional statement. You want to show, for every number n, *if* n has the property, *then* the successor of n also has the property. This is called the **inductive step**. Usually the inductive step will begin like this, where you fill in the blanks with the property you are trying to prove every number has:

> For the inductive step, let *n* be any number, and suppose that *n* is _______________. We want to show that suc *n* is also _______________.

Once you've done both steps, you're done. For in fact, every number is either zero, or else the successor of zero, or the successor of the successor of zero, or .... So by chaining together the conditional you proved in the inductive step some number of times, eventually you prove that every number has the property you wanted.

Until you get used to proof by induction, it can feel a little magical. In particular, the inductive step might seem like cheating. You are *assuming* that something has the property that you are trying to prove *everything* has. But this is okay! Of course it would be useless to prove "for any *n*, if *n* is nice, then *n* is nice". That would amount to a pointlessly circular argument. But that's not what you do in a proof by induction: instead, you prove "for any *n*, if *n* is nice, then *n*'s *successor* is nice". Proving this makes a real advance—an advance of exactly one step. The key insight involved in proof by induction is that the journey to *any* finite number at all is nothing more than many journeys of a single step, one after another.

We'll have lots more examples and opportunities for practice as we go. But first we'll need to introduce another concept, in the next section.

In fact, the validity of proof by induction is usually taken to be part of the *definition* of the natural numbers. The intuitive idea of the natural numbers is that every number can be reached by starting with zero and taking successors some *finite number of times*. This would obviously be circular as a definition of "finite number". But we can make this idea precise using the idea of induction. The key idea of proof by induction is that, for any property, if zero has it, and it is always inherited by successors, then *every* number has the property. There aren't any *infinite* natural numbers which are never reached by the process of repeatedly taking successors.

We don't have any precise theory of properties at this point, so to make this statement official, we'll talk about *sets* instead. So this is another way of putting the important fact about the natural numbers.

2.1.6 Inductive Property of Numbers

Let X be any set. Suppose that (a) 0 is in X, and (b) for each number *n* in X, the successor of *n* is also in X. Then X contains every number.

What this says is just that proof by induction *works*—in particular, induction works for the property of being an element of the set X. Part (a) says that the base case holds for the property of being an element of X; part (b) says that the inductive step also holds for this property. The Inductive Property says that if (a) and (b) both hold, then ("by induction") every number has this property.

We can put these ideas together to say exactly what we are assuming about what the natural numbers are like. These assumptions are called the **Peano Axioms**.[1]

2.1.7 Axiom of Numbers

There is a set $\mathbb{N}$, the set of all **(natural) numbers**. There is an element of $\mathbb{N}$ called **zero**, and a **successor** function $\operatorname{suc} : \mathbb{N} \to \mathbb{N}$. These have the following two properties.

(a) **Injective Property.**

  (i) Zero is not in the range of the successor function. That is, zero is not a successor of any number.

  (ii) The successor function is one-to-one. That is, no two numbers have the same successor.

(b) **Inductive Property.** Let X be any set. Suppose (i) $0 \in X$, and (ii) for each $n \in X$, the successor of n is also in X. Then X contains every number.

2.1.8 Exercise

In this exercise we'll explore the way that the Injective Property and Inductive Property each help pin down the structure of the numbers. Let A be a set, let z be an element of A, and let s be a function from A to A. We'll say A, z, and s have the *Injective Property* iff z is not in the range of s, and s is one-to-one. We'll say A, z, and s have the *Inductive Property* iff, for any set X, if (a) $z \in X$ and (b) for every element $a \in A$ which is in X, sa is also in X, then X includes every element of A.

(a) Give an example of A, z, and s that have neither the Inductive Property nor the Injective Property.

(b) Give an example of A, z, and s that have the Inductive Property, but not the Injective Property.

[1] There is really more than one collection of assumptions that is sometimes called "the Peano Axioms". An important thing about this way of putting the axioms is that they talk about *sets*. In Section 5.4 we'll encounter some other principles that are also called "the Peano Axioms," but which don't say anything about sets.

(c) Give an example of A, z, and s that have the Injective Property, but not the Inductive Property.

2.2 Recursion

> ‘Can you do Addition?’ the White Queen asked. ‘What’s one and one and one and one and one and one and one and one and one and one?’
> ‘I don’t know,’ said Alice. ‘I lost count.’
> ‘She can’t do Addition,’ the Red Queen interrupted.
>
> Lewis Carroll, *Through the Looking Glass* (1871)

Another fundamental technique we’ll use when working with inductive structures such as numbers and strings is *recursive definition*. This is very closely related to inductive proof. A proof by induction is a way of showing that a certain *property* applies to every number. A recursive definition is a way of coming up with a *function* that can be applied to every number. Let’s start with an example.

The **doubling** function takes each number n to the number $2 \cdot n$. That way of describing it assumes we already know how to multiply—but we haven’t officially said what multiplication is. In fact, we can define doubling in way that doesn’t depend on already understanding multiplication—using a recursive definition. We do this in two steps. The two steps are exactly analogous to the two steps in an inductive proof.

First (for the *base case*) we say what the doubling function does to zero. This is easy: the the double of zero is zero.

$$\text{double}\,0 = 0$$

Second (for the *recursive step*) we let n be an arbitrary number, and we *suppose* that we already know how to double n. *Given* this assumption, we say how to to double $\text{suc}\,n$. That is, we suppose that we know $\text{double}\,n$, and we say what $\text{double}(\text{suc}\,n)$ should be in terms of that. For this, we can use the fact that $2 \cdot (n+1) = 2 \cdot n + 1 + 1$. So this is a reasonable rule to use:

$$\text{double}(\text{suc}\,n) = \text{suc}\,\text{suc}\,\text{double}\,n$$

Once we’ve done both of these steps, this is enough to settle what the doubling function does to every number. For example, let’s calculate $\text{double}\,3$ using these rules. We know $3 = \text{suc}\,2$, and $2 = \text{suc}\,1$, and $1 = \text{suc}\,0$. So we can work it out like this:

$$\begin{aligned}
\text{double}\,0 &= 0\\
\text{double}\,1 &= \text{double}(\text{suc}\,0)\\
&= \text{suc}\,\text{suc}(\text{double}\,0)\\
&= \text{suc}\,\text{suc}\,0\\
\text{double}\,2 &= \text{double}(\text{suc}\,1)\\
&= \text{suc}\,\text{suc}\,\text{double}\,1\\
&= \text{suc}\,\text{suc}\,\text{suc}\,\text{suc}\,0\\
\text{double}\,3 &= \text{double}(\text{suc}\,2)\\
&= \text{suc}\,\text{suc}\,\text{double}\,2\\
&= \text{suc}\,\text{suc}\,\text{suc}\,\text{suc}\,\text{suc}\,\text{suc}\,0\\
&= 6
\end{aligned}$$

We have successfully calculated that twice 3 is 6! And it's clear that we can keep going this way, using the result for 3 to get the result for 4, and using the result for 4 to get the result for 5, and so on. By applying the recursive rule over and over again, we eventually reach a value for any number. (But it will take longer and longer to get results for bigger and bigger numbers.) What makes this work is the basic fact about numbers: we can reach every number in exactly one way, by starting from zero, and repeatedly taking successors.

Here's another example.

2.2.1 Definition

Let k be a number. We can recursively define the function that *adds* k to any number. For any number n, we can write the result of this function as $k + n$. For the base case:

$$k + 0 = k$$

For the recursive step, we suppose we already know the result of $k+n$, and we then define the next step, which is the result of adding k to $\text{suc}\,n$.

$$k + (\text{suc}\,n) = \text{suc}(k + n)$$

In this way we can recursively define addition for any two numbers, in terms of the successor function and zero.

We can use the definition of addition to prove something that we've been taking for granted: the successor function is the same thing as adding one.

2.2.2 Example

For any number n, $\operatorname{suc} n = n + 1$.

Proof

Remember that $1 = \operatorname{suc} 0$. So:

$$\begin{aligned} n + 1 &= n + \operatorname{suc} 0 && \text{by the definition of 1} \\ &= \operatorname{suc}(n + 0) && \text{by the recursive step of the definition of } + \\ &= \operatorname{suc} n && \text{by the base case of the definition of } + \end{aligned}$$

□

So from now on, we can go ahead and use either the notation $\operatorname{suc} n$ or the notation $n+1$ equally well: they both mean the same thing. For example, this is an equivalent way of rewriting the recursive definition of addition:

$$\begin{aligned} k + 0 &= k \\ k + (n + 1) &= (k + n) + 1 \end{aligned}$$

2.2.3 Exercise

Use the definition of addition to explicitly show the following:

(a) $1 + 1 = 2$.

(b) $k + 2 = \operatorname{suc} \operatorname{suc} k$, for any number k.

(Remember that 1 is defined to be $\operatorname{suc} 0$ and 2 is defined to be $\operatorname{suc} 1 = \operatorname{suc} \operatorname{suc} 0$.)

Recursive definitions and inductive proofs very often work hand in hand. Often we use recursion to *define* a function, and then we use induction to *prove* that it does what it's supposed to do. Let's look at some examples of this sort of argument.

2.2.4 Example

Prove by induction that $0 + n = n$ for every number n.

(Note that this *doesn't* just follow directly from the first clause of the recursive definition of $+$: that definition tells us about $n + 0$, not $0 + n$, and we haven't shown yet that those are the same thing. Don't worry—we'll show this very soon.)

Proof
We want to show that every number n has the property that $0 + n = n$. The *base case* of the proof is to show that 0 has this property: that is, $0+0 = 0$. This follows immediately from the first clause of the recursive definition of addition.

For the *inductive step*, we want to show that the property is inherited by successors. For this, we'll let n be an arbitrary number, we'll suppose that n has the property, and we'll need show that $\operatorname{suc} n$ has the property as well. That is, for an arbitrary number n, we'll *suppose* that $0 + n = n$, and try to *show* that $0 + \operatorname{suc} n = \operatorname{suc} n$. We can show this using the recursive step of the definition of addition.

$$0 + \operatorname{suc} n = \operatorname{suc}(0 + n) = \operatorname{suc} n$$

(The first equation uses the recursive step of the recursive definition. The second equation uses the *inductive hypothesis*, that $0 + n = n$.) □

2.2.5 Example

Prove that $1 + n = n + 1$ for every number n.

Proof
We'll prove this by induction. For the base case, we need to show that $1+0 = 0+1$. In fact, by the definition of addition, we know $1 + 0 = 1$. And by the previous exercise, we know $1 = 0 + 1$. So the base case is done.

For the inductive step, we *suppose* that $1 + n = n + 1$. (This is the *inductive hypothesis*.) Then we want to *show* that $1 + \operatorname{suc} n = (\operatorname{suc} n) + 1$.

$$\begin{aligned} 1 + \operatorname{suc} n &= \operatorname{suc}(1 + n) && \text{by the definition of addition} \\ &= \operatorname{suc}(n + 1) && \text{by the inductive hypothesis} \\ &= \operatorname{suc}\operatorname{suc} n \end{aligned}$$

The last step uses the fact we showed earlier, that taking the successor of a number is the same as adding one to it: so we know that $\operatorname{suc}(\operatorname{suc} n) = (\operatorname{suc} n) + 1$. That finishes the proof. □

2.2.6 Example

Addition is **associative**: $(k + m) + n = k + (m + n)$ for any numbers k, m, n.

Proof
We'll show by induction that every number n has the property that, for any numbers k and m, $(k + m) + n = k + (m + n)$.

For the base case:

$$(k + m) + 0 = k + m = k + (m + 0)$$

This applies the base case of the inductive definition of addition twice.

For the inductive step, *suppose* $(k + m) + n = k + (m + n)$. We want to *show* that $(k + m) + \operatorname{suc} n = k + (m + \operatorname{suc} n)$.

$$\begin{aligned}
(k + m) + \operatorname{suc} n &= \operatorname{suc}((k + m) + n) && \text{definition of } + \\
&= \operatorname{suc}(k + (m + n)) && \text{inductive hypothesis} \\
&= k + \operatorname{suc}(m + n) && \text{definition of } + \\
&= k + (m + \operatorname{suc} n) && \text{definition of } +
\end{aligned}$$

□

Note a common structural feature of these proofs. In each example, the base case of the *proof* uses the base case of the recursive *definition* of addition. Similarly, in each example the inductive step of the proof uses the recursive step of the definition of addition. This is usually how this kind of proof goes.

With a bit of practice, this kind of inductive proof should end up basically feeling like routine symbol-juggling. Conceptually, the most important part is how to *set up* a proof by induction. Figure out what you need to show, in order to do a proof by induction: identify what property you want to prove every number has (“for every number n, n is nice”), and carefully spell out the base case (“0 is nice”) and the inductive step (“if n is nice, then $\operatorname{suc} n$ is nice”). The details of how you end up *showing* that each of these statements is true are not especially significant for these exercises, though it’s worth working through them to get the feel of it.

2.2.7 Exercise
Prove by induction that addition is **commutative**: $m+n = n+m$, for any numbers m and n.

2.2.8 Definition
We can recursively define *multiplication* of numbers. For any number m, we can define $m \cdot n$ recursively as follows:

$$\begin{aligned}
m \cdot 0 &= 0 \\
m \cdot \operatorname{suc} n &= m \cdot n + m
\end{aligned}$$

For example, let's work out $3 \cdot 2$.

$$\begin{aligned}
3 \cdot 0 &= 0 \\
3 \cdot 1 &= 3 \cdot \operatorname{suc} 0 \\
&= (3 \cdot 0) + 3 \\
&= 0 + 3 \\
&= 3 \\
3 \cdot 2 &= 3 \cdot \operatorname{suc} 1 \\
&= (3 \cdot 1) + 3 \\
&= 3 + 3
\end{aligned}$$

No surprises there.

2.2.9 Example

For any number n, $1 \cdot n = n$. (Again, notice that this isn't the same as the definition, because we haven't shown that $m \cdot n$ and $n \cdot m$ are the same thing.)

Proof
We will prove this by induction.

Base case. By definition, $1 \cdot 0 = 0$.

Inductive step. For the inductive hypothesis, we assume that $1 \cdot n = n$. We will show that $1 \cdot \operatorname{suc} n = \operatorname{suc} n$. In fact, by the definition of multiplication,

$$\begin{aligned}
1 \cdot \operatorname{suc} n &= 1 \cdot n + 1 && \text{by the definition of } \cdot \\
&= n + 1 && \text{by the inductive hypothesis} \\
&= \operatorname{suc} n && \text{by a fact we proved earlier}
\end{aligned}$$

□

2.2.10 Exercise

Show that double $n = 2 \cdot n$, using the recursive definition of the doubling function from the beginning of this section.

Now that we've seen a bunch of examples, let's describe this technique a bit more abstractly.

2.2.11 Technique (defining a function recursively)

Let's say you are trying to come up with a function whose *domain* is the set of all natural numbers: you have some other set A, and you want to come up with some function $f : \mathbb{N} \to A$. (You should fill in A with whatever the codomain of your function should be, and you should replace the letter f with some suitable name for the function you are defining, like `double` or whatever.) You can do this in two steps.

(a) Choose a *starting place*: figure out which value the function should have at zero. Write down:

$$f0 = \underline{\hspace{6em}}$$

Fill in the blank with some description of an element of your set A.

(b) Choose a *step rule*: figure out a general rule for how the value of your function for a number $n + 1$ should *depend* on its value for the previous number n. Again, you'll be filling in the blank:

$$f(n + 1) = \underline{\hspace{6em}}$$

(or alternatively, $f(\mathrm{suc}\, n) = \underline{\hspace{2em}}$, if you prefer to keep this notation). This time, though, you don't just have to describe an element of A out of nowhere. The thing you write down in the blank can use "fn". When you describe the value of f for $n + 1$, you get to assume that you already know fn, the value of f for the previous number n.

Once you've finished both steps, you're done. You will have successfully described what the function f should do guaranteed that there is one (and only one!) function

In Section 2.4 we'll redescribe this general technique more precisely, and we'll *prove* that it really works: if you do both of these two steps, you will have correctly described one and only one function.

2.2.12 Exercise

(a) Write down a recursive definition for the function that takes each number n to the sum of the numbers up to n. For example, if we call this function s:

$$\begin{aligned} s0 &= 0 \\ s1 &= 0 + 1 = 1 \\ s2 &= 0 + 1 + 2 = 3 \\ s3 &= 0 + 1 + 2 + 3 = 6 \\ &\vdots \end{aligned}$$

(b) Use your definition to prove by induction that for every number n,

$$sn + sn = n \cdot (n + 1)$$

2.3 Recursively Defined Properties and Relations

There are many different kinds of recursive definitions. So far we have discussed (so-called "primitive") recursive definitions of *functions.* Another useful thing to do is to recursively define *properties* or *relations.* Let's start with a simple example.

2.3.1 Definition
We can define the *even numbers* recursively, using the following rules.

$$\frac{}{0 \text{ is even}} \qquad \frac{n \text{ is even}}{n + 2 \text{ is even}}$$

Intuitively, what this definition means is that a number is even if and only if we eventually reach that number by applying these two rules finitely many times.

This definition tells us that we can use certain kinds of reasoning in our proofs. First, we can use the rules that we wrote down as part of the definition. At any point in our proofs, for any number n, we can write down "n is even," and justify this step "by Definition 2.3.1." And similarly, at any point in our proofs where we have *already* shown "n is even" we can then also infer "$n + 2$ is even" as well, with the justification "by Definition 2.3.5." For example, we know this:

2.3.2 Example
4 is even.

Proof
By the first part of Definition 2.3.1, 0 is even. By the second part of Definition 2.3.1, it follows that $0 + 2$ is even. By the second part of Definition 2.3.1 again, it then follows that $(0 + 2) + 2$ is even, and this is equal to 4. □

2.3.3 Example
For every number n, $n + n$ is even.

Proof
By induction on numbers.

Base case. $0 + 0 = 0$, which is even.

Inductive step. Suppose that $n + n$ is even. We will show that $(n + 1) + (n + 1)$ is even. In fact,

$$(n + 1) + (n + 1) = (n + n) + (1 + 1) = (n + n) + 2$$

Since $n+n$ is even, we know $(n+n)+2$ is even by the second part of the definition. □

Second, besides these basic rules, we also get to use a new kind of *proof by induction.*

2.3.4 Example

For every even number n, there is some number m such that $n = m + m$.

Proof

Say n is *nice* iff there is some number m such that $n = m + m$. We want to show that every even number is nice. We can prove this using a new proof method: *induction on the recursive definition* Definition 2.3.5. This inductive proof has two parts, one for each part of the recursive definition.

1. First we show that 0 is nice.

   This is true because $0 = 0 + 0$.

2. Next, let n be any number, and *suppose* n is nice. We will *show* that $n + 2$ is nice.

   Since n is nice, there is some m such that $n = m + m$. Thus:

   $$\begin{aligned} n + 2 &= (m + m) + (1 + 1) \\ &= (m + 1) + (m + 1) \end{aligned}$$

   So $n + 1$ is nice.

By induction, every even number is nice. □

In this proof, we showed that every even number is nice, by showing that both of the rules for coming up with even numbers in the recursive definition *preserve niceness:* they only take you from nice numbers to nice numbers. Why does this work? Well, intuitively, the recursive definition tells us that if n is even, then there is some sequence of steps, applying one of the two rules each time, that eventually gets us to n. The two parts of the inductive proof tell us that every time we take one of these steps, we always go from nice numbers to nice numbers. So for any even number n, we can chain together the two parts of the inductive proof some number of times to produce a proof that n is nice.

(Together, the last two examples tell us that the set of even numbers is the range of the doubling function.)

Here is another example. Instead of defining a *property* like evenness, we can recursively define a *relation.*

2.3.5 Definition

The relation m **is no greater than** n, abbreviated $m \le n$, is defined recursively by the following rules (for any numbers m and n):

$$\frac{}{n \le n} \qquad \frac{m \le n}{m \le\ n+1}$$

Again, intuitively what this definition means is that for any numbers m and n, we have $m \le n$ if and only if, by repeatedly applying these two rules finitely many times, we can eventually reach the concluion that $m \le n$.

As with the recursive definition of "even," this definition lets us use certain kinds of reasoning in our proofs. First, we can use the rules that we wrote down as part of the definition. At any point in our proofs, for any number n, we can write down $n \le n$. And at any point in our proofs where we have already shown $m \le n$, we can infer $m \le\ n+1$. For example, we know this:

2.3.6 Example

For each number n, $n \le n$. (That is, the *no greater than* relation is *reflexive.*)

Proof

By the first rule in Definition 2.3.5. □

2.3.7 Example

For each number n, $n \le\ n+2$.

Proof
Let n be any number. By Definition 2.3.5, we have

$$\begin{aligned} & n \leq n \\ \text{so} \quad & n \leq n+1 \\ \text{so} \quad & n \leq (n+1)+1 \\ & \quad = n+2 \end{aligned}$$

□

Second, besides these basic rules, we also get to use a new kind of *proof by induction.* Again, let's start with an example.

2.3.8 Example

For any numbers m and n such that $m \leq n$, either $m = n$ or $m + 1 \leq n$.

Proof
Say that a pair of numbers (m, n) is *nice* iff

either $m = n$ or $m + 1 \leq n$.

We are trying to show that every pair of numbers (m, n) such that $m \leq n$ is nice. We can prove this by *induction on the recursive definition* Definition 2.3.5. This inductive proof has two parts, one for each part of the recursive definition.

1. First, we show that for each number n, the pair (n, n) is nice. This is clear, because $n = n$.

2. Next, we show that for any numbers m and n, if (m, n) is nice, then $(m, n+1)$ is also nice. To show this, we *suppose* that either $m = n$ or $m + 1 \leq n$. We will then *show* that either $m = n + 1$ or $m + 1 \leq n + 1$. There are two cases.

   Suppose $m = n$. Then $m + 1 = n + 1$, and so $m + 1 \leq n + 1$ (by the first rule in Definition 2.3.5).

   Suppose $m+1 \leq n$. Then $m+1 \leq n+1$ by the second rule in Definition 2.3.5.

In either case, the pair $(m, n + 1)$ is nice. This completes the inductive proof. □

In this proof, we tried to show that whenever $m \leq n$, the pair (m, n) is nice. We did this by showing that each of the rules in the recursive definition of $m \leq n$ preserve niceness. Again, intuitively, whenever $m \leq n$, we can reach the conclusion that (m, n) is nice by chaining together the two parts of this proof some finite number of times.

2.3.9 Exercise

Prove by induction that for all numbers m, n, k, if $m \leq n$ and $n \leq k$, then $m \leq k$. (That is, the *no greater than* relation is *transitive.*)

Hint. There are two different options for how to set up your inductive proof. Things work out better if you try to prove that every pair of numbers $m \leq n$ has the following property:

> For every number k such that $n \leq k$, $m \leq k$.

Like other kinds of induction and recursion, recursively defined properties and relations can seem like magic. How is it that saying "by definition" lets us just start pulling statements like "$5 \leq 5$" or "if n is even then $n + 2$ is even" out of the air? In Section 2.4 we state a general result that basically says "recursive definitions of properties and relations *work*"—there always *is* some unique set (or set of pairs, or triples, etc.) which all these statements are true about, and such that proof by induction works. This proof is actually not very difficult, but it is pretty abstract. For now, let's just get the basic idea of how the magic works.

The set of even numbers E is supposed to have certain properties. First, it's supposed to have certain *closure* properties, given by the rules we wrote down.

- (i) $0 \in E$
  (ii) For each number $n \in E$, also $n + 2 \in E$.

For short, we can say that a set with these two properties is **closed.**

Second, E is supposed to have a certain *inductive property,* which tells us that proofs by induction on these rules work. We can write down the inductive property abstractly like this:

> Suppose X is a set of numbers such that (i) $0 \in X$, and (ii) for each number $n \in X$, also $n + 2 \in X$. Then every even number is in X.

We can put this more concisely:

> For any set X, if X is closed, then $E \subseteq X$.

So we can put those two properties together in the following pithy characterization of the set of even numbers: E is the *smallest closed set.* Here is another way of saying this:

2.3.10 Exercise
For any number n, n is even iff n is an element of *every* closed set.

This gives us a way to *define* the set of even numbers *explicitly,* in a way that guarantees that it also satisfies the properties specified in the *recursive* definition. We can say:

> A number n is **even** iff n is an element of every closed set.

We can straightforwardly check that *this* alternative definition of "even" guarantees that the even numbers have the properties we wanted for the recursive definition: the set of even numbers is *closed*, and it has the right inductive property.

All of this is done more carefully and generally in Section 2.4.

2.3.11 Technique (Recursively defining a property or relation)
We can recursively define a property P by writing down certain "derivation rules," like this:

$$\frac{x_1 \text{ has } P \qquad \dots \qquad x_n \text{ has } P}{\underline{\hspace{2.5em}} \text{ has } P}$$

where the blank is filled in with some description of another thing, which can depend on $x_1, \dots, x_n$.

Once we have written down this definition, we can use any of these rules we wrote down anywhere we want in our proofs.

We can also use the following kind of proof by induction. Suppose that we want to prove that *everything* with property P is *nice.* (Here *nice* is a placeholder for some other property.) We can do this by doing one step for each rule in our definition.

> *Suppose* that x_1 is nice, …, and x_n is nice. *Show* that _____ is nice.

(Here we fill in the description from the definition.)

We can do the same thing for *relations*, by writing down derivation rules for *pairs* (or n-tuples).

2.4 The Recursion Theorem*

So far we have relied on intuitive justifications for recursive definitions. In this section we'll back up this intuition by providing more precise proofs that recursive definitions work the way they are supposed to. This section is logically prior to the previous section: there we *assumed* that recursive definition is legitimate. Here we will *prove* it, providing justification for the claims we made before. So in this section we shouldn't rely on any of the things we proved in Section 2.2. We'll only be using the Axiom of Numbers.

We'll do this in two steps. We'll start with recursive definitions for properties and relations (or really, when we get more official here, *sets*). Then we'll talk about recursively defined *functions.*

Let's start by looking more abstractly at how recursive definitions of properties work. Informally we've talked about defining properties and relations. Officially, though, we'll do things in terms of defining a *set*, such as the set of even numbers, or the set of pairs (m, n) such that $m \le n$. In Section 2.2 we talked about *rules.* We can redescribe these rules in terms of *functions.* Consider the recursive definition of *even* again:

$$\frac{}{0 \text{ is even}} \qquad \frac{n \text{ is even}}{n + 2 \text{ is even}}$$

The first rule says that 0 is even. The second rule says that the *function* $n \mapsto n + 2$ *preserves* evenness. Similarly, consider our recursive definition of *no greater than.*

$$\frac{}{n \le n} \qquad \frac{m \le n}{m \le \ n + 1}$$

The first rule says that each pair (n, n) stands in the $\le$ relation, and second rule says that the *function* $(m, n) \mapsto (m, n + 1)$ *preserves* the $\le$ relation.

Let's generalize. (This is where things get a bit abstract.) Say an *operation* on a set A is either an element of A, or a function from A to A, or a two-place function from $A \times A$ to A, or a three-place function from $A \times A \times A$ to A, or .... If we count an element of A as a *zero*-place function, then we can say in general that an operation is an n-place function from A to A, for some n or other.

Suppose F is a set of operations on some set A. These operations correspond to the different rules of a recursive definition of a subset $X \subseteq A$. For each n-place operation $f \in F$, we can write down a rule:

If the set X is *recursively* defined by these rules, then, first, we should be able to use each of the rules in our proofs, and second, we should be able to do proof by induction on these rules. We can state state this carefully as follows.

2.4.1 Definition

Let F be a set of operations on some set A.

- A subset $X \subseteq A$ is **F-closed** iff for each n-place function $f \in F$, and for any elements $x_1, \dots, x_n \in X$, we also have $f(x_1, \dots, x_n) \in X$.
- A subset $X \subseteq A$ has the **F-inductive property** iff for any F-closed set Y, $X \subseteq Y$.
- A subset $X \subseteq A$ is **F-recursive** iff X is both F-closed and F-inductive.

In short, the F-recursive set is the *smallest* F-closed set. So to prove that recursive definitions work, we just have to show that whenever we write down a recursive definition of this kind, we have really described a unique set. This is the sort of fact which is difficult to *state* correctly, in the abstract, but once we have stated it, it really isn't very hard to prove.

2.4.2 Exercise
For any set of operations F on a set A, there is exactly one F-recursive set.

Hint. Put in this form, the inductive property basically tells us exactly what need to do: let

$$X = \{x \in A \mid x \text{ is in every } F\text{-closed set}\}$$

Check that X really is F-recursive.

Before we move on, here is a cool way of thinking about things. If F is a set of operations on A and X is a subset of A, we can let

$$F(X) = \{f(x_1, \dots, x_n) \mid f \in F \text{ is an } n\text{-place operation and } x_1, \dots, x_n \in X\}$$

This is the result of applying all the F-operations to things in X every way we can. For X to be F-closed just means $F(X) \subseteq X$. We can also check that if X is F-recursive, then $X \subseteq F(X)$.

2.4.3 Exercise
(a) For any sets $X \subseteq Y$, $F(X) \subseteq F(Y)$.
(b) If X is F-closed, then $F(F(X)) \subseteq F(X)$.
(c) If X is F-recursive, then $X \subseteq F(X)$.
(d) Therefore, if X is F-recursive,

$$F(X) = X$$

In other words, the set X is a *fixed point* of the F-operations. Applying those operations to X take you right back to the same set we started with. In particular, because of the inductive property, X is the *least fixed point* of F: it is a subset of every fixed point. This idea of a fixed point comes up in several important places in this text (see Section 6.6, Section 7.5), and as we have already glimpsed here, it is intimately connected to recursion and self-reference.

Now for recursively defined *functions*. As an example, recall the recursive definition we gave for the doubling function.

$$\begin{aligned} \operatorname{double} 0 &= 0 \\ \operatorname{double}(\operatorname{suc} n) &= \operatorname{suc}\operatorname{suc}(\operatorname{double} n) \quad \text{for each number } n \end{aligned}$$

This definition has two parts. The first part is a starting place: the value of double 0. The second part is a "step" rule, which tells us how to get from the value of double n to the value of double(suc n). We can represent the shape of this definition more abstractly like this:

$$\begin{aligned} \operatorname{double} 0 &= z \\ \operatorname{double}(\operatorname{suc} n) &= s(\operatorname{double} n) \quad \text{for each number } n \end{aligned}$$

The starting place is z, which in this case is the number 0. The step rule is given by the function s, which in this case is the function that takes each number m to suc suc m. In general, the element $z \in A$ and the function $s : A \to A$ correspond to what we write down in the two blanks when we use Technique 2.2.11 to recursively define a function.

The key fact about the natural numbers is that this always works. Given a starting point z, and a step rule s, there is always *exactly one* function on the natural numbers that they describe. We can put this a bit more precisely.

2.4.4 The Recursion Theorem

Let A be a set, let z be an element of A, and let $s : A \to A$ be a function. Then there is a unique function $f : \mathbb{N} \to A$ with these two properties:

$$\begin{aligned} f\,0 &= z \\ f(\operatorname{suc} n) &= s(f\,n) \quad \text{for each number } n \end{aligned}$$

Call these the Recursive Properties.

In the rest of this section we'll prove the Recursion Theorem. We have already justified recursively defined *relations*, and induction using such definitions—so we can use this technique in our proof. In particular, to get to the recursively defined function f, we'll start by defining a closely related relation. For a number n and an element $a \in A$, we define n **selects** a recursively as follow:

$$\frac{}{0 \text{ selects } z} \qquad \frac{n \text{ selects } a}{\operatorname{suc} n \text{ selects } sa}$$

What we still need to check is that this recursively defined *relation* picks out a *function.* In the terms from Exercise 1.3.6, we will check that the set of pairs (n, a) such that n selects a is *functional:* this means that each number selects exactly one value. Once we have done this, we can check that the corresponding function has the Recursive Properties.

2.4.5 Exercise
Prove each of the following by induction on the definition of "selects."

(a) If 0 selects a, then $a = z$

(b) If suc n selects a, then there is some $b \in A$ such that n selects b and $sb = a$.

Hint. You can restate (a) as "for every pair (n, a) such that n selects a, if $n = 0$, then $a = z$." You can restate (b) in a similar way. These inductive proofs are kind of trivial, which makes them look a little weird. They use the Injective Property.

2.4.6 Exercise

(a) For each number $n \in \mathbb{N}$, there is at least one $a \in A$ such that n selects a.

(b) For each number $n \in \mathbb{N}$, there is at most one $a \in A$ such that n selects a.

Thus we can use this relation to pick out a (unique) function: there is exactly one function $f : \mathbb{N} \to A$ such that

$$f n = a \quad \text{iff} \quad n \text{ selects } a \quad \text{for every } n \in \mathbb{N} \text{ and } a \in A \tag{2.1}$$

2.4.7 Exercise

(a) For any function $f : \mathbb{N} \to A$, f has the Recursive Properties iff f satisfies (2.1).

(b) Explain how this completes the proof of the Recursion Theorem (Principle 2.4.4).

Hint for part (a). Here is another equivalent way of stating the Recursive Properties.

(i) For each $a \in A$, $f\,0 = a$ iff $a = z$.
(ii) For each $a \in A$, $f(\operatorname{suc} n) = a$ iff $a = s(f\,n)$.

2.5 Strings

This text consists of symbols written down in order to form words and sentences, as well as special logical notation. When we express ideas, we almost always do it by stringing together symbols in some order (whether they are written, spoken, signed, or otherwise). So the theory of finite strings of symbols is centrally important for studying language, philosophy, and logic.

Strings bridge between the finite and the infinite. There are only finitely many symbols which can be typed using a standard keyboard. But by typing these symbols in different orders, in sequences of different lengths, they can be used to represent infinitely many different ideas—all the books ever written, and infinitely many merely possible books besides.

A string of symbols is different from a *set* of symbols. The sentences `Dog bites man` and `man bites dog` say different things, but they both are made up from the very same set of symbols. The order of a string matters, and the same symbols can be repeated.

We will use strings to represent language—including words, sentences, logical formulas, programs, and proofs. It will be helpful to fix in advance a standard alphabet for this purpose. We could get by with the twenty-six English letters and a few punctuation marks—or if we wanted to be very austere, we could get away with just dots and dashes like in Morse code, or zeros and ones or some other very simple alphabet. But let's be a little more extravagant.

Since 1991, the Unicode Consortium has standardized a very large "alphabet", called the Unicode Character Set, which includes all the symbols used in most human writing systems. This includes not just letters, punctuation marks, and spaces, but also many technical symbols like `∀` , `→` , and `⊕` , and even emoji. Unicode is nowadays a worldwide standard, especially for representing text on the Internet, which of course is written in many different natural and artificial languages. (This text is also written using Unicode.) Our **standard alphabet** will consist of the entire Unicode 8.0 Character Set. This is a set of about 120,000 different symbols—including all of the symbols used in this text. (But in practice we will use fewer than a hundred symbols or so, so if you want you can use a significantly smaller alphabet without changing anything.) A **symbol** is any element of the standard alphabet. A

string is what you get by putting symbols together one by one in order—we we will be more precise about this in a moment. We will use the notation $\mathbb{S}$ for the set of all strings. (In this text, "string" always means *finite* string.)

Just like we did with numbers, it will be useful to give a careful and precise description of the structure of strings. This will equip us with tools for proving things about *all* strings, and for defining functions on the domain of strings.

In particular, strings also have *inductive structure* like the numbers. This time our starting place is the *empty* string—the length zero string which contains no symbols. (Counting the empty string as a string is handy for similar reasons as counting *zero* as a number.) We'll use the notation () for the empty string. We can build up any finite string at all by starting with the empty string, and adding on symbols one at a time. For example, we can extend the string `BC` by adding `A` to the front to form `ABC` . The general rule is that for any string s and any single symbol a in the standard alphabet, we can form a one-symbol-longer string which consists of the symbol a followed by the string s. We call this longer string $(a : s)$.

We can build up any string this way. For example,

$$\begin{aligned} \texttt{ABC} &= \texttt{A} : \texttt{B} : \texttt{C} : () \\ &= (\texttt{A} : (\texttt{B} : (\texttt{C} : ()))) \end{aligned}$$

(If we leave out the parentheses, they are understood to be added in this way, "associating to the right." This is the only way of filling them in that makes sense.)

Furthermore, this is the *only* way to produce the string `ABC` by adding symbols to the front one at a time. It isn't as if you could put together some other symbols in some other order and end up with the very same string. In general, every finite string can be reached in *exactly one way* by starting with the empty string and adding symbols to the front one by one.

Officially, the notation `ABC` will just be a shorthand for this fully-spelled out expression involving "cons" and the empty string.

We can summarize this fact using "formation rule" notation, similar to what we did for numbers. There are two ways of building up finite strings, which can be described with the following rules:

$$\frac{}{()\text{ is a string}} \qquad \frac{a \text{ is a symbol (in the standard alphabet)} \qquad s \text{ is a string}}{(a : s) \text{ is a string}}$$

Every string can be produced in exactly one way using these two rules.

This means that, just like with numbers, we can do *proofs by induction* for strings. If we want to prove that *every string* has a certain property, it's enough to show two things: (a) The empty string has the property. (b) The property is inherited whenever we add a single symbol. We will look at examples of this in a moment.

Inductive proofs are one important thing that strings have in common with numbers. Here is another thing they have in common. In Section 2.2 we showed how to give a recursive definition for a function whose domain is the set of numbers. Recursive definitions work for strings, too. Every strig can be reached in exactly one way, by starting with the empty string and adding symbols one by one. So we can define an "output" of a function f for *every* string in two steps.

1. We say what the output is for the empty string, $f()$.
2. We assume that we already have the output for a shorter string s, and then we use this value $f\,s$ to define the value of f for a string which is just one symbol longer, $f(a : s)$ for any symbol a in the alphabet.

Here's an example.

2.5.1 Definition

Let's recursively define the *length* of a string. This is a function $\text{length} : \mathbb{S} \to \mathbb{N}$ that takes each string to a number. The definition has two parts. For the *base case*, we define the length of the empty string:

$$\text{length}() = 0$$

For the *recursive step*, we suppose that we already know the length of a string s, and we use this to define the length of the string that results from adding one symbol to s. That is, supposing we know $\text{length}\, s$, we want to define $\text{length}((a : s))$. This is easy: it should be just one more than the length of s.

$$\text{length}(a : s) = \text{suc}(\text{length}\, s)$$

Here's another example. The $(:)$ function lets us add one symbol to a string. But another thing we sometimes want to do is add a bunch of symbols at once, sticking two long strings together end to end. If s and t are both strings, we'll call the result of sticking them together this way $s \oplus t$. We can give an official definition of this operation using recursion. This is closely analogous to the definition of addition for numbers, so it might be helpful to compare the parts of this definition side-by-side with Definition 2.2.1.

2.5.2 Definition

For any string $t \in \mathbb{S}$, we define the function that takes a string $s \in \mathbb{S}$ to a string $s \oplus t$ recursively, as follows.

For the base case, we say how to join the empty string to the beginning of t. This is easy:

$$() \oplus t = t$$

For the recursive step, we suppose that we already know how to join s to the beginning of t, and then use this to define the result for the longer string $(a : s)$. The idea is that we can do this by first joining s to t, and then finally putting in the symbol a as well.

$$(a : s) \oplus t = a : (s \oplus t)$$

2.5.3 Example

Show explicitly using the definition:

$$\texttt{ab} \oplus \texttt{cba} = \texttt{abcba}$$

Proof

Remember that the notation $\texttt{ab}$ is shorthand for

$$\texttt{a} : \texttt{b} : ()$$

Using the base case of the definition of $\oplus$,

$$() \oplus \texttt{cba} = \texttt{cba}$$

Using the recursive step,

$$\begin{aligned} (\texttt{b} : ()) \oplus \texttt{cba} &= \texttt{b} : \texttt{cba} \\ &= \texttt{bcba} \end{aligned}$$

Using the recursive step again,

$$\begin{aligned} \texttt{ab} \oplus \texttt{cba} &= (\texttt{a} : \texttt{b} : ()) \oplus \texttt{cba} \\ &= \texttt{a} : ((\texttt{b} : ()) \oplus \texttt{cba}) \\ &= \texttt{a} : \texttt{bcba} \\ &= \texttt{abcba} \end{aligned}$$

□

Just like with numbers, recursive definitions and inductive proofs for strings work hand in hand.

2.5.4 Example

For any strings s and t,

$$\operatorname{length}(s \oplus t) = \operatorname{length} t + \operatorname{length} s \tag{2.2}$$

Proof

Let t be any string. We'll use induction to prove that every string s has the property (2.2).

Base case. Consider the empty string. By definition, $() \oplus t = t$. So:

$$\begin{aligned} \operatorname{length}(() \oplus t) &= \operatorname{length} t && \text{by the definition of } \oplus \\ &= \operatorname{length} t + 0 && \text{by the definition of } + \\ &= \operatorname{length} t + \operatorname{length}() && \text{by the definition of length} \end{aligned}$$

Inductive step. Suppose that s has the property (2.2). (This assumption is the *inductive hypothesis*.) We want to show that, for any symbol a, the string $(a : s)$ also has the property (2.2).

$$\begin{aligned} \operatorname{length}((a : s) \oplus t) &= \operatorname{length}(a : (s \oplus t)) && \text{by the definition of } \oplus \\ &= \operatorname{suc}(\operatorname{length}(s \oplus t)) && \text{by the definition of length} \\ &= \operatorname{suc}(\operatorname{length} t + \operatorname{length} s) && \text{by the inductive hypothesis} \\ &= \operatorname{length} t + \operatorname{suc}(\operatorname{length} s) && \text{by the definition of } + \\ &= \operatorname{length} t + \operatorname{length}(a : s) && \text{by the definition of length} \end{aligned}$$

□

With numbers, we treated *successor* as the basic notion, and then used this to define addition. Once we had defined addition, though, we showed that $\operatorname{suc} n = n + 1$. From then on we could stick to the more familiar notation. Things go similarly with strings. We treated adding *one* symbol with "cons" as our basic notion. Then we used this to define the more general notion of joining two strings of any length together. Now that we have done this, we can replace cons with simpler and more familiar notation. In particular:

2.5.5 Example

For any symbol a and string s,

$$(a : s) = (a) \oplus s$$

where (a) is the length-one string consisting just of the symbol a—explicitly, $(a) = (a : ())$.

Proof

$$(a) \oplus s = (a : ()) \oplus s = (a : () \oplus s) = (a : s)$$

□

In what follows we won't usually bother to distinguish the *symbol a* from the *length-one string s*. Officially, nothing we have said commits us to a view about whether these are the same thing or two different things. But from now on, we'll treat them as the same thing, for convenience.

2.5.6 Exercise

(a) Is joining strings commutative? That is, does

$$s \oplus t = t \oplus s$$

for all strings $s, t \in \mathbb{S}$? If so, give a proof by induction; otherwise, give a counterexample.

(b) Is joining strings associative? That is, does

$$s \oplus (t \oplus u) = (s \oplus t) \oplus u$$

for all strings $s, t, u \in \mathbb{S}$? If so, give a proof by induction; otherwise, give a counterexample.

Hint. It might be helpful to look back at Example 2.2.6.

2.5.7 Definition

For any string s, we can recursively define the **set of symbols** in s as follows.

$$\text{sym}() = \varnothing$$
$$\text{sym}(a : s) = \{a\} \cup \text{sym}\, s$$

This recursively defines a function

$$\text{sym} : \mathbb{S} \to PA$$

where A is the standard alphabet.

2.5.8 Exercise
Use the definition to show explicitly:

$$\operatorname{sym} \texttt{ABA} = \{\texttt{A}, \texttt{B}\}$$

2.5.9 Exercise
For any strings s and t,

$$\operatorname{sym}(s \oplus t) = \operatorname{sym} s \cup \operatorname{sym} t$$

2.5.10 Exercise
When you log into a website, to protect your privacy your password usually isn't shown directly on your screen: instead, a string of dots with the same length as your password is displayed. Instead of the string `password`, you'll see the string `••••••••`. For each string s, let $\operatorname{dots} s$ be the string of dots with the same length as s.

(a) Write out a recursive definition of the dots function.

(b) Use your definition to show

$$\operatorname{length}(\operatorname{dots} s) = \operatorname{length} s$$

(c) Use your definition to show

$$\operatorname{sym}(\operatorname{dots} s) = \{\texttt{•}\}$$

(d) Use your definition to show

$$\operatorname{dots}(s \oplus t) = \operatorname{dots} s \oplus \operatorname{dots} t$$

(e) Show that

$$\operatorname{length} s = \operatorname{length} t \quad \text{iff} \quad \operatorname{dots} s = \operatorname{dots} t$$

Let's summarize the two main things we have learned to do with strings in this section.

2.5.11 Technique (proof by induction for strings)
Suppose you have some set A, and you want to show:

For every string s, s is nice.

Here “s is nice” is a placeholder for any statement about s. You can do this in two steps.

1. *Base case.* Show that the *empty* string is nice.
2. *Inductive step.* Show that for any string s, and any symbol a, *if* s is nice, *then* the longer string $(a : s) = a \oplus s$ is also nice.

Then you’re done: this is enough to show that *every* string is nice.

2.5.12 Technique (recursively defining a function on strings)
Suppose you want to come up with a function whose *domain* is the set $\mathbb{S}$ of all strings. That is, for some set B, you’re trying to come up with a function $f : \mathbb{S} \to B$. You can do this in two steps.

1. Choose a value of f for the empty string. That is, you’ll write down

   $f() =$ ________________

   Fill in the blank with some description of an element of B.

2. Choose a rule for getting a value of f for a string *using* the value of f for a shorter string. That is, you’ll write down

   For any symbol a and string s, $f(as) =$ ___________________

   Fill in the blank with another description of an element of B, where this description is allowed to use $f(s)$ (as well as a).

Once you have done these two things, you have precisely described one (and only one!) function from $\mathbb{S}$ to B.

The set of numbers and the set of strings are both *inductive structures*. Inductive structures play a starring role throughout logic, and in this text we’ll encounter many others. (For example, formulas of first-order logic, formal proofs, and programs.) For this reason, *proofs by induction* and *recursive definitions* are two very important fundamental skills in logic.

We’ll be talking about strings of symbols a lot. In this written medium, we also *use* strings of symbols in order to talk—strings of symbols that represent English words, as well as technical notation. For instance, this paragraph begins with the string of symbols `We’ll be talking about strings` , and so on. It will be important to be distinguish these two activities, which are standardly called *use* and *mention*: that is, *using* strings of symbols to say things, and *mentioning* strings of symbols to talk about the symbols themselves. So it will be helpful to have some special notation.

2.5.13 Notation

We use the notation `ABC` to refer to the three-letter string consisting of `A` followed by `B` followed by `C` .

Instead of using the join symbol $\oplus$, we can just write two strings next to each other, so st is the same as $s \oplus t$. Likewise, `A`s is the same as `A` $\oplus$ s, and `ABC`s`DEF` is the same as `ABC` $\oplus$ s $\oplus$ `DEF`. This is convenient when we are building up complicated strings out of shorter ones. (This is similar to the convention in algebra of using the shorthand xy instead of $x \cdot y$ for multiplication.)

2.5.14 Exercise

Let $s =$ `tu`. Which of these strings are the same?

(a) `stu`

(b) `s` $\oplus$ `tu`

(c) $s \oplus$ `tu`

(d) `s ⊕ tu`

(e) s `tu`

(f) `s` $\oplus$ s

(g) `s` s

(h) `t` $\oplus$ `u` $\oplus$ `t` $\oplus$ `u`

(i) s ` ⊕ tu`

(j) `s` $\oplus$ ` ⊕ tu`

2.6 Official Principles for Strings*

Just like we did with numbers, we can describe the inductive structure of strings more officially using an *axiom*, which is closely analogous to the Axiom of Numbers. This new axiom is a little more complicated, though, because the adding-one-symbol operation $(:)$ is a little more complicated than the successor function. Likewise, we can more precisely state, and more carefully justify, the technique of recursively defining a function on strings with a *theorem*—a Recursion Theorem for Strings, which is closely analogous to the Recursion Theorem for numbers. The idea is basically the same as with numbers, but again it's a little bit more complicated. I'll state both of these explicitly here for completeness. But for this course, the more important thing to have a handle on is the practical *skills* of inductive proofs and recursive definitions—not the official statements we will give for the Axiom of Strings or the Recursion Theorem for Strings. The Axiom of Strings is just a way of precisely spelling out the main intuitive idea:

> Every string can be reached in exactly one way by starting from the empty string and adding symbols one by one.

Similarly, the Recursion Theorem for Strings is just a way of precisely spelling out the intuitive idea that we can define a function on all strings using a "starting place" and a "step rule."

2.6.1 Axiom of Strings
Let A be the set of symbols in the standard alphabet. There is a set of **strings** $\mathbb{S}$, an **empty string** $()$ which is an element in $\mathbb{S}$, and a **"construct" function** that takes each symbol $a \in A$ and each string $s \in \mathbb{S}$ to a string $a : s$, which have the following properties.

(a) **Injective Property.**

  (i) The empty string $()$ is not in the range of the construct function. That is, there is no element a in A and string s in $\mathbb{S}$ such that $a : s = ()$.

  (ii) The construct function is one-to-one. That is, for any symbols a and a' in A and any strings s and s' in $\mathbb{S}$, if $a : s = a' : s'$, then $a = a'$ and $s = s'$.

(b) **Inductive Property.** Let X be a set. Suppose

  (i) the empty string $()$ is in X, and

(ii) for each symbol a in A and string s in $X \in$ X$, $(a : s)$ is also in X.

Then X includes every string.

2.6.2 The Recursion Theorem for Strings

Let A be the standard alphabet, and let B be any set. For any element $e \in B$ and any function $c : A \times B \to B$, there is a unique function $f : \mathbb{S} \to B$ with the following two *Recursive Properties:*

$$\begin{aligned} f() &= e \\ f(a : s) &= c(a, fs) \quad \text{for each symbol } a \in A \text{ and string } s \in \mathbb{S} \end{aligned}$$

Proof Sketch

We can use the same idea we used for numbers. First, we can use the recursive properties to recursively define a *relation* between strings and values (using Exercise 2.4.2). Second, we can show that this relation is *functional,* and thus that it picks out a function. Finally, we can check that this function has the recursive properties.

For a string s and an element $b \in B$, we define s **selects** b recursively by the following two rules:

$$\frac{}{() \text{ selects } e} \qquad \frac{s \text{ selects } b \qquad a \text{ is a symbol}}{a : s \text{ selects } c(a, b)}$$

As in the proof of the Recursion Theorem for Numbers, we can prove the following things.

1. (a) If () selects b, then $b = e$.
   (b) If $a : s$ selects b, then there is some b' such that s selects b' and $b = c(a, b')$.

2. Any function f has the Recursive Properties iff, for each string s and $b \in B$,

$$fs = b \quad \text{iff} \quad s \text{ selects } b$$

Finally we can prove (by induction on strings)

3. Each string selects exactly one value.

This implies that there really is exactly one function f with the Recursive Properties.

□

As we'll see later, induction and recursion make sense not just for numbers and strings, but also for formulas, proofs, and many other kinds of thing which are important for logic. Each of these inductive structures has both an Inductive Property and a corresponding Recursion Theorem.

2.7 Properties of Numbers and Strings

At this point, we have stated the Axiom of Numbers and the Axiom of Strings: these describe the fundamental structure of finite numbers and finite strings, using the Injective Property and Inductive Property for each of them. We've also given recursive definitions for a few important operations on these structures: especially addition ($+$), multiplication ($\cdot$), concatenation ($\oplus$), and length. In this section we'll summarize some other important facts about how these operations on numbers and strings work, which follow from the axioms and definitions we have already given. Working through all the proofs of the facts in this section would provide good extra exercises for getting practice. Even though I've marked them as "Exercises," though, they won't be assigned as homework and I won't go over them in class—that would just take us too much time, and we want to move on to more interesting things. Still, it's important to know not only that the facts listed here about numbers and strings are *true*, but also that we *can prove* all of these facts from our basic axioms and definitions—even though we won't actually bother to do this.

It will be helpful to refer back to these facts later on.

2.7.1 Exercise (Cancellation Property for Addition)
For any numbers n, k, and k', if $n + k = n + k'$, then $k = k'$.

Hint. Prove that every number n has the property:

> For any $k, k' \in \mathbb{N}$, if $n + k = n + k'$, then $k = k'$.

We defined $m \leq n$ recursively, in Section 2.2, and proved that this relation was reflexive and transitive. Here are some more related facts.

2.7.2 Exercise
For any numbers m and n, $m \leq n$ iff there is some number $k \in \mathbb{N}$ such that $m + k = n$.

Hint. For the left-to-right direction, use induction on the definition of $\leq$. For the right-to-left direction, use induction on k.

2.7.3 Exercise
If $n \leq 0$ then $n = 0$.

Hint. We can restate this: for any pair (n, k) such that $n \leq k$, if $k = 0$, then $n = 0$.

2.7.4 Exercise
If $m \leq n$ and $n \leq m$, then $m = n$. (In other words, $\leq$ is **anti-symmetric**.)

2.7.5 Definition
We say m is (strictly) **less than** n (abbreviated $m < n$) iff $m \leq n$ and $m \neq n$.

2.7.6 Exercise
There is no natural number $n < 0$.

2.7.7 Exercise
$m \leq n$ iff $m < \operatorname{suc} n$, for any numbers m and n.

2.7.8 Exercise
For any numbers m and n, either $m \leq n$ or $n \leq m$. (In other words, $\leq$ is **complete**.)

A relation which is reflexive, transitive, and anti-symmetric is called a **partial order**. A partial order which is also complete is called a **total order**. So we have shown that the natural numbers are totally ordered.

2.7.9 Exercise

(a) If $m \leq \operatorname{suc} n$, then either $m \leq n$ or $m = \operatorname{suc} n$.
(b) If $m < \operatorname{suc} n$, then either $m < n$ or $m = n$.

2.7.10 Exercise (The Least Number Property)
Any non-empty set of numbers X has a least element: that is, there is some $m \in X$ such that $m \leq n$ for every $n \in X$. (Another name for this property is that $\leq$ is a **well-ordering**.)

Hint. Suppose X has no least element, and prove by induction that, for every number n, the set $\{k \in X \mid k < n\}$ is empty.

2.7.11 Exercise
Let X be any set of numbers. Show that X has *at most one* least element: that is, there is at most one $m \in X$ such that, for every number $n \in X$, $m \leq n$.

Let's collect together some of the useful basic facts we've established. Some of these are definitions, and others were proved as examples or in exercises. This particular collection of facts will be useful to refer back to later.

2.7.12 The Minimal Theory of Arithmetic
The following properties hold for all numbers m, n, k:

1. 0 is not a successor.
2. No two numbers have the same successor.
3. $n + 0 = n$.
4. $m + \operatorname{suc} n = \operatorname{suc}(m + n)$
5. $n \cdot 0 = 0$
6. $m \cdot \operatorname{suc} n = (m \cdot n) + m$
7. n is not less than 0
8. $m \leq n$ iff $m < \operatorname{suc} n$
9. $m \leq n$ or $n \leq m$

We can do some similar things for strings.

2.7.13 Definition
For strings s and t, we say s is an **initial substring** of t (abbreviated $s \preceq t$) iff there is some string $u \in \mathbb{S}$ such that $s \oplus u = t$. We say s is a **proper initial substring** of t (abbreviated $s \prec t$) iff $s \preceq t$ and $s \neq t$.

2.7.14 Exercise
$s \preceq t$ iff either s is empty, or for some symbol a, $s = a \oplus s'$, $t = a \oplus t'$, and $s \preceq t'$.

2.7.15 Exercise
If $s \preceq t$ then length $s \leq$ length t.

2.7.16 Exercise (Cancellation Property for Strings)
If $s \oplus t = s' \oplus t$, then $s = s'$. Likewise, if $s \oplus t = s \oplus t'$, then $t = t'$.

2.7.17 Exercise
If $s \preceq t$ and $s' \preceq t$, then either $s \preceq s'$ or $s' \preceq s$.

2.7.18 Exercise (The Shortest String Property)
Let X be any non-empty set of strings. Then X has a minimal-length element: that is, there is some $s \in X$ such that every string in X is at least as long as s.

The Shortest String Property underlies an alternative induction technique for reasoning about strings, which is usually called *strong induction.* Say we want to prove that every string is nice. We can do it like this. Let s be any string, and show the following:

> If every string which is strictly *shorter* than s is nice, then s is also nice.

It then follows that every string is nice. The reason this follows, is because otherwise the set X of non-nice strings would be non-empty, in which case it would have a minimal-length element. But that would mean that s is a non-nice string such that every string *shorter* than s is nice—and what the "strong induction" proof shows is precisely that there is no such string. Strong induction on strings is a bit more flexible than ordinary ("primitive") induction on strings, so it is occasionally handy.

2.7.19 The Minimal Theory of Strings
Let s and t be strings, and let a and b be strings each of which consists of a single symbol.

1. $a \oplus s \neq ()$
2. If $a \oplus s = a \oplus t$, then $s = t$.
3. If a and b are distinct symbols, then $a \oplus s \neq b \oplus t$.
4. $() \oplus s = s$
5. $a \oplus (s \oplus t) = (a \oplus s) \oplus t$
6. $a = a \oplus ()$
7. The empty string () is no longer than s.
8. s is no longer than () iff $s = ()$.
9. $a \oplus s$ is no longer than $b \oplus t$ iff s is no longer than t.
10. Either s is no longer than t, or t is no longer than s (or perhaps both).
11. Either $s = ()$, or else there is some single-symbol string a and string t such that $s = a \oplus t$.

2.8 Review

Key Techniques

- You can use **induction** to prove that every number has a certain property. (Technique 2.1.5)
- You can use a **recursive definition** to come up with a function whose domain is the set of all numbers. (Technique 2.2.11)
- You can use **induction on strings** to prove that every string has a certain property. (Technique 2.5.11)
- You can use **recursion on strings** to come up with a function whose domain is the set of all strings. (Technique 2.5.12)

Key Concepts and Facts

- The **Injective Property of numbers** intuitively says that every number can be reached in *at most one way* by starting from zero and taking successors.
- The **Inductive Property of numbers** intuitively says that every number can be reached in *at least one way* by starting from zero and taking successors.
- The **Recursion Theorem** intuitively says that recursive definitions work. If you give a value for zero, and a “step function” to go from the value for n to the value for $n+1$, then this pins down exactly one function which is defined for every number.
- Together, the Injective Property and Inductive Property for numbers tell us everything we need to know about the structure of numbers: we can use these basic properties to prove familiar facts about how operations like addition and multiplication work.
- The **Injective Property of Strings** intuitively says that every string can be reached in *at most one way* by starting from the empty string and adding symbols one at a time.
- The **Inductive Property of Strings** intuitively says that every string can be reached in *at least one way* by starting from the empty string and adding symbols one at a time.

- The **Recursion Theorem for Strings** intuitively says that recursive definitions on strings work. If you give a value for the empty string, and a rule for going from the value for a string s to a value for any string $a \oplus s$ which is one symbol longer, then this pins down exactly one function which is defined for every string.

- The Injective Property and Inductive Property for strings tells us everything we need to know about the structure of strings: we can use these basic properties to prove facts about operations like joining strings together work.

- We have to be careful to distinguish when we are **using** a string of symbols to say something about the world, and when we are **mentioning** a string of symbols to say something about the string itself. We have some special notation to help with this, like this: `We have some special notation`. (Notation 2.5.13)

Chapter 3

Structures

> Philosophy is written in this grand book—I mean the Universe—which stands continually open to our gaze, but it cannot be understood unless one first learns to comprehend the language and interpret the symbols in which it is written.
>
> Galileo Galilei, *The Assayer* (1623)

3.1 Signatures and Structures

A set is just some things, taken together. But for lots of purposes we don't just want to look at "bare" sets, but rather *structured* sets.

For example, the natural numbers aren't just a bag of featureless marbles. They come equipped with a starting place, and a way of stepping from one number to another. So it's handy to bundle these operations together. The natural numbers *structure* $\mathbb{N}(0, \text{suc})$ intuitively consists of not just the *set* of natural numbers, but also a little sign pointing to zero, and another little sign pointing to the successor function. Of course, there are many different operations we might want to point out. So there are really many different structures which all share the same *domain*—the set of natural numbers. Another example is the structure $\mathbb{N}(0, \text{suc}, +, \cdot, \leq)$, which also has signs pointing out addition, multiplication, and the less-than-or-equal relation. Or we might want to also highlight the exponential function, or the "next largest prime number" function, or whatever we like.

Similarly, we can consider the set of strings. There are different operations on this set which are worth pointing to. One structure just points out the *empty* string as special. Another points out just the empty string and the "join" operation $x \oplus y$ on strings. For any symbol a, we can also pick out the singleton string of just a. We can also point out the "shorter-than" relation between strings.

The definition of a structure has three parts. The first part is the *domain*, which is just a set. The second part is a *signature*, which basically consists of a bunch of signs. The third part (the interesting part) is a way of attaching those signs to various features of interest in the domain.

The "features" come in various flavors. We might point out a special object, like zero, or the empty string, or the string `A`. We might point out a one-place function, like the successor function, or a two-place function, like the join operation. Or we might point out a special subset, like the even numbers, or a special two-place relation, like the less-than relation. A signature is a way of keeping track of how many signs we have, and what sort of things they're each supposed to point to. For example, the signature of the **language of arithmetic** consists of the symbols `0`, `suc`, `+`, `·`, and `≤`, which are respectively a constant, a one-place function symbol, a two-place function symbol, another two-place function symbol, and a two-place relation symbol. (We could think of the constant as a *zero*-place function symbol, since it doesn't take any arguments at all.)

3.1.1 Definition

A **signature** consists of five sets of non-empty strings (with no strings in common between them): a set of **constant symbols**, a set of **one-place function symbols**, a set of **two-place function symbols**, a set of **one-place predicates**, and a set of **two-place predicates** or **relation symbols**.

In principle, we could allow function symbols and predicates with any number of arguments. But we'll be restricting our attention to one-place and two-place functions and relations, just because this often makes things a little simpler, and we won't need need the extra generality for anything in this text.

Note that it is standard to call these "function symbols" and "relation symbols" despite the fact that they don't have to consist of a single symbol from the alphabet. For instance, it's fine to use the length-three string `suc` as a function symbol.

(All of our constants and function symbols are *strings.** As we will see in Section 4.5, this puts some restrictions on how "big" a language can be. In other contexts sometimes it's nice to be less restrictive, and use things other than strings as the "signs" in signatures and structures. For instance, it can occasionally be nice to think about a "Lagadonian language" in which each object serves as a constant symbol for *itself*.[1] But for the purposes of this text it's convenient to be more restrictive: we are focusing on logical languages that can be written down as strings of symbols.)

It will help us out later on if we put some restrictions on what strings we allow in signatures. It would make things a complete mess if we used, say, `) ∧ (x` as the notation for one of our basic function symbols. So strings like this aren't allowed. Our exact rules for what counts as a legitimate function or relation symbol aren't very important, so we won't go into them here—use common sense—but there are details in Section 3.3.

3.1.2 Example

The **signature of the language of arithmetic** has one constant symbol `0` , one one-place function symbol `suc` , two two-place function symbols `+` and `·` , and one relation symbol `≤` .

3.1.3 Example

The **signature of the language of strings** has a two-place function symbol `⊕` , a relation symbol `≤` (for the no-longer-than relation), a constant `""` (representing the empty string), and a constant for the singleton string for each symbol in the standard alphabet. We'll use quotation marks for these "singleton constants." The constant for the singleton string `A` will be `"A"` , the constant for `B` will be `"B"` , and so on. There are a few exceptions to this pattern.

- It would be confusing and potentially ambiguous to use `"""` as the constant

[1] The term "Lagadonian language" comes from Lewis (1986). It is inspired by *Gulliver's Travels*: the professors in the "school of languages" in the city of Lagado proposed the following scheme:

> An expedient was therefore offered, "that since words are only names for things, it would be more convenient for all men to carry about them such things as were necessary to express a particular business they are to discourse on." … [M]any of the most learned and wise adhere to the new scheme of expressing themselves by things; which has only this inconvenience attending it, that if a man's business be very great, and of various kinds, he must be obliged, in proportion, to carry a greater bundle of things upon his back, unless he can afford one or two strong servants to attend him. I have often beheld two of those sages almost sinking under the weight of their packs, like pedlars among us, who, when they met in the street, would lay down their loads, open their sacks, and hold conversation for an hour together; then put up their implements, help each other to resume their burdens, and take their leave.
>
> But for short conversations, a man may carry implements in his pockets, and under his arms, enough to supply him; and in his house, he cannot be at a loss. Therefore the room where company meet who practise this art, is full of all things, ready at hand, requisite to furnish matter for this kind of artificial converse. `[TODO CITE]`

for the quotation mark `"` . So we'll use the constant `quo`, instead.

- To guarantee unique parses (Section 3.3), it's convenient to stick to constants that don't include any commas or parentheses. So we'll use the constant `com` for the comma, `lpa` for `(` and `rpa` for `)` , instead of the more obvious `","` , `"("` , and `")"` .

- In Section 4.1, Section 7.1, and Chapter 8 it will sometimes be convenient to use *multi-line* strings, particularly to write down programs and proofs. For this purpose, we have a symbol in our alphabet that represents the start of a new line—called newline. This symbol is difficult to write down on its own. Our constant that stands for the newline symbol is `new` .

(Why the weird abbreviations for these constants? It turns out that making all of the constants for single symbols the *same length* is a useful hack later on, in Section 6.3.)

Both of these are *finite* signatures. That is, each of them has only finitely many constants and function symbols.

3.1.4 Definition

Suppose L is a signature. A **structure** S with signature L (for short, an L-structure) has the following components.

1. A non-empty set D_S called the **domain** of S.
2. For each constant c in the signature L, an element c_S of the domain of S, which is called the **extension** of c in S.
3. For each one-place function symbol f in the signature L, a function $f_S : D_S \to D_S$, which is called the **extension** of f in S.
4. For each two-place function symbol f in the signature L, a two-place function $f_S : D_S \times D_S \to D_S$, which is called the **extension** of f in S.
5. For each one-place predicate F in the signature L, a subset $F_S \subseteq D_S$, which is called the **extension** of F in S.
6. For each relation symbol R in the signature L, a set of ordered pairs $R_S \subseteq D_S \times D_S$, which is called (surprise) the **extension** of R in S.

(The requirement that the domain of a structure is *non-empty* could be dropped: the empty L-structure is a perfectly fine thing, as long as the signature L doesn't include any constants. But handling the empty structure correctly would sometimes add some extra complications later on, and with very little pay-off, so we won't bother.)

Another name for a structure is a **model**. This term is a bit old-fashioned, but we still use it in certain contexts.

3.1.5 Definition

The **standard model of arithmetic** $\mathbb{N}(0, \mathrm{suc}, +, \cdot, \leq)$, which we just call $\mathbb{N}$ for short, has the following components:

1. The domain $D_{\mathbb{N}}$ is the set of natural numbers $\mathbb{N}$.
2. The extension of the constant `0` (that is, $\mathtt{0}_{\mathbb{N}}$) is the number zero.
3. The extension of the function symbol `suc` is the successor function.
4. The extension of the function symbol `+` is the addition function.
5. The extension of the function symbol `·` is the multiplication function.
6. The extension of the relation symbol `≤` is the set of pairs (m, n) such that m is less than or equal to n.

We also just call this structure $\mathbb{N}$ for short.

We can restate this more concisely:

$$\begin{aligned}
D_{\mathbb{N}} &= \mathbb{N} \\
\mathtt{0}_{\mathbb{N}} &= 0 \\
\mathtt{suc}_{\mathbb{N}} &= \mathrm{suc} \\
\mathtt{+}_{\mathbb{N}}(m, n) &= m + n \quad \text{for each } m, n \in \mathbb{N} \\
\mathtt{\cdot}_{\mathbb{N}}(m, n) &= m \cdot n \quad \text{for each } m, n \in \mathbb{N} \\
\mathtt{\leq}_{\mathbb{N}} &= \{(m, n) \in \mathbb{N} \times \mathbb{N} \mid m \leq n\}
\end{aligned}$$

Note that we need to be careful about use and mention here as well. The word `Obama` is a different thing from President Obama. Similarly, we shouldn't confuse the number 0, which is an element of the *domain* of this structure (a certain number), with the constant `0` which is a symbol in the *signature* (a certain string). The constant `0` *stands for* the number 0—that is, `0` has the number 0 as its extension. But they are not the same thing. This same kind of note applies to all the other arithmetical symbols.

This note also applies to the symbols in the language of *strings*, where things are a little subtler. For example, `"A"` and `A` are both strings, but they are not the same string. The string `"A"` is a constant in the language of strings. The string `A` is not a constant, though—it is the string that the constant `"A"` has as its *extension.* Even when we are naming parts of language—indeed, even when we are naming names!—it is important to keep track of the difference between names and what is named.

Another example of a structure is $\mathbb{N}(0, \text{suc})$. This structure also has as its domain the set of all natural numbers, and in this structure also the constant symbol `0` stands for the number zero, and the function symbol `suc` stands for the successor function. (But unlike $\mathbb{N}(0, \text{suc}, +, \cdot, \leq)$, this structure doesn't have the symbols `+`, `·`, or `≤` in its signature.)

There's also an even simpler structure $\mathbb{N}(0)$ which only labels zero, and doesn't label any operations at all. This one isn't very practically useful, but it's sometimes helpful as an example.

3.1.6 Definition

The **standard string structure** $\mathbb{S}$ is a structure with the signature of the language of strings specified above (Example 3.1.3). Its domain is the set of all strings. The extension of `""` is the empty string. For each symbol a in the standard alphabet, the corresponding constant symbol has as its extension the singleton string of just a. (For example, the extension of the constant symbol `"A"` is the singleton string `A`, and the extension of the constant symbol `quote` is the singleton string `"`.) The extension of the two-place function symbol `⊕` is the function that joins two strings together. The extension of the relation symbol `≤` is the set of pairs of strings (s, t) such that length $s \leq$ length t.

In other words:

$$\begin{aligned} D_{\mathbb{S}} &= \mathbb{S} \\ \texttt{⊕}_{\mathbb{S}}(s, t) &= s \oplus t \qquad \text{for each } s, t \in \mathbb{S} \\ \texttt{≤}_{\mathbb{S}} &= \{(s, t) \mid \text{length } s \leq \text{length } t\} \\ \texttt{""}_{\mathbb{S}} &= () \quad \text{(that is, the empty string)} \end{aligned}$$

And for each symbol a in the standard alphabet, if c is its corresponding singleton constant, then

$$c_{\mathbb{S}} = (a)$$

For example, `"A"`$_{\mathbb{S}}$ = `A`, and `quo`$_{\mathbb{S}}$ = `"`.

In this example we need to be *even more* careful about use and mention—because now strings are not only the things we are using as labels, but strings are also things that some of our labels *stand for*. So we don't just have to pay attention to what kind of thing we are talking about (strings, sets of strings, functions, etc.), but what we are doing with it.

We standardly use the symbols `0` , `+` , and so on to talk about numbers. But we could also interpret them in other ways. There are *non-standard* structures for the signature of arithmetic. Here's a simple example:

3.1.7 Example

There is a structure S with the signature (`0`, `suc` , `( + )`) given as follows:

1. The domain of S consists of all of the buildings in Los Angeles.

2. The extension of `0` in S is the Natural History Museum of Los Angeles County.

3. The extension of `suc` in S is the function that takes each building to the nearest building directly east of it. (This will map buildings at the eastern edge of LA all the way around the world to the West Side again.)

4. The extension of `+` in S is the function that takes two buildings to whichever one of them contains the most dinosaur skeletons (or the building farthest east in the case of a tie).

The main point of the language of arithmetic is to talk about the *standard* number structure. But non-standard structures are also important. As we will see later on, one way of investigating how much we have managed to say about an *intended* structure, is to look at what *unintended* interpretations are still compatible with what we have said so far.

3.2 Terms

One of the overarching themes of this course is the relationship between language and the world, and in particular the way that languages can describe (or fail to describe) different structures. Here we'll work out the details of a very simple kind of precise language. We've already begun: a *signature* is already a very simple kind of language. It is basically just a "bag of words", without any structure that holds different words together. We'll now take a step to a slightly more complicated language, putting symbols together to build up expressions that have *syntactic* structure.

A signature gives us some basic symbols for picking out features of interest. Take the standard model of arithmetic $\mathbb{N}(0, \mathrm{suc}, +, \cdot, \leq)$ as an example: we have a label for zero, and a label for the successor function. But once we have these, we can put them together to pick out other numbers as well. We know that the number one is $\mathrm{suc}\,0$, so we can use the expression `suc 0` to pick it out. Similarly, we can use `suc suc 0` to stand for the number two, and so on. Here `suc 0` is a *complex* symbol, built out of two basic symbols `suc` and `0`. The things we get by putting these symbols together are called **numerals**: they are *labels* for numbers. The numerals are these expressions:

`0`, `suc 0`, `suc suc 0`, `suc suc suc 0`, ...

We can also describe a number in other ways: for example, the number two isn't just $\mathrm{suc}\,\mathrm{suc}\,0$, but it's also $(\mathrm{suc}\,0)+(\mathrm{suc}\,0)$ (that is, $1+1$). We also have `+` in the language of arithmetic, so we can also build up the expression `(suc 0) + (suc 0)` as an alternative way to refer to the number two. In general we can build up arbitrarily complicated *terms* by putting these symbols together in different ways.

(Relation symbols do not ever appear in terms. We will bring them back in Chapter 5.)

Hopefully that gets across the intuitive idea of what a term for a certain signature is. The next thing we'll do is give a more precise description of terms.

In the language of arithmetic, one term is `suc suc 0 · (suc 0 + suc 0)`. We can visualize its structure as in Fig. 3.1. This has the form of a **labeled tree**, where each node of the tree is labeled with some symbol in the language of arithmetic. The key idea here is that every term can be represented by a syntax tree like this, and in exactly one way.

Another way of representing the same structure is with a **syntax derivation**, which

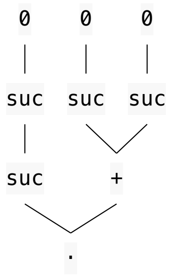

Figure 3.1: The syntactic structure of a term, as a labeled tree.

shows how each stage is built up using one of the basic symbols.

$$\dfrac{\dfrac{\dfrac{\texttt{0}}{\texttt{suc 0}}\,\texttt{suc}}{\texttt{suc suc 0}}\,\texttt{suc} \qquad \dfrac{\dfrac{\texttt{0}}{\texttt{suc 0}}\,\texttt{suc} \qquad \dfrac{\texttt{0}}{\texttt{suc 0}}\,\texttt{suc}}{\texttt{suc 0 + suc 0}}\,\texttt{+}}{\texttt{suc suc 0} \cdot \texttt{(suc 0 + suc 0)}}\,\cdot$$

We can think of a derivation as a complex argument, consisting of statements of the form "a is a term", where each step of the argument follows from some basic *formation rule* for building up terms.

We can define the terms in the language of arithmetic *inductively*. These are the rules for putting together terms in this language.

$$\frac{}{\texttt{0} \text{ is a term}} \qquad \frac{t \text{ is a term}}{\texttt{suc } t \text{ is a term}}$$

$$\frac{t_1 \text{ is a term} \qquad t_2 \text{ is a term}}{(t_1 \texttt{ + } t_2) \text{ is a term}} \qquad \frac{t_1 \text{ is a term} \qquad t_2 \text{ is a term}}{(t_1 \cdot t_2) \text{ is a term}}$$

As in Section 2.2, each rule means that, given the facts written above the line, we can derive the fact written below the line. A list of rules like this is sometimes called a **grammar**.

Let's state this idea more abstractly, not just for the language of arithmetic, but for an *arbitrary* signature. Before we do this, though, we should talk about some notational issues. In practice, we write down function symbols in several different styles. Some two-place function symbols, like + and · , look best in between the two things they apply to (“infix” notation). Other two-place function symbols, like `cons` or `f`, look best in front of them (“prefix” notation). In practice, we use both notations, depending on which one is more convenient. But when we give an official definition of the syntax of a formal language, it’s a nuisance to keep track of two different ways of writing things down, and this would add annoying and useless complications to our proofs. So we won’t do that. Instead, we’ll make one official choice: because it happens to be a little less cumbersome in general, our official choice will be “prefix” notation: we’ll write two-place function terms like `f(x, y)` , rather than like `(x f y)` . *Officially*, we’ll apply this convention to all function terms, even `+` and `·` and `⊕` . So when we’re being totally official, the terms of the language of arithmetic will look like `+(0,0)` , rather than `(0 + 0)` . But we will almost never bother being totally official. In practice, we can freely write our terms whichever way is most convenient, trusting that this won’t lead to confusion. (It isn’t as if there is some *other* term that you might plausibly mean by `(0 + 0)` .)

There are similar issues that come up with parentheses and spaces. Again, our official definition of terms is going to commit us to one particular choice of where to put parentheses and spaces. Our official choices are mainly driven by the goal of keeping things simple in the general case. But in practice, things often look better and are clearer to human readers if we leave out parentheses that are officially called for (as long as this doesn’t make things ambiguous), and put in extra spaces. Computer programs might make a fuss over this, but since we’re all humans it shouldn’t make too much trouble.

That means that often when we write down a term—for example, as `suc 0 + 0` —*officially* we are really talking about a different, closely related string—in this case, `+(suc(0),0)` . In practice, this shouldn’t really be a big deal. (There will be other notational issues like this that come up from time to time.)

3.2.1 Definition

The **closed terms** for a signature L are defined inductively by the following rules:

$$\frac{c \text{ is a constant}}{c \text{ is a term}}$$

$$\frac{f \text{ is a one-place function symbol} \qquad t \text{ is a term}}{f\texttt{(}t\texttt{)} \text{ is a term}}$$

$$\frac{f \text{ is a two-place function symbol} \qquad t_1 \text{ and } t_2 \text{ are terms}}{f\texttt{(}t_1\texttt{,}t_2\texttt{)} \text{ is a term}}$$

It isn't hard to generalize this to arbitrary *n*-place function symbols. But we won't bother: we won't need them, and they would make our notation a bit more complicated.

It will become clear in Section 3.5 why the definition says "*closed* terms."

It's worth pausing here on a philosophical question. Our definition says that terms are certain strings. But is that really what terms are? This is similar to some questions that came up earlier: whether sequences are really functions from numbers, whether functions are really sets of ordered pairs, or whether ordered pairs are really certain sets. There are some reasons to think that the answer is no. After all, we had to make some arbitrary notational choices in order to decide *which* string was the term `(0 + 0)` (that is, officially, `+(0,0)`). The *nature* of the term—which basic symbols are put together in what syntactic structure—doesn't seem tied to one notation or another. We could have used `(+ 0 0)` or any other unambiguous notational system to write down the same term. But it will make things harder for us down the road if we are always distinguishing between a term and its (somewhat arbitrary) string representation in a certain system of notation. So we will proceed as if the philosophical myth were true, that terms (and syntactic structures more generally) just *are* strings.

The important thing about terms is not the notational details of how specific symbols are put together. The important thing is more structural: each term can be put together out of constants and function symbols, in just one way. An important thing for us to check is that our official system of notation really does have this structural feature—so these strings can at least play the *role* of terms.

As with numbers and strings, the important structural features of terms can be spelled out in two parts: an *Inductive Property* and an *Injective Property.* Intuitively, the Inductive Property says that we can reach every term using these rules in *at least one way,* and the Injective Property says that we can reach every term using these rules in *at most one way*. Let's take a closer look at each of these properties.

The Inductive Property for Terms follows directly from our recursive definition (which is underlied by Exercise 2.4.2). This definition tells us that each of our syntactic rules for forming terms takes you from terms to terms, and it also tells us that every term can be reached somehow or other by applying these rules. Practically, what this gives us is a new kind of inductive proof: induction on syntactic structure. This is an extremely important tool in logic.

3.2.2 Technique (Induction on Closed Terms)

Suppose that we want to show that every closed term is *nice*. We can do this in three steps.

1. Let c be any constant. *Show* that c is nice.

2. Let f be any one-place function symbol, and let t be any closed term. *Suppose* that t is nice. (This is the *inductive hypothesis.*) Then *show* that the term $f(t)$ is also nice.

3. Let f be any two-place function symbol, and let t_1 and t_2 be any closed terms. *Suppose* that t_1 is nice and t_2 is nice. (Again, this is the *inductive hypothesis.*) Then *show* that the term $f(t_1,t_2)$ is also nice.

3.2.3 Example

Every closed term contains at least one constant.

Proof

The proof is by induction on the structure of closed terms. There are three parts to this proof.

Let c be a constant. Then it's obvious that c contains a constant.

Let f be a one-place function symbol, and let t be a term. Suppose, for the inductive hypothesis, that t contains a constant. Then clearly $f(t)$ also contains whatever constants appear in t, since it has t as a substring.

Let f be a two-place function symbol, and let t_1 and t_2 be closed terms. Suppose, for the inductive hypothesis, that t_1 contains a constant, and t_2 contains a constant. Then it's clear that $f(t_1,t_2)$ also contains those constants. □

3.2.4 Exercise

Say a string is **balanced** iff it includes the same number of left parentheses `(` as right parentheses `)`. Prove by induction that every closed term is balanced (as long as there are no parentheses in any constant or function symbols).

The second important thing about terms is that they are not syntactically ambiguous. For example, you can't have one term which is *both* a constant and *also* of the form $f(t)$ for some function symbol f and term t. So something we need to check, to make sure that our official definition of terms really has the right structural features, is that the formation rules are unambiguous. Each term should only be reached in *one way* using these rules. That means that terms also have an Injective Property, analogous to the Injective Properties for numbers and strings. This fact seems kind of obvious when you look at the definition: we've inserted parentheses and commas between things, so you should always be able to find those "delimiters" and use them to break up a term into parts in just one way. But precisely stating this fact and officially proving it is surprisingly fiddly. The Injective Property for Terms is spelled out and proved in Section 3.3. (This fact also goes by other names, including the Parsing Theorem and the Unique Readability Theorem.)

Because terms have both an inductive property and an injective property, this also gives us an important new kind of *recursive definition*, using syntactic structure. Say we want to define a function that assigns a value to every term. In order to define the value of the function for a certain term, we can assume that we have already defined the function for each of its *subterms.*

3.2.5 Example

Here's an example of a recursively defined function: the **complexity** function, which assigns a number to each term. The idea is that the complexity of a term is its total number of constants and function symbols. (This is *not* the same as its *length* as a string.) Here are some examples of some terms in the language of arithmetic with their complexities.

$$\begin{aligned} \texttt{0} &\mapsto 1 \\ \texttt{suc 0} &\mapsto 2 \\ \texttt{suc 0 + 0} &\mapsto 4 \\ \texttt{suc 0 · (suc 0 + 0)} &\mapsto 7 \end{aligned}$$

Here is the recursive definition:

$$\begin{aligned} \operatorname{complex} c &= 1 \\ \operatorname{complex}(f\,\texttt{(}t\texttt{)}) &= 1 + \operatorname{complex} t \\ \operatorname{complex}(f\,\texttt{(}t_1\texttt{,}t_2\texttt{)}) &= 1 + \operatorname{complex} t_1 + \operatorname{complex} t_2 \end{aligned}$$

As usual, recursive definitions work hand in hand with inductive proofs.

3.2.6 Example

For any closed term t,

$$\operatorname{complex} t \leq \operatorname{length} t$$

Proof

We will prove this by induction on the structure of terms.

1. Let c be any constant. Then

$$\text{complex}\, c = 1 \leq \text{length}\, c$$

since the constant c is required to be a non-empty string.

2. Let f be a one-place function symbol, and let t be any term. Suppose for the inductive hypothesis:

$$\text{complex}\, t \leq \text{length}\, t$$

Then

$$\begin{aligned}
\text{complex}\, f(t) &= 1 + \text{complex}\, t && \text{definition of complexity}\\
&\leq 1 + \text{length}\, t && \text{inductive hypothesis}\\
&< \text{length}\, f + 1 + \text{length}\, t + 1 &&\\
&= \ \text{length}\, f(t) &&
\end{aligned}$$

(since $f(t)$ consists of f, t, and the two parentheses joined together).

3. Let f be a two-place function symbol, and let t_1 and t_2 be terms. Suppose for the inductive hypothesis:

$$\begin{aligned}
\text{complex}\, t_1 &\leq \text{length}\, t_1\\
\text{complex}\, t_2 &\leq \text{length}\, t_2
\end{aligned}$$

Then

$$\begin{aligned}
\text{complex}\, f(t_1,t_2) &= 1 + \text{complex}\, t_1 + \text{complex}\, t_2\\
&\leq 1 + \text{length}\, t_1 + \text{length}\, t_2\\
&< \text{length}\, f + 1 + \text{length}\, t_1 + 1 + \text{length}\, t_2 + 1\\
&= \ \text{length}\, f(t_1,t_2)
\end{aligned}$$

□

Here's another important example of a recursively defined function on terms. In many ways, this is the most important example: it spells out how terms can be meaningful. Terms stand for objects in structures. For example, in the standard number structure, the term `suc 0 + 0` stands for the number 1. The same term can also stand for other things in other structures. What a term stands for depends on how we interpret its basic symbols. For example, in the structure from Example 3.1.7 which has Los Angeles buildings in its domain, the term `suc 0 + 0` stands for the Natural History Museum.

If we have an L-structure S, then we can map each L-term to the object in S which it is supposed to stand for. In general, each closed term **denotes** some object in S. Remember that a structure S provides some important information. For each constant, S gives us an extension c_S, which is a certain object in the domain of S. For each function symbol f, S gives us an extension f_S, which is a certain function from objects in the domain to other objects in the domain. We will use these extensions for the primitive symbols to build up the denotations of complex terms.

We can define the *denotation function* recursively. For a constant symbol c, the structure already tells us what it's supposed to stand for—this is its extension c_S. For a term built up using a function symbol f, we first work out what its component terms each denote, and then we apply the function f_S to the results. Here's the official definition.

3.2.7 Definition

Let L be a signature, and let S be an L-structure. The **denotation** of a term is defined recursively as follows.

1. Each constant symbol c denotes c_S, which is the extension of c in S.

2. Suppose that t denotes d. Then for any one-place function symbol f, $f(t)$ denotes $f_S d$, which is the result of applying the function which is the extension of f in S to d.

3. Suppose that t_1 denotes d_1 and t_2 denotes d_2. Then for any two-place function symbol f, the term $f(t_1,t_2)$ denotes $f_S(d_1, d_2)$, which is the result of applying the function which is the extension of f in S to d_1 and d_2.

The denotation of a term t in a structure S is labeled $[\![t]\!]_S$. (Accordingly, we can label the denotation *function* $[\![\cdot]\!]_S$, with a dot indicating where to write the function's argument.) Using this notation, we can rewrite the recursive definition more concisely.

$$\begin{aligned}
[\![c]\!]_S &= c_S && \text{for each constant } c \\
[\![f(t)]\!]_S &= f_S\,[\![t]\!]_S && \text{for each one-place function symbol } f \text{ and term } t \\
[\![f(t_1,t_2)]\!]_S &= f_S([\![t_1]\!]_S, [\![t_2]\!]_S) && \text{for each two-place function symbol } f \text{ and terms } t_1 \text{ and } t_2
\end{aligned}$$

We'll often leave off the S subscripts from the denotation function to keep our notation tidier, when it's clear in context which structure we're talking about.

3.2.8 Example

Use the definition of the denotation function to show that the term `suc suc 0 + suc 0` denotes the number three, in the standard model of arithmetic $\mathbb{N}$. That is,

$$[\![\texttt{suc suc 0 + suc 0}]\!]_{\mathbb{N}} = 3$$

(In our totally official notation, this term would be written

```
+(suc(suc(0)),suc(0))
```

But you don't have to bother with this, unless you really want to.)

Proof

$$\begin{aligned}
[\![\texttt{suc suc 0 + suc 0}]\!] &= [\![\texttt{suc suc 0}]\!] + [\![\texttt{suc 0}]\!] && \text{by the clause for the function symbol } \texttt{+} \\
&= \operatorname{suc}[\![\texttt{suc 0}]\!] + \operatorname{suc}[\![\texttt{0}]\!] && \text{by the } \texttt{suc} \text{ clause (twice)} \\
&= \operatorname{suc}\operatorname{suc}[\![\texttt{0}]\!] + \operatorname{suc}[\![\texttt{0}]\!] && \text{by the } \texttt{suc} \text{ clause again} \\
&= \operatorname{suc}\operatorname{suc}0 + \operatorname{suc}0 && \text{by the clause for the constant symbol } \texttt{0} \\
&= 2 + 1 = 3
\end{aligned}$$

□

3.2.9 Exercise

Use the definition of the denotation function in the standard string structure $\mathbb{S}$ to show the following:

(a) The term `("" ⊕ "A") ⊕ "B"` denotes the string `AB` in $\mathbb{S}$. That is,

$$[\![\texttt{("" ⊕ "A") ⊕ "B"}]\!]_{\mathbb{S}} = \texttt{AB}$$

(b) For any term t, the term t `⊕ ""` has the same denotation in $\mathbb{S}$ as t. That is,

$$[\![t \texttt{ ⊕ ""}]\!]_{\mathbb{S}} = [\![t]\!]_{\mathbb{S}}$$

3.2.10 Definition

For each number, there is a corresponding term in the language of arithmetic, which is called its **numeral**. The numeral for the number zero is the term `0`, the numeral for the number one is the term `suc 0`, the numeral for the number two is the term `suc suc 0` , and so on. For a number n, we'll call its numeral $\langle n\rangle$. We can make the definition of numerals explicit using a recursive definition—that is, a recursive definition on *numbers*.

$$\begin{aligned}\langle 0\rangle &= \mathtt{0}\\ \langle \operatorname{suc} n\rangle &= \mathtt{suc}\langle n\rangle \quad \text{for every } n \in \mathbb{N}\end{aligned}$$

(Use and mention can be a little confusing here, so I'll spell it out. Notice that the 0 on the left side of the definition is the *number* zero, while the `0` on the right side is a constant in the language of arithmetic. Similarly the suc on the left side is a function on numbers, while the `suc` on the right side is a one-place function symbol in the language of arithmetic.)

3.2.11 Exercise

(a) Prove by induction that for any number n, the numeral $\langle n\rangle$ denotes the number n, in the standard model of arithmetic. In short:

$$[\![\langle n\rangle]\!]_{\mathbb{N}} = n \quad \text{for every } n \in \mathbb{N}$$

(b) No two numbers have the same numeral. That is, for any numbers m and n, if $\langle m\rangle = \langle n\rangle$, then $m = n$. In other words, the numeral function is one-to-one.

In our official language of arithmetic, we have adopted a very simple "unary" notation for numbers. There are other ways of writing numbers down which are more familiar, such as *decimal notation*:

$$\begin{aligned}0 &\mapsto \mathtt{0}\\ 1 &\mapsto \mathtt{1}\\ 2 &\mapsto \mathtt{2}\\ &\vdots\\ 10 &\mapsto \mathtt{10}\\ 11 &\mapsto \mathtt{11}\\ &\vdots\end{aligned}$$

But while this notation is familiar, actually writing down the official rules for this system of notation is quite a bit trickier than our simple system. Other notational systems, like Roman numerals, are even more complicated. For our official purposes, it's helpful to keep things as simple as we can.

3.2.12 Definition

We'll call an L-structure **explicit** iff every element of its domain is denoted by some L-term.

3.2.13 Exercise

(a) Give an example of a structure which is not explicit.

(b) Show that the natural number structure $\mathbb{N}(0, \text{suc})$ is explicit.

(c) Show that the string structure $\mathbb{S}$ is explicit, by recursively defining a function that takes each string $s \in \mathbb{S}$ to some term $\langle s \rangle$ in the standard language of strings such that $[\![\langle s \rangle]\!]_{\mathbb{S}} = s$ (as in Exercise 3.2.11).

3.3 Parsing Terms*

We have seen some examples of inductive proofs and recursive definitions for closed terms, and a bit of practice applying these tools. These practical tools are the most important thing to understand about terms. But it's also worthwhile to understand the foundations on which these tools based.

The crucial property of terms is that every term can be built up from constants and function symbols in exactly one way. As with numbers and strings, we can split that property up into two properties. The **Inductive Property** says that every term can be built up in *at least one way* using these rules. The **Injective Property** says that no term can be built up in two different ways using these rules. In this section, first we'll state these properties more carefully. Next, we'll see how they can be proved.

The idea of the Inductive Property for terms is very similar to our official Inductive Properties for numbers and strings. It is a version of the kind of inductive property for recursively defined sets in Section 2.2. There are certain rules for building up terms: the rules for constants, one-place function symbols, and two-place function symbols. The idea of the inductive property is that every term can be reached in at least one way using these rules. This means that any set that includes everything you eventually reach by applying these rules includes every term.

Say you have a set X, and you want to prove by induction that every closed term is in X. You would have to show three things:

1. Every constant is in X.

2. For any one-place function symbol f and any t, if t is in X, then $f(t)$ is in X.
3. For any two-place function symbol f and any t_1 and t_2, if t_1 and t_2 are both in X, then $t_1(t_2)$ is in X.

What it means to say that proof by induction works is that, if X has these three properties, then X contains every term.

3.3.1 Definition

Let L be a signature. A set of strings X is **L-hereditary** iff

(a) Each constant in L is in X;

(b) For any one-place function symbol f and any $s \in X$, the string $f(s)$ is in X;

(c) For any two-place function symbol f and any $s_1, s_2 \in X$, the string $f(s_1,s_2)$ is in X.

(In fact, this is an application of the definition of "F-closed" in Definition 2.4.1 to certain operations for joining together strings.)

3.3.2 Inductive Property for Terms

Let L be a signature. If X is any L-hereditary set, then X contains every closed L-term.

This fact immediately follows from the recursive definition of terms, using Exercise 2.4.2.

The Inductive Property means that every term can be formed in *at least* one way using the formation rules for constants and function symbols. The other thing to check is that each term can be formed in *at most* one way from these rules: that is, no two different formation rules ever give the same result. This amounts to the fact that our system of notation does not have any syntactic ambiguity. (It is also called the *Unique Readability Theorem*, or the *Parsing Theorem.*) This is rather complicated to state.

3.3.3 Injective Property for Terms

Let L be a signature.

(a) c, $f(t)$, and $g(t_1,t_2)$ are all distinct from one another (for any constant c, one-place function symbol f, two-place function symbol g, and closed L-terms t, t_1, and t_2).

(b) If $f(t)$ is the same as $f'(t')$, then f is f' and t is t' (for any one-place function symbols f and f' and closed L-terms t and t').

(c) If $g(t_1,t_2)$ is the same as $g'(t'_1,t'_2)$, then g is g', t_1 is t'_1, and t_2 is t'_2 (for any two-place function symbols g and g' and closed L-terms t_1, t_2, t'_1, t'_2).

Like the Injective Properties for numbers and sequences, we can state this more elegantly in terms of functions. If L is a signature, let L_0 be the set of constants, let L_1 be the set of one-place function symbols, and let L_2 be the set of two-place function symbols. Let T be the set of L-terms. Consider three “term-forming” functions:

- $T_0 : L_0 \to T$ takes each constant to itself;
- $T_1 : L_1 \times T \to T$ takes each pair of a one-place function symbol f and a term t to the term $f(t)$;
- $T_2 : L_2 \times T \times T \to T$ takes each triple (f, t_1, t_2) of a two-place function f and two terms t_1 and t_2 to the term $f(t_1,t_2)$.

Then we can succinctly restate the Injective Property like this:

> Each of the term-forming functions T_0, T_1, and T_2 is one-to-one, and their ranges have no elements in common.

We can go a bit further, making this even more elegant and abstract, by combining all three of these term-builders into one big function

$$T_* : L_0 \cup (L_1 \times T) \cup (L_2 \times T \times T) \to T$$

where

$$\begin{aligned} T_*(c) &= T_0(c) && \text{for } c \in L_0 \\ T_*(f,t) &= T_1(f,t) && \text{for } (f,t) \in L_1 \times T \\ T_*(f,t_1,t_2) &= T_2(f,t_1,t_2) && \text{for } (f,t_1,t_2) \in L_2 \times T \times T \end{aligned}$$

For short, let

$$L(T) = L_0 \cup (L_1 \times T) \cup (L_2 \times T \times T)$$

This is the set of basic *term-constructors*—all of the different ways that a term can be build up. Then this is an even more concise way to state the Injective Property:

> The term-building function $T_* : L(T) \to T$ is one-to-one.

This brings us to a nice way of *proving* the Injective Property. What we are trying to prove is that our notation for terms is unambiguous: there is just one way of “parsing” a term. We can prove that every term can be parsed uniquely by actually writing a *parsing* function. This will be a function that breaks up a term into its parts.

What the parsing function should do is tell us, for each term, whether it is a constant, or if it is a term of the form $f(t)$, what the strings f and t are, or if it is a term of the form $f(t_1,t_2)$, what the strings f, t_1, and t_2 are. In other words, this function should take us *back* from a term to its *term-constructor.*

$$\text{parse} : T \to L_0 \cup (L_1 \times T) \cup (L_2 \times T \times T)$$

This function should have the following properties:

$\text{parse}(c) = c$ for each constant $c \in L_0$

$\text{parse}(f(t)) = (f,t)$ for each function symbol $f \in L_1$ and term t $\text{parse}(f(t_1,t_2)) = (f,t_1,t_2)$ for each function

A function like this will guarantee that we can recover the pieces a term is built up from, and so there must not be more than one way to build up the same term. We can restate those properties:

$\text{parse}\, T_0(c) = (c)$ for each constant $c \in L_0$

$\text{parse}\, T_1(f,t) = (f,t)$ for each function symbol $f \in L_1$ and term t $\text{parse}\, T_2(f,t_1,t_2) = (f,t_1,t_2)$ for each functio

Or we can state all three of these properties in one go:

$$\text{parse}(T_*(x)) = x \quad \text{for each term-constructor } x \in L(T)$$

In short, a parsing function is a one-sided *inverse* of the term-building function. If there is any function like this, then this shows that the term-building function T_* must be one-to-one (Example 1.2.24).

So in order to prove the Injective Property, what we need to do is write a parser. This a little bit complicated, and has a software engineering flavor. Programmers deal with this kind of string-parsing problem a lot, whenever they are dealing with structured data that has been flattened out (or “serialized”) in one long sequence of symbols.

Before we get into it, we will need to start by being a bit more explicit about our rules for what strings are allowed to be used as constants or function symbols. If you chose something perverse like `suc(x)` as one of your *constants*, you really could get ambiguities. To avoid this, we’ll just say that you aren’t allowed to use any parentheses or commas in your constants or function symbols.

3.3.4 Definition

(a) A symbol is a **delimiter** iff it is one of `,` , `(` , or `)` .

(b) A string is a **token** iff it does not contain any delimiters.

(c) A string is **delimiting** iff it begins with a delimiter or else is empty.

So our rule for signatures is that constants and function symbols have to be *tokens.*

(Another issue that comes up later is that we want to make sure that constants can be distinguished from *variables*, and also from the logical connectives in first-order logic. So officially we might want to be even more restrictive about what we get to use as constants or function symbols. But let's not worry about this right now.)

We'll say more about what exactly this means in a moment, but let's start with something basic. Consider the term `suc(+(0,0))` (written in official "prefix" notation). In order to break this down, the first thing we need to notice is that it starts with the function symbol `suc` . This is the initial *token* of the string. It's important that each string has a *unique* initial token. That is, if we take a token and stick a *delimiting* string onto the end of it, we can always figure out what the original token was.

3.3.5 Exercise

Recursively define a function $\mathrm{tok} : \mathbb{S} \to \mathbb{S} \times \mathbb{S}$ which splits up a string into two parts: a token, and a delimiting string. Prove by induction that for any token s and delimiting string t,

$$\mathrm{tok}(s \oplus t) = (s, t)$$

This means that if we have a term, we can figure out what *kind* of term it is, by identifying its first token, and whether this is a constant, a one-place function symbol, or a two-place function symbol. That's a good start, but we're not done. We also need to work out what the component *parts* of the term are. The hard part is the case of a term f`(`t_1`,`t_2`)`, which has two component terms. The terms t_1 and t_2 might include commas themselves. So in order to break up this term we need to figure out which is the "right" comma, the one that belongs to the outermost function symbol.

To write our parser in general, we'll start by writing what programmers call a "helper function." This solves a slightly more general problem: instead of just taking a string that represents a term and breaking it down, this will take a string that represents a term together with some extra stuff (maybe just a bunch of garbage symbols, for all we know) and gives us back *both* the term-constructor for the first part, and also the remaining symbols.

3.3.6 Lemma

There is a partial function p that takes strings to pairs of a term-constructor and a string (a partial function from $\mathbb{S}$ to $L(T) \times \mathbb{S}$), such that, for each term-constructor $x \in L(T)$ and delimiting string s,

$$p(T_*(x) \oplus s) = (x, s)$$

That is to say,

$$\begin{aligned} p(c \oplus s) &= (c, s) && \text{for each } c \in L_0 \\ p(f\,(t) \oplus s) &= ((f, t), s)) && \text{for each } f \in L_1 \text{ and } t \in T \\ p(f\,(t_1,t_2) \oplus s) &= ((f, t_1, t_2), s) && \text{for each } f \in L_2 \text{ and } t_1, t_2 \in T \end{aligned}$$

Proof

The function p can be defined using a *partial recursive* definition [REF]. Recall from Section 4.6 that there is a function $\mathrm{tok} : \mathbb{S} \to (\mathbb{S}, \mathbb{S})$ that splits up a string into a *token* (without any delimiters) and a delimiting string. Then the idea is, for any string s, there are three different cases, corresponding to the three ways of building up terms.

- Suppose there is a constant $c \in L_0$ and a string s' such that

  $$\mathrm{tok}\, s = (c, s')$$

  In that case,

  $$p(s) = (c, s')$$

- Suppose there are a function symbol $f \in L_1$, a term-constructor $x \in L(T)$, and strings s' and s'' such that

  $$\begin{aligned} \mathrm{tok}\, s &= (f, (\oplus s') \quad \text{and} \\ p(s') &= (x,) \oplus s'') \end{aligned}$$

  In that case,

  $$p(s) = ((f, T_*(x)), s'')$$

  The idea here is that if we find a function symbol followed by a left parenthesis, we consume as much of the remaining string as is necessary to produce a term, after which we should find the matching right parenthesis.

- Suppose there are a two-place function symbol $f \in L_2$, term-constructors $x_1, x_2 \in L(T)$, and strings $s', s'', s''' \in \mathbb{S}$ such that

$$\begin{aligned} \operatorname{tok} s &= (f, \texttt{(} \oplus s') && \text{and} \\ p(s') &= (x_1, \texttt{,} \oplus s'') && \text{and} \\ p(s'') &= (x_2, \texttt{)} \oplus s''') \end{aligned}$$

  In that case,

$$p(s) = ((f, T_*(x_1), T_*(x_2)), s''')$$

  That is, if the string starts with a two-place function symbol followed by a left parenthesis, we should consume a term from the remaining string, then a comma, then another term, and then a right parenthesis, leaving alone whatever is left after all of that.

If none of these three cases apply, then $p(s)$ is undefined. (So as we said, p is a partial function on strings.)

Once we have this definition of p in place, we can prove that it has the property we want: for any string which has the form $T_*(x) \oplus s$ for some term-constructor x and delimiting string s,

$$p(T_*(x) \oplus s) = (x, s)$$

We will prove this by "strong induction" on strings: we assume for our inductive hypothesis that every string *shorter* than $T_*(x) \oplus s$ has this property. Now, there are three cases to consider, depending on which kind of term-constructor x is.

1. If x is a constant c, then $T_*(x) = c$. In this case,

$$\operatorname{tok}(c \oplus s) = (c, s)$$

   (This uses the "tokenization lemma" Exercise 3.3.5: c is a token and s is a delimiting string.) So it is clear from the first part of the definition of p that

$$p(c \oplus s) = (c, s)$$

2. If x is (f, t) for some one-place function symbol f and term t, then $T_*(x) = f \texttt{(} t \texttt{)}$. In this case,

$$\operatorname{tok}(f \texttt{(} t \texttt{)} \oplus s) = (f, \texttt{(} \oplus t \oplus \texttt{)} \oplus s)$$

Here t is a term, so there is some term-constructor $y \in L(T)$ such that $t = T_*(y)$. Then, by the inductive hypothesis,

$$p(t \oplus \mathtt{)} \oplus s) = (y, \mathtt{)} \oplus s)$$

So by the second part of the definition of p,

$$p(\mathtt{f(t)} \oplus s) = ((f, T_*(y)), s) = ((f, t), s)$$

3. The third case is left as an exercise. □

3.3.7 Exercise
Use the "helper function" p to define the parsing function parse (a partial function from strings to term-constructors), and use the properties of p to complete the proof of the Injective Property for Terms.

3.4 Recursion for Terms*

In Section 3.2 we saw some examples of functions which are defined recursively using the syntactic structure of terms, like the complexity function and the denotation function. This is analogous to recursive definitions for numbers and sequences. Just like those kinds of recursive definition, recursive definitions for terms are based on a general Recursion Theorem. Intuitively, what this theorem says is just that *recursive definitions work*: that is, if you write down a recursive definition, you will have successfully described one and only one function defined for every closed term. This theorem a bit tricky to state in general, because the structure of terms is a bit more complicated than the structure of numbers. But it works very similarly.

First, recall that when we stated the Injective Property for Terms, we used the functions T_0, T_1, and T_2, which build up terms from constants, one-place function symbols, and two-place function symbols, respectively. We'll use these functions again to state the Recursion Theorem for Terms.

3.4.1 The Recursion Theorem for Terms
Let L be a signature. Let L_0, L_1, and L_2 be its set of constants, one-place function symbols, and two-place function symbols, respectively. Let T be the set of closed

L-terms. We have three term-building functions:

$$\begin{aligned}
T_0 &: L_0 \to T \\
T_1 &: L_1 \times T \to T \\
T_2 &: L_2 \times T \times T \to T
\end{aligned}$$

Now, let A be any set, and consider any three functions with the same shape:

$$\begin{aligned}
f_0 &: L_0 \to A \\
f_1 &: L_1 \times A \to A \\
f_2 &: L_2 \times A \times A \to A
\end{aligned}$$

Then there is a unique function $r : T \to A$ with the following Recursive Properties:

$$\begin{aligned}
rc &= f_0 c && \text{for each constant } c \\
r(f(t)) &= f_1(f, rt) && \text{for each one-place function symbol } f \text{ and term } t \\
r(f(t_1, t_2)) &= f_2(f, (rt_1, rt_2)) && \text{for each two-place function symbol } f \text{ and terms } t_1 \text{ and } t_2
\end{aligned}$$

Here is another more elegant and abstract way of putting that. The term-building functions can be squished together into a single function

$$T_* : L(T) \to Tthattakesterm - constructorsin$$

This takes "term-constructors" (elements of $L(T) = L_0 \cup (L_1 \times T) \cup (L_2 \times T \times T))$ to terms. We can similarly squish together the "step functions" for the recursively defined functions into a single function defined on "constructors,"

$$f_* : L(A) \to A$$

In particular, we can define this function f_* piecewise, just by sticking together the three functions f_0, f_1, and f_2.

$$\begin{aligned}
f_*(c) &= f_0(c) && \text{for each } c \in L_0 \\
f_*(f, a) &= f_1(f, a) && \text{for each } f \in L_1 \text{ and } a \in A \\
f_*(f, a_1, a_2) &= f_2(f, a_1, a_2) && \text{for each } f \in L_1 \text{ and } a_1, a_2 \in A
\end{aligned}$$

The Recursion Theorem for Terms tells us that we can build up a recursive function using any set A and function $f_* : L(A) \to A$. This recursive function should "hook up" the term-constructors T_* with the function-constructors f_*, using the Recursive Properties.

We can also restate these Recursive Properties in an elegant way, using the concept of a *functor.* The signature L gives us a way of "lifting" any set X to a set $L(X)$ of "X-constructors. It also gives us a way of"lifting" *functions,* in a natural way. For any sets X and Y, and any function $g : X \to Y$, we can define a new function $Lg : L(X) \to L(Y)$. The idea is that this function takes each kind of constructor, and uses g to convert all of its pieces, turning the X-constructor into a Y-constructor.

$$\begin{aligned} Lg(c) &= c && \text{for every } c \in L_0 \\ Lg(f, x) &= (f, gx) && \text{for every } f \in L_1 \text{ and } x \in X \\ Lg(f, x_1, x_2) &= (f, gx_1, gx_2) && \text{for every } f \in L_2 \text{ and } x_1, x_2 \in X \end{aligned}$$

Using all of this repackaging, we can state the Recursion Theorem all in one go.

> For any set A and function $f_* : L(A) \to A$, there is a unique function $r : T \to A$ such that, for each term-constructor $x \in L(T)$,
>
> $$r(T_*(x)) = f_*(Lr(x))$$

We can summarize the Recursion Theorem for Terms in a neat little diagram.

$$\begin{tikzcd} L(T) \arrow[r, "T_*"] \arrow[d, dotted, "Lr"] & T \arrow[d, dotted, "r"] \\ L(A) \arrow[r, "f_*"] & A \end{tikzcd}$$

The diagram means that if you follow either path from a constructor $x \in L(T)$ to an element of A (either first construct a term using T_* and then apply r, or else first use Lr to turn it into an A-constructor and then apply the "step rule" f_*), you end up at the same place. (This is called a *commutative diagram.*)

Of course, *anything* can be restated very concisely if you just introduce complicated enough definitions. The point of restating the Recursion Theorem this way is that it uses notational tricks to hide away the messy details of constructing and parsing *terms*, and puts out in the open the basic structure that makes the function r *recursive.* For in fact, all recursively defined functions—including those on numbers and strings—can be represented using the same kind of diagram.

Take numbers. Numbers are build up using a zero element, and a successor function. So the "signature" of this inductive definition has one constant and one one-place function. Let L be this signature (`0`, `suc`). Then we can package the basic operations on numbers into a single function

$$s : L(\mathbb{N}) \to \mathbb{N}$$

which is defined by

$$\begin{aligned} s(\mathtt{0}) &= 0 \\ s(\mathtt{suc}, n) &= \operatorname{suc} n \qquad \text{for every } n \in \mathbb{N} \end{aligned}$$

Then the Recursion Theorem for Numbers says:

> For any set A and any function $f_* : L(A) \to A$, there is a unique function $r : \mathbb{N} \to A$ such that
>
> $$r(sx) = f_*(Lr(x)) \quad \text{for every } x \in L(\mathbb{N})$$
>
> $$\begin{tikzcd} L(\mathbb{N}) \arrow[r, "s"] \arrow[d, dotted, "Lr"] & \mathbb{N} \arrow[d, dotted, "r"] \\ L(A) \arrow[r, "f_*"] & A \end{tikzcd}$$

Basically, this statement just repackages the "starting point" and "step rule" in a recursive definition of a function on numbers, sticking both of them together in a single function.

You can do something similar with strings, by thinking about the strings as a structure with a signature which has a constant for the empty string, and a one-place function symbol for the function $s \mapsto a \oplus s$ for each symbol a in the alphabet.

We can also restate the Inductive Properties for these different structures. Given a function $s : L(A) \to A$, we can call a set $X \subseteq A$ s-**closed** iff, for each $x \in L(X)$, sx is an element of X. For example, if s is the function that bundles together zero and successor, this says that 0 is in X and every successor of an element of X is also in X. Then the generic *Inductive Property* for $s : L(A) \to A$ says:

> If X is any s-closed set, then X includes every element of A.

And the generic *Injective Property* for $s : L(A) \to A$ says:

> s is one-to-one.

So far, we have only *stated* the Recursion Theorem for Terms (in several ways); we still need to *prove* it. In fact, we'll go ahead and prove a slightly more general and abstract recursion theorem, which *also* will cover the Recursion Theorem for Numbers and the Recursion Theorem for Strings, all at once.

3.4.2 The Recursion Theorem (General Version)

Let L be any signature, let A be any set, and suppose that

$$s : L(A) \to A$$

has the Inductive Property and the Injective Property. Then for any set B and function

$$f : L(B) \to B$$

there is a unique function $r : A \to B$ with the Recursive Property

$$r(sx) = f(Lr(x)) \quad \text{for every } x \in L(A)$$

In a diagram:

$$\begin{tikzcd}
L(A) \arrow[r, "s"] \arrow[d, dotted, "Lr"] & A \arrow[d, dotted, "r"] \\
L(B) \arrow[r, "f"] & B
\end{tikzcd}$$

The proof of this theorem is structurally exactly parallel to the proof of the Recursion Theorem for Numbers (Principle 2.4.4). It is just a lot more abstract.

Proof
First we can recursively define a relation. For $a \in A$ and $b \in B$, we recursively define what it is for a to **select** b. (Since the letter f is taken, we'll use the letter c for function symbols as well as constants here.)

$$\frac{}{s(c) \text{ selects } f(c)} \text{ for each } c \in L_0 \qquad \frac{a \text{ selects } b}{s(c,a) \text{ selects } f(c,a)} \text{ for each } c \in L_1 \qquad \frac{a_1 \text{ selects } b_1 \qquad a_2 \text{ selects } b_2}{s(c,a_1,a_2) \text{ selects } f(c,a_1,a_2)} \text{ f}$$

We can sum these up with one general rule. We can define an extended notion of "L-selects" that applies to "constructors" in $L(A)$ and $L(B)$. In particular,

$$\begin{array}{rcll}
c \ L\text{-selects } c & & & \text{for every } c \in L_0 \\
(c,a) \ L\text{-selects } (c,b) & \text{iff} & a \text{ selects } b & \text{for every } c \in L_1 \\
(c,a_1,a_2) \ L\text{-selects } (c,b_1,b_2) & \text{iff} & a_1 \text{ selects } b_1 \text{ and } a_2 \text{ selects } b_2 & \text{for every } c \in L_2
\end{array}$$

In these terms, the inductive definition of "selects" says simply, for each $x \in L(A)$ and $y \in L(B)$,

$$\frac{x \ L\text{-selects } y}{sx \text{ selects } fy}$$

This inductive definition straightforwardly implies:

> For any $a \in A$ and $b \in B$, a selects b iff there is some $x \in L(A)$ and $y \in L(B)$ such that x L-selects y, $a = sx$ and $b = fy$.

Next we can prove:

> For each $a \in A$, there is exactly one $b \in B$ such that a selects b.

We prove this by the Inductive Property of s. Let $x \in L(A)$. We can suppose for the inductive hypothesis that there is exactly one $y \in L(B)$ such that x L-selects y. (Putting the hypothesis this way is skipping over some "packing" and "unpacking" details for constructors.) We have to show that there is also exactly one $b \in B$ such that sx selects b. The existence part is clear: by the definition, sx selects $f y$. Now for uniqueness, let $b \in B$, and suppose that sx selects b. We have already noted that this implies that there is some y' such that x L-selects y' and $b = f y'$. Our inductive hypothesis tells us that if x L-selects y', then $y = y'$. So $b = f y$, which is what we wanted to show.

We can conclude that there is a unique function $r : A \to B$ such that

$$r(a) = b \quad \text{iff} \quad a \text{ selects } b \quad \text{for every } a \in A, b \in B$$

This also implies (by unpacking the definition of "L-selects" for each kind of constructor):

$$Lr(x) = y \quad \text{iff} \quad x \ L\text{-selects } y \quad \text{for every } x \in L(A), y \in L(B)$$

Finally, we can check that r has the Recursive Property. In fact, for any $b \in B$, we have

$$r(sx) = b \quad \text{iff} \quad sx \text{ selects } b$$

The right-hand side holds iff there is some $y \in L(B)$ such that x L-selects y and $b = f y$. And this holds iff $Lr(x) = y$. That is, $r(sx) = b$ iff $b = f(Lr(x))$. □

3.5 Variables

So far our term language is pretty limited. We can use it to label particular objects in a structure—and that's it. In this section we'll extend our language to make it more flexible, so we can also build up complex labels for *functions*, going beyond just the basic function symbols. The key idea is to use symbols which don't have a *fixed* interpretation. They're called "variables", because their denotations can vary within a single structure.

In the language of arithmetic, we can use `suc 0 + suc suc 0` to label the number three; and we can use `+` to label the addition function, or `suc` to label the successor function. But how about the "add two" function?

$$[\quad 0 \mapsto 2, \quad 1 \mapsto 3, \quad 2 \mapsto 4, \quad \dots \quad]$$

Or how about the doubling function?

$$[\quad 0 \mapsto 0, \quad 1 \mapsto 2, \quad 2 \mapsto 4, \quad \dots \quad]$$

We can represent these functions using a language with variables. For instance, the "add two" function can be represented by the term `suc suc x` . ("For each *x*, take the successor of the successor of *x*.") Similarly, the doubling function can be represented by the term `x · suc suc 0` . ("For each *x*, multiply *x* by 2.") Of course, these aren't the only options. We could also use `x + suc suc 0` for the "add two" function, or `x + x` for the doubling function. One of the important questions we'll consider is when two different terms are *equivalent*, in the sense of representing the same function.

In what follows, we'll suppose that we have a fixed infinite set of **variables**. Officially, we'll say that each variable is the letter `x` , `y` , or `z` , perhaps followed by some subscripted numerals, like $\texttt{x}_0$, $\texttt{x}_{12}$, etc. But unofficially, sometimes we'll use other expressions for variables when it happens to be convenient.

We're going to extend our definition of the term language. In Section 3.2, we defined the *closed* terms—in this context, "closed" just means "with no variables". Now we'll define the terms in general. We can do this in just the same way as before, by adding one extra formation rule to the three we had before.

3.5.1 Definition

The **terms** for a signature L are recursively defined by the following four rules:

$$\frac{x \text{ is a variable}}{x \text{ is a term}}$$

$$\frac{c \text{ is a constant}}{c \text{ is a term}}$$

$$\frac{f \text{ is a one-place function symbol} \qquad t \text{ is a term}}{f\,\texttt{(}t\texttt{)} \text{ is a term}}$$

$$\frac{g \text{ is a two-place function symbol} \qquad t_1 \text{ and } t_2 \text{ are terms}}{g\,\texttt{(}t_1\texttt{,}t_2\texttt{)} \text{ is a term}}$$

(If you're paying very close attention, you might notice something tricky about use and mention in this definition. In the formation rule for variables, we are using a "meta-linguistic" variable *x*, which can stand for any "object language" variable. For example, the variable rule tells that, since `y` is a variable, `y` is also a term, and since $\texttt{z}_2$ is a variable, $\texttt{z}_2$ is also a term. As one instance of the rule, `x` is a variable, so `x` is a term. But in the rule, *x* can be any variable, not just `x` !)

Terms with variables also have an Inductive Property and an Injective Property, just like closed terms, but with some extra clauses about variables. But we won't worry about making this totally official, since hopefully you have the hang of the idea. The key thing about this definition is that our two key tools still work: inductive proof, and recursive definition. (When it comes to the Injective Property, one detail is that we have to impose extra restrictions on the language to make sure that constants and variables don't clash.)

3.5.2 Technique (Induction on Terms)

Suppose that we want to show that every L-term is *nice* (for some signature L). We can do this in four steps. (Three of these steps are exactly the same as in Technique 3.2.2.)

1. Let x be any variable. *Show* that x is nice.
2. Let c be any constant. *Show* that c is nice.
3. Let f be any one-place function symbol, and let t be any closed term. *Suppose* that t is nice. (This is the *inductive hypothesis.*) Then *show* that the term $f(t)$ is also nice.
4. Let f be any two-place function symbol, and let t_1 and t_2 be any closed terms. *Suppose* that t_1 is nice and t_2 is nice. (Again, this is the *inductive hypothesis.*) Then *show* that the term $f(t_1,t_2)$ is also nice.

Here is an example of a recursive definition using the full definition of terms.

3.5.3 Definition

For a variable x, we define the terms that x **occurs in** recursively as follows:

1. For each variable y, x occurs in y iff y just is x.
2. For each constant c, then x does not occur in c.
3. For each one-place function symbol f and each term t, x occurs in $f(t)$ iff x occurs in t.
4. For each two-place function symbol f and terms t_1 and t_2, x occurs in $f(t_1,t_2)$ iff x occurs in t_1 or x occurs in t_2.

Here's an alternative way of stating this definition which makes its recursive form a bit more explicit. We can recursively define a function Var that takes each term to the *set* of variables that occur in that term:

$$
\begin{aligned}
\operatorname{Var} x &= \{x\} && \text{for each variable } x\\
\operatorname{Var} c &= \varnothing && \text{for each constant } c\\
\operatorname{Var} f(t) &= \operatorname{Var} t && \text{for each one-place function symbol } f \text{ and term } t\\
\operatorname{Var} f(t_1,t_2) &= \operatorname{Var} t_1 \cup \operatorname{Var} t_2 && \text{for each two-place function symbol } f \text{ and terms } t_1 \text{ and } t_2
\end{aligned}
$$

Then, finally, we say x occurs in t iff $x \in \operatorname{Var} t$.

3.5.4 Definition

We say t is a **term of one variable** iff at most one variable occurs in t. Similarly, t is a **term of two variables** iff at most two variables occur in t, and so on. We'll often use the label $t(x)$ for a term in which at most the variable x occurs, and similarly $t(x, y)$ for a term of two variables in which at most x and y occur, etc.

3.5.5 Technique (Recursively definining a function on terms)

Suppose that we want to come up with a function r whose domain is the set of all L-terms. We can do this in four steps.

1. Fill in the blank:

   For each variable x, $r(x) =$ _______________

2. Fill in the blank:

   For each constant c, $r(c) =$ _______________

3. Fill in the blank:

   For each one-place function symbol f and term t, $r(f(t)) =$ _______________

   This time, the description you write down can depend on the value of $r(t)$.

4. Fill in the blank:

   For each two-place function symbol f and terms t_1 and t_2, $r(f(t_1,t_2)) =$ _______________

   This time you can use both $r(t_1)$ and $r(t_2)$.

3.5.6 Example

A variable is like a hole in a term. One useful thing to do is plug the hole up with another term. Here are some examples of what happens when we plug the term `0 + 0` into the `x`-spot in various terms:

```
suc suc x   ↦   suc suc (0 + 0)
x + suc x   ↦   (0 + 0) + suc (0 + 0)
x + suc y   ↦   (0 + 0) + suc y
    y + y   ↦   y + y
```

We'll now give a precise definition of the "plugging in" operation. Once again, this definition is recursive. The intuitive idea is that whenever we meet a function term $t_1(t_2)$, we just apply the substitution to each of its inner terms, until eventually we reach the constants and variables. At this point, if it's the variable we want, then we replace it; otherwise we leave it alone.

3.5.7 Definition

Suppose that x is a variable and a is term. Then for any term t, the **substitution instance** $t[x \mapsto a]$ is the result of replacing each occurrence of x in t with a. We can recursively define the function that takes each term t to $t[x \mapsto a]$ as follows.

1. For each variable y,

$$y[x \mapsto a] \quad = \quad \begin{cases} a & \text{if } y \text{ is } x \\ y & \text{otherwise} \end{cases}$$

2. For each constant c,

$$c[x \mapsto a] \quad = \quad c$$

3. For each one-place function symbol f and term t,

$$f(t)[x \mapsto a] \quad = \quad f(t[x \mapsto a])$$

4. For each two-place function symbol f and terms t_1 and t_2,

$$f(t_1,t_2)[x \mapsto a] \quad = \quad f(t_1[x \mapsto a],t_2[x \mapsto a])$$

3.5.8 Notation

This "function-style" notation $t[x \mapsto a]$ isn't very standard. It's more common to use the "slash" notation $t[a/x]$. But I've always found this a bit harder to read. (Everyone forgets which side of the slash the variable is supposed to go on.)

Sometimes we'll use a more concise notation for variable substitution. Suppose $t(x)$ is a formula of one variable x. Calling the term "$t(x)$" tells us that x is the important variable. So instead of writing $t[x \mapsto a]$, we can more simply just write $t(a)$. In this case, it's clear in context which variable is supposed to be replaced by a. If instead we were talking about a term $u(y)$, then $u(a)$ would mean $u[y \mapsto a]$.

Similarly, if $t(x, y)$ is a formula of two variables, then $t(a, y)$ would mean the same thing as $t[x \mapsto a]$, and $t(x, a)$ would mean the same thing as $t[y \mapsto a]$. Again, this notation relies on making the choice of variables (and their order) clear in context.

We will use the "function-style" notation $t[x \mapsto a]$ whenever we need to avoid ambiguity about which variable we are talking about.

But the $t(x)$ style notation raises a concern. Say we have a three terms $t(x)$, $u(y)$, and a. Then what does the notation $t(u(a))$ mean? It's potentially ambiguous. Does it mean to plug $u(a)$ into $t(x)$? Or does it mean to plug a into $t(u(y))$? Fortunately, this ambiguity is harmless, because of the following fact.

3.5.9 Proposition
Suppose $t(x)$ and $u(y)$ are terms of one variable, and a is a term. Then these are the very same term:

$$t[x \mapsto u][y \mapsto a] \quad = \quad t\big[x \mapsto u[y \mapsto a]\big]$$

The left-hand side is what you get by plugging a into $t(u(y))$, and the right-hand side is what you get by plugging $u(a)$ into $t(x)$.

Proof
We can prove this by induction on the structure of $t(x)$. The notation here gets messy. To simplify it a bit, in this proof let's just write $[u]$ instead of $[x \mapsto u]$, and $[a]$ instead of $[y \mapsto a]$. So what we're trying to show is that, for any term t,

$$t[u][a] = t\big[u[a]\big] \tag{3.1}$$

Even though the notation is kind of awkward and ugly, this proof really just amounts to straightforward mechanical checking. We just have to be very careful to set up our inductive steps correctly, apply our recursive definitions carefully, and pay close attention to the brackets.

1. First we'll show that each variable has the property (3.1). Since we are only showing that (3.1) holds for terms whose only variable is x, we only need

to show this for the variable x. We know that plugging anything into x just gives the same thing straight back: $x[u] = u$, and $x[u[a]] = u[a]$. Putting that together:

$$x[u][a] = u[a] = x[u[a]]$$

2. Let c be a constant. In this case, plugging in any term just produces c again. So:

$$c[u][a] = c[a] = c = c[u[a]]$$

3. Let f be a one-place function symbol and let t be a term. For this step, we can assume for our inductive hypothesis:

$$t[u][a] = t[u[a]]$$

We want to show that (3.1) applies to $f(t)$: that is, we want to show

$$f(t)[u][a] = f(t)[u[a]]$$

We'll use the recursive definition of substitution three times now.

$$\begin{aligned} f(t)[u][a] &= f(t[u])[a] && \text{definition of substitution} \\ &= f(t[u][a]) && \text{definition of substitution} \\ &= f(t[u[a]]) && \text{inductive hypothesis} \\ &= f(t)[u[a]] && \text{definition of substitution} \end{aligned}$$

4. The step for two-place function symbols is similar.

This completes the induction. □

3.5.10 Exercise
Prove by induction that if x doesn't occur in t, then $t[x \mapsto a] = t$.

3.5.11 Exercise
If $t(x)$ is a term of one variable, then the variables that occur in $t(a)$ are the same as the variables that occur in a.

3.5.12 Exercise
Let $t(x)$ be a term of one variable, and let a and a' be closed terms. Prove by induction on the structure of $t(x)$ that, if a and a' denote the same value in a

structure S, then $t(a)$ and $t(a')$ also denote the same value in S. That is,

$$\text{If} \quad [\![a]\!]_S = [\![a']\!]_S \quad \text{then} \quad [\![t(a)]\!]_S = [\![t(a')]\!]_S$$

In Section 3.2 we defined the *denotation* of a term in a structure: the object that the term stands for in that structure. Our next job is to extend this definition to apply to terms with variables. But this time it's a little trickier. If we are finding the denotation of the term `suc x`, what should the variable `x` stand for? A variable doesn't pick out any one thing once and for all. So we won't define a "once and for all" denotation of a term that contains variables. Instead, we can interpret a term *with respect to a choice of values for for its variables.* First, let's define what we mean by a "choice of values."

3.5.13 Definition

Let S be a structure, and let D be the domain of S. A **variable assignment** (or just an *assignment*, for short) is a partial function from V to D: that is, it picks out values in S for (some of) the variables in V.

We call an assignment **adequate** for a term t iff its domain includes every variable that occurs in t.

(It's common in other texts to require that variable assignments be *total* functions, defined for every variable. But this is unnecessary, and it can occasionally be a bit of a nuisance, so we won't require it.)

We can now interpret terms with variables simply by adding one extra clause to our old recursive definition of the denotation function.

3.5.14 Definition

Let S be a structure and let g be an assignment. We recursively define the **denotation of t with respect to g (in S)** as follows.

1. Each variable x denotes gx, with respect to g.

2. Each constant c denotes c_S, with respect to g.

3. For each one-place function symbol f and term t, if t denotes d with respect to g, then $f(t)$ denotes $f_S d$ with respect to g.

4. For each two-place function symbol f and terms t_1 and t_2, if t_1 denotes d_1 with respect to g and t_2 denotes d_2 with respect to g, then $f(t_1,t_2)$ denotes $f_S(d_1, d_2)$ with respect to g.

As in Section 3.2, we also use the more concise notation $[\![t]\!]_S\, g$. So we can restate the definition more concisely.

$$\begin{aligned}
[\![x]\!]_S\, g &= g(x) \\
[\![c]\!]_S\, g &= c_S \\
[\![\texttt{f(}t\texttt{)}]\!]_S\, g &= f_S([\![t]\!]_S\, g) \\
[\![\texttt{f(}t_1\texttt{,}t_2\texttt{)}]\!]_S\, g &= f_S([\![t_1]\!]_S\, g, [\![t_2]\!]_S\, g)
\end{aligned}$$

We often drop the S subscript when it's clear in context which structure we're talking about. So here it is again, a bit tidier:

$$\begin{aligned}
[\![x]\!]\, g &= g(x) \\
[\![c]\!]\, g &= c_S \\
[\![\texttt{f(}t\texttt{)}]\!]\, g &= f_S([\![t]\!]\, g) \\
[\![\texttt{f(}t_1\texttt{,}t_2\texttt{)}]\!]\, g &= f_S([\![t_1]\!]\, g, [\![t_2]\!]\, g)
\end{aligned}$$

The definition only really "looks at" the assignment in the case of variables, but we had to modify the other parts of the definition to make sure they pass the assignment down to their parts, so that we have it available when we reach the variables.

3.5.15 Example

Recall that $\mathbb{S}$ is the standard string structure. Let g be the assignment $[\texttt{x} \mapsto \texttt{ABC}]$. The term `x ⊕ "D"` denotes `ABCD` with respect to g in $\mathbb{S}$. We can show this explicitly using the definition.

$$\begin{aligned}
[\![\texttt{x ⊕ "D"}]\!]\, g &= [\![\texttt{x}]\!]\, g \oplus [\![\texttt{"D"}]\!]\, g && \text{since } \texttt{⊕}_{\mathbb{S}} \text{ is } \oplus \\
&= [\![\texttt{x}]\!]\, g \oplus \texttt{D} && \text{since } \texttt{"D"}_{\mathbb{S}} = \texttt{D} \\
&= \texttt{ABC} \oplus \texttt{D} && \text{since } g(x) = \texttt{ABC} \\
&= \texttt{ABCD}
\end{aligned}$$

3.5.16 Exercise

Suppose g and h are variable assignments in some structure S which have the same value for each variable that occurs in t. (In particular, they are both adequate for t.) Prove by induction that t has the same denotation with respect to g as it has with respect to h. That is to say:

$$[\![t]\!]\, g = [\![t]\!]\, h$$

Note in particular that if t is a *closed* term, then this shows that t denotes the same value with respect to any assignment at all. This is the same as the denotation we defined in the last section.

3.5.17 Notation

A variable assignment is a way of associating some objects with some variables. But often it will be clear in context which variables are important. In this case, we can keep our notation cleaner and simpler by just talking about the objects, and keeping the variables in the background. This is similar to our simplified notation for variable substitution, $t(a)$, where we leave the variable x in the background and only mention the term a.

If it is clear in context that the important variable is x—for instance, because we have been talking about a term $t(x)$—then we'll sometimes just talk about an object d in the domain of a structure, as a shorthand for the assignment $[x \mapsto d]$. Similarly, if it is clear in context that the important variables are x and y, in that order—for instance, because we have been talking about a term $t(x, y)$—then we can use a pair of objects (d_1, d_2) as a stand-in for the assignment $[x \mapsto d_1, y \mapsto d_2]$. This often simplifies our notation quite a bit.

Here are some contexts in which we will often do this.

3.5.18 Definition

Let S be a structure, and let $t(x)$ be a term of one variable. For any d in the domain of S, the **denotation of** $t(x)$ **at** d (in S) is the denotation of t with respect to the variable assignment $[x \mapsto d]$ (in S).

Similarly, suppose $t(x, y)$ is a term of two variables. Then for any $d_1, d_2 \in D$, the **denotation of** t **at** (d_1, d_2) (in S) is the denotation of t with respect to the assignment $[x_1 \mapsto d_1, x_2 \mapsto d_2]$ (in S).

This generalizes straightforwardly to terms of n variables and sequences of n objects.

3.5.19 Definition

Let S be a structure and let $t(x)$ be a term of one variable. The **extension** of $t(x)$ (in S) is the function that takes each $d \in D_S$ to the denotation of $t(x)$ at d. We use the notation $[\![t]\!]_S$ for the extension of t in S. That is,

$$[\![t]\!]_S(d) = [\![t]\!]_S[x \mapsto d]$$

(Notice that in this notation the variable x is left implicit.)

Similarly, if $t(x_1, x_2)$ is a term of two variables, then the **extension** of t (in S) is the two-place function given by

$$\llbracket t \rrbracket_S(d_1, d_2) = \llbracket t \rrbracket_S[x_1 \mapsto d_1, x_2 \mapsto d_2]$$

It's clear how to extend this to terms of n variables.

We have two different things we can do with a term $t(x)$ with a free variable. Earlier we defined a *syntactic* operation of plugging a term a in where the free variable is, to produce another term $t(a)$. Now we have also defined a *semantic* operation of evaluating the denotation of $t(x)$ at a certain object d. It's important to keep track of the difference between these two operations. Intuitively, substitution relates bits of language to other bits of language, while denotation relates bits of language to things "out in the world." But these two ideas are closely related, in the following way.

3.5.20 Exercise

Let $t(x)$ be a term of one variable, and let a be a closed term. Suppose a denotes d. Prove by induction that the denotation of $t(x)$ at d is the same as the denotation of $t(a)$. That is, $\llbracket t \rrbracket(d) = \llbracket t(a) \rrbracket$. To sum this fact up very concisely:

$$\llbracket t \rrbracket(\llbracket a \rrbracket) = \llbracket t(a) \rrbracket$$

Or in notation which is more explicit about the variable x:

$$\llbracket t \rrbracket[x \mapsto \llbracket a \rrbracket] = \llbracket t[x \mapsto a] \rrbracket$$

(Notice that in this equation, the arrow notation on the left stands for an *assignment*, which maps x to an object in the domain of a structure, while the similar notation on the right stands for a *substitution*, which maps x to another term in the language.)

Using just our primitive symbols like `suc`, `+`, or `⊕`, we could describe a few basic functions. But now that we have complex terms and variables, we can now describe lots more. Say D is the domain of a structure, and f is a function from D to D. We can "describe" or "express" f if we can find a term $t(x)$ such that "applying" $t(x)$ has the same effect as applying f. For any d, the denotation of $t(x)$ at d should be the same as the value of $f d$. In other words, f should be the *extension* of $t(x)$.

3.5.21 Definition

Let D be the domain of an L-structure S, and suppose $f : D^n \to D$ is a function.

We say f is **simply definable** (in S) iff there is some L-term of n variables whose extension is f.

(The word "simply" is there to signal that this is just our preliminary definition of "definable". We'll give another definition later on, in Section 6.1, when we have introduced more expressive languages.)

3.5.22 Exercise
Show that the doubling function is simply definable in the standard model of arithmetic.

3.5.23 Exercise
Let a be a symbol in the standard alphabet. Show that the function that takes each string s to $(a : s)$ (the result of adding the single symbol a to the beginning of s) is simply definable in the string structure $\mathbb{S}$.

3.6 Review

Key Techniques

- We can use **structures** to represent the relationship between simple languages and the world.
- You can prove that every term has a certain property by **induction on terms.** (Technique 3.5.2)
- You can **recursively define** a function whose domain is the set of all terms. (Technique 3.5.5)

Key Concepts and Facts

- A **structure** consists of a set of objects called its **domain**, together with **extensions** for certain constants and function symbols. (Definition 3.1.4)
- **Terms** are built up inductively out of variables, constants, and function symbols. (Definition 3.5.1)
- You can plug a term into a another term using **substitution.** (Definition 3.5.7)

- In any structure, each term **denotes** an object, with respect to an **assignment** of values for its variables. (Definition 3.5.18)

- We can use terms to represent functions. The **extension** of a term $t(x)$ in a structure is the function that takes each object d to the denotation of $t(x)$ at d (that is, for the assignment $[x \mapsto d]$). (Definition 3.5.19)

- These ideas (substitution, denotation, and extension) are closely related: for any term $t(x)$ and any term a,

  $$[\![t]\!][x \mapsto [\![a]\!]] = [\![t[x \mapsto a]]\!]$$

  (in any structure). (Exercise 3.5.20)

Chapter 4

The Uncountable

> Unity added to infinity adds nothing to it, any more than does one foot added to infinite length. The finite is annihilated in presence of the infinite, and becomes pure nothingness. So does our mind before God …
>
> Blaise Pascal, *Pensees* (1670)

People have been mystified by the infinite every since they first noticed it in antiquity—for example, that there are infinitely many numbers, and (perhaps) infinitely many different positions in space or time. In the quotation above, Pascal is playing up the mystery (for theological purposes) but he is pointing to something genuinely weird. Consider an infinite ray, extending from a point A in one direction, forever. Next consider another point B which is one foot from A along this ray.

Suppose we just delete the part of the ray between A and B. What are we left with? It's another infinite ray starting from B—a ray which looks exactly like the ray we began with. In particular, it is the *same size* as the ray we began with—infinite.[1]

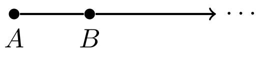

Figure 4.1: An infinite ray

[1] This puzzle is inspired by one of Zeno's paradoxes—though it is hard to tell from the surviving fragments what exactly Zeno's original puzzle was. In his commentary, Simplicius quotes (or para-

But if the ray from A and the ray from B are the same, then the difference between them—the foot from A to B—seems to make no difference at all. A finite distance seems to become "pure nothingness"!

But while we may be mystified, this doesn't mean we can make no further progress. This puzzle is not a stopping place, but a starting place. The puzzle points us to an essential feature of infinity. If something is infinitely large, then it is *the same size as one of its parts*. (Since that part is also infinite, it is then also the same size as one of *its* parts—we can delete the next mile after B, and the next mile after that, and so on, and at every step we are left with another infinite ray just like the first. So in fact, something infinite is the same size as *infinitely many* different parts.)

We can make the idea of the "size" of a set more precise using one-to-one correspondences (which we introduced in Section 1.2). If we can pair off the elements of two sets, so that neither of them has any extra elements left over, then this gives us a precise sense in which the two sets are the same "size", or have the *same number of elements*. Then this is the basic observation: an infinite set can be put in *one-to-one correspondence* with one of its proper subsets.

Once we see this, though, this raises many questions. What sets can be put in one-to-one correspondence? As we'll see in Section 4.5, it's lots more than you might initially think. For example, you might think it's obvious that there are *more* numbers than there are even numbers, and that there are more *pairs* of numbers than there are numbers. But these thoughts are wrong.

You might also be tempted by the opposite thought, that all infinite sets are the *same* size. But as it turns out, just as finite sets come in many different sizes, there are also infinite sets which have different sizes. In fact, there are infinitely many different sizes of infinite sets. This chain of ever-vaster infinities is very beautiful, but it also turns out to be a surprisingly practical tool. Just as it's helpful to use individual numbers as a measuring stick against finite sets—we call this *counting*, and we've done it since prehistory—the set of *all* natural numbers is a useful measuring stick for infinite sets. The set of natural numbers is the smallest kind of infinity.

These beautiful facts follow pretty straightforwardly from Cantor's Theorem (Exercise 1.5.3), which was proved using the "self-application" (or "diagonalization") trick.

phrases) Zeno: "But if when it is subtracted, the other thing is no smaller, nor is it increased when it is added, clearly the thing being added or subtracted is nothing." (sec. 139.9) `TODO CITE`

4.1 Counting

Galileo wrote, "It is wrong to speak of infinite quantities as being the one greater or less than or equal to the other."[2] The history of philosophy, science, and mathematics is full of warnings like this—cautions which, as Georg Cantor wrote in 1883[3], are

> intended to contain the flight of mathematical speculation and conceptualization within its true limits, where it runs no danger of falling into the abyss of the "transcendent," the place where supposedly, as is said in order to inspire fear and wholesome terror, "all is possible."

But in fact, infinity is not a conceptual "abyss" about which it is impossible to even think clearly. One key discovery (made precise by Cantor) is that there really *is* a reasonable way of making size comparisons between infinite sets. We really can speak of infinite quantities as *greater* or *less* or *equal* in a way that makes perfect sense. There are some ways in which our intuitions (guided by our experience with finite sets) might lead us astray—so we so have to be careful. But "wholesome terror" would be overreacting!

Consider this function (from Example 1.2.3)

$$[\quad 1 \mapsto \text{Los Angeles}, \quad 2 \mapsto \text{San Diego}, \quad 3 \mapsto \text{San Jose} \quad]$$

This function is a one-to-one correspondence between the sets

$$A = \{1, 2, 3\}$$
$$B = \{\text{Los Angeles}, \text{San Diego}, \text{San Jose}\}$$

It pairs off each element of A with exactly one element of B, with no elements left over. This is possible because A and B are both three-element sets. They are two different sets with the same number of elements—they are the same *size*.

We can generalize this idea. In general, if A and B are any sets, then if there is a *one-to-one correspondence* between A and B, this pairs off each element of A with exactly one element of B, with no elements left over. We can line up the elements of A with the elements of B, and neither set sticks out past the other. This gives us a reasonable sense in which A and B have *the same size*, or *the same number of elements*. In fact, we will use this as a definition.

[2] 1638, *Two New Sciences* CITE

[3] CITE "Fundamentals of a General Theory of Manifolds"

4.1.1 Definition

If A and B are sets, A and B have the **same number of elements** iff there is a one-to-one correspondence between A and B. This is abbreviated $A \sim B$.

4.1.2 Example

(a) Exercise 1.3.3 showed that there is a one-to-one correspondence between $A \times B$ and $B \times A$. That is,

$$A \times B \sim B \times A$$

(b) Exercise 1.3.5 showed tht there is a one-to-one correspondence between $(A \times B) \times C$, and $A \times (B \times C)$. That is,

$$(A \times B) \times C \sim A \times (B \times C)$$

(c) Exercise 1.3.4 showed that there is a one-to-one correspondence between any set A and the diagonal of $A \times A$. That is,

$$A \sim \{(a_1, a_2) \in A \times A \mid a_1 = a_2\}$$

(d) Exercise 1.4.7 showed that there is a one-to-one correspondence between the subsets of a set A and the functions from A to a two-element set. That is,

$$PA \sim \mathbb{2}^A$$

(e) Example 1.4.8 showed that for any set A,

$$A^{\mathbb{2}} \sim A \times A$$

(f) Exercise 1.4.9 showed that for any set A,

$$A^{\mathbb{1}} \sim A$$

(g) Exercise 1.4.10 showed that for any sets A, B, and C,

$$C^{A \times B} \sim (C^A)^B$$

Notice that our definition of “same number” doesn’t actually say anything about *numbers*. You can think of *has-the-same-number-of-elements-as* if it was just one word, which describes a certain relationship between sets. In particular, this notion of “same number of elements” even makes sense when we apply it to *infinite* sets, which don’t have any finite number of elements. For example, the facts in the previous example are not restricted just to finite sets. For example, the infinite set of functions $\mathbb{N}^{\mathbb{2}}$ has-the-same-number-of-elements-as the infinite set of pairs of numbers $\mathbb{N} \times \mathbb{N}$.

But we have to be careful: there are some ways in which this can be counterintuitive.

4.1.3 Exercise (Hilbert's hotel)
There is a one-to-one correspondence between $\mathbb{N}$ (the set of all numbers) and its proper subset $\mathbb{N} - \{0\}$. That is, $\mathbb{N}$ has the same number of elements as $\mathbb{N} - \{0\}$.

Imagine that there is an infinite hotel, with one room for each natural number. Every room has one person in it. What Exercise 4.1.3 shows is that it is still possible to accommodate another person, without kicking anybody out or doubling up. We can use the one-to-one correspondence in Exercise 4.1.3 to rearrange people within the hotel, giving each person a new room, in a way that leaves room 0 unoccupied for a new guest. An infinite hotel can be full, but still have room for more people.

In fact, this is the defining feature of infinite sets.

4.1.4 Definition
A set A is **infinite** iff there is a one-to-one correspondence from A to some proper subset of A. That is, A is infinite iff there is some proper subset $B \subsetneq A$ such that $A \sim B$.

So Exercise 4.1.3 shows that, in this official sense, there are infinitely many numbers.

Here is another way of putting this definition.

4.1.5 Proposition
A is infinite iff there is a function $f : A \to A$ which is one-to-one, but not onto.

Proof
Suppose that A is infinite: that is, for some proper subset $B \subsetneq A$, there is a one-to-one correspondence $f : A \to B$. In that case, f is a one-to-one function, and the range of f is not the same as A, so there is a function from A to A which is one-to-one but not onto.

For the other direction, suppose that $f : A \to A$ is one-to-one and not onto. Then, by Exercise 1.2.14, there is a one-to-one correspondence between A and the range of f, which by assumption is a proper subset of A. So A is infinite. □

4.1.6 Exercise
The set $\mathbb{S}$ of all strings is infinite.

As we already discussed, the set

$$A = \{1, 2, 3\}$$

has the same number of elements as the set

$$B = \{\text{Los Angeles}, \text{San Jose}, \text{San Diego}\}$$

Meanwhile, the set

$$C = \{1, 2\}$$

has *fewer* elements than B. In general, we can also describe relationships like this using functions. If we do it this way, then again we can make size comparisons that even make sense for infinite sets.

4.1.7 Definition

For any sets A and B, A **has no more elements than** B (abbreviated $A \lesssim B$) iff A has the same number of elements as some subset of B.

In Exercise 1.2.26 we showed that the following are equivalent:

- There is a one-to-one correspondence between A and some subset of B.
- There is a one-to-one function from A to B.
- There is an onto partial function from B to A.

So these are all different ways of saying that A has no more elements than B. Since these different properties are all equivalent, in practice you might as well just forget which of them was the official *definition* of "$A \lesssim B$". You can think of the definition of "$A \lesssim B$" as being whichever one of these properties happens to be most useful for the problem you are solving.

While we're at it, let's give a fourth equivalent way of putting this.

4.1.8 Proposition

$A \lesssim B$ iff either A is empty or there is an onto function from B to A.

Proof

If $A \lesssim B$, then there is an onto partial function f from B to A. Suppose that A is not empty. That is, A has at least one element, so let a_0 be an element of A. Then we can define a *total* function f^+ from B to A that *extends* f. We can define this function f^+ piecewise.

$$f^+ b = \begin{cases} f\,b & \text{if } b \text{ is in the domain of } f \\ a_0 & \text{otherwise} \end{cases}$$

Finally, let's check that f^+ is onto. For any $a \in A$, because f is onto, there is some $b \in B$ such that $f\,b = a$. That means $f^+ b = a$ as well. □

4.1.9 Example

(a) Exercise 1.3.4 showed that, for any set A, there is a one-to-one correspondence between A and the diagonal subset of $A \times A$. That is to say,

$$A \lesssim A \times A$$

(b) Exercise 1.4.3 showed that there is a one-to-one function from any set to its power set. That is to say, for any set A,

$$A \lesssim PA$$

4.1.10 Exercise

Let A and B be sets. If B is not empty, then

$$A \lesssim A \times B$$

We should check that the "no more elements" relation we have defined has some properties we would reasonably expect.

4.1.11 Exercise

Let A, B, and C be sets.

(a) If $A \sim B$, then $A \lesssim B$ and $B \lesssim A$.
(b) $A \lesssim A$.
(c) If $A \lesssim B$ and $B \lesssim C$, then $A \lesssim C$.
(d) If $A \sim B$ and $A \lesssim C$, then $B \lesssim C$.

Hint. Use facts from Section 1.2 for some parts of this exercise.

4.1.12 Exercise

(a) For any function f, the range of f has no more elements than the domain of f.

$$\text{range}\, f \lesssim \text{domain}\, f$$

(b) For any sets A and B, $A \lesssim B$ iff there is some function f such that

$$A \subseteq \text{range}\, f \quad \text{and} \quad \text{domain}\, f \subseteq B$$

So far, we have looked at the concept "same size", and the concepts "at least as big" and "at least as small". We'll also need one more important size concept—the concept of a *bigger* set.

4.1.13 Definition

For any sets A and B, A **has (strictly) fewer elements than** B (abbreviated $A < B$) iff A has no more elements than B ($A \lesssim B$), but A does not have at least as many elements as B ($A \not\gtrsim B$).

We have to be careful about this definition. Here are two other things which you might think were right in general, based on your experience with *finite* sets. You might be tempted to think that $\mathbb{N}$, the numbers starting from zero, has strictly more elements than $\mathbb{N} - \{0\}$, the numbers starting from one. But that's not right—in the sense of "number" we are talking about right now, these two sets have the *same* number of elements. As "Hilbert's hotel" shows, there is a one-to-one correspondence between them. More generally, you might be tempted by this thought:

- **Wrong.** If there is a one-to-one correspondence between A and a proper subset of B, then $A < B$.
- **Wrong.** If there is a function from A to B which is one-to-one but *not* onto, then $A < B$.

Both of these claims are true if A and B happen to be finite sets, but they aren't true in general. Your intuitions probably need some retraining when it comes to counting the elements of infinite sets.

If you can easily come up with a function from A to B that is one-to-one, but not onto, then it's very tempting to conclude that $A < B$. But infinite sets don't work that way! Finding a way of mapping A into B with some stuff left over does tell you that $A \lesssim B$—that B has *at least* as many elements as A—but it could still turn out that $B \lesssim A$ as well, in which case they would be the same size after all. To show that $A < B$, it isn't enough to find *some* one-to-one function from A to B which *isn't* a one-to-one correspondence. You have to prove that *no* function from A to B *is* a one-to-one correspondence. We'll talk about how to do this in Section 4.7.

Again, we should check that this notion of "bigger" has some properties that we would reasonably expect.

4.1.14 Exercise
Let A, B, and C be sets.

(a) If $A \lesssim B$ and $B < C$, then $A < C$.
(b) If $A < B$ and $B \lesssim C$, then $A < C$.

Before we move on, here are a couple more facts about comparing sizes of sets. These ones are quite a bit trickier to prove than the others we have done, but they are useful to know, and occasionally we'll want to rely on them.

4.1.15 Theorem (Schröder–Bernstein Theorem)
For any sets A and B, if A has no more elements than B and B has no more elements than A, then A and B have the same number of elements. That is, if $A \lesssim B$ and $B \lesssim A$, then $A \sim B$.

4.1.16 Theorem (Cardinal Comparability)
For any sets A and B, either A has no more elements than B, or B has no more elements than A (or perhaps both). That is, $A \lesssim B$ or $B \lesssim A$.

Together, these facts tell us that for any sets A and B, there are just three possibilities: A has strictly fewer elements than A, A has strictly more elements than B, or else A and B have the *same* number of elements. We always have exactly one of the following:

$$A < B$$

$$A \sim B$$

$$A > B$$

This is sometimes called the **law of trichotomy**. This seems very intuitive based on our experience with sizes, but showing that the Cardinal Comparability Theorem and the Schröder-Berstein Theorem are true, just using our basic assumptions, is not easy. (Furthermore, the proof of the Cardinal Comparability Theorem relies on the controversial part of the Axiom of Choice.) Proofs are included in Section 4.8, if you are interested.

4.2 Sequences

A finite string lists symbols one by one, in order. It has a first symbol, a second symbol, and so on. We can also list other things. For example, we can list numbers in a finite sequence, like this:

$$1, 2, 3, 4, 5$$

or this:

$$1, 2, 3, 2, 1$$

There are also sequences of cities, like this:

$$\text{Los Angeles}, \text{San Jose}, \text{Los Angeles}$$

Or any other kind of thing. Like sets, sequences have *elements,* but unlike sets, the elements of a sequence come in a particular order, and they can repeat.

A finite sequence has a zeroth element (we start counting from zero), a first element, and so on up to some finite number. We can capture all of the important information about a finite sequence by specifying the element it has in each position. For example, we can describe the sequence of cities above with the function

$$[0 \mapsto \text{Los Angeles}, 1 \mapsto \text{San Jose}, 2 \mapsto \text{Los Angeles}]$$

We'll call a function like this an **enumeration.**

4.2.1 Definition
A **finite enumeration** is a function whose domain is the set of numbers $\{k \in \mathbb{N} \mid k < n\}$, for some number n.

If we want to be ontologically parsimonious, we might suppose that a finite sequence *just is* its enumeration. But as with the other ontological questions that have come up (what are functions? what are pairs?) it is a hard philosophical question whether this is the right thing to say. But in any case, a sequence can be perfectly well *represented* by its enumeration.

Like numbers and strings, finite sequences also form an inductive structure, with an Inductive Property and an Injective Property. We won't go into all the foundational details again this time.

Besides *finite* sequences like these, we can also consider *infinite* sequences of things. For example, consider this sequence of strings

$$\texttt{A}, \texttt{AA}, \texttt{AAA}, \texttt{AAAA}, \cdots$$

We can also describe infinite sequences using enumerations.

4.2.2 Definition
An **infinite enumeration** is a function whose domain is the set of all numbers $\mathbb{N}$.

4.3 The Infinite*

We have encountered some examples of finite sets (such as $\{\text{Silver Lake}, \text{Echo Park}\}$) and some examples of infinite sets (such as the set of natural numbers). In this section we'll look more systematically at the distinction between these two kinds of sets. What is the essential difference between finiteness and infinity? In this section we'll examine three different answers to this question. We'll then show that all three answers are equivalent.

In Section 4.1, we defined an *infinite* set as one that has the "Hilbert's Hotel" property: an infinite set is the same size as one of its proper subsets.

Another way of understanding finite sets, which might be more intuitive, is in terms of counting. A finite set is one that has some finite number of elements. We know that $\{\text{Silver Lake}, \text{Echo Park}\}$ is a finite set because, in particular, it has two elements. We know *this* by *counting:* when we count, we point to Silver Lake and say "one," and then point to Echo Park and say "two." When we do this, we are demonstrating that there is a one-to-one correspondence between $\{\text{Silver Lake}, \text{Echo Park}\}$ and the set of *numbers* $\{1, 2\}$. Since our counting numbers start from zero, though, for official purposes it's a little more convenient to do thigns in terms of a correspondence with the set $\{0, 1\}$.

4.3.1 Definition
A set A is **finitely enumerable** iff there is some finite enumeration of A, without repetitions: that is, there is some number n such that

$$A \sim \{k \in \mathbb{N} \mid k < n\}$$

In this case we say that A **has** n **elements.**

(Something we'll need to check, to make sure that everything is working right, is that this can never happen with two *different* finite numbers. Otherwise we might have a set that *both* has 12 elements and *also* has 15 elements, for example. This can't happen—we'll show it a bit later.)

A third perspective on infinity uses the set of *all* numbers as a yardstick, instead of just some of them. A different way of saying a set is infinite is to say it has at least as many elements as there are numbers—that is, iff there is some one-to-one function from the natural numbers to the set in question. We can think of a function from the numbers as representing an *infinite sequence*—we pick a zeroth element, a first, a second, and so on for each number. So another way of saying this is that a set A is infinite when there is some infinite sequence of elements of A that never repeats. (Note, though, it is not required that *every* element of A shows up somewhere in the list. We will come back to this in Section 4.5.)

The main thing we will show in this section is that these three perspectives on infinity are all different ways of looking at the same property.

4.3.2 Theorem
The following are equivalent:

(a) A is infinite: there is a one-to-one correspondence between A and a proper subset of A;

(b) A is not finitely enumerable: there is no number n such that A is in one-to-one correspondence with the numbers less than n;

(c) $\mathbb{N} \lesssim A$: there is a one-to-one function from $\mathbb{N}$ to A.

We will spend the rest of this section proving this equivalence.

4.3.3 Exercise
If $A \sim B$ and A is infinite, then B is infinite.

4.3.4 Exercise
If A is infinite and $A \subseteq B$, then B is infinite.

4.3.5 Exercise
If $\mathbb{N} \lesssim A$ then A is infinite.

4.3.6 Exercise
For each number n, the set of numbers

$$\{k \in \mathbb{N} \mid k < n\}$$

is finite.

Hint. Use induction. Use the fact that $k < n + 1$ iff either $k < n$ or $k = n$. So if there were a "Hilbert's hotel" function f for the numbers less than $n + 1$, this could also be used to build a "Hilbert's hotel" function g for the numbers less than n. The key trick is to say what $g(k)$ should be in the case where $f(k) = n{+}1$.

4.3.7 Exercise
No infinite set is finitely enumerable.

4.3.8 Proposition
If A is not finitely enumerable, then there is a one-to-one function from $\mathbb{N}$ to A.

Proof
Intuitively, we want to show that there is an infinite sequence of elements of A without any repetitions ($\mathbb{N} \lesssim A$). The idea is that we can first choose any element of A, and then choose any other element of A, and then choose any third element of A which we haven't already chosen, and so on. At each step, we'll only have used up a finitely enumerable set of elements of A, which means there must be more remaining.

Suppose A is not finitely enumerable. That means that any finitely enumerable subset of A must be a *proper* subset of A. Another slightly tricky way of putting that is that, for each subset $A_0 \subseteq A$, there is some $a \in A$ such that, *if* A_0 is finitely enumerable, then a is not in A_0.

Then we can recursively define a function $f : \mathbb{N} \to A$ such that, for each number n, we let fn be some element of A which is not in the set $\{fk \mid k < n\}$. We can then show by induction that, for each number n, the restriction of f to numbers $k < n$ is one-to-one. (If this holds, then in particular the set $\{fk \mid k < n\}$ is finitely enumerable.) And from this it follows that f itself is one-to-one. □

4.3.9 Exercise
Put these facts together to complete the proof of Theorem 4.3.2.

4.4 Induction and Infinity*

In the last chapter, we considered two fundamental infinite sets: the set of numbers, and the set of strings. Each of those had an axiom that went with it: the Axiom of Numbers and the Axiom of Strings. In this chapter we have considered a more general perspective on infinity, based on Hilbert's hotel. This suggests an alternative, more abstract axiom we might have used.

4.4.1 Axiom of Infinity
There is an infinite set.

It's an important foundational fact that we don't really need to assume all three of these as axioms: in fact, any one of them is strong enough to prove the other two as consequences.

(One little detail here is that in order to derive the Axiom of Strings, we will need the basic assumption that the symbols of the standard alphabet can be collected into a set.)

4.4.2 Exercise
Explain why the Axiom of Numbers implies the Axiom of Infinity, and why the Axiom of Strings implies the Axiom of Infinity.

4.4.3 Theorem
The Axiom of Numbers, the Axiom of Strings, and the Axiom of Infinity are equivalent (assuming there is a set containing the symbols of the standard alphabet).

Proof
Given Exercise 4.4.2, it's enough to show that the Axiom of Infinity implies the Axiom of Numbers, and that the Axiom of Numbers implies the Axiom of Strings.

Suppose that the Axiom of Infinity is true: there is a set A which is Dedekind-infinite, which means that there is a function $f : A \to A$ which is one-to-one but not onto. We want to show that the Axiom of Numbers is true, which means that there is a set N that has an element we can call "zero" and a function we can call "successor", such that together these obey the Injective Property and the Inductive Property. Since $f : A \to A$ is not onto, there is some element of A which is not in the range of f. Call this z. Then we'll define N in such a way that it is guaranteed to have the Inductive Property, with respect to the function f. Let's call a subset $X \subseteq A$ f**-hereditary** iff for any $a \in X$, we also have $f a \in X$. Then we can let N

be the following set:

$$N = \{a \in A \mid \text{for every } f\text{-hereditary set } X, \text{ if } z \in X, \text{ then } a \in X\}$$

It's clear from the definition that $z \in N$, since obviously z is in every f-hereditary set that contains z. It also follows from the way we picked N that, if X is f-hereditary and $z \in X$, then every element of N is in X. And this is exactly what the Inductive Property requires, if N is the set we call "the natural numbers", z is the element we call "zero", and f is the function we call "successor". The last thing we need to check is that N, z, and f also has the Injective Property. This is clear: f is a one-to-one function, and we picked z so it wouldn't be in the range of f, which means that our "zero" is not a "successor". Thus, if there is an infinite set, there is a suitable set that has the right properties for the natural numbers.

There is a philosophical question worth asking: is this set N *really* the natural numbers, and is z really zero, and f really the successor function? If there is an infinite set, then in fact there are many *different* choices of z and f which would work for the argument above—and surely not every choice of z is really the number zero, since the number zero is just one thing. But the Axiom of Numbers was a claim about the *existence* of a set $\mathbb{N}$, an element 0, and a function suc with the right properties—and we have now proved that this existence claim follows from existence of any infinite set at all. We don't need to answer the philosophical question in order to use this existence claim to prove other interesting facts that just depend on the *existence* of numbers with the right structure. In what follows, we can regard our use of number-words as arbitrarily picking out the elements of some structure with the right properties—and we don't care exactly which things they happen to be. But in general this is a deep issue.

The second part is to show that the Axiom of Numbers implies the Axiom of Strings. We can do this by finding a way to "encode" strings using numbers and things based on numbers. One way to do this uses the codings from Section 4.6, with some arithmetic. But here is a different way. We can completely describe the string

```
ABCBA
```

by saying "Symbol 0 is `A`, symbol 1 is `B`, symbol 2 is `C`, symbol 3 is `B`, and symbol 4 is `A`". (We're counting from zero for convenience.) So the string is completely described by specifying a certain *function* from the first five numbers $\{0, 1, 2, 3, 4\}$ to symbols, which says which symbol appears at each position in the sequence. In other words, we can represent the sequence with this function:

$$[0 \mapsto \texttt{A}, \quad 1 \mapsto \texttt{B}, \quad 2 \mapsto \texttt{C}, \quad 3 \mapsto \texttt{B}, \quad 4 \mapsto \texttt{A}]$$

It's clear that we can do this with any string.

To prove the Axiom of Strings from the Axiom of Numbers, we need to show that there is some set $\mathbb{S}$, an element $() \in \mathbb{S}$, and for each symbol a and $s \in \mathbb{S}$, we have some element $(a : s) \in \mathbb{S}$, where these have the Injective Property and Inductive Property for Strings. So, using the idea we just described, we can let $\mathbb{S}$ be a certain set of *functions*. Let $()$ be the empty function from $\varnothing$ to the alphabet A. For any partial function s from $\mathbb{N}$ to A, and for any symbol $a \in A$, let $(a : s)$ be the function

$$\begin{aligned} 0 &\mapsto a \\ n+1 &\mapsto sn \quad \text{if } n \text{ is in the domain of } s \end{aligned}$$

Finally, we'll use the same trick as we did for the numbers. A **cons-hereditary set** is a set X such that, for any $a \in A$ and $s \in X$, $(a : s)$ is in X. Then let $\mathbb{S}$ be the set of all partial functions s from $\mathbb{N}$ to A such that *every* cons-hereditary set X that contains $()$ also contains s. This guarantees that $\mathbb{S}$ has the Inductive Property for sequences.

The last thing to check is that $\mathbb{S}$ also has the Injective Property. First, it's clear that for any symbol a and function $s \in \mathbb{S}$, the function $(a : s)$ at least has 0 in its domain, while $()$ has an empty domain. So $() \neq (a : s)$. Checking that if $(a : s) = (a' : s')$, then $a = a'$ and $s = s'$ is left as an exercise.

The same philosophical question arises for strings: is this really what a finite sequence *is*—a certain function from numbers to symbols? That might seem kind of weird. While questions like these about the nature of abstract objects are philosophically important, fortunately we don't have to answer them in order to use the Axiom of Strings for technical purposes—because again, all we will really care about is that there is *some* set $\mathbb{S}$, element $()$, and "cons" operation $(:)$ with the right structural features. It won't really matter what that set's elements really *are*, as far as our proofs go. That doesn't answer the philosophical question of what sequences really are. But we can sidestep that question for most of what we're up to. □

4.5 Countable Sets

Our abstract definitions of "same number of elements" and "no more elements," using functions, are especially useful when it comes to infinite sets. In this case, our intuitions about counting, and about which sets are the "same size" or "different sizes," are not very reliable. We need these precise abstract tools to make progress.

Let's start by focusing on $\mathbb{N}$, the set of numbers. In Section 4.1 we discussed one counterintuitive feature of this set: it has the same number of elements as one of its proper subsets. If we have infinitely many people in hotel rooms numbered $0, 1, 2, 3, \ldots$, then we can shift each of them down to the next room. This puts them in the rooms $1, 2, 3, 4, \ldots$, leaving room 0 unoccupied.

We can go much further than this. Suppose Hilbert's hotel is full, and then *infinitely* many guests arrive. Can we accommodate them? That is, can we rearrange the guests in rooms $0, 1, 2, 3, \ldots$ in such a way as to leave *infinitely* many rooms unoccupied? Yes, we can.

4.5.1 Exercise

The set of even numbers $\{0, 2, 4, \ldots\}$ has the same number of elements as $\mathbb{N}$. That is, if E is the set of even numbers, then $E \sim \mathbb{N}$.

Hint. What it means to be *even* is to be the result of doubling some number.

This shows that we can move the guests in Hilbert's hotel into just the rooms $0, 2, 4, 6, \ldots$, without kicking anyone out, or doubling people up in the same room, leaving all of the odd-numbered rooms unoccupied. This is counterintuitive. The set of even numbers E leaves out infinitely many numbers. It's very intuitive to think think that this means there aren't as many even numbers as natural numbers: that is, that $E < \mathbb{N}$. This intuitive thought is wrong!

Very similarly, one can show that the set of all odd numbers is in one-to-one correspondence with $\mathbb{N}$. So in fact, the set of all numbers $\mathbb{N}$ is a union of two sets, each of which has the same number of elements as $\mathbb{N}$. We can take two "copies" of $\mathbb{N}$ (the even numbers and the odd numbers) and stick them together, and we end up with a set which is the same size as we started with.

This raises an interesting question: of the many different infinite sets out there, which of them are the same size as the set of numbers? Are there just as many strings as numbers, or is one of these infinite sets bigger than the other? What about the set of all *sets* of numbers? What about sets of strings?

A *countably infinite* set is a set that has the same number of elements as there are numbers. We can build up this idea from two parts—*infinite*, and *countable.* If a set is infinite, this means it has *at least* as many elements as there are numbers. If it is *countable*, this means that it does not have *more* elements than there are numbers.

4.5.2 Definition

A set A is **countable** iff $A \lesssim \mathbb{N}$. Remember that these are all equivalent ways of saying this (see **??**):

1. There is a one-to-one correspondence between A and some set of numbers.

2. There is a one-to-one function from A to $\mathbb{N}$.
3. There is an onto partial function from $\mathbb{N}$ to A.
4. There is an onto function from $\mathbb{N}$ to A, or else A is empty.

(Countable sets are also called *enumerable* or *denumerable*.)

When we count things, we point at each one of them and say a number. If the set is finite, then we can count all of them using just finitely many numbers. That is, if A is a finite set, then there is a one-to-one correspondence between A and the numbers 0, 1, 2, …, n, for some number n. If A is an infinite set, then this process can never stop. But suppose we don't have to stop—we just keep counting forever. Then we might succeed in "counting" A in the sense of assigning every element of A a unique number. We might put A in one-to-one correspondence with *all* the numbers. A countable set is one that can be counted, in this sense, using either some or all of the counting numbers. Many important sets can be "infinitely counted" this way—but as we will see, not all of them!

Another useful way to think about a countable set is as one that can be *listed* in a finite or infinite sequence. Putting things into an ordered sequence basically amounts to the same thing as assigning each of them a number—its position in the sequence. This works for both finite and infinite sequences. For example, consider this infinite sequence:

$$\texttt{A}, \texttt{B}, \texttt{C}, \texttt{A}, \texttt{B}, \texttt{C}, \dots$$

To represent this sequence, we just need to specify which letter appears at each place in the sequence: at position 0 we have `A`, at 1, `B`, at 2, `C`, and so on. So we can represent this sequence with the function

$$[0 \mapsto \texttt{A}, \quad 1 \mapsto \texttt{B}, \quad 2 \mapsto \texttt{C}, \quad \dots \quad]$$

4.5.3 Definition

For any set A, an **infinite sequence** of elements of A is a function from $\mathbb{N}$ to A. So $A^{\mathbb{N}}$ is the set of all infinite sequences in A.

(Really, there can also be infinite sequences that are even *longer* than this sort of sequence. A more precise name for this particular kind of infinite sequence is an **omega-sequence**, or ω-sequence.)

So another way of putting Definition 4.5.2 is that a countable set is one whose elements can be listed in an infinite sequence (or else the empty set).

Let's check some basic things.

4.5.4 Proposition

Every finite set is countable.

Proof

Theorem 4.3.2 showed that every finite set is in one-to-one correspondence with the set of all numbers less than n, for some number n. Any set which is one-to-one correspondence with a set of numbers is countable. □

4.5.5 Exercise

Any subset of a countable set is countable.

4.5.6 Definition

A set A is **countably infinite** iff A is both countable and infinite.

4.5.7 Theorem

A is countably infinite iff $A \sim \mathbb{N}$.

Proof

A is countable iff $A \lesssim \mathbb{N}$. A is infinite iff $\mathbb{N} \lesssim A$. So A is countably infinite iff $A \lesssim \mathbb{N}$ and $\mathbb{N} \lesssim A$. By the Schröder-Bernstein Theorem (Theorem 4.1.15) this holds iff $A \sim \mathbb{N}$.

We can also prove this fact directly, without appealing to the Schröder-Bernstein Theorem. If A is the same size as $\mathbb{N}$, then A is clearly countable ($A \lesssim \mathbb{N}$) and infinite ($\mathbb{N} \lesssim A$). For the other direction, suppose that A is countably infinite. This means that A is the same size as some infinite subset $X \subseteq \mathbb{N}$. So it is enough to show that X has the same number of elements as $\mathbb{N}$—that is, there is a one-to-one correspondence between $\mathbb{N}$ and X. Here is the intuitive idea. If X is an infinite set of numbers, then X has a least element, and a second-least element, and a third-least element, and so on. This defines a one-to-one correspondence between X and the numbers: in particular, for each $x \in X$, we can let $f(x)$ be the number of elements of X which are less than x. (We are using the fact that for any number x, there are only finitely many numbers less than x.) We can show that this function is a one-to-one correspondence.

(This function f is *one-to-one,* because if x and y are elements of X such that $x < y$, then every element of X which is less than x is less than y, and x is an element of X which is less than y but not less than y. So $f(y)$ is at least $f(x)+1$. We can show that f is *onto* by induction. By the Least Number Property, there is some $x \in X$

such that $f(x) = 0$. And for any number n, if $f(x) = n$, then $\{y \in X \mid y > x\}$ is a non-empty set of numbers, which also has a least element y. So $f(y) = n + 1$.) □

Let's look at another important example. How does the set of *strings* compare in size to the set of numbers? Is it countable? We could try to list the strings one by one like this:

```
A
AA
AAA
AAAA
⋮
```

But this is not encouraging—putting the strings in this order, we'll never reach any string that includes the letter `B`. But remember, just because we can find some infinite list that *doesn't* include every string, that doesn't mean there isn't some other way of listing strings that *does* include them all. In fact, we can use this trick. First list all the length-zero strings (the empty string). Then list all the length-one strings, in alphabetical order. Then list all the length-two strings, in alphabetical order. And so on. For any number n, there are only finitely many length-n strings. So eventually, going on this way, we will reach every string.

If we imagine that the alphabet just includes the symbols `A`, `B`, and `C`, this sequence of strings will look like this:

(),
`A`, `B`, `C`,
`AA`, `AB`, `AC`, `BA`, `BB`, `BC`, `CA`, `CB`, `CC`,
`AAA`, `AAB`, `AAC`, `ABA`, `ABB`, `ABC`, …

This should be enough to get across the main idea, showing that it is possible to list all of the strings in one infinite list. For most purposes, we don't need to bother with making this proof completely official.

4.5.8 Proposition
The set of strings $\mathbb{S}$ is countably infinite. That is $\mathbb{S} \sim \mathbb{N}$.

To make the proof of this fact more precise, we can explicitly define the functions this infinite sequence corresponds to—a function from numbers to strings, and another function from strings to numbers which is its inverse.

$$\begin{array}{llllll} 0 \mapsto (), & 1 \mapsto \texttt{A}, & 2 \mapsto \texttt{B}, & 3 \mapsto \texttt{C}, & 4 \mapsto \texttt{AA}, & \dots \\ () \mapsto 0, & \texttt{A} \mapsto 1, & \texttt{B} \mapsto 2, & \texttt{C} \mapsto 3, & \texttt{AA} \mapsto 4, & \dots \end{array}$$

Writing out this correspondence carefully and checking it is kind of fiddly and involves a bit of basic number theory; the details are given in Section 4.6.

These details aren't very important for our current purposes—but the basic idea is important. Each string can be assigned a unique numerical *code*. There are two functions

$$\begin{aligned} \text{code} &: \mathbb{S} \to \mathbb{N} \\ \text{decode} &: \mathbb{N} \to \mathbb{S} \end{aligned}$$

which are inverses to each other. This is the underlying idea of how computers represent data. When text is stored on a computer, it is represented as a sequence of bits—zeros and ones. Essentially, text is thus represented by a very large number. Represent strings with number is called "arithmetization." This makes it possible to talk about properties of strings obliquely, in the language of numbers, by talking about arithmetical properties of their numerical codes. (This idea will come up again in Section 6.9. Numerical codes for strings are also called *Gödel numbers.*)

Because there is a one-to-one correspondence between numbers and strings, this gives us another useful way of thinking about countable sets.

4.5.9 Exercise
A set A is countable iff there is a one-to-one function from A to $\mathbb{S}$.

This way of thinking about what it means to be countable is particularly important in logic: a countable set is a set whose elements can be *written down* in some system of notation or other. If we can assign each element of A a unique string as a *label,* this shows that A is countable. So the countable sets include all the sets that can be "named" in any language whose expressions can be written down as a finite sequence of symbols. In general, we'll call a one-to-one function from A to $\mathbb{S}$ a **string representation** for A, or a **labeling** for A.

This tells us, in particular, that all of the *languages* we discussed in Chapter 3 are countable. The terms in the language of arithmetic are strings of symbols. So are the terms in the language of strings, and any other language we might discuss in this class.

4.5.10 Proposition
For any signature L, the set of L-terms is countable.

For another example, we already know that there are one-to-one functions from numbers to strings. This immediately follows from the fact that there are infinitely many strings. But another way to get here is from the fact that there is a *labeling* function, which represents each number with a unique string. The system we have used in our official language of arithmetic is the simple "unary" notation (Definition 3.2.10):

$$\begin{aligned}
0 &\mapsto \texttt{0} \\
1 &\mapsto \texttt{suc 0} \\
2 &\mapsto \texttt{suc suc 0} \\
&\vdots \\
n &\mapsto \langle n \rangle
\end{aligned}$$

Exercise 3.2.11 told us that this representation is unique: the function that takes each number to its numeral is one-to-one.

Many things can be written down as strings—even things that might initially appear to be more complicated than strings. Consider *ordered pairs* of strings. This set is also countable. One way to show this would be to come up with a way of arranging all these pairs in an infinite list. But a different approach is to think about how we can *write down* an ordered pair of strings, as a single string.

One tempting way to do this would be to write down the two strings side by side. For example, we could try to represent the pair (`ABC`, `DE`) with the single string `ABCDE` . But this won't quite work: this representation loses los track of where one string ends and the other begins. We'll need to mark the boundary between them. Another thing to try is to put a *delimiting* symbol in between them, like a comma. But this also won't quite work. Consider the string `A„C` . This would be ambiguous between two different pairs of strings: (`A,`, `B`) and (`A`, `,B`).

Here's a nice trick to avoid this problem. Instead of just joining up the strings, we can join up *labels* for the two strings. In Exercise 3.2.13, we showed that for each string, we can choose a *term* in the language of strings that denotes it. For example, for the string `ABC` , we might choose the label

```
"A" ⊕ "B" ⊕ "C" ⊕ ""
```

Arbitrary strings might be annoying to stick together, but these *labels* for the strings are guaranteed to be nice expressions in a well-behaved language. Now we can represent a pair of strings by sticking their *labels* together. For neatness, we'll put a *newline* symbol in between them—this symbol is guaranteed not to appear in any term in the language of strings, so we can’t get any ambiguities this way.[4] For example, we can represent the ordered pair of strings (`AB`, `C`) with the two-line string

```
”A” ⊕ ”B” ⊕ ””
”C” ⊕ ””
```

And this even works for strings that have newlines in them. For example, consider the ordered pair whose first element is this two-line string

```
AB
C
```

and whose second element is `D`. This pair is unambiguously represented by

```
”A” ⊕ ”B” ⊕ new ⊕ ”C” ⊕ ””
”D” ⊕ ””
```

This is still just a two-line string, rather than blowing up into an ambiguous three-line string. Then to recover the original strings, we just have to split up the lines, and figure out which string each line represents.

(To be clear, there is nothing magical about the newline symbol, and this trick does not essentially rely on “multiline strings.” The newline symbol just happens to be a convenient symbol that we weren’t already using for other purposes, which makes for pretty formatting.)

4.5.11 Proposition

The set of ordered pairs of strings $\mathbb{S} \times \mathbb{S}$ is countable.

Proof

Let $\langle \cdot \rangle$ be a label function as in Exercise 3.2.13, which assigns each string s a term that denotes s in the standard string structure. We can define a string representation function

[4] Remember that in our official notation, recall that this label for `ABC` is really

```
⊕(”A”,⊕(”B”,⊕(”C”,””)))
```

This has commas in it. If we also used commas for pairs, we wouldn’t actually get any ambiguities, but it would make parsing a little more complicated—particularly when we get a bit fancier in Section 6.2.]

In general, if $\langle s \rangle$ and $\langle t \rangle$ are the labels for the strings s and t, then we can represent the ordered pair of strings (s, t) with

$$r(s,t) = \langle s \rangle \oplus \text{newline} \oplus \langle t \rangle \quad \text{for each pair of strings } (s,t) \in \mathbb{S} \times \mathbb{S}$$

This function r is one-to-one, so $\mathbb{S} \times \mathbb{S}$ is countable. (Checking this involves another little parsing fact: since each string in the range of r contains exactly one newline symbol, there is only one way of breaking it up into two lines. See Section 4.6.) □

Since we can write down an ordered pair of strings as a single string, it follows that we can also write down ordered pairs of anything that can be *represented* by strings. For example, we can write down pairs of numbers. One obvious way to do this would be with strings like `(3, 1)`. But another way to do it is using the fact we have just proved: each number can be written as a string, and we have a general strategy for representing pairs of strings. This trick works for anything that has a string representation.

4.5.12 Exercise
If A and B are each countable sets, then $A \times B$ is countable.

In particular, this tells us that the set of ordered pairs of numbers $\mathbb{N} \times \mathbb{N}$ is countable. It's worth pausing to think about this fact, because it's rather striking. You might try to start listing the ordered pairs like this:

$$(0,0), (0,1), (0,2), \ldots$$

But this will never get you to any of the pairs

$$(1,0), (1,1), (1,2), \ldots$$

or

$$(2,0), (2,1), (2,2), \ldots$$

and so on. Each of these lists of numbers looks just like a copy of the numbers. So $\mathbb{N} \times \mathbb{N}$ looks like infinitely many copies of $\mathbb{N}$ put together. The fact that $\mathbb{N} \times \mathbb{N}$ is countable tells us that we can squish all of these infinitely many copies of $\mathbb{N}$ into just one copy of $\mathbb{N}$. Not only can Hilbert's hotel accommodate some extra guests, and not only can it accommodate infinitely many extra guests, but in fact it can hold *infinitely many Hilbert's hotels* full of guests. This might seem pretty weird. How could you flatten out this two-dimensional grid of numbers in a single one-dimensional list? It can be done. One way, suggested by the proofs we have just

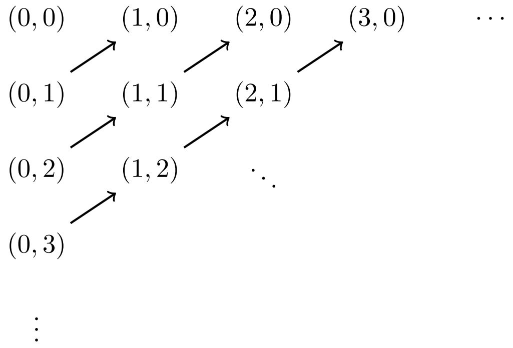

Figure 4.2: A way of listing ordered pairs

given, is to first write out each pair of numbers as a string, and then list those strings in order from shortest to longest, with ties broken alphabetically. An alternative way of doing this is suggested by Fig. 4.2.

We can straightforwardly generalize the idea of Proposition 4.5.11: we can use exactly the same trick to write down any *finite sequence* of strings.

4.5.13 Exercise
Explain why the set of all finite sequences of strings is countable, by describing a string representation function.

4.6 Coding and Parsing Details*

In this section we'll fill in some of the details from the previous section.

First, we'll discuss the details of the correspondence between numbers and strings, in order to show that the set of strings is countable. What we need to do is define a numerical *code* for each string, and then show that we can convert back and forth between strings and their codes. That is, we will define a *coding* function from strings to numbers, and we can prove that this function is one-to-one by also defining a *decoding* function. The key thing to check is that if you decode a code for a string, you get the original string back. (Numerical codes for strings are also sometimes called *Gödel numbers.*)

The details of the coding involve a little bit of basic arithmetic. In particular, we will use the fact that it is possible to *divide* one number by another, with a remainder.

4.6.1 Exercise (Division Lemma)
Let $n > 0$. For any number m, there are unique numbers q and r such that $r < n$ and $m = n{\cdot}q{+}r$. The number q is the *quotient*, and the number r is the *remainder*.

4.6.2 Proposition
The set of strings $\mathbb{S}$ is countably infinite. That is $\mathbb{S} \sim \mathbb{N}$.

Proof
We will start by choosing a standard way of ordering the symbols in the alphabet A—an alphabetical order for individual symbols. This amounts to choosing a number for each symbol, which means we have a function $d : A \to \{0, 1, \ldots, N - 1\}$, where N is the number of symbols in A (which is a finite set).

Next we will recursively define a function $f : \mathbb{S} \to \mathbb{N}$, which says what position in the infinite list each string gets. We can do it like this:[5]

$$f() = 0 f(a \oplus s) \quad = N \cdot f(s) + d(a) + 1 \quad \text{for each string } s \text{ and symbol } a$$

Now we just need to prove that f is one-to-one. We can do this by also defining an *inverse* function for f. This will be a “decoder” function $g : \mathbb{N} \to \mathbb{S}$ that takes each number to the string that it encodes.

We’ll define the function g recursively.[6] For the base case, clearly $g(0)$ should be the empty string. For the recursive part, we’ll need a bit of elementary school arithmetic. The Division Lemma (Exercise 4.6.1) tells us that we can *divide* the number $n - 1$ by the number N (which is not zero), with some remainder. That is, there is a number q (the *quotient*) and a number r (the *remainder*) such that

$$n - 1 = N \cdot q + r$$

and $r < N$. Furthermore, these numbers q and r are unique. The quotient q is guaranteed to be smaller than n. (Otherwise,) That means we can assume we already know how to decode q for our recursive definition. So we can let

$$g(n) = d^{-1}(r) \oplus g(q)$$

[5] A small detail: this definition of f actually corresponds to listing the strings of length n in alphabetical order reading from *right to left*, rather than from left to right as in the list we gave in Section 4.5. This happens to be slightly more convenient, because we are using the convention of building up strings to the left.

[6] The alert reader will notice that this definition uses *strong* recursion on numbers. `TODO. Explain.`

where $d^{-1} : \{0, \dots, n-1\} \to A$ is the inverse function of A: that is, $d^{-1}(r)$ is the rth symbol in the alphabet.

Now we have to prove that $g(f(s)) = s$ for each string s, by induction on strings. For the base case, it is clear from the definitions that $g(f()) = g(0) = ()$. For the inductive step, let s be any string, let a be any symbol, and suppose that $g(f(s)) = s$. Then

$$f(a \oplus s) = N \cdot f(s) + d(a) + 1$$

If we divide $N \cdot f(s) + d(a)$, by N, the (unique) quotient and remainder are $f(s)$ and $d(a)$. So by the definition of g, we have

$$g(f(a \oplus s)) = g(N \cdot f(s) + d(a) + 1) = d^{-1}(d(a)) \oplus g(f(s))$$

Furthermore, $d^{-1}(d(a)) = a$ and by the inductive hypothesis $g(f(s)) = s$. So this simplifies to $a \oplus s$, as we intended.

That much shows that $\mathbb{S}$ is countable. It's also clear that $\mathbb{S}$ is infinite, and thus $\mathbb{S}$ is countably infinite. □

The next thing we'll do is show that the string representation for *pairs of numbers* that we discussed in Section 4.5 is unique. This will also introduce some of the basic machinery we will use for more complicated string representations and parsing facts in Section 3.3 and Section 6.2.

One further detail is to check some facts we took for granted about breaking strings into lines.

4.6.3 Exercise

There is a function line $: \mathbb{S} \to \mathbb{S} \times \mathbb{S}$ such that for each pair of strings s and t such that s does not contain any newlines, and t either is empty or begins with a newline,

$$\mathrm{line}(s \oplus t) = (s, t)$$

Hint. Compare Exercise 3.3.5 for "tokenizing" strings.

4.6.4 Exercise

There is a function lines $: \mathbb{S} \to \mathbb{S}^*$ such that, for any strings $t_1, \dots, t_n$ which do not contain any newline symbols,

$$\mathrm{lines}(t_1 \oplus \mathrm{newline} \oplus \cdots \oplus \mathrm{newline}\, t_n) = (t_1, \dots, t_n)$$

4.7 Uncountable Sets

So far we have considered how it is possible to compare the sizes of infinite sets, using one-to-one correspondences. We have also considered a bunch of *countably infinite* sets, which all have the same number of elements as there are counting numbers. These include the set of numbers $\mathbb{N}$ itself (obviously), any infinite subset of $\mathbb{N}$, as well as the set of strings $\mathbb{S}$, the set of pairs of numbers $\mathbb{N} \times \mathbb{N}$. We can also use similar methods to show many other sets are countable: for example, $\mathbb{S} \cup \mathbb{N}$, $\mathbb{S} \times (\mathbb{N} \times \mathbb{S})$, the set of finite sets of strings, the set of pairs of a number . As we said, any set of things that can be *written down*, in principle, using finite strings of symbols, is a countable set. So you might wonder, does this include *all* of the infinite sets? Are all infinite sets the same size? We know that there aren't any infinite sets that are *smaller* than the set of numbers. But are there infinite sets which are *bigger*?

In fact, we have basically already answered this question. In Section 1.5, we considered Russell's Paradox, which showed that not just any things can be collected into a set. (In particular, there is no set containing all sets.) We also considered how the argument can be modified slightly to prove *Cantor's Theorem* (Exercise 1.5.3):

> For any set A, there is no onto function from A to PA.

Here is another way of saying that.

4.7.1 Exercise (Cantor's Theorem (Version 2))
For any set A, there are strictly more subsets of A than elements of A. That is,

$$A < PA$$

4.7.2 Definition
A set A is **uncountable** iff A is not countable. Equivalently, a set A is uncountable iff A has strictly more elements than there are numbers: $\mathbb{N} < A$.

4.7.3 Exercise

(a) The set of all sets of natural numbers, $P\mathbb{N}$, is uncountable.

(b) If A is an infinite set, then A has uncountably many subsets: that is, PA is uncountable.

4.7.4 Exercise

(a) If A is infinite and B has at least two elements, then the set of all functions from A to B is uncountable.

(b) For any set A, if A has at least two elements, the set of all infinite sequences in A (that is, $A^{\mathbb{N}}$) is uncountable.

Notice that the same argument that tells us that there are more *sets* of numbers than there are numbers ($\mathbb{N} < P\mathbb{N}$) can be applied again. There are more *sets of sets* of numbers than there are sets of numbers: that is, $P\mathbb{N} < P(P\mathbb{N})$. And we can keep on going:

$$\mathbb{N} < P\mathbb{N} < PP\mathbb{N} < PPP\mathbb{N} < \cdots$$

So just starting with *one* infinite set ends up introducing us to *infinitely many* different sizes of infinity.

In fact, this ladder of infinite sets only reaches up to some of the lowest levels of abstract reality. *All* of these infinitely many infinite sets are puny compared to some of the vast collections that mathematical set theory has to offer. In this course we'll explore some "small" infinite sets, but we won't climb very high at all into Plato's heaven.

Another very interesting question is whether there are other sizes of infinity *in between* the rungs in this ladder we get by taking sets of sets of sets … The claim that there are no infinite sets intermediate in size between $\mathbb{N}$ and $P\mathbb{N}$ is called *Cantor's Continuum Hypothesis.* The question of whether it is true is a famous open question. One reason the question is so striking is that as it turns out, we know that it is *impossible* to answer this question using standard mathematical methods. See Chapter 10 for more about this.

4.7.5 Technique (Counting Arguments)

The natural numbers are an infinite yardstick for measuring sets. Whether a set is countable or uncountable provides a good first approximation of what that set is like. One common way we use this is based on a very simple principle: if A is countable, and B is uncountable, it follows that A and B are *not the same set*. In particular, If A is a countable subset of B, and B is uncountable, then it follows that B has elements *besides* those in A. (Indeed, B has *uncountably infinitely many* elements which aren't in A.) So a handy trick for showing that there are B's that aren't A's is to show that A is countable, and B is uncountable.

This a more specific version of the general kind of counting argument we introduced in Section 1.5.

4.7.6 Exercise

Suppose that L is some set of strings, which we'll call *descriptions*. (For example, L could consist of strings that make grammatical English noun-phrases, like `the set of all prime numbers`.) Suppose furthermore that there is a function d that takes each description in L to a set of numbers: for any string s, we'll call $d s$ the set **described** by s. Show that there are infinitely many sets of numbers which are not described by *any* description in L.

4.7.7 Exercise

Let L be a signature, and let S be an L-structure with an infinite domain D. Use a counting argument to show that there is some function $D \to D$ which is *not* simply definable in S.

4.7.8 Exercise

Let I be the set of **real numbers** between 0 and 1. We won't need to worry too much about what real numbers are like, but here is one fact about them:
we can represent a real number using an infinite sequence of digits. Let D be the set of base 10 digits, $D = \{0, 1, \dots, 9\}$. The standard way of representing numbers with sequences of decimal digits isn't quite one-to-one: for example, 0.4999… and 0.5 are both the same number. In order to get a one-to-one representation, we'll need to block this case. So let X be the set of all infinite sequences of digits that eventually end in just 9's. That is, if $s \in D^*$ is a *finite* sequence of digits, let $s \oplus \overline{9}$ be the result of adding an infinite sequence of 9's to the end of s. Then

$$X = \{t \in D^{\mathbb{N}} \mid t = s \oplus \overline{9} \text{ for some } s \in D^*\}$$

In other words, X is the range of the function $[s \mapsto s \oplus \overline{9}]$ from D^* to $D^{\mathbb{N}}$. This is the key fact that you can take for granted about the decimal representation of real numbers: there is a one-to-one correspondence between $D^{\mathbb{N}} - X$ and I.

There is also a **division** function. This function takes each ordered pair of natural numbers $(m, n) \in \mathbb{N} \times \mathbb{N}$ such that $m < n$ to a real number in I (namely, the number m/n). A real number in I is called **rational** iff it is in the range of this division function. Otherwise, it is called **irrational**.

Prove that there are uncountably many irrational numbers.

4.8 More Counting Facts*

```
UNDER CONSTRUCTION
```

Here we'll give proofs of two other important facts about the sizes of sets, which we mentioned at the end of Section 1.5. Both of these facts sound obvious, but they are surprisingly tricky to prove in general. The first fact is that if A has at least as many elements as B, *and* B has at least as many elements as A, then they have the same number of elements. The second fact is that any two sets are *comparable* in size: this means you can always find a one-to-one function in one direction or the other. Together, they amount to what is sometimes called the "law of trichotomy". For any two sets A and B, exactly one of the following three conditions holds: (a) A has more elements than B, (b) B has more elements than A, or (c), A and B have the same number of elements.

4.9 Review

Key Techniques

- One way to show that there is some element of A which is *not* an element of B is to use a **counting argument:** show that A has *strictly more* elements than B ($A > B$). One way to do this is to show that A is *uncountable*, and B is *countable*. (Technique 4.7.5)

- Cantor's Theorem is proved using the **self-application trick** (which is also called **diagonalization**). If there was an onto function $f : PA \to A$, then you could use this to "apply" sets to themselves (by checking whether $f(X) \in X$). But then if you consider the set of all sets that do not "apply" to themselves in this sense, you can derive a contradiction.

Key Concepts and Facts

- A **has the same number of elements as** B ($A \sim B$) iff there is a one-to-one correspondence between A and B.

- A **has no more elements than** B ($A \lesssim B$) iff any of the following equivalent conditions hold:

  (a) A has the same number of elements as some subset of B.
  (b) There is a one-to-one function from A to B.
  (c) Either there is an onto function from B to A, or else A is empty.
  (d) There is an onto partial function from B to A.

- ***B*** **has strictly more elements than** A ($A < B$) iff either of the following equivalent conditions holds.

  (a) $A \lesssim B$ and $B \not\lesssim A$.
  (b) $A \lesssim B$ and $A \nsim B$.

- **Cantor's Theorem.** Every set has strictly more subsets than it has elements. That is, $A < PA$.

- The set $\mathbb{N}$ of all numbers is **infinite.** So is the set $\mathbb{S}$ of all strings.

- Any infinite set has a **"Hilbert's hotel" function:** intuitively, you can rearrange the elements of the set in a way that leaves some extra room left over. More officially, if A is an infinite set, there is a function from A to A which is *one-to-one but not onto.*

- A **countable** set has no more elements than there are natural numbers. Some infinite sets are countable, and some infinite sets are **uncountable.**

- The following sets are countable:

  - $\mathbb{N}$, the set of all numbers
  - $\mathbb{N} \times \mathbb{N}$, the set of all ordered pairs of numbers
  - $\mathbb{N}^*$, the set of all finite sequences of numbers
  - $\mathbb{S}$, the set of all strings

- In general, if A can be divided up into countably many countable sets, then A is countable. In particular, $A \times A$ and A^* are both countable.

- The following sets are uncountable:

  - $P\mathbb{N}$, the set of all sets of numbers
  - $P\mathbb{S}$, the set of all sets of strings
  - $A^{\mathbb{N}}$, the set of all infinite sequences of elements of A, as long as A has at least two elements

- In general, if A is an infinite set, then the following sets are uncountable:

  - PA, the set of all subsets of A
  - B^A, the set of all functions from A to B, as long as B has at least two elements

- In an infinite structure, not every function is the extension of a term. (Exercise 4.7.7)

Chapter 5

First-Order Logic

> The proposition is not a mixture of words (just as the musical theme is not a mixture of tones). The proposition is articulate.
>
> Wittgenstein, *Tractatus Logico-Philosophicus* (1921)

As I've said before, one of the central topics of this course is the relationship between language and the world. In order to understand this relationship, we are working out the details of a simple precise language.

The languages we described in Chapter 3 was very simple. We have terms which we can use to refer to particular objects, or to describe functions in a structure (by using variables). Now we'll build up our language a bit more so that we can *say* things about these objects and functions and how they are related to each other. The expressions we use to say things about the world are called *sentences*.

In particular, we'll be studying sentences in a *first-order* language (with function symbols and identity). What this means is that we have a way of making generalizations about all of the objects in an entire structure, using "for all" statements. ("First-order" contrasts with "higher-order" languages, which can also say things about all *sets* or *properties* of objects in a structure. See Chapter 9.)

Just like with terms, when it comes to sentences there are two main things we need to look at. The first thing is the internal structure of the language: the way its simple pieces can be put together to produce complicated expressions. This is called *syntax*. The second thing is the way the language is related to the world, and in particular the way that sentences can be true or false. This is called *semantics*.

Once we have looked at these two aspects of the first-order language, we can apply them to look at the *logic* of this language, which is about special relationships between different sentences. For example, we can ask whether some sentences are inconsistent with each other, or whether a conclusion follows from some premises.

Nowadays first-order logic is a standard part of the philosopher's toolkit (as well as the mathematician's toolkit, the linguist's toolkit, and the computer scientist's toolkit). You can do a lot with it: it's a pretty powerful tool. But it has its limits. In later chapters, we will examine some things it *can't* do.

5.1 Syntax

I hope the language of first-order logic is already familiar to you (though some of the details in this section will probably be new). Here are some examples of sentences in first-order logic (with identity) in the language of arithmetic.

```
∀x ∀y ∀z (x + (y + z) = (x + y) + z)
∃x (∀y (x + y = y)  ∧  ∀y (y + x = y))
∀x ∀y ∃z (x + z = y)
```

The first sentence says that addition is associative, in the sense that the order of parentheses doesn't matter. The second says there is an additive identity—something which makes no difference when added to anything—that is, zero. The third says that any two things have a difference, which can be added to one to reach the other. (This principle is *false* about the natural numbers, because if y is smaller than x then there is no natural number you can add to x to reach y. But the principle is true about the *integers*, which include negative numbers.)

These examples are all *sentences*, but they have as part of their internal structure things which aren't sentences, like

```
∀y (x + y = y)
```

Here `x` is what we call a *free variable*, which doesn't correspond to any quantifier within the expression. A "sentence fragment" like this is called a **formula**. In order to explicitly define what a first-order *sentence* is, it's helpful to start by defining the more general class of *formulas*. After we've done that, we'll pick out the sentences as certain special formulas that tie up all their loose variables.

5.1.1 Definition

Let L be a signature. We have already defined the structure of terms. Just to refresh our memory, the L-terms are recursively defined by the following rules. (We'll generally leave the signature L implicit.)

$$\frac{x \text{ is a variable}}{x \text{ is a term}} \qquad \frac{c \text{ is a constant}}{c \text{ is a term}}$$

$$\frac{f \text{ is a one-place function symbol} \qquad a \text{ is a term}}{f\texttt{(}a\texttt{)} \text{ is a term}}$$

$$\frac{f \text{ is a two-place function symbol} \qquad a \text{ and } b \text{ are terms}}{f\texttt{(}a\texttt{,}b\texttt{)} \text{ is a term}}$$

The **first-order formulas in the language L,** or **L-formulas** for short, are recursively defined by the following six rules. (We drop the L when it is clear in context which language we are talking about.) First, three rules for forming the simplest formulas.

$$\frac{a \text{ is a term} \qquad b \text{ is a term}}{\texttt{(}a\texttt{=}b\texttt{)} \text{ is a formula}}$$

$$\frac{F \text{ is a one-place predicate} \qquad a \text{ is a term}}{F\texttt{(}a\texttt{)} \text{ is a formula}}$$

$$\frac{R \text{ is a two-place predicate (relation symbol)} \qquad a \text{ and } b \text{ are terms}}{R\texttt{(}a\texttt{,}b\texttt{)} \text{ is a formula}}$$

(It's easy, and standard, to generalize these rules to allow predicates with more than two places. But we'll stick to one and two, because that's all we happen to need, and it will keep our notation simpler.)

Once again, there is a notational issue. Normally we write some relation symbols, like `≤`, in between the two terms it applies to, as in `0 ≤ suc 0`. But again, it is more convenient to only ever do things one way in our official notation, and our official choice is "prefix" notation. So again, *officially*, we would write that formula as `≤(0, suc(0))`. But we will hardly ever be official.

(Notice that as far as the *syntax* goes, we could treat the identity symbol `=` as just another basic relation symbol. But identity gets its own special treatment because it has a special interpretation.)

Next, three more rules for building up more complex formulas.

$$\frac{A \text{ is a formula}}{\neg A \text{ is a formula}}$$

$$\frac{A \text{ is a formula} \qquad B \text{ is a formula}}{\texttt{(}A\texttt{∧}B\texttt{)} \text{ is a formula}}$$

$$\frac{A \text{ is a formula} \qquad x \text{ is a variable}}{\forall x\ \ A \text{ is a formula}}$$

Again, while we want to have official rules, in practice we often take some notational liberties when we are writing down formulas—just like we did with terms. We may drop parentheses or modify spacing a bit, if that makes things more reader-friendly.

Another thing to notice is that many standard connectives don't appear in these official formation rules, such as the conditional `→`, or the existential quantifier `∃`. That's because we can define them up from the basic materials in the definition. For example, later on we'll define `∃x (x + x = x)` to be just a notational shortcut for the official formula `¬∀x ¬(x + x = x)`. We'll go over these abbreviations in the next section. It's helpful to do things this way, because it means that when we are proving things inductively about formulas we only need to consider a small number of formation rules.

As with numbers, strings, and terms, formulas have an Inductive Property and an Injective Property. The Inductive Property intuitively says that every formula can be produced in at least one way from these rules, and the Injective Property intuitively says that every formula can be produced in at most one way from these rules. At this point, we won't go through the fuss of being totally official about these properties. The thing that matters is that our two familiar friends—inductive proof and recursive definition—also work for formulas. We'll do some examples of induction and recursion for formulas very soon.

(The *Inductive Property* for Formulas follows from our recursive definition of "*L*-formula." The *Injective Property* for Formulas is more complicated to prove: we would have to prove another parsing theorem, like the one for terms in Section 3.3. This parsing theorem would show that our particular system of notation for writing down formulas is not ambiguous: no two different ways of applying the rules ever produce the same string by accident. Since this proof of the parsing theorem for formulas is very similar to the analogous proof for terms, we won't bother going into it.)

5.1.2 Exercise
Write out the Inductive Property and the Injective Property for Formulas, based on the formation rules in Definition 5.1.1. (You can use the Inductive Property and the Injective Property for terms as a model.)

5.1.3 Example

The **first-order language of arithmetic** consists of the first-order formulas with the signature of the language of arithmetic: `0` , `suc` , `+` , `·` , and `≤` .

The **first-order language of strings** consists of the first-order formulas with the signature of the language of strings: `⊕` , `≤` , the empty-string constant `""` , and the constant for each symbol in the standard alphabet.

5.1.4 Example

Let A be this formula in the language of arithmetic:

```
¬((x = y) ∧ ¬∀z ¬((x + y = x + z) ∧ ¬(x = z)))
```

This formula has two *free variables*, `x` and `y` , which aren't "bound" by any quantifiers. In contrast, the variable `z` is *bound* by the quantifier `∀z` , so it is not a free variable.

5.1.5 Definition

We recursively define the relation x **is free in** A, for a variable x and a formula A, as follows:

1. x is free in $(a{=}b)$ iff x occurs in the term a or x occurs in the term b (as we defined in Definition 3.5.3).
2. x is free in $R(a,b)$ iff x occurs in the term a or x occurs in the term b.
3. x is free in $\neg A$ iff x is free in A.
4. x is free in $(A \wedge B)$ iff x is free in A or x is free in B.
5. x is free in $\forall y\ A$ iff x is a distinct variable from y and x is free in A.

5.1.6 Definition

A **first-order sentence** in the language L—or an L**-sentence**, for short—is a first-order L-formula with no free variables.

A **formula of one variable** is a formula with at most one free variable. Similarly, a **formula of n variables** is a formula with at most n free variables.

Like we did for terms, we'll often use the notation $A(x)$ for a formula of one variable, namely x, and similarly $B(x, y)$ for a formula of two variables, etc.

A formula $A(x)$ is like a sentence with a hole in it. One thing we often want to do is plug a term into that hole, to see what the formula "says about" a certain thing. For instance, take the formula

```
¬(x = 0) ∧ ∀y ¬(x + y = y)
```

This says, "x is not zero, and adding x to anything never gives you the same thing back". We can plug the term `suc 0` into the `x` slot, to get the sentence

```
¬(suc 0 = 0) ∧ ∀y ¬(suc 0 + y = y)
```

which says: "one is not zero, and adding one to anything never gives you the same thing back." The basic idea is that each free occurrence of the variable `x` gets replaced with the term `suc 0`. But there are a couple of details to be careful about.

First, what if the quantifier `∀x` appears in the formula? For instance, what happens when we plug `suc 0` in for `x` in this formula, instead?

```
¬(x = 0) ∧ ¬∀x (x + x = x)
```

Only the first occurrence of `x` is free here—the other occurrences are "bound" by the quantifier `∀x`. So when we plug in `suc 0`, we get

```
¬(suc 0 = 0) ∧ ¬∀x (x + x = x)
```

The second conjunct doesn't say anything in particular about `suc 0`. The "bound" `x`'s are left untouched.

What if the term we're plugging in contains variables of its own? Say we plug the term `suc y` in for `x` in the formula

```
¬∀y (x = y)
```

This formula says "not everything is x" (in other words, "there is something other than x"). So we'd like the result of substituting in `suc y` to say "not everything is y's successor". But if we just naïvely replace each free `x` with `suc y`, we'd end up with

```
¬∀y (suc y = y)
```

This says "not everything is its own successor," which is a very different idea. The trouble is that the free variable `y` in `suc y` has been *captured* by the quantifier `∀y`.

Handling this edge case correctly is tricky. But in practice, we can always avoid dealing with this messy case by just using different bound variables instead, since this never makes a difference to the meaning of the formula. For instance, instead of plugging `suc y` into `¬∀y (x = y)`, we could plug it into `¬∀z (x = z)` instead. Then the result is `¬∀z (suc y = z)`, which has the meaning we wanted ("not everything is y's successor"), because the variable `y` escapes being captured.

Since we can always avoid the hard case of captured variables by judicious relettering, we will simply leave substitution undefined in this case: that is, substitution will be undefined when a *free* variable in the term we are plugging in is also *bound* in the expression we are plugging it into. (But this is never going to come up—we'll make sure to avoid this edge case—so you don't really have to worry about it.)

5.1.7 Definition

Let A be a formula, let x be a variable, and let t be a term. We say x **is free for** t **in** A iff there is no subformula of A with the form $\forall y\ B$, where y occurs in t and x is free in B.[1]

5.1.8 Definition

Suppose A is a formula, x is a variable, t is a term, and x is free for t in A. The **substitution instance** $A[x \mapsto t]$ is defined recursively as follows.

$$\begin{aligned}
(a{=}b)[x \mapsto t] &\text{ is } (a[x \mapsto t]{=}b[x \mapsto t]) \\
R(a,b)[x \mapsto t] &\text{ is } R(a[x \mapsto t],b[x \mapsto t]) \\
(\neg A)[x \mapsto t] &\text{ is } \neg(A[x \mapsto t]) \\
(A \wedge B)[x \mapsto t] &\text{ is } (A[x \mapsto t] \wedge B[x \mapsto t]) \\
(\forall y\ A)[x \mapsto t] &\text{ is } \begin{cases} \forall y\ A & \text{if } x \text{ and } y \text{ are the same variable} \\ \forall y\ A[x \mapsto t] & \text{otherwise, as long as } y \text{ does not occur in } t \end{cases}
\end{aligned}$$

Note that this definition doesn't say what to do in the case of a bound variable which *does* occur in t. If there are any bound variables like that, then x is not free for t in

[1] We haven't officially defined "subformula", but this is the only place where we happen to need it. We can recursively the set of subformulas of a formula in a fairly obvious way.

A, and the substitution instance is undefined. (But we won't always bother stating this qualification explicitly.)

5.1.9 Notation

Just as we did with terms, we'll often use simplified notation for substitution in formulas. If $A(x)$ is a formula of one variable, then $A(a)$ means the same thing as $A[x \mapsto a]$. Similarly if $B(x, y)$ is a formula of two variables, then $B(a, y)$ means the same thing $B[x \mapsto a]$, and $B(x, a)$ means the same thing as $B[y \mapsto a]$.

5.1.10 Exercise

Prove by induction on the structure of formulas that, for any formula A and variable x, if x is not free in A, then $A[x \mapsto t]$ is the same formula as A.

5.2 Semantics

Consider the standard model of arithmetic $\mathbb{N}$. A truth about this structure is that every number has a successor, and not every number *is* a successor. This is a truth which we can formalize in the first-order language of arithmetic:

```
∀x ∃y (y = suc x)  ∧  ¬∀x ∃y (suc y = x)
```

(We haven't officially introduced the existential quantifier `∃` yet, but we will very soon.)

First-order sentences are a useful tool for describing structures. In order to use them this way, though, we need to be explicit about what makes *this* sentence a good description of $\mathbb{N}$, and this *other* sentence a bad description of $\mathbb{N}$:

```
∀x ∀y ∃z (x + z = y)
```

As we noted in Section 5.1, this says that for any numbers, there is a *difference* which added to the first produces the second. This is a false claim about the natural numbers structure: for example there is no natural number you can add to 3 to get 1. Of course, there is another sense in which any two numbers *do* have a difference, which is formalized by this sentence:

```
∀x ∀y ∃z ((x + z = y) ∨ (y + z = x))
```

(This is called the *absolute* difference between two numbers.) Our goal in this section is to give a precise definition of "The first-order sentence A is true in the structure S," and then check that this definition works the way it should.

Just like for terms, we'll want to define the semantics for the first-order language recursively. But sentences aren't just built out of sentences: in general, they're built out of *formulas*, which can contain free variables. So to get to a definition of the semantics for sentences, we'll need to go by way of a definition of the semantics for formulas more generally.

But what would it even mean to say that an open formula like `suc y = x` is *true* in a structure like $\mathbb{N}$? This formula isn't true or false all on its own: first we need to choose values for the variables. So in order to achieve our goal of defining "true sentence", we'll work through the intermediate goal of defining what it is for a *formula* to be true in a structure *with respect to some choice of values for the variables.* We have already used this idea of a choice of values for the variables, when we defined our semantics for terms with free variables in Section 3.5. Here is a reminder:

5.2.1 Definition

Let S be a structure. A **variable assignment function** (in S) is a partial function from variables to elements of the domain of S.

If A is a formula, a variable assignment g is **adequate for** A iff g is defined for every variable which is free in A.

5.2.2 Definition

Suppose S is a structure, g is a variable assignment, x is a variable, and d is an element of the domain of S. Then the **variant** assignment $g[x \mapsto d]$ modifies the assignment g in just one place, by assigning a new value to the variable x. That is to say, $g[x \mapsto d]$ is the variable assignment function such that, for each variable y,

$$g[x \mapsto d](y) = \begin{cases} d & \text{if } x \text{ is } y \\ g(y) & \text{if } x \text{ is distinct from } y \text{ and } y \text{ is in the domain of } g \end{cases}$$

5.2.3 Definition

Let S be a structure, let A be a formula, and let g be a variable assignment. We recursively define the relation g **satisfies** A **in** S as follows. (The structure S is usually left implicit.)

1. Suppose a and b are terms. Then g satisfies an identity formula $(a=b)$ iff a and b denote the same element of the domain of S, with respect to the assignment g. That is, g satisfies $(a=b)$ iff $[\![a]\!]\,g = [\![b]\!]\,g$.

2. Suppose F is a one-place predicate and a is a term. Then g satisfies the formula $F(a)$ iff $[\![a]\!]\,g$ is in the extension F_S.

3. Suppose R is a two-place predicate and a and b are terms. Then g satisfies the formula $R(a,b)$ iff the ordered pair $([\![a]\!]\,g, [\![b]\!]\,g)$ is in the extension R_S.

4. Suppose A is a formula. Then g satisfies the negation $\neg A$ iff g does not satisfy A.

5. Suppose A and B are formulas. Then g satisfies the conjunction $(A \wedge B)$ iff g satisfies A and g satisfies B.

6. Suppose A is a formula and x is a variable. Then g satisfies the generalization $\forall x\ A$ iff, for every d in the domain of S, the variant assignment $g[x \mapsto d]$ satisfies A.

5.2.4 Definition

If A is a sentence, then A is **true in** S iff the empty assignment satisfies A in S. Otherwise A is **false in S.**

5.2.5 Definition

As we discussed in Section 3.5, when it is clear in context which variables are important, we can often talk about assignments in a way that leaves the variables implicit.

If $A(x)$ is a formula with at most one free variable x, and d is in the domain of S, then $A(x)$ **is true of** d **in** S iff the assignment $[x \mapsto d]$ satisfies $A(x)$ in S.

Similarly, if $A(x, y)$ is a formula with at most two free variables, and d_1 and d_2 are in the domain of S, then $A(x, y)$ **is true of** (d_1, d_2) **in** S iff the assignment $[x \mapsto d_1, y \mapsto d_2]$ satisfies $A(x, y)$ in S.

It's easy to generalize this to formulas of any number of free variables.

Using this alternative way of talking about assignments, it's helpful to restate what Definition 5.2.3 says about the special case of *sentences*. This is a bit simpler and more intuitive than the general case of formulas, and it is by far the most important case in practice.

5.2.6 Proposition
Let S be a structure.

1. Suppose a and b are closed terms. Then $(a = b)$ is true in S iff a and b have the same denotation in S; that is, $[\![a]\!]_S = [\![b]\!]_S$.
2. Suppose F is a one-place predicate and a is a closed term. Then $F(a)$ is true in S iff $[\![a]\!]_S \in F_S$.
3. Suppose R is a two-place predicate and a and b are closed terms. Then $R(a, b)$ is true in S iff $([\![a]\!]_S, [\![b]\!]_S) \in R_S$.
4. Suppose A is a sentence. Then $\neg A$ is true in S iff A is not true in S.
5. Suppose A and B are sentences. Then $(A \wedge B)$ is true in S iff A is true in S and B is true in S.
6. Suppose $A(x)$ is a formula of one variable x. Then $\forall x\ A(x)$ is true in S iff, for every element d in the domain of S, $A(x)$ is true of d in S.

It can also sometimes be handy to describe the semantics for sentences in a way that's more closely analogous to the denotations of terms.

5.2.7 Definition
Let S be a structure, A a formula, and g an assignment which is adequate for A. The **truth-value** of A with respect to g in S, written $[\![A]\!]_S g$ for short, is True if g satisfies A in S, and otherwise it is False. (Usually we leave off the S subscript from $[\![A]\!]_S$ for simplicity.) That is:

$$[\![A]\!] g = \begin{cases} \text{True} & \text{if } g \text{ satisfies } A \text{ in } S \\ \text{False} & \text{otherwise} \end{cases}$$

5.2.8 Proposition
Suppose that g and h are assignments that assign the same value to every variable which is free in A. Then g satisfies A iff h satisfies A. In other words, in this case $[\![A]\!] g = [\![A]\!] h$.

Proof
We can prove this using a straightforward but tedious inductive proof, using the recursive definition of first-order formulas. Even though it is tedious, I'll go through this in detail as an example of how inductive proofs about first-order semantics go.

To be explicit, we are proving that every formula A has the following property:

> For any assignments g and h that have the same value for each free variable in A, g satisfies A iff h satisfies A.

We can do this in six steps, one for each way of building up formulas (three "base cases", and three "inductive steps").

1. *Identity.* Let a and b be terms. Suppose that g and h are assignments with the same value for each free variable in $(a=b)$. Then, since every variable that occurs in a or in b is free in $(a=b)$, we know g and h agree on the variables in each of these terms. So, by Exercise 3.5.16 (which showed the analogous fact for *terms*), we know that $[\![a]\!]\,g = [\![a]\!]\,h$, and likewise that $[\![b]\!]\,g = [\![b]\!]\,h$. So:
$$\begin{aligned} g \text{ satisfies } (a=b) \quad &\text{iff} \quad [\![a]\!]\,g = [\![b]\!]\,g \\ &\text{iff} \quad [\![a]\!]\,h = [\![b]\!]\,h \\ &\text{iff} \quad h \text{ satisfies } (a=b) \end{aligned}$$

2. *One-place predicates.* The step for one-place predicates is essentially the same as the step for two-place predicates. So we'll skip ahead to that step.

3. *Two-place predicates.* Let R be a two-place predicate, and let a and b be terms. Suppose that g and h are assignments that agree on each free variable in $R(a,b)$. Again, g and h agree on each variable that occurs in a or in b. So again by Exercise 3.5.16,
$$[\![a]\!]\,g = [\![a]\!]_h \quad \text{and} \quad [\![b]\!]\,g = [\![b]\!]\,h$$
Thus:
$$\begin{aligned} g \text{ satisfies } R(a,b) \quad &\text{iff} \quad ([\![a]\!]\,g, [\![b]\!]\,g) \in R_S \\ &\text{iff} \quad ([\![a]\!]\,h, [\![b]\!]\,h) \in R_S \\ &\text{iff} \quad h \text{ satisfies } R(a,b) \end{aligned}$$

4. *Negation.* Let A be any formula. *Suppose for the inductive hypothesis:*

   > For any formulas g and h that agree on all free variables in A, g satisfies A iff h satisfies A.

   Now, suppose g and h are assignments that agree on all the free variables in $\neg A$. Then (since every free variable in A is also free in $\neg A$) by the inductive hypothesis g satisfies A iff h satisfies A. Thus:
$$\begin{aligned} g \text{ satisfies } \neg A \quad &\text{iff} \quad g \text{ does not satisfy } A \\ &\text{iff} \quad h \text{ does not satisfy } A \\ &\text{iff} \quad h \text{ satisfies } \neg A \end{aligned}$$

5. *Conjunction.* Let A and B be any formulas. *Suppose for the inductive hypothesis:*

    > For any formulas g and h that agree on all free variables in A, g satisfies A iff h satisfies A.

    > For any formulas g and h that agree on all free variables in B, g satisfies B iff h satisfies B.

    Suppose that g and h are assignments that agree on all the free variables in a conjunction $(A \wedge B)$. Then g and h agree on all the free variables in A, and g and h also agree on all the free variables in B (since all of these variables are still free in the conjunction $(A \wedge B)$). So in this case our inductive hypothesis tells us that g satisfies A iff h satisfies A, and likewise that g satisfies B iff h satisfies B. Thus:

    $$\begin{aligned} g \text{ satisfies } (A \wedge B) \quad & \text{iff} \quad g \text{ satisfies } A \text{ and } g \text{ satisfies } B \\ & \text{iff} \quad h \text{ satisfies } A \text{ and } h \text{ satisfies } B \\ & \text{iff} \quad h \text{ satisfies } (A \wedge B) \end{aligned}$$

6. *Quantification.* This is the trickiest step, so let's take it slow. Let x be a variable, and let A be a formula. In this case our inductive hypothesis says:

    > For any assignments g' and h' that agree on all free variables in A, g' satisfies A iff h' satisfies A.

    (We've switched to g' and h' in this "for any" statement, not because it makes any difference to the meaning, but because it will help keep us from getting this generalization tangled up with the one we are about to state.) What we want to show is:

    > For any assignments g and h which agree on all of $\forall x \; A$'s free variables, g satisfies $\forall x \; A$ iff h satisfies $\forall x \; A$.

    To show this, suppose that g and h are assignments which agree on all of the free variables in $\forall x \; A$. Now, if d is any element of the domain of S, we can consider the assignments $g[x \mapsto d]$ and $h[x \mapsto d]$. These have the same value for d, and they also have the same value for each free variable in $\forall x \; A$. This means that they have the same value for each free variable in A (since, by Definition 5.1.5, if a variable is free in A then it is either x or else a free variable in $\forall x \; A$). So our inductive hypothesis tells us that, for each $d \in D_S$,

    $$g[x \mapsto d] \text{ satisfies } A \quad \text{iff} \quad h[x \mapsto d] \text{ satisfies } A$$

Thus:

$$\begin{aligned} g \text{ satisfies } \forall x\ A \quad & \text{iff} \quad g[x \mapsto d] \text{ satisfies } A \text{ for every } d \in D_S \\ & \text{iff} \quad h[x \mapsto d] \text{ satisfies } A \text{ for every } d \in D_S \\ & \text{iff} \quad h \text{ satisfies } \forall x\ A \end{aligned}$$

That completes the inductive proof. □

5.2.9 Exercise

If A is a sentence which is true in a structure S, then every variable assignment g satisfies A in S.

5.2.10 Lemma (Satisfaction Lemma)

Let S be a structure, and let A be a formula. Let g be an assignment which is adequate for $A[x \mapsto t]$. (That is, it has values for every variable in a and A except possibly x.) Let t be a term in which x does not occur, and which can be substituted for x in A. (That is, t does not include any variables which are bound in A.) Suppose furthermore that t denotes d in S with respect to g. That is, $[\![t]\!]g = d$. Then:

$$g \text{ satisfies } A[x \mapsto t] \quad \text{iff} \quad g[x \mapsto d] \text{ satisifies } A$$

Or in our alternative notation:

$$[\![A[x \mapsto t]]\!]g = [\![A]\!](g[x \mapsto d])$$

Here is the most important special case of this fact. Let S be a structure, let $A(x)$ be a formula of one variable x, and let t be a closed term. Then:

$$A(t) \text{ is true in } S \quad \text{iff} \quad A(x) \text{ is true of the denotation of } t \text{ in } S$$

Proof

We can prove this by induction on the structure of formulas. (Again, I'm afraid this is kind of tedious. But here goes.)

Before we begin, let's recall Exercise 3.5.20, which showed very closely related fact about terms. This exercise says that, for any closed term t and term of one variable a, if t denotes d, then

$$[\![a[x \mapsto t]]\!] = [\![a]\!][x \mapsto d]$$

We can use a very similar inductive proof to show a more general version of this fact. (I won't go through it, but it's worth going back to check how you would modify the solution to Exercise 3.5.20 to show this more general version.) This more general version allows other free variables to show up in the terms, and uses an assignment to evaluate them.

> For any variable x, any terms a and t (such that x does not occur in t), and any assignment g which is adequate for $a[x \mapsto t]$ (and thus which is adequate for t as well)
>
> $$\text{If} \quad [\![t]\!]\, g = d \quad \text{then} \quad [\![a[x \mapsto t]]\!]\, g = [\![a]\!](g[x \mapsto d])$$

The basic idea is the same as before; we're just adding in an assignment g to give values to the other free variables that might be around. Putting it a little bit roughly, the idea is still that if t stands for d, then plugging t into a term's x spot has the same effect as assigning d as the value for x. We will use this generalized version of Exercise 3.5.20.

1. *Identity.* Let a and b be terms. Suppose that g is adequate for $(a{=}b)$, and thus g is adequate for each of a and b. Now our generalization of Exercise 3.5.20 tells us:

   $$[\![a[x \mapsto t]]\!]\, g = [\![a]\!](g[x \mapsto d])$$
   $$[\![b[x \mapsto t]]\!]\, g = [\![b]\!](g[x \mapsto d])$$

   Thus:

   $$\begin{aligned} g \text{ satisfies } (a{=}b)[x \mapsto t] \quad &\text{iff} \quad g \text{ satisfies } (a[x \mapsto t]{=}b[x \mapsto t]) \\ &\text{iff} \quad [\![a[x \mapsto t]]\!]\, g = [\![b[x \mapsto t]]\!]\, g \\ &\text{iff} \quad [\![a]\!](g[x \mapsto d]) = [\![b]\!](g[x \mapsto d]) \\ &\text{iff} \quad g[x \mapsto d] \text{ satisfies } (a{=}b) \end{aligned}$$

The steps for one-place predicates and two-place predicates both go basically the same way as the identity step, and the steps for negation and conjunction are straightforward so we'll skip over them. These steps are left as an exercise. That just leaves universal generalizations—as usual, this is the trickiest step.

5. Let y be a variable and let A be a formula. There are two cases to consider: y might be the same variable as x, or it might be a different variable.

   For the first case, all of the occurrences of x in A are *bound* by the quantifier. So (by definition) the substitution instance $(\forall x\ A)[x \mapsto t]$ is the same as

$\forall x \ A$, and, since x is not free in $\forall x \ A$, the assignments g and $g[x \mapsto d]$ have the same value for every variable which is free in $\forall x \ A$. Thus:

$$\begin{aligned} g \text{ satisfies } (\forall x \ A)[x \mapsto t] \quad &\text{iff} \quad g \text{ satisfies } \forall x \ A \\ &\text{iff} \quad g[x \mapsto d] \text{ satisfies } \forall x \ A \end{aligned}$$

Finally we consider the case where y is a distinct variable from x. In this case, the substitution instance $(\forall y \ A)[x \mapsto t]$ is the formula $\forall y \ A[x \mapsto t]$. For the inductive hypothesis, we assume that for any adequate assignment g',

$$g' \text{ satisfies } A[x \mapsto t] \quad \text{iff} \quad g'[x \mapsto d] \text{ satisfies } A$$

Thus,:

$$\begin{aligned} & g \text{ satisfies } (\forall y \ A)[x \mapsto t] \\ & \quad \text{iff} \quad g \text{ satisfies } \forall y \ A[x \mapsto t] \\ & \quad \text{iff} \quad g[y \mapsto d'] \text{ satisfies } A[x \mapsto t] \quad \text{for every } d' \in D_S \\ & \quad \text{iff} \quad g[y \mapsto d'][x \mapsto d] \text{ satisfies } A \quad \text{for every } d' \in D_S \\ & \quad \text{iff} \quad g[x \mapsto d][y \mapsto d'] \text{ satisfies } A \quad \text{for every } d' \in D_S \\ & \quad \text{iff} \quad g[x \mapsto d] \text{ satisfies } \forall y \ A \end{aligned}$$

The third step uses the inductive hypothesis, using the assignment $g[y \mapsto d']$ as our g'. The fourth step follows because, since x and y are distinct variables, $g[x \mapsto d][y \mapsto d']$ and $g[y \mapsto d'][x \mapsto d]$ are the very same assignment.

That completes the inductive proof. □

5.2.11 Exercise

Suppose t and t' are closed terms, and $A(x)$ is a formula of one variable. If t and t' denote the same object (in a structure S) then $A(t)$ and $A(t')$ have the same truth-value (in S). In short:

$$\text{If} \quad [\![t]\!] = [\![t']\!] \quad \text{then} \quad [\![A(t)]\!] = [\![A(t')]\!]$$

So far we've just been working with our "primitive" logical symbols: $\forall$, $\neg$, $\wedge$, and $=$. These are the only logical symbols in our official first-order language. But this isn't a serious limitation. For example, consider "or": say we want to formalize the claim "either $x = 0$ or $x = 1$". We can unpack this statement in terms of "and" and "not":

```
¬(¬(x = 0) ∧ ¬(x = 1))
```

This has exactly the same truth conditions as the “or” statement: the only way it can be false is if both `x = 0` and `x = 1` are false. But in practice, we don’t want to write out this complicated expression every time we want an “or” statement. So we’ll just introduce a handy shorthand: we’ll write $(A \lor B)$ as an abbreviation for the official formula $\neg(\neg A \land \neg B)$. This means that when we write down certain strings, we’re really officially talking about the string you get by unpacking all of the abbreviations. But we’ve already been allowing ourselves a bit of this kind of laziness—for example, by leaving off parentheses.

We can use similar tricks for other logical connectives.

5.2.12 Definition

For any formulas A and B, terms a and b, and variable x:

(a) The **material conditional** $A \to B$ abbreviates the formula $\neg(A \land \neg B)$.

(b) The **biconditional** $A \leftrightarrow B$ abbreviates the formula $(A \to B) \land (B \to A))$.

(c) The **disjunction** $A \lor B$ abbreviates the formula $\neg A \to B$.

(d) The **standard truth** $\top$ is the formula `∀x (x = x)` .

(e) The **standard falsehood** $\bot$ is the formula $\neg\top$.

(f) The **existential generalization** $\exists x\ A$ abbreviates the formula $\neg\forall x\ \neg A$.

(g) The **unique existential** $\exists ! x\ A$ abbreviates the formula

$$\exists x\ \forall y\ ((x = y) \leftrightarrow A[x \mapsto y])$$

(where y is a distinct variable from x).

(h) $a \neq b$ abbreviates the formula $\neg(a = b)$.

5.2.13 Notation

We’ll also have some conventions for leaving out parentheses.

(a) $A \land B \land C$ means $A \land (B \land C)$. (Not that it really matters which way we add parentheses.)

(b) $A \to B \to C$ means $A \to (B \to C)$. (In this case it does matter!)

In *informal* proofs, sometimes you'll see people write things like

```
∀A: A is a set → A ⊆ A
```

or

```
A is consistent ↔ ∃S (S is a structure ∧ A is true in S)
```

In many contexts, this kind of shorthand is convenient and harmless. But in the context of talking about logic, this can be extremely confusing, and it often leads to mistakes. (This holds especially when we start looking at formal languages that can talk about formal languages themselves, in Chapter 6.) So I recommend that you just don't do it, at least for this course. When you want to make a statement about what all things are like, write the word `all`, not the abbreviation `∀`. Only use the symbol `∀` when this symbol is part of a *first-order formula* that you are talking about. Even though it takes a little longer to write down, it's worth it to be clear. You want to be clear to the people you are talking to (and to yourself) when you are *using* some language to say things, and when you are talking *about* a bit of language. That is, you want to be clear what is part of the "meta-language"—the language you are using to make statements about how things are—and what is part of the "object language"—the formal language that you are saying things about. It isn't always absolutely essential to use different symbols in the two languages (like `all` and `∀`) in order to keep them separate, but it can help a lot, and, at least when it comes to the subject matter of this text, I think it's usually a good idea.

5.2.14 Example

Let S be any structure.

(a) For any sentences A and B, the sentence $A{\to}B$ is true in S iff either A is false in S, or B is true in S.

(b) For any formula of one variable $A(x)$, the sentence $\exists x\ A(x)$ is true in S iff there is some d in the domain of S such that $A(x)$ is true of d.

Proof of (a)

Let A and B be sentences. By Definition 5.2.12, $A{\to}B$ abbreviates the sentence

$$\neg(A{\wedge}\neg B)$$

Using Proposition 5.2.6,

$$\begin{aligned} &\neg(A \wedge \neg B) \text{ is true in } S \\ &\quad \text{iff} \quad (A \wedge \neg B) \text{ is not true in } S \\ &\quad \text{iff} \quad A \text{ and } \neg B \text{ are not both true in } S \\ &\quad \text{iff} \quad \text{either } A \text{ is false in } S \text{ or } B \text{ is true in } S \end{aligned}$$

□

Proof of (b)
By Definition 5.2.12, $\exists x \; A(x)$ abbreviates

$$\neg\forall x \; \neg A(x)$$

Using Proposition 5.2.6 again,

$$\begin{aligned} &\neg\forall x \; \neg A(x) \text{ is true in } S \\ &\quad \text{iff} \quad \forall x \; \neg A(x) \text{ is not true in } S \end{aligned}$$

This holds iff it is *not* the case that for every $d \in D$, $\neg A(x)$ is true of d in S. That is, this holds iff there is *some* $d \in D$ such that $\neg A(x)$ is *not* true of d in S. Furthermore $\neg A(x)$ is not true of d (in S) iff $A(x)$ is true of d (in S). Putting this together, $\neg\forall x \; \neg A(x)$ is true in S iff there is some $d \in D$ such that $A(x)$ is true of d in S, which is what we wanted to show. □

5.2.15 Exercise
Prove the following, using Definition 5.2.12 and Proposition 5.2.6.

(a) $A \leftrightarrow B$ is true in S iff A and B have the same truth-value in S.

(b) $A \vee B$ is true in S iff at least one of A and B is true in S.

(c) $\top$ is true in every structure.

(d) $\bot$ is false in every structure.

(e) $\exists! x \; A(x)$ is true in S iff there is exactly one d in the domain of S such that $A(x)$ is true of d.

5.3 Logic

A sentence can be true or false in a structure. One reason we care about this is because we care about the truth. If we are using a certain language to talk about a certain structure S, then we can find out what is true by investigating which sentences are true in S. But there is also another important reason we care about truth-in-a-structure: we also care about what *follows* from what. Which arguments are logically valid? Which theories are logically consistent? One of the neat insights of modern logic (from `CITE` Tarski 1936) is that that we can understand logical consequence and logical consistency by looking at what is true in different structures. (This isn't the only way to do it. In Chapter 8 we'll consider an alternative, older approach to logical consequence and logical consistency, using *proofs* instead of structures.)

Here's the basic idea. The sentence

```
Snow is white
```

is true. But the sentence

```
If snow is white, snow is white
```

is not only true, but *logically* true. We can tell that it is true without knowing anything about the color of snow, and indeed without even knowing what the word "snow" means. This is because, no matter *what* "snow" and "white" happen to mean, the sentence will still be true. Typically, whether a sentence is true depends on its actual "intended" interpretation. But if a sentence is *logically* true, then it is true on *every* possible reinterpretation—including alternative "unintended" interpretations. Even if we perversely interpret `snow` to mean "water" and `white` to mean "impenetrable", we would still understand `If snow is white, snow is white` to express something true (namely, if water is impenetrable, water is impenetrable). The basic idea is that a sentence is logically true if it is true according to *every* interpretation. Since a structure (for a signature L) provides us with a way of interpreting a sentence (in the language with signature L), this means that if an L-sentence is logically true, it should be true in every L-structure.

(But what if we also perversely interpret the word `if` to mean "unless"? Then we could end up understanding the sentence as saying something false. The more precise idea is that a logical truth is true according to every reinterpretation of its *non-logical* expressions. But this raises the difficult question of what is supposed to count as a *logical* expression. What do we hold fixed, and what do we allow to vary? It's not clear how to answer this question in general. But for our present purposes, we do have a precise answer. We are talking specifically about first-order sentences, and instead of "interpretations" in general we are talking specifically about *structures*. That means we are only looking at reinterpretations of the basic symbols in the *signature* of our language: the basic constants, function symbols, and relation symbols. These are the only bits of the language whose extensions are allowed to vary from structure to structure. You could explore how things go with other choices of what to reinterpret. For example, maybe you don't like fixing the same interpretation of *identity* in every structure, or maybe you think we should also fix the interpretation of some extra things. You can do that, and if you do, you will end up with different logical systems with different things counting as "logical consequences." But for now we are just studying one particular logical system: *first-order logic* with identity.)

Our definitions of *logical consistency* and *logical consequence* are based on the same idea as this definition for logical truth. Logical *consistency* means being true according to at least one interpretation. Logical *consequence* means having no *counterexamples*, where a counterexample to an argument is an interpretation according to which every premise is true, but the conclusion is false.

Now let's do this officially.

5.3.1 Definition
Let L be a signature, and let A be an L-sentence.

(a) A is a **logical truth** (or **valid**) iff A is true in every L-structure.

(b) A is **logically consistent** iff A is true in some L-structure.

We can also extend these notions to sets of sentences, in a natural way.

5.3.2 Definition
Let L be a signature. Let X be a set of L-sentences, and let A be an L-sentence. We'll leave the L's implicit.

(a) A structure S is a **model of** X iff every sentence in X is true in S.

(b) X is **(semantically) consistent** iff X has a model. (That is, some structure is a model of X.) Otherwise X is **(semantically) inconsistent**.

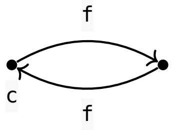

Figure 5.1: An example model.

(c) A is a **logical consequence** of X (for short, $X \vDash A$) iff A is true in every model of X.

5.3.3 Example

Consider a signature with one constant `c`, and one one-place function symbol `f`. This set of sentences is consistent:

{ `∀x ¬(f x = x)`, `(f(f c) = c)` }

Proof
We can show this by explicitly providing a model, like the one in Fig. 5.1.

The domain of this structure S has two elements—for concreteness, say the domain is $\{0, 1\}$. The value of $\mathtt{c}_S$ (the extension of the constant `c`) is 0, and $\mathtt{f}_S$ is the function that takes 0 to 1 and 1 to 0.

There are, of course, many other models for this set of sentences. But to prove they are consistent, we just have to provide one. □

5.3.4 Example

This set of sentences is inconsistent:

{ `∀x ¬(f x = x)`, `(f(f c) = f c)` }

Proof
We can't prove this just by providing examples of structures which are *not* models: rather, we have to give a general argument that there is *no* structure where both of these sentences are true. Here's one way of arguing for this.

Suppose (for reductio) that S is a model of these sentences. Then in particular, `∀x ¬(f x = x)` is true in S. This means that for each d in the domain of S, `¬(f x = x)` is true of d in S. In particular, let d be the element that is denoted by `f c` in S. Since `¬(f x = x)` is true of d in S, and `f c` denotes d, it follows (from the Satisfaction Lemma Lemma 5.2.10) that `¬(f(f c) = f c)` is true in

S. Thus `f(f c) = f c` is false in S. Since S was an arbitrary structure, we have shown that no structure is a model of both `∀x ¬(f x = x)` and `f(f c) = f c`
. □

5.3.5 Exercise

Let `c` be a constant and let `f` be a one-place function symbol. Show whether each of the following sets of sentences is consistent or inconsistent.

(a) { `∃x (f x = c)`, `∃x ¬(f x = c)` }

(b) { `f c = c`, `∀x ¬(f x = c)` }

(c) { `∀x ¬(f c = x)` }

(d) { `∀x ¬(fx = c)`, `∀x ∀y ((f x = f y) → (x = y))` }

5.3.6 Notation

When we use the “turnstile” notation $X \models A$ for logical consequence, it’s common to take a few notational shortcuts. In this context, we usually leave out set brackets, and we use commas instead of union signs. If X and Y are sets of sentences, and A, B, and C are sentences, then instead of these—

$$\{A, B\} \models C \qquad X \cup \{A\} \models B \qquad X \cup Y \cup \{A, B\} \models C \qquad \varnothing \models A$$

—we’ll usually write these simplified versions:

$$A, B \models C \qquad X, A \models B \qquad X, Y, A, B \models C \qquad \models A$$

(For these shortcuts to make sense, we have to make it clear in context which letters stand for sentences and which letters stand for sets of sentences.)

5.3.7 Example

Let X be a set of sentences and let A be a sentence. $X \models A$ iff $X \cup \{\neg A\}$ is inconsistent.

Proof

$X \cup \{\neg A\}$ is *consistent* iff some structure S is a model of $X \cup \{\neg A\}$. This means that every sentence in X is true in S and $\neg A$ is true in S, which means that S is a model of X in which $\neg A$ is true, or equivalently, S is a model of X in which A is not true. So $X \cup \{\neg A\}$ is *inconsistent* iff there is no such S: that is, A is true in every model of X. That’s just what $X \models A$ means. □

5.3.8 Exercise

(a) $\{\bot\}$ is inconsistent.

(b) X is inconsistent iff $X \vDash \bot$.

(c) A is a logical truth iff $\vDash A$.

5.3.9 Example

Prove the following facts about logical consequence, where A and B are any sentences, and X and Y are any sets of sentences.

(a) **Assumption**

$$X, A \vDash A$$

(b) **Weakening**

$$\text{If} \quad X \vDash A \quad \text{then} \quad X, Y \vDash A$$

(c) **Conjunction Introduction**

$$\text{If} \quad X \vDash A \text{ and } X \vDash B \quad \text{then} \quad X \vDash A \wedge B$$

(d) **Modus Ponens**

$$A,\ A \to B \vDash B$$

Proof of (a)

We want to show that A is true in every model of $X \cup \{A\}$. This is obvious: if S is a model of $X \cup \{A\}$, that means that every element of $X \cup \{A\}$ is true in S, so in particular A is true in S. So we're done. } □

Proof of (b)

Suppose that $X \vDash A$: that is, A is true in every model of X. We want to show that $X, Y \vDash A$: that is, that A is true in every model of $X \cup Y$. So suppose that S is a model of $X \cup Y$. That means that every sentence in $X \cup Y$ is true in S. But every sentence in X is a sentence in $X \cup Y$, so S is also a model of X. So A is true in S. This is what we wanted to show. □

Proof of (c)

Suppose that $X \vDash A$ and $X \vDash B$, and suppose that S is a model of X. Then since $X \vDash A$, it follows that A is true in S, and since $X \vDash B$, it follows that B is true in S. By Proposition 5.2.6, this means that $A \wedge B$ is true in S. So $A \wedge B$ is true in every model of X, which is what we wanted to show. □

Proof of (d)
Suppose that S is a model of $\{A,\ A \to B\}$: that is, A is true in S, and $A \to B$ is true in S. By Exercise 5.2.15, the truth of the conditional tells us that either A is false in S, or B is true in S. But A is not false in S, so B must be true in S. This shows that B is true in every model of $\{A,\ A \to B\}$. □

5.3.10 Exercise
Prove the following facts about logical consequence, where A, B, and C are any sentences, and X and Y are any sets of sentences.

(a) **Cut**

$$\text{If} \quad X \vDash A \text{ and } Y, A \vDash B \quad \text{then} \quad X, Y \vDash B$$

(b) **Conjunction Elimination**

$$\text{If} \quad X \vDash A \wedge B \quad \text{then} \quad X \vDash A \text{ and } X \vDash B$$

(c) **Double Negation Elimination**

$$\text{If} \quad X \vDash \neg\neg A \quad \text{then} \quad X \vDash A$$

(d) **Explosion**

$$A, \neg A \vDash B$$

(e) **Proof by Contradiction** (Reductio)

$$\text{If} \quad X, A \vDash B \text{ and } X, A \vDash \neg B \quad \text{then} \quad X \vDash \neg A$$

(f) **Conditional Proof**

$$\text{If} \quad X, A \vDash B \quad \text{then} \quad X \vDash A \to B$$

5.3.11 Example (Leibniz's Law)
Let a and b be terms, and let $A(x)$ be a formula of one variable.

$$\text{If} \quad X \vDash a = b \text{ and } X \vDash A(a) \quad \text{then} \quad X \vDash A(b)$$

Proof
Let S be any model of X. Given that $X \vDash a = b$, we know $a = b$ is true in S. This

implies tells us that $[\![a]\!]_S = [\![b]\!]_S$. Given that $X \vDash A(a)$, we know that $A(a)$ is true in S. By the Satisfaction Lemma Lemma 5.2.10 this implies that $[\![a]\!]_S$ satisfies $A(x)$ in S. Thus $[\![b]\!]_S$ satisfies $A(x)$ in S, and so (using the Satisfaction Lemma a second time) $A(b)$ is true in S. □

5.3.12 Exercise

Let a be a closed term, and let $A(x)$ be a formula of one variable.

(a) **Reflexive Property**

$$X \vDash a{=}a$$

(b) **Universal Instantiation**.

$$\text{If} \quad X \vDash \forall x \; A(x) \quad \text{then} \quad X \vDash A(a)$$

5.3.13 Definition

Let a and b be terms, and let A and B be sentences.

(a) a and b are **logically equivalent** (abbreviated $a \equiv b$) iff a and b denote the same thing in every structure. That is,

$$a \equiv b \quad \text{iff} \quad [\![a]\!]_S = [\![b]\!]_S \quad \text{for every structure } S$$

(b) A and B are **logically equivalent** (also abbreviated $A \equiv B$) iff A and B have the same truth-value in every structure. That is,

$$A \equiv B \quad \text{iff} \quad [\![A]\!]_S = [\![B]\!]_S \quad \text{for every structure } S$$

(c) a and b are **logically equivalent given** X (abbreviated $a \underset{X}{\equiv} b$) iff a and b denote the same thing in every model of X.

(d) A and B are **logically equivalent given** X (abbreviated $A \underset{X}{\equiv} B$) iff A and B have the same truth-value in every model of X.

(This means that $A \equiv B$ means the same things as $A \underset{\varnothing}{\equiv} B$, since every structure is trivially a model of $\varnothing$. Obviously the same goes for terms.)

For the following exercises, let X be a set of sentences, let A, B, and C be sentences, and let a, b, and c be terms.

5.3.14 Exercise

(a) Show:

$$a \underset{X}{\equiv} b \quad \text{iff} \quad X \vDash a{=}b$$

(b) Show:

$$A \underset{X}{\equiv} B \quad \text{iff} \quad X \vDash A{\leftrightarrow}B$$

5.3.15 Exercise

(a) Show that if $X \vDash A{\leftrightarrow}B$, then $X \vDash A$ iff $X \vDash B$.

(b) Is the converse true? That is, suppose $X \vDash A$ iff $X \vDash B$. Does it follow that $X \vDash A{\leftrightarrow}B$?

5.3.16 Exercise

(a) The relation $\underset{X}{\equiv}$ is an equivalence relation: that is, for any sentences A, B, C:

i. $A \equiv A$

ii. If $A \equiv B$ then $B \equiv A$.

iii. If $A \equiv B$ and $B \equiv C$ then $A \equiv C$.

(b) Similarly, for any terms a, b, c,

i. $a \equiv a$.

ii. If $a \equiv b$ then $b \equiv a$.

iii. If $a \equiv b$ and $b \equiv c$ then $a \equiv c$.

5.3.17 Exercise

The following are equivalent:

$$X \vDash A$$

$$A \underset{X}{\equiv} \top$$

$$A{\wedge}B \underset{X}{\equiv} B \quad \text{for every sentence } B$$

5.3.18 Exercise
$A \underset{X}{\equiv} B$ iff $\neg A \underset{X}{\equiv} \neg B$

5.3.19 Exercise
The following are equivalent:

$$\begin{array}{ll} X \text{ is inconsistent} & \\ A \underset{X}{\equiv} B & \text{for all sentences } A \text{ and } B \\ A \underset{X}{\equiv} \neg A & \text{for some sentence } A \\ \top \underset{X}{\equiv} \bot & \end{array}$$

5.3.20 Exercise
Let X be a set of sentences, let $A(x)$ be a formula of one variable, and let a and b be terms.

$$\text{If} \quad a \underset{X}{\equiv} b \quad \text{then} \quad A(a) \underset{X}{\equiv} A(b)$$

5.3.21 Exercise
$A \to (B \to C)$ and $(A \wedge B) \to C$ are logically equivalent, for any sentences A, B, and C.

5.3.22 Definition
We can also generalize these logical notions from *sentences* to arbitrary *formulas*. Let X be a set of formulas, and let A and B be formulas.

(a) A pair (S, g) of a structure and a variable assignment is a **model of** X iff g satisfies every formula in X in S.

(b) X is **consistent** iff X has a model.

(c) A is a **logical consequence** of X iff $X \cup \{\neg A\}$ is inconsistent.

(d) A and B are **logically equivalent** given X iff the models of $X \cup \{A\}$ are just the same as the models of $X \cup \{B\}$.

All of the facts we've proved in this section straightforwardly extend to arbitrary formulas, and not just sentences. Since the arguments are almost identical, we won't bother repeating them. We should just note what the generalized version of Universal Instantiation says.

5.3.23 Proposition (Universal Instantiation)
For any formula A and any term a,

$$\forall x \ A \vDash A[x \mapsto a]$$

5.3.24 Exercise
Let X be a set of formulas, let A and B be formulas, and let x be a variable.

(a) Suppose x is not free in B. If g satisfies B in S, then for any d in the domain of S, $g[x \mapsto d]$ also satisfies B in S.

(b) Suppose that x is not free in any formula in X (though it may be free in A). If $X \vDash A$, then $X \vDash \forall x \ A$.

Hint. Let S be a structure and g be an assignment. If $X \vDash A$, then there is no d in S for which $(S, g[x \mapsto d])$ is a model of $X \cup \{\neg A\}$.

This exercise proves that the rule of **Universal Generalization** preserves validity. It corresponds to a certain common pattern of reasoning. If we want to prove *everything is awesome,* we can reason as follows:

> Let x be an arbitrary thing. Then [insert reasoning here]. It follows from this reasoning that x is awesome. So, since x was arbitrary, it follows that everything is awesome.

The condition "x is an arbitrary thing" corresponds to the constraint that x *is not free* in any of the premises of this argument. Here "arbitrary" means that we are making no assumptions at all about what x is like, and we are using "x is free in B" as a way of formalizing the intuitive notion "B says something about what x is like".

5.4 Theories and Axioms

The ancient Greeks knew a lot about geometry. Around 300 BCE, the Greco-Egyptian mathematician Euclid systematized this knowledge by showing how a huge variety of different facts about figures in space could be derived from a very small collection of basic principles—or *axioms*—about points, lines, and circles. It was a beautiful accomplishment, and since then Euclid's "axiomatic method" has been deeply influential. It's a wonderful thing when we can find a simple set of basic principles with far-reaching implications—and this kind of thing has been done over and over again with remarkable success in mathematics, in empirical science, and in philosophy. Consider just a few examples from the history of philosophy. In the 18th century Isaac Newton (among others) gave elegant principles describing space, time, and the motion of material objects. In the 19th century John Stuart Mill (among others) gave elegant principles describing which actions are best. In the 20th century, Ruth Barcan Marcus (among others) gave elegant principles describing essence and contingency—about what particular objects could have been like.

(Of course in each case, there are important questions about whether the principles these philosophers gave are *true*. Lots of false statements are "axioms" in some theory or other. Calling certain statements "axioms" and their consequences a "theory" isn't taking any stand on whether they are true or false.)

We now have some tools to help us understand how this works. Later on we will also encounter some striking ways that it *doesn't* work (especially in Section 7.7 and Section 8.5).

There are two parts to this deep idea: "a simple set of basic principles," and "far-reaching implications". In the previous section we worked out an account of implications—that is, an account of first-order logical consequence. The set of everything that logically follows from certain principles is called a *theory*.

5.4.1 Definition

Let T be a set of sentences.

(a) Let X be a set of sentences. We say X **axiomatizes** T iff T includes all and only the logical consequences of X. That is,

$$T = \{A \mid X \vDash A\}$$

We call the elements of X **axioms** for T, and we call the elements of T **theorems** of T.

(b) T is a **theory** iff there is some set of sentences X that axiomatizes T.

Here are some examples.

5.4.2 Definition

The minimal theory of arithmetic, called Q for short, is axiomatized by the following sentences. (Here and throughout, whenever we present an axiom with free variables, we should understand this as implicitly adding universal quantifiers to the front as needed to turn the open formula into a sentence.)

```
             0 ≠ suc x
suc x = suc y  →  x = y

         x + 0 = x
     x + suc y = suc (x + y)

         x · 0 = 0
     x · suc y = (x · y) + x

           ¬(x < 0)
    x < suc y  ↔  (x < y  ∨  x = y)

   x < y  ∨  x = y  ∨  y < x
   x = 0  ∨  ∃y (x = suc y)
```

(See CITE BBJ 16.2.) The first two axioms capture the Injective Property of Numbers. The next three pairs capture the recursive definitions of addition, multiplication, and less-then, respectively. The last two axioms give us a kind of exhaustiveness conditions. (But they are not nearly as powerful as our full-fledged exhaustiveness condition, the Inductive Property of numbers.)

5.4.3 Definition

The minimal theory of strings, or S for short, has the following axioms.

Remember that the language of strings includes a constant for each one-symbol string. We start off with a long list of axioms of the form

c_1 ≠ c_2

for each pair of distinct constants for single symbols. (For example, this includes the axioms `"A" ≠ "B"`, `"A" ≠ "C"`, `"B" ≠ "C"`, and so on.) Next we have axioms corresponding to the Injective Property of strings. We will use `Sym(x)` as an abbreviation for the long formula

```
x = "A"  ∨  x = "B"  ∨  ⋯  ∨  x = c_n  ∨  ⋯
```

which lists out all of the constants that stand for single symbols. The axioms:

```
Sym(x)  → x ⊕ y ≠ ""
Sym(x₁) → Sym(x₂)  →  x₁ ⊕ y₁ = x₂ ⊕ y₂  →  (x₁ = x₂ ∧ y₁ = y₂)
```

We then state some basic properties of joining together strings.

```
"" ⊕ x = x
x ⊕ "" = x
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)
```

Next, we have some axioms for the "no-longer-than" relation `≲`.

```
"" ≲ x
x ≲ "" ↔ x = ""
Sym(x₁) → Sym(x₂)  →  x₁ ⊕ y₁ ≲ x₂ ⊕ y₂  ↔  y₁ ≲ y₂
x ≲ y  ∨  y ≲ x
```

Finally, we have an axiom that says *every* string is either empty, or else the result of adding some symbol to the beginning of another string.

```
x = ""  ∨  ∃y ∃z (Sym(y)  ∧  x = y ⊕ z)
```

The theory Q does not include all of the truths of arithmetic—just some of them. Likewise, the theory S just includes a small fragment of the first-order truths in the standard string structure 𝕊. These theories are important because, while they are both pretty simple,[2] at the same time they also turn out to be strong enough to represent lots of interesting structure. They will be important players in Chapter 6 and Chapter 7.

[2] The list of axioms for S is not short: because we are using such an extravagantly large alphabet with over 100,000 basic symbols, the full list of axioms would take more than 10 billion symbols to write out explicitly! Of course, if we cared about doing things more efficiently we could really do everything important with a much, much smaller alphabet. If we decided to be super-efficient and write everything using a two-symbol alphabet ("binary code"), then the fully-written out sentence that axiomatizes the theory analogous to S would comfortably fit on a single page.

5.4.4 Exercise
The following are equivalent:

(a) T is a theory.
(b) T axiomatizes T.
(c) For each sentence A, if $T \vDash A$ then $A \in T$.
(d) For each sentence A, $A \in T$ iff $T \vDash A$.

Notice that it isn't built into the definition of a theory that its axioms have to be *simple*. For example, we could count every single truth of arithmetic as an "axiom", as far as the definition goes. But theories with simple axioms are especially nice. These two examples of theories do have simple axioms: in particular, we can give all of the axioms in a short list.

5.4.5 Definition
A theory T is **finitely axiomatizable** iff there is some finite set of sentences X that axiomatizes T.

5.4.6 Example
The minimal theory of arithmetic Q and the minimal theory of strings S are each finitely axiomatizable.

5.4.7 Exercise
Suppose that T is a finitely axiomatizable theory with axioms $A_1, \dots, A_n$. Then for any sentence B, B is a theorem of T iff

$$(A_1 \wedge \cdots \wedge A_n) \to B$$

is a logical truth.

You might think that *only* finitely axiomatizable theories are simple enough to be useful for humans. But that isn't true: some infinite sets of axioms are also practically useful. Here is an important example:

5.4.8 Definition
First-order Peano arithmetic PA is the theory with the following axioms. There are two parts. The first part:

```
        suc x ≠ 0
  suc x = suc y → x = y

        x + 0 = x
      x + suc y = suc (x + y)

        x · 0 = 0
      x · suc y = (x · y) + x

         ¬(x < 0)
     x < suc y  ↔  (x < y  ∨  x = y)
```

These axioms are the same as in minimal arithmetic Q. For the second part, we have some axioms that are intended to capture the Inductive Property of Numbers. For each formula *A*(x), we have an axiom of this form:

```
A(0) ∧ ∀x (A(x) → A(suc x)) → ∀x A(x)
```

Axioms of this form are called **instances of the induction schema**.

First-order Peano arithmetic has infinitely many axioms. So we can't simply list all of the axioms. But we can still *describe* all of the axioms using a simple rule. It is easy to tell whether a sentence is an instance of the induction schema just by looking at its syntactic structure. A theory like this is called *effectively axiomatizable*: what counts as an axiom can be checked using some straightforward procedure. But we won't give an official definition of this notion until after we have said more about the idea of a "straightforward procedure" in Chapter 7.

Here's another example of a theory which is effectively axiomatizable, but not finitely axiomatizable. (The details really don't matter for the purposes of this course, so don't get hung up. The main thing to notice is just that we *can* formalize our ordinary reasoning about sets using a first-order theory, with a bit of work.)

5.4.9 Definition

The **first-order language of pure set theory** is a first-order language with just one relation symbol ∈ . **First-order set theory**, or ZFC, is the theory in this language with the following axioms. As usual, we add universal quantifiers to bind the free variables, and *A* can be any formula in this first-order language of sets. We'll use `z ⊆ x` as an abbreviation for `∀w (w ∈ z → w ∈ x)`.

(This axiomatization is pretty old-school. It's stated in a way which avoids mentioning ordered pairs or functions directly, which makes things a bit harder than you might expect.) Each axiom has a name.

Extensionality.

```
∀z(z ∈ x ↔ z ∈ y)  →  x = y
```

Separation.

```
∃y ∀z (z ∈ y  ↔  (z ∈ x ∧ A))
```

Power Set.

```
∃y ∀z (z ∈ y  ↔  z ⊆ x)
```

Union.

```
∃y ∀z (z ∈ y  ↔  ∃w (w ∈ x  ∧  z ∈ w))
```

Choice.

```
∀y (y ∈ x  →  ∃z (z ∈ y))  →
    ∃w ∀y (y ∈ x  →  ∃!z (z ∈ y  ∧  z ∈ w))
```

Infinity.

```
∃x (∃y (y ∈ x)  ∧
    ∀y (y ∈ x  →  ∃z (z ∈ x  ∧  y ⊆ z  ∧  y ≠ z)))
```

Foundation.

```
∃y (y ∈ x) → ∃y (y ∈ x  ∧  ¬∃z(z ∈ y  ∧  z ∈ x))
```

Replacement.

```
∀y (y ∈ x  →  ∃!z A)  →  ∃w ∀y (y ∈ x  →  ∃z (z ∈ w  ∧  A))
```

Not all theories are simple. One way of describing a theory is "bottom-up", by starting with some nice set of axioms that generates the whole theory. But we can also describe a theory "top-down", by starting with a *structure*. For example, the set of *all truths of arithmetic* is a theory, since anything that follows from the truths of arithmetic is another truth of arithmetic.

5.4.10 Definition

Let S be a structure. The **first-order theory of** S is the set of all sentences which are true in S. We call this $\mathrm{Th}\, S$ for short.

5.4.11 Example

(a) The **first-order theory of arithmetic** is $\mathrm{Th}\, \mathbb{N}$, the set of all sentences that are true in the standard model of arithmetic.

(b) The **first-order theory of strings** is $\mathrm{Th}\, \mathbb{S}$, the set of all sentences that are true in the standard string structure.

Notice that we've used the word "theory" in this definition, but we haven't really justified using this word yet. Is the theory of a structure really a theory in the sense of Definition 5.4.1—a set of sentences that are the consequences of some axioms (perhaps infinitely many)? Yes: the first-order theory of any structure is a theory. But not every theory is the theory of some structure. The following exercises show these facts.

5.4.12 Definition

A set of L-sentences X is **(negation) complete** iff for every L-sentence A, either $A \in X$ or $\neg A \in X$ (or both).

5.4.13 Exercise

Suppose X is a set of sentences.

(a) Suppose X is the first-order theory of some structure S. Then (i) X is consistent, (ii) X is negation-complete, and (iii) X is a theory.

(b) If X is consistent and complete, then X is the theory of some structure:

that is, there is some structure S such that $X = \text{Th}\, S$.

5.4.14 Exercise
For each of the following, either give an example or explain why there is no example.

(a) A theory which is not negation-complete.

(b) A theory which is not consistent.

(c) A consistent and negation-complete set of sentences which is not a theory.

Here's an important question: when can a structure be completely described using simple axioms? Can we come up with a simple system of axioms from which we can derive *all* of the truths? For example, First-Order Peano Arithmetic looks like a reasonable candidate for a set of axioms that might capture all of the truths of arithmetic. (In that case, First-Order Peano Arithmetic would be the very same theory as the complete first-order theory of arithmetic $\text{Th}\, \mathbb{N}$.) Similarly, ZFC looks like a reasonable candidate for a set of axioms that captures all of the truths of pure set theory. But do they really? We will answer this question in Section 8.5. But here's a preliminary result.

5.4.15 Exercise
If S is a finite structure, then $\text{Th}\, S$ is finitely axiomatizable.

Note. This exercise is harder than most.

5.5 Definite Descriptions

We're now going to consider an extension to standard first-order logic: the word "the". This extension is convenient and useful, and we'll help ourselves to it in what follows. But it doesn't really extend the expressive power of sentences of first-order logic. We will prove an "elimination theorem": this shows that whatever we can assert about structures using the word "the", we could also assert without using that word. The idea, which comes from Bertrand Russell [CITE], is that "the F is G" is logically equivalent to "There is exactly one F, and it is G".

But just because we *can* eliminate "the" doesn't mean we have to. Definite descriptions are still useful: even though they don't increase the expressive power of our *formulas*, they do increase the expressive power of our *terms*. This makes it a bit easier for us to describe particular objects and functions in a structure.

5.5.1 Definition

The **definite description language** with signature L is defined inductively using all of the syntax rules for first-order terms and formulas, and one additional syntax rule:

$$\frac{A \text{ is a formula} \qquad x \text{ is a variable}}{\texttt{the}\ x\ A \text{ is a term}}\ \text{Definite Description}$$

The definite description language is called Def L, for short.

(Definite descriptions are sometimes instead written $\iota x\ A$ by people who like more Greek letters in their notation.)

Notice an important difference from our old language. Before, formulas could have terms as parts; but now, we can also have *terms* with *formulas* as parts.

Now that we have extended our formal language, we also need to extend some of our earlier definitions to handle the extra case of definite descriptions. First, we'll need to modify the recursive definition of *free variables*. Notice another important change: now we have a way of binding variables in a *term*, and not just in a formula. So some of the variables that occur somewhere in a term might not be free in that term.

5.5.2 Definition

(a) If x is a distinct variable from y, then x is free in $\texttt{the}$ y A iff x is free in A.

(b) The variable y is not free in $\texttt{the}$ y A.

The rest of the clauses of the definition of free variables are exactly the same as for ordinary terms and first-order formulas (Definition 3.5.3 and Definition 5.1.5). (But we have to say "free in the term a" instead of "occurs in the term a".)

Second, we need to extend the recursive definition of *substitution*. (This extends Definition 3.5.7 and Definition 5.1.8).

5.5.3 Definition

(a) If x is distinct from y, then the substitution instance

$$(\texttt{the}\ y\ A)[x \mapsto t] \quad \text{is} \quad \texttt{the}\ y\ (A[x \mapsto t])$$

(b) The substitution instance (`the` y A)[$y \mapsto t$] is just `the` y A again.

The rest of the clauses are unchanged from before.

Third, we need to extend our definitions of *denotation* and *satisfaction* to handle definite descriptions. This part is a bit trickier. Some cases are easy. For example, the description

```
the x (”A” + x = ”ABC”)
```

should denote `BC` in the standard string structure: this is the only thing that satisfies the formula `”A” + x = ”ABC”` in $\mathbb{S}$. More generally, in the context of a structure where exactly one thing that satisfies $A(x)$, then the definite description `the` x $A(x)$ should denote that thing. In this case we call the definite description *proper*; this is the central case, where it is clear what to do.

But in order to give a general definition of the denotation function, we also need to handle *improper* definite descriptions, and this case is less clear. For example, in the string structure, nothing satisfies the formula

```
”A” ⊕ x = ”B”
```

So there is no obvious denotation for `the x (”A” ⊕ x = ”B”)`. The most natural thing to say is that this term just doesn't denote anything at all in $\mathbb{S}$. It is an *empty term.* But this is a perfectly well-formed sentence in the definite description language:

```
the x (”A” ⊕ x = ”B”)  ≤  ”C”
```

What should we do with it? Is it true? Does it have a truth-value at all? And what about slightly more complicated sentences like this one?

```
¬(the x (”A” ⊕ x = ”B”)  ≤  ”C”)
```

There is more than one way to go here—there are a number of different systems of "free logic", the logic of empty terms. For our purposes, we will choose one that happens to make it especially simple to *eliminate* definite descriptions. The idea is that each *atomic* formula that contains an empty term is *false*. Then we use our standard rules to work out the truth-values of more complicated sentences.

(This kind of system is called "negative free logic": no "positive" sentence involving an empty term is true. There are alternative versions of the logic of empty terms that allow some sentences to be true—for instance, identities like "the present king of France is the present king of France". A third option is to say that sentences with empty terms are *neither true nor false.*)

5.5.4 Definition

Suppose A is a formula, x is a variable, S is a structure, and g is an assignment. If there is exactly one d in the domain D_S such that $g[x \mapsto d]$ satisfies A in S, then `the` x A denotes d with respect to S, g. If there is no $d \in D_S$ such that $g[x \mapsto d]$ satisfies A in S, or there is more than one such d, then `the` x A doesn't denote anything (for S, g).

$$[\![\texttt{the}\ x\ \ A]\!]_{S,g} = d \quad \text{iff} \quad d \text{ is the unique element of } D_S \text{ such that } g[x \mapsto d] \text{ satisfies } A \text{ in } S$$

We should clarify what some of our old definitions are supposed to mean, when some terms don't denote anything at all.

- If the term a doesn't denote anything (for S, g), then for any function symbol f, $f\,a$ doesn't denote anything either (for S, g). (And similarly for two-place function symbols.)
- If a doesn't denote anything, or b doesn't denote anything (for S, g), then g does not satisfy $(a = b)$ in S. Similarly, in this case g does not satisfy $R(a, b)$ in S, for any relation symbol R.

Logical consequence, logical equivalence, etc. still make sense in the same way for the definite description language as they did for the ordinary first-order language. For instance, a logical truth in Def L is a formula that is true in all structures, for all adequate assignments.

Every formula in ordinary first-order logic is still a formula in Def L. Furthermore, in this extended language the ordinary formulas still have the same free variables, they are true in the same structures, and so on.

Now for our main theorem about definite descriptions: we can always get rid of them. We can replace any sentence that includes definite descriptions with a *logically equivalent* ordinary sentence that doesn't include any definite descriptions. As I mentioned, the basic idea comes from Bertrand Russell. Russell argued that the sentence "Scott is the author of Waverley" means the same thing as "There is exactly one author of Waverley, and Scott is an author of Waverley.” That is, we can translate a sentence

```
s = the x W(x)
```

to a sentence of the form

```
∃!x W(x) ∧ W(s)
```

The general strategy is to replace definite descriptions with variables that are bound by ordinary quantifiers.

It turns out to be convenient to use an alternative logically equivalent statement, rather than Russell’s original version:

```
∀x (W(x) ↔ s = x)
```

“All and only authors of Waverley are identical to Scott.” This version is slightly simpler, because the predicate “author of Waverley” only shows up in it once, instead of twice.

The idea is that we can go through a complex formula systematically replacing each atomic formula containing a definite description with an ordinary first-order formula. Making this idea precise requires a pretty elaborate inductive argument.

First, the simple case.

5.5.5 Exercise
For any formula B,

$$(x=\mathtt{the}\ y\ B) \quad \equiv \quad \forall y\ (B \leftrightarrow x=y)$$

Next, we can prove that every formula in the definite-description language is equivalent to an ordinary first-order formula, by *induction on the number of definite descriptions*—that is, the number of times the word `the` occurs in a formula.

For the base case, a formula that contains zero definite descriptions is already an ordinary first-order formula, and obviously it is equivalent to itself.

For the inductive step, we suppose:

> For each formula A, if A contains at most n occurrences of `the`, then A is equivalent to an ordinary first-order formula.

Then we will use this assumption to show:

> For each formula A, if A contains at most $n + 1$ occurrences of `the`, then A is equivalent to an ordinary first-order formula.

Now we can show *this* generalization by induction on the complexity of formulas.

For this "inner" induction, we start with the base case:

> If A is an atomic formula, if A contains at most $n + 1$ occurrences of `the`, then A is equivalent to an ordinary first-order formula.

The idea here is that we can "extract" the definite descriptions from an atomic formula, replacing them with variables. For example, the formula

```
the y ("A" ⊕ y = "B")  ≤  "C"
```

is equivalent to

```
∃x (x = the y ("A" ⊕ y = "B")  ∧  x ≤ "C")
```

More generally, suppose that A is an atomic formula which contains at most $n + 1$ occurrences of `the`. Then A is equivalent to a formula of the form

```
∃x (x = the y B  ∧  C)
```

where B and C each contain fewer occurrences of `the` than A—that is, they each have at most n occurrences of `the`. (We could prove this fact carefully, but we won't bother.)

By the "outer" inductive hypothesis, B and C are respectively equivalent to formulas B^* and C^* with no definite descriptions. So A is equivalent to

```
∃x (x = the y B*  ∧  C*)
```

Furthermore, by applying Exercise 5.5.5, this is equivalent to

```
∃x (∀y(B* ↔ x = y) ∧ C*)
```

and this contains no definite descriptions.

Next, we consider a formula of the form $\neg A$. By the "inner" inductive hypothesis (on formula-complexity), A is equivalent to some formula A^* which contains no definite descriptions. So $\neg A$ is equivalent to $\neg A^*$, which also contains no definite descriptions.

The steps for $\wedge$ and $\forall$ go basically the same. Thus, by the "inner" inductive argument (on formula-complexity), we have shown every formula that contains at most $n + 1$ definite descriptions is equivalent to a formula with no definite descriptions. This completes the inductive step for "outer" inductive argument (on the number of definite descriptions). So we have also shown that for any number n, every formula that contains at most n definite descriptions is equivalent to one with no definite descriptions. Since for every formula A there is *some* number n such that A has at most n definite descriptions, this shows:

5.5.6 Theorem (Russell's Elimination Theorem)
Every formula in the first-order language with definite descriptions is logically equivalent to a formula in the ordinary first-order language with no definite descriptions.

In practice, then, it's often convenient to consider formulas that involve definite descriptions as *abbreviations* for their ordinary first-order translations—in basically the same way as we treat formulas that involve $\vee$ as abbreviations for formulas just containing $\neg$ and $\wedge$. The main difference is that the translation procedure for `the` is a little bit more complicated.

Here's a fact that will be particularly handy for moving back and forth between definite descriptions and ordinary first-order formulas, which follows from Russell's Elimination Theorem and Exercise 5.5.5.

5.5.7 Exercise
For every definite description term $t(x)$, there is an ordinary first-order formula $A(x, y)$ such that

$$(t(x) = y) \equiv \forall z\ (A(x, z) \leftrightarrow z = y) \tag{5.1}$$

(where z is a distinct variable from x and y). Likewise, for every first-order formula $A(x, y)$, there is a definite description term $t(x)$ such that (5.1) holds.

Chapter 6

The Inexpressible

> ‘You are sad,’ the Knight said in an anxious tone: ’let me sing you a song to comfort you.
> ‘Is it very long?’ Alice asked, for she had heard a good deal of poetry that day.
> ‘It’s long,’ said the Knight, ‘but it’s very, very beautiful. Everybody that hears me sing it—either it brings the tears into their eyes, or else—’
> ‘Or else what?’ said Alice, for the Knight had made a sudden pause.
> ‘Or else it doesn’t, you know. The name of the song is called *“Haddocks’ Eyes”*.’
> ‘Oh, that’s the name of the song, is it?’ Alice said, trying to feel interested.
> ‘No, you don’t understand,’ the Knight said, looking a little vexed. ‘That’s what the name is *called.* The name really *is “The Aged Aged Man”*.’
>
> Lewis Carroll, *Through the Looking Glass* (1871)

This chapter works up to two important results about the limits of what can be said. *Truth* in a structure cannot be described within that same structure. More generally no theory can fully describe itself. (For concreteness, we’re working with *first-order* theories, but it turns out that not very many of the main points in this chapter turn on that detail. Similar facts hold for many other kinds of precise logical theories.)

Before we can state these limits on expressibility, we’ll think about the “expressive power” of logic more generally. What kinds of things—in particular, what sets and functions—can be described within a structure? What sets and functions can a theory represent?

6.1 Definable Sets and Functions

Back in Chapter 3 we considered functions that are *simply definable* in a structure. Take the standard model of arithmetic $\mathbb{N}(0, \text{suc}, +, \cdot)$. Even though this structure doesn't include a primitive function symbol for *doubling*, even so the doubling function is definable using the complex term `x + x` (or `2 · x` , or many other choices).

Now we have a more powerful language than the simple term language: the language of first-order logic. We can use a first-order formula to describe a *set* of objects in a structure. For example, we can describe the set of even numbers using this formula

```
∃y (y + y = x)
```

In the standard model of arithmetic $\mathbb{N}$, this formula is satisfied by all and only the even numbers. Or we can describe the set of prime numbers:

```
∀y (∃z (y · z = x) → (y = 1  ∨  y = x))
```

This is a formula of one variable, `x` , which is satisfied by all and only the prime numbers. (It says, "For any number y, if x is divisible by y, then either y is 1 or y is x.")

We can also define *relations* in this structure, by picking out just the pairs which satisfy a certain formula. For example, we can define the relation "x is divisible by y":

```
∃z (y · z = x)
```

Let's abbreviate this formula `Div(x, y)` .

We can also use first-order logic to define more *functions* in a structure than we could before. Instead of just using simple terms like `x + x` to pick out functions, we can also use definite descriptions. For example, we can define the function that takes two numbers m and n to their *greatest common divisor*. (This is the largest number that *m* and *n* are both divisible by: for example, the greatest common divisor of 12 and 15 is 3). One way is using this definite description, with free variables *x* and *y* (using our abbreviation `Div(x, y)` for divisibility):

```
the z (Div(x, z)  ∧  Div(y, z)  ∧
    ∀z’ ((Div(x, z’)  ∧  Div(y, z’))  →  z’ ≤ z))
```

(“The *z* such that *x* and *y* are each divisible by *z*, and which is at least as large as any z' such that *x* and *y* are each divisible by z'.”)

For another example, consider the string structure $\mathbb{S}$. The set of *one-symbol* strings is definable in $\mathbb{S}$ using this formula:

```
x ≲ ”•”  ∧  x ≠ ””
```

This says that *x* is no longer than a one-symbol string `•`, but not empty. Abbreviate this formula `Sym(x)`. We can use this to define the function that takes each non-empty string to its first symbol. We can use this definite description (with a free variable `x`):

```
the y (Sym(y) ∧ ∃z (x = y ⊕ z))
```

That is, the first symbol of a sequence *x* is the string *y* which is one symbol long, and such that *x* consists of *y* followed by another (perhaps empty) string of symbols. Let’s call this term head(`x`). Note that if we plug the empty string in for `x` in this term, the resulting term head(`””`) is an *improper* definite description: there isn’t any *y* which you can join to the front of a string and get the empty string. So the term head(`””`) has no denotation. But that’s fine, because the first-element function is also undefined for the empty string—it is a *partial* function.

Now let’s give some more explicit definitions of these ideas.

6.1.1 Definition

Let S be an L-structure with domain D.

A formula $A(x)$ **defines** a set $X \subseteq D$ in S iff, for each $d \in D$,

$$A(x) \text{ is true of } d \text{ in } S \quad \text{iff} \quad d \in X$$

A set X is **definable** in S iff there is some formula that defines X in S.

Similarly, a set of pairs $X \subseteq D \times D$ is definable in S iff there is some first-order formula $A(x, y)$ such that, for each pair $(d_1, d_2) \in D \times D$,

$$A(x, y) \text{ is true of } (d_1, d_2) \text{ in } S \quad \text{iff} \quad (d_1, d_2) \in X$$

The same goes for sets of n-tuples of elements of D, for any number n.

A term $t(x)$ **defines** a (partial) function $f : D \to D$ in S iff

$$[\![t]\!]_S(d) = f\,d \qquad \text{for each element } d \text{ in the domain of } f$$

That is, the denotation of $t(x)$ with respect to the assignment $[x \mapsto d]$ in S is the same as the "output" of the function f for the "input" d, whenever this is defined. A (partial) function f is **definable** in S iff there is some definite description term $t(x)$ that defines f.

It's also kind of nice to rewrite this definition in different notation which make the parallels between sets and functions more obvious. Recall that the *truth-value* $[\![A]\!]_S(d)$ is True if $A(x)$ is true of d in S, and False otherwise. Similarly, recall that the *characteristic function* char X is the function that takes d to True is $d \in X$, and False otherwise. So a formula $A(x)$ defines a set X in a structure S iff,

$$[\![A]\!]_S(d) = (\text{char } X)d \quad \text{for every } d \in D$$

Something to be careful about is that different textbooks use slightly different definitions and terminology for "definable", particularly when it comes to *partial* functions. (The same goes even more so for "representable", which we will encounter in Section 6.7.) One thing to notice about this definition is that it doesn't say anything at all about what happens to the term $t(x)$ for objects d which *aren't* in the domain of f. We don't require that $t(x)$ even has a denotation in that case—but we also don't require that it *doesn't* have a denotation.

6.1.2 Exercise
Show that in the standard string structure $\mathbb{S}$, the following are definable.

(a) The partial function that takes each non-empty string to its last symbol.

(b) The set of non-empty strings.

(c) The set of pairs (s, t) such that s is an initial substring of t. (Recall, this means that t is the result of adding zero or more symbols onto the end of s.)

(d) The set of pairs (s, t) such that s is a single symbol that appears somewhere in t.

(e) The "dots" function from Exercise 2.5.10.

(f) The set of pairs of strings (s, t) such that t is twice as long as s.

We used *definite description* terms in our definition of definable functions, because this makes things a bit easier. But the logic of definite descriptions is more complicated than ordinary first-order logic, so sometimes it's handy to eliminate definite descriptions. The following facts are useful for this.

6.1.3 Exercise

Let $f : D \to D$ be a (partial) function. Then f is definable iff there is an ordinary first-order formula $A(x, y)$ (with no definite descriptions) such that, for any element d in the domain of f, the following formula is true of $(d, f d)$ in S:

$$\forall z \; (A(x, z) \leftrightarrow y = z)$$

(where z is a distinct variable from x and y).

Hint. We proved a useful fact at the end of Section 5.5.

6.1.4 Exercise

Recall that the *graph* of a function f is the set of pairs $(d, f d)$ for each element d in the domain of f. Let D be the domain of a structure S.

(a) If f is a total function from D to D, the graph of f is definable in S iff f is definable in S.

(b) If f is a partial function from D to D, then the graph of f is definable in S iff f is definable in S *and* the domain of f is definable in S.

Hint. Be careful about the "negative" case, when a pair (d_1, d_2) is *not* in the graph of f.

6.1.5 Definition

A set of numbers X is **arithmetically definable** iff X is definable in the standard model of arithmetic, $\mathbb{N}(0, \mathrm{suc}, +, \cdot)$. Similarly for sets of tuples of numbers and functions.

6.1.6 Exercise

Show that the following sets and functions are arithmetically definable.

(a) The function that takes a pair of numbers (m, n) to the remainder after dividing m by n.

(b) Any finite set of numbers.

6.1.7 Exercise

Suppose that S is an infinite structure. Show that infinitely many subsets of the domain of S are undefinable.

6.1.8 Definition

(a) Recall from Exercise 3.2.11 that every number has a label in the standard number structure $\mathbb{N}$. These are the terms `0` , `suc 0` , `suc suc 0` , and so on. The label for a number is called its **numeral**, and we use the notation $\langle n \rangle$ for the numeral which denotes the number n. This is defined recursively:

$$\begin{aligned} \langle 0 \rangle &= \texttt{0} \\ \langle \operatorname{suc} n \rangle &= \texttt{suc } \langle n \rangle \quad \text{for every } n \in \mathbb{N} \end{aligned}$$

(b) Similarly, in Exercise 3.2.13 we showed that every string has a label in the standard string structure $\mathbb{S}$. Here's one standard way of doing it. The string `ABC` is the same as

$$\texttt{A} \oplus \texttt{B} \oplus \texttt{C} \oplus ()$$

which is built up by joining together one-symbol strings and the empty string. So we can label this string with the term

```
("A" ⊕ ("B" ⊕ ("C" ⊕ "")))
```

in the language of strings. We call this the **canonical label** (or **quotation name**) for `ABC` . In general, we can define canonical labels recursively. Just like with numerals and numbers, we'll use the notation $\langle s \rangle$ for the canonical label for the string s (in the language of strings).

$$\begin{aligned} \langle () \rangle &= \texttt{""} \\ \langle a \oplus s \rangle &= \texttt{(}c \texttt{ ⊕ } \langle s \rangle\texttt{)} \quad \text{where } c \text{ is the constant for the single symbol } a \end{aligned}$$

(c) We can generalize this idea. Recall from Definition 3.2.12 that a structure S is **explicit** iff every object in the domain of S is denoted by some term.

Thus, if S is explicit, there is a **label function** that takes each object d in the domain of S to a term that denotes d, the **label** for d. We'll use the notation $\langle d \rangle$ for this as well. So we can call the label function $\langle \cdot \rangle$ (with a dot indicating where to write its argument).

6.1.9 Notation

In what follows, when I'm talking about labels, I'll sometimes hide extra brackets. Things like $A(\langle d \rangle)$ look ugly, so I'll instead write this as $A\langle d \rangle$. Similarly, instead of $A(\langle d_1 \rangle, \langle d_2 \rangle)$ I'll write the simplified version $A\langle d_1 \rangle \langle d_2 \rangle$. I'll try to put the parentheses back in when it would otherwise be confusing what something means.

6.1.10 Exercise

Let S be an explicit structure, and let $\langle \cdot \rangle$ be a labeling function for S. Let $t(x)$ be a term of one variable that defines a partial function f in S. Show that, for each element d in the domain of f,

$$t\langle d \rangle \underset{\mathrm{Th}\, S}{\equiv} \langle f d \rangle$$

6.1.11 Exercise

Let S be an explicit structure, and let $\langle \cdot \rangle$ be a labeling function for S. Let $A(x)$ be a formula of one variable that defines a set X in S. Then, for each object d in the domain of S,

$$\begin{array}{llll} \text{If} & d \in X & \text{then} & \mathrm{Th}\, S \vDash A\langle d \rangle \\ \text{If} & d \notin X & \text{then} & \mathrm{Th}\, S \vDash \neg A\langle d \rangle \end{array}$$

6.1.12 Notation

We can use a notational trick to make the analogies clearer between these facts about sets and functions. (This is cute, but entirely optional, so feel free to skip it.) For a subset X of D, and an element $d \in D$, we can define

$$\langle d \in X \rangle = \begin{cases} \top & \text{if } d \in X \\ \bot & \text{if } d \notin X \end{cases}$$

(where $\top$ is the standard truth and $\bot$ is the standard falsehood). Then we can rewrite the conclusion of the previous exercise like this:

$$A\langle d \rangle \underset{\mathrm{Th}\, S}{\equiv} \langle d \in X \rangle$$

Exercise 6.1.10 and Exercise 6.1.11 give us another way to think about definable sets and functions, when we are looking at *explicit* structures, like $\mathbb{N}$ and $\mathbb{S}$. If a formula $A(x)$ defines a set X, then the sentence $A\langle d\rangle$ is true for each d which is in X, and false for each d which is not in X. A nice thing about this alternative way of thinking about definability is that it only talks about the truth of *sentences*, rather than the extensions of formulas and terms with free variables. This will turn out to be helpful when we generalize this notion in Section 6.7.

6.2 Representing Pairs and Sequences

We can use the string structure to talk about strings. But strings of symbols are a very handy general purpose tool for representing other things as well. We can use strings to write down numbers, or formulas and terms, or computer programs, or many other things. Once we have given some things names, we can use the language of strings to talk about those things indirectly, by talking about their names. We can ask whether a certain set of things is *definable* in the string structure, by way of the set of names for those things.

For example, there are many different notation systems for numbers, such as Arabic numerals, Roman numerals, binary code, and so on. In this text we have been using a simple "unary notation," in which, for example, the number 3 is represented by the term `suc suc suc 0`. (Or more officially, `suc(suc(suc(0)))`.)

6.2.1 Exercise

The set of pairs (s, t) such that, for some number n, s is the numeral for n and t is the numeral for $\operatorname{suc} n$ is definable in the string structure $\mathbb{S}$.

In other words, the successor relation for *string representations* for numbers is definable in the string structure. This gives us a way of "talking about" this relation between numbers, entirely in the language of strings.

Once we have chosen a way of representing numbers with strings, it is natural to speak a bit informally, and say things like "the successor function is definable in $\mathbb{S}$." Of course, the successor function isn't really a function on *strings*. When we say this, what we mean is really that the function that takes the *string representations* of a number to the *string representation* of its successor is definable.

Here's another important example. In Section 4.5 we also considered how to write down *pairs* of strings. We can't just stick the two strings together—that would leave it ambiguous how to break them up. We can't just put a special symbol in between them (like a comma or newline)—the special symbol might appear in the original strings. What we *can* do is first “quote” each of the strings, by switching over to its canonical label, and then stick *those* together. For tidiness, we’ll separate the two strings with a newline. In this system, we represent the pair (`AB`, `C`) with the string

```
"AB"
"C"
```

In general, we can represent any pair of strings (s, t) with the string

$$\langle s\rangle \oplus \text{newline} \oplus \langle t\rangle$$

using the “quotation-labels” for s and t. This is the **string representation** for (s, t).

This is an unambiguous notation. But now we also want to make sure that it is a notation that the *language of strings* is expressively powerful enough to handle. For example, consider the function that takes a string representation for a pair of strings to its first element. Is this function *definable* in $\mathbb{S}$? In fact, it is—but this takes a bit of work to show.

The first step is to check that the *label* function for strings is definable. Our official definition of this function was *recursive*. Soon we’ll figure out a general way to capture recursive definitions in the language of strings. But we don’t have that yet—and that strategy will rely on what we are doing now, which means using it here would be circular. So we have to be a bit devious, and figure out a way to redescribe the label function without any recursion—just in terms of the basic operations of joining together strings and comparing lengths. This is bit of logic engineering involves a bit of hackery, and the details aren’t super important, so we’ll tuck them away in the starred section Section 6.3. For the moment, let’s take this point on faith.

6.2.2 Proposition
The function that takes each string s to its canonical label $\langle s\rangle$ is definable in $\mathbb{S}$.

Proof
See Section 6.3. □

Using this fact, we can check that our string representations for pairs of strings can be “packed” and “unpacked” within the language of strings.

6.2.3 Exercise

(a) The set of all string representations for ordered pairs of strings is definable in $\mathbb{S}$.

(b) The function that takes the string representation for the pair (s, t) to the first element s in this pair is definable in $\mathbb{S}$. Likewise, the function that takes the string representation for (s, t) to the second element t is also definable in $\mathbb{S}$.

(c) The two-place function that takes strings s and t to the string representation for the pair (s, t) is definable in $\mathbb{S}$.

Hint. Remember that the strings $\langle s \rangle$ and $\langle t \rangle$ are guaranteed not to include any newline symbols.

We now have two different ways of talking about definability for a set of pairs. One way (Definition 6.1.1) uses the extension of a formula of two variables. The other way uses the set of *string representations* for those pairs. Let's check that these two different versions match up.

6.2.4 Exercise

For any set or pairs of strings $X \subseteq \mathbb{S} \times \mathbb{S}$, the set of pairs X is definable in $\mathbb{S}$ (using a formula of two variables) iff the set of string representations for each pair in X is definable (using a formula of one variable).

We can do more with this same kind of idea. There's no reason we have to stop with just *two* strings: we can also write down any finite sequence of strings.

6.2.5 Definition

The **string representation** for a sequence of strings $(s_1, \dots, s_n)$ is the string

$$\langle s_1 \rangle \oplus \text{newline} \oplus \langle s_2 \rangle \oplus \text{newline} \oplus \cdots \oplus \text{newline} \oplus \langle s_n \rangle$$

using the canonical labels for each string in the sequence.

(We could make this definition more explicit with a recursive definition, but we won't bother.)

This is also an unambiguous notation. Using this string representation, certain important operations on sequences are definable in the language of strings.

6.2.6 Exercise
The set of all string representations for sequences of strings is definable in $\mathbb{S}$.

6.2.7 Exercise
The relation of being an element of a sequence is definable in $\mathbb{S}$. That is to say, the set of pairs (s, t) such that s is an element of the sequence of strings represented by t is definable in $\mathbb{S}$.

In what follows, let `Elem(x, y)` be a formula that defines this relation.

6.2.8 Exercise
The function that takes a sequence to its last element is definable in $\mathbb{S}$. That is, the (partial) function that takes each string that represents a sequence of strings $(s_1, \dots, s_n)$ to s_n is definable in $\mathbb{S}$.

In what follows, let `last(x)` be a term that defines this (partial) function in $\mathbb{S}$.

6.2.9 Exercise
The initial subsequence relation is definable in $\mathbb{S}$. That is, the set of pairs (s, t) such that s is the string representation for an initial subsequence of the sequence represented by t is definable in $\mathbb{S}$.

In what follows, let `Init(x, y)` be a formula that defines this set of pairs in $\mathbb{S}$.

We can use these operations on sequences to do something cool: we can capture *recursive* definitions in $\mathbb{S}$. Let's start with an example. Consider the set of terms in the language of arithmetic. This set of strings is defined recursively using the following four rules:

$$\frac{}{\texttt{0} \text{ is a term}} \qquad \frac{t \text{ is a term}}{\texttt{suc } t \text{ is a term}}$$

$$\frac{t_1 \text{ is a term} \qquad t_2 \text{ is a term}}{\texttt{(}t_1 \texttt{ + } t_2\texttt{)} \text{ is a term}} \qquad \frac{t_1 \text{ is a term} \qquad t_2 \text{ is a term}}{\texttt{(}t_1 \texttt{ ·} t_2\texttt{)} \text{ is a term}}$$

That is, every term in the language of arithmetic is the result of applying these rules some number of times. Each term can be built up step by step, where each step uses one of these rules to build up a more complicated string from earlier steps. We can describe a step-by-step process like this using a sequence of strings. We call these step-by-step sequences *derivations.*

Intuitively, a derivation spells out the results of repeatedly applying the formation rules. For example, a derivation for the term `0 + (suc 0 + 0)` is the sequence

(`0`, `suc 0`, `(suc 0 + 0)`, `(0 + suc 00)`)

In this sequence (s_1, s_2, s_3, s_4), we get s_1 from the zero rule, we get s_2 by applying `suc` to s_1, we get s_3 by applying `+` to s_1 and s_2, and finally we get s_4 by applying `+` to s_1 and s_3. Each element of the sequence is the result of applying one of the formation rules to earlier elements of the sequence.

6.2.10 Exercise

A sequence of strings $(s_1, \dots, s_N)$ is a **derivation** (for the terms in the language of arithmetic) iff for each $k \leq N$, one of the following four conditions holds:

- $s_k =$ `0`,
- For some $i < k$, $s_k =$ `suc` s_i,
- For some $i < k$ and $j < k$, $s_k =$ `(`s_i `+` s_j`)`, or
- For some $i < k$ and $j < k$, $s_k =$ `(`s_i `·`s_j`)`.

Show that for any string s, s is a term in the language of arithmetic iff s is an element of some derivation.

Hint. Use a different kind of induction for each direction of the “iff.” It may be helpful to use the following facts, which follow straightforwardly from the definition.

(a) If s and t are derivations, then $s \oplus t$ is also a derivation.

(b) Any initial subsequence of a derivation is also a derivation.

6.2.11 Exercise

The set of terms in the language of arithmetic is definable in $\mathbb{S}$.

Hint. You can define *derivation* in $\mathbb{S}$ using `Elem(x, y)` , `last(x)` , and `Init(x, y)` .

We can use the same idea to show that for any finite signature L, the set of L-terms is definable. A particularly important case is the language of strings—we will come back to this in Section 6.5.

We can also extend this same idea to capture other kinds of recursive definitions—such as recursive definitions for *functions*. The basic idea is to use ordered pairs. A function $f : \mathbb{S} \to \mathbb{S}$ is definable iff the set containing the pair $(s, f\,s)$ for each string $s \in \mathbb{S}$ is definable (Exercise 6.1.4). So we can also capture a recursive definition of a *function*, by turning it into a definition of the *set* of string representations for these ordered pairs.

6.2.12 Example

The length function is definable in $\mathbb{S}$. That is, the function that takes each string to the string representation for its length is definable in $\mathbb{S}$.

For example, this function maps

$$\begin{aligned} \texttt{•} &\mapsto \texttt{suc 0} \\ \texttt{ABC} &\mapsto \texttt{suc suc suc 0} \end{aligned}$$

Proof Sketch

The function that takes a string to the numeral for its length can be recursively defined as follows:

$$\begin{aligned} \text{len}() &= \texttt{0} \\ \text{a} \oplus \text{s}() &= \texttt{suc}\ \ \text{len}\, s \quad \text{for each symbol } a \text{ and string } s \end{aligned}$$

(This just sticks together the recursive definition of the length function from $\mathbb{S}$ to $\mathbb{N}$ and the numeral function from $\mathbb{N}$ to $\mathbb{S}$.)

A *derivation* for this recursive definition is a sequence of strings $(s_1, \ldots, s_N)$ in which each s_k is the string representation for some *ordered pair* of strings (x, y) which has one of the following two properties:

(i) $x = ()$ and $y = \texttt{0}$, or else

(ii) There is some $i < k$ such that s_i is the string representation for an ordered pair (x', y'), and

$$\begin{aligned} x &= a \oplus x' \quad \text{for some symbol } a, \\ y &= \texttt{suc}\ \ y' \end{aligned}$$

We can then check (using two simple inductive arguments) that for any strings x and y,

> len $x = y$ iff the string representation for the pair (x, y) is an element of some derivation sequence.

Then all we need to do is translate this definition of a *derivation sequence* for len into the language of strings, in the same way as in Exercise 6.2.11. □

In general, any property, relation, or function that has a *recursive* definition in terms of definable operations on strings can also be defined in the string structure $\mathbb{S}$. But the precise statement of this general claim, and its proof, is rather abstract. This is in the starred section Section 6.4. The main idea is that whenever we want to translate a recursive definition into the language of strings, we can always do it by defining the corresponding kind of derivation sequence. In Section 6.5 we will consider some other important applications of this idea.

6.3 Defining Quotation*

In Section 6.2, we skipped the proof that the *label* (or *quotation*) function, which takes each string s to a term $\langle s \rangle$ in the language of strings that stands for it, is definable in $\mathbb{S}$. In this section we'll fill that in.

We have usually been writing out terms using unofficial shortcuts, like "infix" notation for `⊕`, and dropping parentheses. For this part, though, we had better write out our official labels explicitly, since it's the *official* label function that we are trying to define. For example, the label function officially does this:

`•` ↦ `("•" ⊕ "")`

`ABC` ↦ `("A" ⊕ ("B" ⊕ ("C" ⊕ "")))`

Our original definition of this function was recursive. But looking at how it works, we can describe the pattern of what it does in a different way.

6.3.1 Exercise

For any string s, there are strings $t_1, \ldots, t_n$ and u such that the canonical label $\langle s \rangle$ has the form

$$t_1 \oplus \cdots t_n \oplus \texttt{""} \oplus u$$

where

(i) Each string t_i has the form `⊕(` c_i `,` , where c_i is the constant in the language of strings that denotes the ith symbol in s, and

(ii) u is the same length as s and contains only right parentheses `)`

Hint. This can be proved by induction, using the original recursive definition of the label function (Definition 6.1.8).

6.3.2 Exercise
Each symbol in the standard alphabet has a corresponding constant symbol in the language of strings. The relation

c is the constant for the symbol a

is definable in $\mathbb{S}$

Finally, we can use a hack. In the language of strings, all of the constants for single symbols have the same *length*: each of them is three symbols long. That means that each "chunk" t_i in the canonical label which corresponds to the ith symbol is exactly six symbols long. We can use this fact to "line up" each symbol with the relevant part of its string representation. In order to check if a string t is the canonical label for s, we just have to compare initial substrings of the string s with initial substrings of t which are *exactly six times as long.*

6.3.3 Exercise
The relation

s is exactly six times as long as t

is definable in $\mathbb{S}$.

6.3.4 Exercise
Let s be any string. Then the canonical label $\langle s \rangle$ is the unique string of the form $t \oplus \texttt{""} \oplus u$ for some strings t and u such that

(i) For each pair of an initial substring $s' \preceq s$ and an initial substring $t' \preceq t$ which is exactly six times as long as s', for any symbol a, if $s' \oplus a$ is also an initial substring of s, and c is the constant that stands for a in the language of strings, then the string $t' \oplus \texttt{(}c\texttt{,}$ is also an initial substring of t.

(ii) u has the same length as s and contains only the symbol $\texttt{)}$

6.3.5 Exercise
The function that takes each string s to its canonical label $\langle s \rangle$ is definable in $\mathbb{S}$.

6.4 Recursive Definitions*

Let's consider how we can generalize the idea from Section 6.2 to other recursive definitions.

Suppose F is a set of operations on some set A. That is, each element $f \in F$ is a function from $A^n \to A$ for some n or other. Recall from Section 2.4 that there is a (unique) F*-recursively defined* set $X \subseteq A$, which is closed under all of the F operations and which has inductive property that corresponds to F. That inductive property tells us that we can show that *every* element of X has a certain property by showing

6.4.1 Definition
Let F be a set of operations on a set A. An F**-derivation** is a sequence $(x_1, \dots, x_N)$ such that for each number $k \leq N$, there is some n-place operation $f \in F$ and some numbers $i_1, \dots, i_n < k$ such that

$$x_k = f(x_{i_1}, \dots, x_{i_n})$$

(a) For any F-derivations s and t, the joined sequence $s \oplus t$ is also an F-derivation.

(b) If s is an F-derivation, any initial subsequence of s is also an F-derivation.

6.4.2 Exercise
Let F be a set of operations on a set A. Let $X \subseteq A$ be the F-recursively defined subset of A. Then for each $a \in A$,

$a \in X$ iff a is an element of some F-derivation.

6.4.3 Exercise
Let F be a set of operations on $\mathbb{S}$. Suppose that each operation $f \in F$ is definable in $\mathbb{S}$. Then the F-recursively defined set of strings X is also definable in $\mathbb{S}$.

We can also apply this to recursively defined functions.

`TODO. Fill this in.`

6.5 Representing Language

We use language to represent the world. But language is also *part* of the world, and it is one of the things that we talk about. We don't just use words to talk about other things; we can also use words to talk about words themselves.

We have been doing this a lot: as we have been building up little formal languages, we have talked about them a lot, using ordinary English. But these little formal languages can *also* talk about language. We can look at formal languages which include names for sentences, and which include sentences that say things about names, and so on. This means we can use these formal languages as a model of what we ourselves have been doing all along, as logicians. We can turn the tools of logic onto logic itself.

In fact, we have already been working with a theory with these capabilities since early on. We represent terms and formulas of formal languages as strings of symbols. These are all elements of the domain of $\mathbb{S}$, the standard string structure. So the string structure $\mathbb{S}$ and the first-order language of strings are good tools for a *formalized theory of syntax*. The language of strings is a formal language that can describe formal languages.

Here are some important facts:

6.5.1 Lemma
Let L be a finite signature. The following sets are definable in $\mathbb{S}$.

(a) The set of L-terms.
(b) The set of L-formulas.

6.5.2 Lemma
Let L be a finite signature. The **substitution function** is the two-place function that takes an L-formula $A(x)$ and a closed L-term b to the L-sentence $A(b)$. The substitution function is definable in $\mathbb{S}$. That is to say, there is a term $\text{sub}(x, y)$ in the language of strings (with definite descriptions), such that

$$[\![\text{sub}]\!]_{\mathbb{S}}(A(x), b) = A(b)$$

for each L-formula $A(x)$ and L-term b. (Remember, formulas and terms are *strings*—so $A(x)$, b, and $A(b)$ are each elements of the domain of the string structure $\mathbb{S}$.)

Lemma 6.5.1 and Lemma 6.5.2 can both be proved using the tricks we discussed for translating *recursive* definitions into the language of strings (in Section 6.2 and Section 6.4). We won't go through these details here.

In fact, Lemma 6.5.1 and Lemma 6.5.2 follow from an even more general fact about the expressive power of the string language. It turns out that *any* operation on strings which can be systematically worked out step by step is definable in $\mathbb{S}$. (This is called the Definability Theorem, Exercise 7.7.5.) Since the substitution function is systematic in this way, the fact that it is definable in $\mathbb{S}$ will follow as one particular application of this general result.

Recall that we have also shown that each string s has a *canonical label* (or *quotation name*), a term in the language of strings that denotes s, which we call $\langle s \rangle$ (Definition 6.1.8). Since formulas and terms are strings of symbols, they have canonical labels in the language of strings. If A is any L-formula, $\langle A \rangle$ is a term in the language of strings that *denotes* A (in the string structure $\mathbb{S}$). Similarly, if t is an L-term, then $\langle t \rangle$ is a term in the language of strings that denotes the string representation for a.

We can use these canonical labels, together with Exercise 6.1.10, to describe the definability of substitution another way. For every L-formula $A(x)$ and every L-term b, the following sentence is true in $\mathbb{S}$:

$$\text{sub}\langle A(x)\rangle\langle b\rangle = \langle A(b)\rangle$$

(Or, equivalently, the ordinary first-order formula that results from eliminating definite descriptions from this sentence is true $\mathbb{S}$.)

We can describe the syntax of any language L that has finitely many primitive symbols in the first-order theory of the string structure $\mathbb{S}$. In particular, then, we can apply all of these ideas to the language of strings itself. This language only has finitely many primitives (`⊕` , `≤` , `""` , and one constant for each symbol in the standard alphabet, which is finite). Thus the language of strings is a language that can describe itself. The string language has terms that denote the very strings of symbols that we use to write that language down. In the language of strings, for each *formula* A, there is a *term* $\langle A \rangle$ that denotes A.

6.5.3 Example

Consider the formula `x = x` : in its completely official form, this is `(x=x)` . The canonical label for this in the language of strings, $\langle$`(x=x)`$\rangle$, is

```
("(" ⊕ ("x" ⊕ ("=" ⊕ ("x" ⊕ (")" ⊕ "")))))
```

6.5.4 Exercise
Use the definition of the canonical label function $\langle\cdot\rangle$ for strings (from Definition 6.1.8) to explicitly write out each of the following expressions.

(a) ⟨`¬∀x (x = 0)`⟩, which is the canonical label (in the language of strings) of the formula `¬∀x (x = 0)` (in the language of arithmetic).

(b) $\langle\langle()\rangle\rangle$, which is the canonical label of the *canonical label* of the empty string.

(c) $A\langle A(x)\rangle$, where $A(x)$ is the formula `(x = x)` .

(They are pretty long!)

Since we began talking about strings, it's been important for us to be careful about the difference between *use* and *mention*: when we are using some symbols to say something, and when we want to talk about those symbols themselves. We've used notation like `We've used notation like` to mark this distinction. Now, though, we need to be extra careful, because the formal language we are talking about—the language of strings—can *also* talk about language. *Within* this "object language" there is also a distinction between use and mention: between ways formulas and terms come up as part of the language, and ways formulas and terms come up as part of what this language is understood as being *about*.

For example, for any formula A, this sentence is true in $\mathbb{S}$:

```
⟨A⟩ ⊕ "" = ⟨A⟩
```

This says, intuitively, that joining the empty string to the formula A gives you back the same formula A. But *this* isn't even a well-formed sentence:

```
A ⊕ "" = A
```

This sticks a *formula* A in a spot where a term should be, and the result is gibberish. Intuitively, it doesn't say anything.

For another example, let's examine these first-order sentences.

```
∀x (x ⊕ ”” = x)
∀x (x ⊕ ⟨()⟩ x)
∀x (x ⊕ ⟨””⟩ = x)
```

The first sentence is true (in $\mathbb{S}$): it says that appending the empty string to the end of any string gives you the same thing back (a generalization of the fact above). The second sentence is, in fact, the very same sentence as the first. Since $\langle()\rangle$, the label for the empty string, is `””`, this notation just means to stick the string `””` in the middle of the string. The third sentence, in contrast, is false (in $\mathbb{S}$). Intuitively, it says that appending the *label* for the empty string—the length-two string `””`—to any string gives you the same thing back. If we unpack it, what the third sentence says is

```
∀x(x ⊕ (quote ⊕ quote ⊕ ””) = x)
```

This is false in $\mathbb{S}$: for example, the string `ABC””` is not the same as the string `ABC`. This kind of issue can be subtle, and it's important to get the hang of these distinctions.

For any formula A, its label $\langle A\rangle$ is some complex term. We can use this term $\langle A\rangle$ just like any other term to build up other formulas, such as this one:

```
∃x (x ≲ ⟨A⟩ ∧ ¬(⟨A⟩ ≲ x))
```

("Some string is strictly shorter than the formula A"). We can also substitute $\langle A\rangle$ into another formula $B($`x`$)$, just like any other term, to get a formula $B\langle A\rangle$. For example, suppose $B($`x`$)$ is the formula `(x = x)`, and A is the sentence `(0 = 0)`. Then the substitution instance $B\langle A\rangle$ is `(`$\langle A\rangle$`=`$\langle A\rangle$`)`. Since $\langle A\rangle$ is the term

```
”(” ⊕ ”0” ⊕ ”=” ⊕ ”0” ⊕ ”)” ⊕ ””
```

the fully spelled out sentence $B\langle A\rangle$ is this monstrosity:

```
”(” ⊕ ”0” ⊕ ”=” ⊕ ”0” ⊕ ”)” ⊕ ”” = ”(” ⊕ ”0” ⊕ ”=” ⊕ ”0” ⊕ ”)” ⊕ ””
```

(As usual, to simplify the notation, I'm leaving out some redundant parentheses.)

Furthermore, since the labels for expressions in the string language are themselves part of the string language, we can even plug formulas into themselves, in a sense.

If $A(\texttt{x})$ is a formula of one variable, we can substitute the term $\langle A(\texttt{x})\rangle$ into the formula $A(x)$, to get the formula $A\langle A(x)\rangle$. For an example in English, if $A(\texttt{x})$ is the formula `x is great`, and $\langle A(\texttt{x})\rangle$ is its quotation-name `”x is great”`, then the substitution instance $A\langle A(\texttt{x})\rangle$ is the sentence `”x is great” is great`. This is a sentence that says that a certain English formula is great.

This provides us with more examples of syntactic operations which are definable in $\mathbb{S}$.

6.5.5 Lemma

The function that takes each formula A in the language of strings to its standard label $\langle A\rangle$ is definable in $\mathbb{S}$. That is to say, there is a term label(x) in the language of strings (with definite descriptions) such that

$$[\![\text{label}]\!]_{\mathbb{S}}(A) \quad = \quad \langle A\rangle$$

for each formula A in the language of strings. (Again, recall that the formula A and the term $\langle A\rangle$ are each strings, and thus elements of the structure $\mathbb{S}$.)

Equivalently, for each formula A, the following sentence is true in $\mathbb{S}$:

$$\text{label}\langle A\rangle = \langle\langle A\rangle\rangle$$

Proof

This immediately follows from Proposition 6.2.2 and Lemma 6.5.1. □

6.5.6 Exercise

Let the **application function** be the function that takes a pair of formulas $A(x)$ and $B(x)$ to the sentence $A\langle B(x)\rangle$, which results from plugging the label for $B(x)$ into $A(x)$. The application function is definable in $\mathbb{S}$. That is to say, there is a term apply(x, y) in the language of strings (with definite descriptions) such that,

$$[\![\text{apply}]\!]_{\mathbb{S}}(A(x), B(x)) \quad = \quad A\langle B(x)\rangle$$

for any formulas $A(x)$ and $B(x)$. Equivalently, for any formulas $A(x)$ and $B(x)$, the following sentence is true in $\mathbb{S}$:

$$\text{apply}\langle A(x)\rangle\langle B(x)\rangle = \langle A\langle B(x)\rangle\rangle$$

Hint. Use Lemma 6.5.2 and Lemma 6.5.5.

Intuitively, the result of apply is a sentence which says something *about* the original formula B. This apply term gives us a systematic way to put together sentences that say things about formulas—a way of describing language in language.

6.6 Self-Reference and Paradox

Remember our old friend the *Liar sentence*, the sentence L which says `L is not true` . Is L true? If L is true, then since what L says is that L is not true, it should follow that L is not true. That's a contradiction, so it must be that L is not true. But again, since what L says is that L is not true, and L is not true, it should follow that L is true. That's a contradiction. Moreover, we derived that contradiction just using the following principles:

- There is a sentence $L =$ `L is not true`
- `L is not true` is true if and only if L is not true.

The second principle is an instance of a more general schema. Here is another famous instance:

- `Snow is white` is true iff snow is white.

On the left hand side we are *mentioning* a certain sentence. On the right hand side we are *using* that very sentence to say something about snow, rather than saying something about a sentence. In general, for any sentence A, if $\langle A \rangle$ is a label for A, then the schema says:

- $\langle A \rangle$ is true iff A

The left hand side of the biconditional uses a label for a sentence, and the right hand side uses that very sentence. This is called the **T Schema**.

For a long time, many people assumed that the problem of the Liar Paradox arose because there was something defective about "self-referential" sentences like L. In English we can say things like `this very sentence` , or `what I am saying right now` . The trick, many people thought, was to just avoid saying things like this, at least whenever we were speaking "seriously", like in mathematics. In proper official languages, there just shouldn't be any sentence like $L =$ `L is not true`. Sentences shouldn't be allowed to mention themselves.

But it turns out that this natural idea won't work: in an important sense, *self-reference is inevitable*. This follows from **Gödel's Fixed Point Theorem** (which is also known as the *Diagonal Lemma*). One caveat: what the theorem really shows is not *exactly* that there is a sentence which mentions itself, but rather that there is a sentence which is *equivalent* to one that mentions it. But this is plenty to raise the interesting problems.

Let's start with a warm-up. There is a paradox called "Grelling's Paradox" which is very similar to the Liar Paradox, but doesn't involve any self-reference. (We already discussed this paradox in Section 1.5, but now we have some logical resources that will help us make it a little more precise.) Instead of self-reference, we can use *self-application*.

Instead of just asking which sentences are true, we can ask what a one-variable formula is true *of*. A formula $A(x)$ is true of an object d iff $A\langle d\rangle$ is true, where $\langle d\rangle$ is a name for d. For example,

- `x is a city` is true of Los Angeles iff `Los Angeles is a city` is true.

Notice that this relies on the fact that `Los Angeles` is a name for the city of Los Angeles.

Now, let $H(x)$ be the English formula `x is not true of x`. What happens when we apply this formula to itself?

- $H(x)$ is true of $H(x)$ iff `H(x) is not true of H(x)` is true.

Notice that this relies on the fact that `H(x)` is a name for the formula $H(x)$. But then, using the T-schema, it's easy to derive a contradiction.

We can use the first-order language of strings to formalize Grelling's paradox. In this language, we can apply formulas to formulas. Given any formula $A(x)$ and any other formula $B(x)$, we can apply $A(x)$ to $B(x)$ to get a sentence $A\langle B(x)\rangle$. As we discussed in the previous section, this syntactic operation is definable in the string structure $\mathbb{S}$. There is a term $\text{apply}(x, y)$ in the language of strings such that

$$\llbracket\text{apply}\rrbracket(A(x), B(x)) \quad = \quad A\langle B(x)\rangle$$

in the string structure $\mathbb{S}$. In particular, we can consider *self-application*—which is commonly called *diagonalization*. Let $\text{diag}(x)$ be the term $\text{apply}(x, x)$. Then

$$\llbracket\text{diag}\rrbracket(A(x)) \quad = \quad A\langle A(x)\rangle$$

Now suppose we also had a formula True(x) that only applies to the true sentences (in $\mathbb{S}$). In that case, we could formalize `x is not true of x` as

$$\neg \text{True}(\text{diag}(x))$$

Call this formula $H(x)$. Intuitively, this says, "The result of applying x to itself is not true," just like the informal Grelling formula. So again we can ask: what happens when we apply this formula $H(x)$ to itself?

Working out the substitution instance, we have:

$$H\langle H(x)\rangle \quad \text{is} \quad \neg \text{True}(\text{diag}\langle H(x)\rangle)$$

Furthermore, since the term $\langle H(x)\rangle$ denotes the formula $H(x)$, the term diag$\langle H(x)\rangle$ denotes $H\langle H(x)\rangle$. So the right-hand-side is true in $\mathbb{S}$ iff $\neg$True(x) is true of $H\langle H(x)\rangle$. But that means that $H\langle H(x)\rangle$ is true if and only if $H\langle H(x)\rangle$ is *not* true! This sentence $H\langle H(x)\rangle$ is just as bad as the self-referential Liar sentence L.

The tricky sentence $H\langle H(x)\rangle$ is a "fixed point" of the predicate $\neg$True(x). But most of this reasoning (all of it except the very last step) doesn't depend on anything special about *truth*. We can use the same idea to prove a more general, quite beautiful theorem.

6.6.1 Exercise (Gödel's Fixed Point Theorem (the Diagonal Lemma) Version 1) Let $F(x)$ be any formula in the language of strings. Then there is some first-order sentence A in the language of strings such that

$$A \text{ is true in } \mathbb{S} \quad \text{iff} \quad F(x) \text{ is true of } A \text{ in } \mathbb{S}$$

Or in more concise notation,

$$[\![A]\!]_{\mathbb{S}} = [\![F]\!]_{\mathbb{S}}(A)$$

(Recall that $[\![A]\!]_{\mathbb{S}}$ is the truth-value of A in $\mathbb{S}$.)

(Here's a little detail. The proof is easiest if we help ourselves to definite descriptions, and you should feel free to use them; that lets us represent diagonalization with a *term* diag(x). But we don't really need definite descriptions, because we have Russell's Elimination Theorem. In particular, the formula F(diag(x)), which uses definite descriptions, is logically equivalent to some formula $H(x)$ *without* any definite descriptions.)

6.6.2 Exercise (Tarski's Theorem Version 1)
The set of sentences which are true in the standard string structure $\mathbb{S}$ is not definable in $\mathbb{S}$.

In other words, *truth is undefinable.* The property of being a true sentence in the string structure is not expressible by any first-order formula in the language of strings.

These are our first, prelimary versions of Gödel's Fixed Point Theorem and Tarski's Theorem. Next we'll generalize these ideas.

6.7 Representing Sets and Functions in a Theory

Definability lets us pick out sets and functions in a particular structure. Effectively, we are using *all* of the truths in that structure in order to pin down facts about particular sets and functions. It is also useful to generalize this idea. We might want to see what we can pin down using just *some* of the facts. One reason this is important is that picking out *all* of the truths in a structure can be very difficult in practice, while picking out just a few useful truths is much easier.

Here's an example. We've been focusing mainly on definability in the string structure $\mathbb{S}$. As we'll see in Section 7.7, there is a sense in which the true statements in this structure are intractably complicated. But it turns out that we can do a lot with a lot less. We can consider some simple axioms that don't pick out all of the truths about sequences, but do pick out enough of them for many purposes. For instance, there are still enough facts there to describe operations on numbers, sequences, and syntax. Not only can the full *structure* $\mathbb{S}$ *define* these operations, but there is a much simpler *theory* of strings that can *represent* these operations.

Remember that if a set X is definable in $\mathbb{S}$, this means that there is a first-order formula $A(x)$ in the language of strings such that, for each string s, if s is in X then $A(x)$ is true of s, and if s is not in X then $\neg A(x)$ is true of s. Remember also that every string s has a *canonical label*: a term that denotes s (Definition 6.1.8). So here is another way of saying that X is definable (using Exercise 6.1.11): there is a formula $A(x)$ such that, for every string s,

$$\begin{array}{llll} \text{If} & s \in X & \text{then} & \text{Th}\,\mathbb{S} \vDash A\langle s\rangle \\ \text{If} & s \notin X & \text{then} & \text{Th}\,\mathbb{S} \vDash \neg A\langle s\rangle \end{array}$$

Likewise, if a function f is definable in $\mathbb{S}$, this means there is a term $t(x)$ (possibly using definite descriptions) such that for every string s in the domain of f,

$$\mathrm{Th}\,\mathbb{S} \vDash t\langle s\rangle = \langle f\,s\rangle$$

This way of putting things suggests a natural way of generalizing the idea of definability. Instead of using the full theory of strings Th $\mathbb{S}$, we can try to do something similiar with some simpler theory T. We say that T *represents* a set of strings X iff there is some formula $A(x)$ such that, for every string s,

$$\begin{array}{llll} \text{If} & s \in X & \text{then} & T \vDash A\langle s\rangle \\ \text{If} & s \notin X & \text{then} & T \vDash \neg A\langle s\rangle \end{array}$$

Similarly, we say that T *represents* a function f iff there is a term $t(x)$ such that, for every string s,

$$T \vDash t\langle s\rangle = \langle f\,s\rangle$$

Or in other words,

$$t\langle s\rangle \underset{T}{\equiv} \langle f\,s\rangle$$

In Section 5.4 we introduced the *minimal theory of strings* S (Definition 5.4.3). This is a finitely axiomatized theory that includes some important basic facts about how strings are put together. Let's look at a simple example of how a set can be represented in S. For this example, we just need to recall that S includes an axiom of this form,

```
∀x ∀y (Sym(x) → x ⊕ y ≠ "")
```

where `Sym(x)` is an abbreviation for the long formula

```
x = "A"  ∨  x = "B" ∨ \cdots
```

which lists out all of the constants for one-symbol strings. In particular, for each such constant c, we have a theorem of S of the form

```
∀y (c ⊕ y ≠ "")
```

6.7.1 Example

The minimal theory of strings S represents the set of all non-empty strings.

Proof
We can use the obvious formula

```
x ≠ ""
```

Call this formula $A(\texttt{x})$, and suppose s is any string. We need to show two things.

First:

$$\text{If } s \text{ is non-empty then } \mathsf{S} \vDash A\langle s\rangle$$

If s is non-empty, then $s = \text{cons}(a, t)$ for some string t and some symbol a in the standard alphabet, and so $\langle s\rangle$ is the term `(`c `⊕` $\langle t\rangle$`)`, where c is the constant for the symbol a. So $A\langle s\rangle$ is the formula

```
c ⊕ ⟨t⟩ ≠ ""
```

This immediately follows by universal instantiation from the theorem of S:

```
∀y (c ⊕ y ≠ "")
```

Second:

$$\text{If } s \text{ is empty then } \mathsf{S} \vDash \neg A\langle s\rangle$$

This is true because s is empty, then the label $\langle s\rangle$ is the term `""`. So $\neg A\langle s\rangle$ is the formula `¬¬("" = "")`, which is a logical truth—and thus of course it is a logical consequence of S.

So what we've shown is that if X is the set of non-empty strings, then for any $s \in X$, the theory S implies $A\langle s\rangle$, and for any string $s \notin X$, S implies $\neg A\langle s\rangle$. This is the sense in which S represents X. □

We don't need the *whole* theory of strings to get these consequences about particular strings being non-empty. Just a little bit of this theory is plenty to work with. Now let's state a more general definition of what it means for a theory to represent a set.

First, recall that in Definition 6.1.8 we gave a definition of a labeling function for a particular explicit structure. But for a *theory* to represent a set or a function, we don't have to be tied down to any particular choice of structure. So we can generalize that definition a bit.

6.7.2 Definition

A **labeling** for a set D in a language L is a one-to-one function $\langle\cdot\rangle$ that takes each object $d \in D$ to some L-term $\langle d\rangle$.

6.7.3 Definition

Let T be a theory in a language L, and let $\langle\cdot\rangle$ be a labeling for D in L. Let X be a subset of D, and let $A(x)$ be a formula of one variable. Then $A(x)$ **represents** X **in** T (with respect to $\langle\cdot\rangle$) iff, for each $d \in D$,

$$\begin{array}{llll} \text{If} & d \in X & \text{then} & T \vDash A\langle d\rangle \\ \text{If} & d \notin X & \text{then} & T \vDash \neg A\langle d\rangle \end{array}$$

(We almost always leave out the explicit reference to the labeling function, because it should be clear in context which one we mean.)

Using the special notation we introduced at the end of Section 6.1, we can put this more succinctly:

$$A\langle d\rangle \underset{T}{\equiv} \langle d \in X\rangle$$

Similarly, if $X \subseteq D^2$ is a set of pairs, then a formula $A(x, y)$ represents X in T iff, for each d_1 and d_2 in D,

$$\begin{array}{llll} \text{If} & (d_1, d_2) \in X & \text{then} & T \vDash A\langle d_1\rangle\langle d_2\rangle \\ \text{If} & (d_1, d_2) \notin X & \text{then} & T \vDash \neg A\langle d_1\rangle\langle d_2\rangle \end{array}$$

Or more succinctly:

$$A\langle d_1\rangle\langle d_2\rangle \underset{T}{\equiv} \langle (d_1, d_2) \in X\rangle$$

A set X is **representable in** T (or T **represents** X) iff there is some L-formula that represents X in T.

6.7.4 Exercise

What should the definition be for a representable *function* from D to D (in a theory T with a labeling for D)?

6.7.5 Proposition

If X is any set of strings, then X is definable in $\mathbb{S}$ iff X is representable in Th $\mathbb{S}$.

Proof

This follows from the definition of a representable set, using Exercise 6.1.11. □

6.7.6 Example

Consider the function that appends a stroke to the beginning of a string:

$$\text{delimit } x = | \oplus x$$

This function is representable in the minimal theory of strings S. The term that represents this function is the obvious one: `"|" ⊕ x`. To show that this really does represent the function in question, what we need to show is that for any string s,

$$\texttt{"|"} \oplus \langle s \rangle = \langle | \oplus s \rangle$$

is a theorem of S. In fact, the right-hand side $\langle | \oplus s \rangle$ is defined to be `"|" ⊕` $\langle s \rangle$ (because `"|"` is the constant for the symbol $|$). So this identity sentence is a logical truth of the form $a{=}a$. So of course it is a logical consequence of the axioms of S.

6.7.7 Exercise

Let T be a theory with a labeling for D. If X and Y are subsets of D which are each representable in T, then the following sets are also representable in T:

(a) The union $X \cup Y$.

(b) The intersection $X \cap Y$.

(c) The difference $X - Y$.

6.7.8 Exercise

Suppose a theory T' *extends* T: that is, the set of theorems of T is a subset of the set of theorems of T'. If T represents X, then T' represents X.

In Section 6.5, we discussed (but did not prove) the fact that some important syntactic operations are definable in the standard string structure $\mathbb{S}$. The string structure can describe how to substitute terms into formulas, and in particular it can describe how labels for *formulas* can be plugged into formulas in the language of strings. As it turns out, the minimal theory of strings S can do this, too. This theory includes enough information to *represent* these basic syntactic operations.

The reason this is true is that, as it turns out, the minimal theory of strings includes every truth-in-$\mathbb{S}$ which is *syntactically simple enough*. Section 6.8 explains what this means in more detail, and walks through the details of how to prove it. The important upshot is that any set or function that is definable in $\mathbb{S}$ using a *simple enough* expression can also be represented in the minimal theory of strings S. (The officially stated version of this fact is #ex:sigma-one-definable.) This includes the expressions we use to define syntactic operations. It also includes enough formulas to define lots of other interesting operations, as we will discuss in Section 7.7.

For the moment, we will take the following facts on faith:

(a) The substitution function, which takes a formula $A(x)$ and a term b to the sentence $A(b)$, is representable in S.

(b) The label function, which takes a formula A to its canonical label $\langle A\rangle$, is representable in S.

Thus:

(c) The application function, which takes a formula $A(x)$ and a formula B to the sentence $A\langle B\rangle$, is representable in S.

Representing application turns out to be especially important. So it will be helpful later on to have concise way of referring to this property of S. Let's restate it:

6.7.9 Definition

Let L be a finite signature, and let T be an L-theory. Suppose that there is a labeling of the L-formulas of one variable in L. That is, for each L-formula $A(x)$, there is a corresponding closed term $\langle A(x)\rangle$. Suppose there is a also a term $\text{apply}(x, y)$ in Def L, such that, for any L-formulas $A(x)$ and $B(x)$,

$$\text{apply}\langle A(x)\rangle\langle B(x)\rangle \underset{T}{\equiv} \langle A\langle B(x)\rangle\rangle$$

In this case we say that T **represents syntax**.

So this is another way of stating the main fact that we are taking on faith, for now:

6.7.10 Theorem

The minimal theory of strings S represents syntax.

For a theory to represent syntax, it needs three things. First, it needs "quotation terms": canonical labels for the formulas in its language. Second, it needs an "apply" term. Third, the theory needs to be *strong enough* to imply certain sentences involving those terms: the identity sentences

$$\text{apply}\langle A(x)\rangle\langle B(x)\rangle = \langle A\langle B(x)\rangle\rangle \tag{*}$$

(Or equivalently, the theory needs to be strong enough to imply the ordinary first-order formulas that result from eliminating definite descriptions from these identity sentences. That is, there needs to be a formula Apply(x, y, z) such that each sentence of the form

$$\forall z\ (\text{Apply}(\langle A(x)\rangle, \langle B(x)\rangle, z) \leftrightarrow (z = \langle A\langle B(x)\rangle\rangle))$$

is a theorem of T.)

Again, we haven't proved Theorem 6.7.10 yet, though in principle there's nothing stopping us. (It's a matter of writing out the complicated term that represents application, and verifying that each of the identity sentences given by (*) are theorems of S.) Rather than giving a proof now, we'll wait until we prove the more general Representability Theorem in Section 6.7.

We can use this to prove powerful generalizations of the theorems from Section 6.6. If we have a term apply(x, y) that represents application in a theory T, then as before we can let diag(x) be the term apply(x, x). So

$$\text{diag}\langle A(x)\rangle \underset{T}{\equiv} \langle A\langle A(x)\rangle\rangle$$

Then we can plug this into a formula in order to come up with a "self-referential" sentence, basically the same way as before.

6.7.11 Exercise (Gödel's Fixed Point Theorem Version 2)
Suppose that T is a theory that represents syntax. Let $F(x)$ be any formula. Then there is some first-order sentence A such that

$$A \underset{T}{\equiv} F\langle A\rangle$$

6.7.12 Exercise (Tarski's Theorem Version 2)
Suppose T represents syntax. Let True(x) be a formula, and suppose that for

each sentence A,

$$\mathrm{True}\langle A\rangle \underset{T}{\equiv} A$$

Then T is inconsistent.

6.7.13 Exercise (Tarski's Theorem Version 3)
Suppose T represents syntax, and suppose furthermore that T is representable in T. Then T is inconsistent.

Notice that our earlier version of Tarski's Theorem, the undefinability of truth-in-the-string-structure (Exercise 6.6.2), follows from this generalized version. We can let T be Th $\mathbb{S}$, the set of all sentences which are true in $\mathbb{S}$. This theory is consistent (it has $\mathbb{S}$ as a model). It also represents syntax (as we discussed in Section 6.6). So Th $\mathbb{S}$ must not be representable in Th $\mathbb{S}$, which means that Th $\mathbb{S}$ is not definable in the string structure $\mathbb{S}$.

Before we move on, here's one more "bonus" version of the Fixed Point Theorem. It allows you to more directly formalize statements like

L = `L is not true`

in the language of strings—with definite descriptions.

6.7.14 Exercise (Gödel's Fixed Point Theorem Version 3)
Suppose that T is a theory that represents syntax. Let $F(x)$ be any formula. Then there is some definite description *term* a such that

$$a \underset{T}{\equiv} \langle F(a)\rangle$$

Hint. You can use the same formula $H(x)$ as before. This time, though, you want to come up with a *term* that represents the result of "diagonalizing" $H(x)$, instead of a sentence.

Unlike the other versions of the Fixed Point Theorem, the use of definite descriptions can't easily be eliminated from this version. Without definite descriptions (or something else that does a similar job), we might not have enough expressive power in our *terms* to come up with a "self-referential" term a.

Notice that Version 3 of the Fixed Point Theorem implies Version 2, as well. If you have a term a which is equivalent to $\langle F(a)\rangle$ in T, then $F(a)$ is equivalent to $F\langle F(a)\rangle$ in T. But the converse doesn't hold: the sentence version does not imply the term version. You might think that if G is a fixed-point sentence, then $\langle G\rangle$ is a fixed-point term. But that doesn't follow. Just because G is *equivalent* to $F\langle G\rangle$ in T—that is, $G \leftrightarrow F\langle G\rangle$ is a theorem of T—that doesn't mean that the *identity* $\langle G\rangle = \langle F\langle G\rangle\rangle$ is a theorem of T. Different sentences can be logically equivalent to each other. So the term version of the Fixed Point Theorem is going a bit beyond the sentence version. In a language *without* definite descriptions, you might not have a fixed point term, even if you do have a fixed point sentence.

6.8 What the Minimal Theory of Strings Can Represent*

The full theory of strings (Th $\mathbb{S}$) is very complicated, and the G theory of strings (S) is relatively simple—it has just finitely many axioms. (In Chapter 7 and Chapter 8 we will be able to say more about the ways in which the minimal theory of strings is "simpler.") Even so, it turns out that the minimal theory of strings can do a whole lot of what the full-fledged theory of strings can. It is expressively pretty powerful. So lots of interesting things that can be *defined* in the standard string structure can also be *represented* in the minimal theory of strings. This includes the syntactic operations that Gödel's Fixed Point Theorem relies on—but that's just one example. In this section we will examine how this works in general.

There are lots of truths about strings which are not theorems of the minimal theory of strings S. But as it turns out, the minimal theory of strings includes all of the truths about strings which are *syntactically simple enough.* For example, S includes every truth about strings that can be expressed in a sentence (in the language of strings) without any *quantifiers*. (Later in this section we will precisely say what we mean by "syntactically simple enough".)

If a set of strings X is definable in the string structure $\mathbb{S}$, this means that there is a certain formula $A(x)$ which is true of each string which is in X, and false of each string which is not in X. Equivalently, $A\langle s\rangle$ is true in $\mathbb{S}$ for each string $s \in X$, and $\neg A\langle s\rangle$ is true in $\mathbb{S}$ for each string $s \notin X$. This means that if the formula $A(x)$ is *syntactically simple enough*, then these sentences are each theorems of the minimal theory of strings, as well. So this tells us that any set which is definable in the standard string structure *using a syntactically simple enough formula* is also representable in the minimal theory of strings. Something similar goes for representing functions, as well.

For now, the most important application of these facts is about representing syntax. The syntactic operations of substitution and labeling are definable in the standard string structure. But if we pay attention to how we do it, we can show that they are definable using *syntactically simple* formulas. (To use the technical term we will define later in this section, they are Σ_1-*definable*.) This tells us the main fact we took on faith in Section 6.7: the minimal theory of strings represents syntax. In Section 7.7 there will be another very important application, which generalizes this. It turns out that *any* operation on strings that can be systematically computed step by step is representable in the minimal theory of strings.

How can we show that S includes all the "syntactically simple enough" truths about strings? Let's start small. We'll start by showing that the minimal theory S knows enough to "unpack" each closed term in the language of strings. Remember that a term in this language is either `""`, a constant for some single symbol a, or else a term (t_1 `⊕` t_2) for some terms t_1 and t_2. Remember also that each string s has a canonical label $\langle s \rangle$ (Definition 6.1.8). For example, the canonical label for `ABC` is the term

```
"A" ⊕ ("B" ⊕ ("C" ⊕ ""))
```

We can repeatedly apply the axioms of S to convert an arbitrary term to its canonical form. In particular, S has these axioms for joining strings:

```
"" ⊕ x = x
x ⊕ "" = x
(x ⊕ y) ⊕ z = x ⊕ (y ⊕ z)
```

We can apply these axioms to "normalize" the term `("A" ⊕ "") ⊕ "B"`. Using the last of these axioms, instantiating x with `"A"`, y with `""` and z with `"B"`):

```
("A" ⊕ "") ⊕ "B" = "A" ⊕ ("" ⊕ "B")
```

Then, since `"" ⊕ "B" = "B"` and `"B" ⊕ ""` follow from the first and second axioms, by Leibniz's Law we have this as a theorem of S:

```
("A" ⊕ "") ⊕ "B" = "A" ⊕ ("B" ⊕ "")
```

The right-hand term is the canonical label for the string ⟨`ABC`⟩, which is the string denoted by the left-hand term.

6.8.1 Exercise

(a) Let s_1 and s_2 be strings. Show by induction on the length of s_1 that

$$\langle s_1 \rangle \oplus \langle s_2 \rangle = \langle s_1 \oplus s_2 \rangle$$

is a theorem of S.

(b) Let t be any term in the language of strings, and let $s = [\![t]\!]_{\mathbb{S}}$ be the denotation of t in $\mathbb{S}$. Show by induction on the structure of the term t that

$$t = \langle s \rangle$$

is a theorem of S.

(c) Let t_1 and t_2 be any terms in the language of strings. If

$$t_1 = t_2$$

is true in $\mathbb{S}$, then it is a theorem of S.

We can also show similar things about *distinctness*. We have these axioms of S (which correspond to the Injective Property for strings).

```
Sym(x)  → x ⊕ y ≠ ""
Sym(x₁) → Sym(x₂)  →  x₁ ⊕ y₁ = x₂ ⊕ y₂  →  (x₁ = x₂ ∧ y₁ = y₂)
```

And for each pair of *distinct* constants c_1 and c_2 for single symbols, we have an axiom:

$$c_1 \neq c_2$$

We can use these axioms to show the following. It follows that S has these theorems, for any distinct single-symbol constants c_1 and c_2:

$$c_1 \oplus \texttt{x} \neq \texttt{""}$$
$$c_1 \oplus \texttt{x} \neq c_2 \oplus \texttt{y}$$

6.8.2 Exercise

(a) Show by induction that for any string s_1, if s_2 is a distinct string from s_1, then

```
⟨s₁⟩ ≠ ⟨s₂⟩
```

is a theorem of S.

(b) Let t_1 and t_2 be any terms in the language of strings. If

```
t₁ ≠ t₂
```

is true in $\mathbb{S}$, then it is a theorem of S.

We can do similar things with our other basic kind of formulas. The theory S also has some axioms that say how the “no-longer-than” relation should work:

```
"" ≤ x
x ≤ ""  ↔  x = ""
Sym(x₁) → Sym(x₂)  →  x₁ ⊕ y₁ ≤ x₂ ⊕ y₂  ↔  y₁ ≤ y₂
```

6.8.3 Exercise

Show by induction that for any strings s_1 and s_2:

(a) If s_1 is no longer than s_2, then

```
⟨s₁⟩ ≤ ⟨s₂⟩
```

is a theorem of S.

(b) If s_1 is longer than s_2, then

```
¬(⟨s₁⟩ ≤ ⟨s₂⟩)
```

is a theorem of S.

(c) Let t_1 and t_2 be terms in the language of strings. If the sentence

```
t₁ ≤ t₂
```

is true in $\mathbb{S}$, then it is a theorem of S. If it is false in $\mathbb{S}$, then

$$\neg(t_1 \lesssim t_2)$$

is a theorem of S.

This shows that the minimal theory S "knows" the truth-value of every basic sentence in the language of strings, which is either an identity sentence or a "no-longer-than" sentence. Next we can extend this to slightly more complex sentences, which also use the propositional connectives `¬` and `∧`.

6.8.4 Exercise

Let A be any *quantifier-free* sentence in the language of strings: that is, A is built up using just identity sentences, relational sentences (using `≲`), negation, and conjunction. If A is true in $\mathbb{S}$, then $\mathsf{S} \vDash A$, and if A is false in $\mathbb{S}$, then $\mathsf{S} \vDash \neg A$.

Hint. Use induction.

This means S knows the truth-value of every sentence in the first-order language of strings that doesn't use any quantifiers. But we'll need more than this—the formulas we used to define computable functions use quantifiers, too. It would be natural to try adding the quantifiers back in as well—but in fact, this won't work. There are some sentences using quantifiers that are true in $\mathbb{S}$, but are not theorems of S. (We won't prove this now, but it will turn out to be a consequence of Gödel's First Incompleteness Theorem, Exercise 8.5.6.)

But not all sentences using quantifiers are out of reach. For example, consider this sentence:

```
∀x ((x ≲ "••") → ((x ≲ "A") ∨ ("BB" ≲ x)))
```

This uses a universal quantifier. But the quantifier is *restricted* to just the strings of length at most two. So, effectively, instead of quantifying over the infinite domain of all strings, this sentence only "cares about" those finitely many strings which are no longer than `••`. It turns out that the minimal theory S can handle sentences like this just fine. The trick is that, since there are only finitely many different strings of length at most two, we can list them all out (though it's a long finite list, because our alphabet is large):

$$s_1, s_2, \dots, s_n$$

Then, if we abbreviate the right-hand side `((x ≲ "A") ∨ ("BB" ≲ x))` as $A(x)$, we can rewrite the quantified sentence as a long conjunction, like this:

$$A\langle s_1\rangle \wedge A\langle s_2\rangle \wedge \ldots \wedge A\langle s_n\rangle$$

The quantified sentence is true in $\mathbb{S}$ if and only if this long conjunction is true in $\mathbb{S}$. Furthermore, we can show that S “knows” this equivalence. And since the conjunction doesn’t have any quantifiers, we have already shown that S knows its truth-value, too. Thus this particular quantified sentence is also a theorem of S.

In general, we can use this idea to show that any sentence that uses only *bounded* quantifiers is still within the ken of the minimal theory S.

6.8.5 Definition

Let t be a term, let A be a formula, and let x be a variable. Let

$$\forall (x \leq t)\ A$$

abbreviate the **bounded universal generalization**

$$\forall x\ (x \leq t \to A)$$

Similarly,

$$\exists (x \leq t)\ A$$

abbreviates the **bounded existential generalization**

$$\exists x\ ((x \leq t) \wedge A)$$

Call a formula in the language in the language of strings (without definite descriptions) **bounded** iff it is built up just using identity formulas, length formulas, conjunction, negation, and *bounded* universal quantification.[1]

Here is the final axiom of the minimal theory S.

```
x = ""  ∨  ∃y ∃z (Sym(y)  ∧  x = y ⊕ z)
```

We can use this, along with things we have already shown about what S knows about the no-longer-than relation, to show the following.

[1] Other standard names for bounded formulas include Δ_0-formulas, Σ_0-formulas, and Π_0-formulas.

6.8.6 Exercise

(a) Let s be any string, and let $s_1, \dots, s_n$ be *all* of the strings which are no longer than s. Prove by induction on s that

$$\forall x\ (x \leq \langle s \rangle \leftrightarrow (x = \langle s_1 \rangle \vee \cdots \vee x = \langle s_n \rangle))$$

is a theorem of S.

(b) Let t be a term, and let $A(x)$ be a quantifier-free formula of one variable x. There is a quantifier-free formula B such that

$$B \leftrightarrow \forall (x \leq t)\ A(x)$$

is a theorem of S.

6.8.7 Exercise

Let A be any bounded sentence. If A is true in $\mathbb{S}$, then $\mathsf{S} \vDash A$, and if A is false in $\mathbb{S}$, then $\mathsf{S} \vDash \neg A$.

Finally, we can go one step further, by adding some *unbounded* quantifiers. But this time we can't do quite as much. We can only add *existential* quantifiers, we can only do it once, and we only get half as strong a conclusion. So far, we have shown that S knows the truth-value of every bounded sentence. But for the final step, we will only get one direction: for each of these slightly more complicated sentences, if it is *true*, then S knows it is true—but if it is *false*, then S might not know it. (That's why we can't use this result to keep building up to even *more* complicated sentences. We have reached a limit.)

6.8.8 Exercise

Suppose $A(x)$ is a bounded formula. If

$$\exists x\ \ A(x)$$

is true in $\mathbb{S}$, then it is a theorem of S.

It's helpful to have a word for formulas which are slightly more complicated than bounded formulas in this way.

6.8.9 Definition

A formula is Σ_1 (pronounced "sigma-one") iff it has the form $\exists x\ A$, for some bounded formula A. That is, a Σ_1 formula is a bounded formula with an unrestricted existential quantifier in front.[2]

So in other words, what Exercise 6.8.8 tells us is that, if A is Σ_1, and A is true in $\mathbb{S}$, then A is a theorem of S. But, to reiterate, in general if A is *false* in $\mathbb{S}$, we *don't* know that $\neg A$ is a theorem of S.

Intuitively, if there *is* an example of something that satisfies a bounded formula $B(x)$, then eventually S can find it, by plugging away through the structure of individual strings. But if there is *no* example of something that satisfies $B(x)$, then no matter how long you plug away finding consequences of S, you may never succeed in "proving the negative."[3]

6.8.10 Exercise

Say a formula is Σ_1**-equivalent** iff it has the same extension in $\mathbb{S}$ as some Σ_1-formula. If A and B are Σ_1 formulas, and t is a term, then the following are Σ_1-equivalent formulas.

(a) $A \vee B$

(b) $A \wedge B$

(c) $\exists x\ A$

(d) $\forall (x \leq t)\ A$

6.8.11 Exercise

(a) Let X be a set of strings which is definable in $\mathbb{S}$ using a bounded formula.

[2] The Greek letter capital sigma is often used to represent existential quantification, and the subscript one indicates that we have just used existential quantification once.

The Greek letter capital pi Π is often used to represent universal quantification. So similarly, a Π_1-formula is a bounded formula with an unbounded universal quantifier in front.

This is just the beginning of a hierarchy of more and more complex formulas. A Σ_2-formula is what you get by adding an existential quantifier to a Π_1 formula. A Π_2-formula is what you get by adding a universal quantifier to a Σ_1-formula. And you can go on this way to recursively define Σ_n and Π_n formulas for every number n. Every formula is equivalent (in S) to something that shows up at some stage in this hierarchy. This gives us a useful general notion of a formula's "quantificational complexity".

[3] In Section 7.6 we will discuss a very closely analogous distinction, between *decidable* sets and *semi-decidable* sets. In a sense, the bounded sentences are "decidable in S", while the Σ_1 sentences are only "semi-decidable in S". (But this is an alternative sense of "decidable" and "semi-decidable" that has to do with *logical consequences*, rather than programs.)

That is, there is a bounded formula $A(x)$ which is true of each string in X, and false of each string not in X (in the structure $\mathbb{S}$). Then $A(x)$ also represents X in S.

(b) Let X be a set of strings which is definable in $\mathbb{S}$ using a Σ_1-formula $A(x)$. Then $A(x)$ also represents X in S "in one direction". That is, for each string $s \in X$, $A\langle s\rangle$ is a theorem of S, and for each string $s \notin X$, $A\langle s\rangle$ is not a theorem of S.

(Similar facts hold for sets of n-tuples and formulas of n variables, but there is no need to show this separately.)

We'll also need to show some related things about representable *functions*, rather than sets.

6.8.12 Definition

Let f be a partial function from strings to strings. Say that f is Σ_1-**definable** iff there is a Σ_1 formula $A(x, y)$ such that, for each string s in the domain of f, $f\,s$ is the unique string such that $A(x, y)$ is true of $(s, f\,s)$ in $\mathbb{S}$.

We'll need to use one more axiom of S:

```
x ≤ y  ∨  y ≤ x
```

6.8.13 Exercise

Let A(x) be the formula

```
B(x) ∧ ∀(x' ≤ x)(B(x') → x = x')
```

Then

```
∀x ∀y (A(x) → A(y) → x = y)
```

is a theorem of S.

6.8.14 Exercise

If f is Σ_1-definable, then f is representable in S.

Hint. This is a tricky problem. To show that f is representable in S, it's enough to show that there is a formula $A(x, y)$ such that, for each string s, $A\langle s\rangle\langle f\,s\rangle$ and

$$\forall y\ \forall z\ (A(\langle s\rangle,\ y) \to A(\langle s\rangle,\ z) \to y = z)$$

are both theorems of S. But the simplest strategy for showing this doesn't work: we *can't* just let $A(x, y)$ be the same Σ_1-formula that defines f. If we did that, the uniqueness condition wouldn't be a Σ_1 formula (it has unbounded universal quantifiers in the front), and so there is no guarantee that it is a theorem of S. We have to use a different formula $A(x, y)$, instead.

If f is Σ_1-definable, this means that there is a bounded formula $B(x, y, z)$ such that, for each s, $f\,s$ is the unique value such that $(s, f\,s)$ satisfies $\exists z\ B(x, y, z)$. What we can do is let $A(x, y)$ be a modified formula that "builds in" the uniqueness condition we need. In particular, we can use this Σ_1-formula:

$$\begin{aligned}&\exists z\ (B(x,\ y,\ z)\ \ \land\\ &\quad \forall(y' \le y)\ \forall(z' \le z)(B(x,\ y',\ z')\ \to\ y = y')\end{aligned}$$

Basically, this says that y is the *shortest* string such that, for some z, $B(x, y, z)$. Since by assumption $f\,s$ is the only string such that $(s, f\,s)$ satisfies $\exists z\ B(x, y, z)$, it follows that it is also the only string such that $(s, f\,s)$ satisfies this modified formula $A(x, y)$. Furthermore, the theory S can tell that $A(x, y)$ has the uniqueness condition.

Here's the key application of all of this. The substitution function is not just *definable* in the string structure $\mathbb{S}$: in fact, it is Σ_1*-definable*. The same goes for the labeling function. Thus by Exercise 6.8.14, these two functions are representable in the minimal theory of strings S. That is, S represents syntax.

There will be a second important application in Section 7.7, which is much more general.

6.9 Syntax and Arithmetic

We have one central example of a theory that represents syntax: the minimal theory of strings S (though, again, we have deferred the proof of this until Chapter 7). But there are many other theories that will do the same job. First, it's clear that any theory in the language of strings that *extends* S also represents syntax. For a theory to represent syntax, it just needs to include the identity sentences

$$\text{apply}\langle A(x)\rangle\langle B(x)\rangle = \langle A\langle B(x)\rangle\rangle$$

for each pair of formulas $A(x)$ and $B(x)$. Since S includes each of these sentences, any theory that extends S also includes them. So, for example, the *complete* theory of strings $\mathrm{Th}\,\mathbb{S}$ also represents syntax.

But what about theories in other languages? Many of these also represent syntax. To see this, note that it doesn't matter whether the symbols ””, (⊕), and so on that appear in S are really *primitive* symbols. You could replace each of them with some more complex term—indeed, with some complex term in another language. The result of doing this is called a *translation*. If a theory includes suitable *translations* of the sentences in S, then in particular its language includes a translation of the term $\text{apply}(x, y)$, and the theory includes corresponding translations of each of the identity sentences (*). So a theory like this also represents syntax.

We'll call a theory like this **sufficiently strong**: a sufficiently strong theory is one that includes some suitable translation of the minimal string theory S. Thus any sufficiently strong theory represents syntax.

Let's make this idea a little more precise. (We won't give proofs of everything: they aren't hard, but they are a bit tedious.)

6.9.1 Definition

Let L and L' be languages. A **translation manual** from L to L' is a function that assigns each primitive n-place function symbol f in the language L some term $f'(x_1, \dots, x_n)$ in the language L', and which assigns each primitive n-place relation symbol R in the language L some formula $R'(x_1, \dots, x_n)$ in the language L'.

Given a translation manual, the **translation** of an L-formula is the result of replacing each occurrence of $f(a_1, \dots, a_n)$ with the corresponding term $f'(a_1, \dots, a_n)$, and each occurrence of $R(a_1, \dots, a_n)$ with the corresponding formula $R'(a_1, \dots, a_n)$. (This can be defined more precisely using recursion, but we won't bother going through the details.)

A **translation function** is a function that takes each L-formula A to its translation in L', with respect to some fixed translation manual.

6.9.2 Definition

Let L and L' be languages, let T be an L-theory, and let T' be an L'-theory. T' **interprets** T with respect to a translation function φ iff, for each sentence A in T, its translation $\varphi(A)$ is in T'.

6.9.3 Lemma

Let T be an L-theory and let T' be an L'-theory. Suppose that T' interprets T with respect to a translation functon φ from L to L'. Let D be a set, and suppose furthermore that each $d \in D$ has a label $\langle d \rangle$ in L. For each $d \in D$, let $\varphi\langle d \rangle$ be the label for d in L'.

If T represents a set X, then T' also represents X (with respect to the labeling we just defined). Similarly, if T represents a function f, then T' represents f as well.

Proof

If T represents X, then L includes a formula $A(x)$ such that

$$\begin{array}{llll} \text{If} & d \in X & \text{then} & T \vDash A\langle d \rangle \\ \text{If} & d \notin X & \text{then} & T \vDash \neg A\langle d \rangle \end{array}$$

Since T' interprets T with respect to the translation function φ, we know:

$$\begin{array}{llll} \text{If} & d \in X & \text{then} & T' \vDash \varphi(A\langle d \rangle) \\ \text{If} & d \notin X & \text{then} & T' \vDash \varphi(\neg A\langle d \rangle) \end{array}$$

Now, $\varphi(A\langle d \rangle)$ is the result of systematically replacing each symbol in $A\langle d \rangle$ with some term or formula. In particular, then, this is the same as the result of doing the replacement for the formula $A(x)$ and the term $\langle d \rangle$ separately, and then putting the results together. (This could be shown more carefully using induction.) So if we let $A'(x)$ be the translation $\varphi(A(x))$, it follows that

$$\begin{array}{llll} \text{If} & d \in X & \text{then} & T' \vDash A'(\varphi\langle d \rangle) \\ \text{If} & d \notin X & \text{then} & T' \vDash \neg A'(\varphi\langle d \rangle) \end{array}$$

So, since $\varphi\langle d \rangle$ is the label for d in L', this shows that $A'(x)$ represents X in T'. Things go similarly for sets of n-tuples and functions. □

6.9.4 Definition

A theory T is **sufficiently strong** iff it interprets the minimal string theory S.

6.9.5 Exercise (Tarski's Theorem Version 4)

(a) If T is sufficiently strong, then T represents syntax.

(b) For any sufficiently strong theory T, if T represents T, then T is inconsistent.

Now let's turn to an especially important example of a sufficiently strong theory.

We've been using strings to represent syntax. But Gödel originally did something a bit different. Gödel was primarily interested in the foundations of mathematics, rather than the philosophy of language, and so he was especially interested in *arithmetic*. So Gödel came up with a way of describing syntax *in arithmetic*. This is called "the arithmetization of syntax"—or "Gödel numbering". We won't be making any extensive use of this, because arithmetic isn't really our central focus, but it's good to know about it, because this is a much more common way of presenting Gödel's and Tarski's results.

In Definition 5.4.2 we presented the *minimal theory of arithmetic* Q, which is a very simple theory with just ten axioms. As it turns out:

6.9.6 Theorem
The minimal theory of arithmetic Q is sufficiently strong.

The proof of this fact involves finding a way to uniquely represent *strings* using *numbers*. This involves some non-trivial number theory—in particular, some facts about prime factors and remainders. Since this isn't a number theory course, we won't go into these details. (You can find a sketch of the proof in CITE BBJ, Lemma 16.5.)

One thing to note, though, is that you really only need the fancy number theory if you insist on just using the primitive operations $(0, \text{suc}, +, \cdot)$. If you help yourself to other operations—such as *exponentiation*—then things get a lot easier. If you have that (and a few more axioms about how exponentiation works), then instead of Gödel's fancy encoding based on prime factors, you can use the same kind of *binary* encoding that computers use. The basic idea is to think of numbers as sequences of "bits" (one and zero, or "on" and "off"); then you can use those sequences to encode sequences of sequences of bits, and so on. The basic reason exponentiation helps with this is because the "join" operation for sequences of bits, that takes, say, `10110` and `110` to `10110110`, corresponds to the operation on *numbers* defined by $x \cdot 2^n + y$, where n is the length of the binary representation of y. But calculating 2^n (for arbitrary n) uses exponents, and not just straightforward addition and multiplication.

6.9.7 Exercise
Any theory that interprets Q is sufficiently strong. In particular, the theory of arithmetic $\text{Th}\,\mathbb{N}$ is sufficiently strong.

6.9.8 Definition

Let φ be the translation from the language of sequences to the language of arithmetic, with respect to which Q (minimal arithmetic) interprets S (the minimal sequence theory). For each formula A, the canonical label for A in the sequence language is $\langle A\rangle$. Thus each formula A also has a label in the language of arithmetic, namely $\varphi\langle A\rangle$.

The **Gödel number** of a formula A is the *number* denoted by this numerical label for A. That is, the Gödel number for A is the number $[\![\varphi\langle A\rangle]\!]_{\mathbb{N}}$.

6.9.9 Exercise (Tarski's Theorem Version 5)

The set of Gödel numbers of true first-order sentences of arithmetic is not arithmetically definable.

Chapter 7

The Undecidable

> But the science of operations … is a science of itself, and has its own abstract truth and value; just as logic has its own peculiar truth and value, independently of the subjects to which we may apply its reasonings and processes. … [A] few fundamental relations being true, certain other combinations of relations must of necessity follow; combinations unlimited in variety and extent if the deductions from the primary relations be carried on far enough.
>
> Ada Lovelace, notes on *Sketch of The Analytical Engine Invented by Charles Babbage* (1842)

For some questions, there is a systematic procedure you can follow that will eventually bring you to an answer. For example, suppose you want to know whether a certain number n is prime. To answer this, you can try dividing n by each number less than it, one by one, and see if there is a remainder in each case. If you find a number $k < n$ such that dividing n by k leaves no remainder, then n is prime. Otherwise, n is not prime. What we have just described is an *algorithm* for answering the question of whether a number is prime. An algorithm is a list of instructions for how to find the answer to a question. If a question can be systematically answered somehow or other, then it is called **effectively decidable**.

We can also think about questions which have different sorts of answers. For instance, the question "What is the remainder when m is divided by n?" has a number as its answer. Many of us learned the long-division algorithm in elementary school, which provides a systematic way of answering any question of this form. A family of questions like this, the answers to which can be arrived at systematically, is called **effectively computable**.

We can describe these "families of questions" as *functions*, whose values are answers to the question. The remainder function takes a pair of numbers (m, n) to the number which is the remainder when m is divided by n. Similarly, the question "Which numbers are prime?" can be represented by the function takes each number n to either True or False. Alternatively, using the correspondence between sets and two-valued functions that we discussed in Chapter 1, we can represent this question with the *set* of all prime numbers.

The main thing we'll be working up to in this chapter is a central result about certain undecidable questions in logic. It turns out that the question "Which first-order sentences are true in the standard model of arithmetic?" is *undecidable*: there is no systematic way of answering it in general. The question "Which first-order sentences are logically consistent?" is also undecidable. (So there will always be work left for logicians to do!)

7.1 Programs

An algorithm is a general systematic "recipe" for answering a question. (This is also called an "effective procedure".) For example, given a string like `ABC`, what is the string of the same symbols in reverse order? For this example, the answer is `CBA`. How can we work out the answer in general, for an arbitrary string? One approach is to follow these steps.

1. Set the result to the empty string.
2. Go through the symbols in the string one by one, from left to right. For each symbol x, set the result to be the old result with x appended to the end.

We can also describe algorithms using formal languages: these are called **programming languages**, and a formal description written in such a language is called a **program**. The first programs were written by Ada Lovelace (and a few other people) in the 1840's (about a century before the first programmable computers were built). Nowadays programs are everywhere. There are millions of lines of programming code that make your phone work, and about a hundred million for a new car. Hundreds of different programming languages have been developed for different purposes: Javascript, C++, Python, Lisp, Haskell, and so on. Here are two examples of programs that describe (or "implement") the counting algorithm we just described. (Don't worry about the details yet—these are just meant to give you the general flavor of what programs can look like.)

Here's a program written in Python:

```python
def reverse(x):
    result = ""
    for symbol in x:
        result = symbol + result
    return result
```

Here's a program written in Javascript:

```javascript
function reverse(x) {
    var result = "";
    for (i = 0; i < x.length; i++) {
        result = x[i] + result;
    }
    return result;
}
```

Each of these languages is relatively easy to use to write complex programs—that's exactly what they're designed for. The downside is that giving a full description of the syntax and semantics for any of these languages would take a whole lot of work, because they are so complicated and have so many features.[1] What we're primarily interested in isn't *writing* programs, but rather *analyzing* programs—showing certain properties they have. So for our purposes, it makes sense to look at a much simpler programming language than any of these. It turns out that this simple programming language can answer any question that any of the others can. (We won't prove this ourselves, since that would involve the very complicated task of saying precisely what questions Javascript or Python or Haskell can answer. But computer scientists have done this—and it turns out that the answer is: exactly the same questions as our simple language.) In fact, for most purposes we could think of *any* of these languages as just our little language, with a whole lot of convenient abbreviations.

So our first technical job is to describe a simple programming language. Here's what the "reverse" program will look like in this little language:

[1] For example, formally defining a denotation function for a simplified version of the Python language is the topic of a hundred-page master's thesis (see Smeding 2009).

```
result = ””
while x != ””:
    result = head(x) + result
    x = tail(x)
```

Programs are expressions in a formal language. This language is very similar in spirit to the first-order languages we've been using already—for example, this language also uses variables. But the details are a bit different. For example, we don't have any quantifiers—because typically, finding out whether *there is* something of a certain sort practically involves *looking* for it, in some systematic way. If our domain is infinite, then there's no guarantee ahead of time that a search through the whole domain will ever end. In fact, as we'll show later on, some things we can say using quantifiers aren't decidable at all. So quantifiers aren't a good fit for programming languages.

The most important thing about these basic programs we'll describe is that they only do things that can be worked out mechanically and systematically—given enough time and space to write things down. So if we can write a program in this little language that answers a certain question, this shows that the question is *effectively decidable*.

The language we'll use is a very simple subset of the Python language. We'll call it **Py**. Because it's a subset of real Python, that means you can enter our programs into any Python interpreter and they should run. This is a useful way to check your work.[2]

There is a very small set of basic rules for forming Py-programs. This is convenient for proving things about the language: we don't have to go through zillions of special cases in our proofs. But to actually write programs in this language, it will be useful to introduce shorthand expressions that encapsulate common patterns. This situation is analogous to what we did for the syntax of first-order logic: we used a very small set of basic syntax rules, and then we treated other symbols (like `→`, `∨`, and `∃`) as abbreviations for expressions that just use the basic symbols. We'll discuss some of these shorthands along the way.

In this section we'll take an informal tour of how programs can be written in Py, looking at some examples and getting a bit of practice writing programs. In the next section we'll give a more formally precise description of the syntax and semantics for programs.

Py has three different syntactic categories. This is analogous to the distinction in first-order logic between terms and formulas. In Py, the three kinds of expressions are called **terms**, **statements**, and **programs** (also called *blocks*).

[2] For example, you can use this one: `https://repl.it/languages/python3`

There's one catch: we have a few operations that aren't built into standard Python. So to make our programs work in a standard Python interpreter, you need to add these lines to the beginning of your code:

```
def head(x): return x[0]
def tail(x): return x[1:]
quo = ”\””
com = ”,”
rpa = ”(”
lpa = ”)”
new = ”\n”
```

After that, everything should work ok.

You can use the statement `print(x)` in your programs to show the value of the variable `x` on the screen at any stage of computation. This is helpful for keeping track of how your program is working.

Another thing to watch out for is that Python interpreters are picky about white space. If you are typing programs into a Python interpreter, you should make sure to always indent using four spaces—not “tabs”—and watch out that your `while`, `for`, and `if` blocks are correctly lined up. A good text editor will help take care of these things for you automatically. See `https://en.wikipedia.org/wiki/Source-code_editor#Notable_examples`

Terms

Terms stand for things—in particular, terms in Py stand for strings. Here are some examples of terms:

```
””
x
”A”
x + y
head(x)
tail(y)
```

The terms in Py are very similar to the terms in the language of strings, but there are some slight differences to fit with Python conventions. We use `+` rather than $\oplus$ to represent the result of joining two strings together end to end. (Python syntax uses the same symbol `+` both for adding together numbers and also for joining together strings.) The term `””` denotes the empty string.

The term `x` is a variable, and it denotes whatever value happens to be assigned to the variable. In first-order logic we officially use the strings `x`, $\texttt{y}_1$, `z`, etc., for variables. When we're writing programs it's customary to use longer and more informative variable names. For instance, we might use names like `result`, or `sequenceOfPrimeNumbers`, or `awesomeString`, or pretty much whatever we want.

The term `”A”` denotes the singleton string `A`. We have a term like this for each symbol in the standard alphabet, just like in the first-order language of strings. (Remember, our standard alphabet is the Unicode Character Set. Conveniently, this is the same standard alphabet that Python interpreters use.) In almost every case, we get this term by putting quotation marks around the symbol itself. (Once again, there are a few exceptions: we use `quo` for `”`, `com` for `,`, `lpa` for `(`, `rpa` for `)`, and `new` for the newline symbol.)[3]

All of these terms so far are basically familiar from the language of strings. Besides these, we have two new term-formers. The term `head(x)` denotes the string containing just the first symbol from the string denoted by `x`, as long as that string is non-empty. Otherwise, the program will crash with an error message. The term `tail(x)` denotes all of the rest of the string denoted by `x`, *except* the first symbol—again, unless the string denoted by `x` is empty, in which case we crash with an error message.

[3] In fact, there are a few other exceptions for how Python interpreters handle some other special symbols, but we can ignore these.

We can build up complex terms in Py by putting together these basic pieces in arbitrary combinations, just like before. For example, we can build up these complex terms:

```
”A” + (x + ””)
head(y + ”A” + ”B”)
tail(head(tail(head(”A” + ”B” + new + ”C” + ”D”))))
```

Let Statements

A **statement** is an instruction, which says to do something. This is a bit different from the sentences we've been talking about so far, which describe how things already are. A statement describes a way of *changing* the way things are.

There are two basic kinds of statement. The first kind of statement is a "let" statement, which looks like this:

```
x = a
```

You should read this as an *imperative* sentence—"let x be a from now on"—and *not* as a *declarative* sentence "x is a". (It's a bit confusing that programmers use the `=` sign this way—rather than something else for the purpose, like `x := a` —but unfortunately this is almost completely standard. "Let" statements are also called "assignments", which is also unfortunately confusing terminology.)[4] So we can write things like

```
x = x + ”A”
```

If we read this as a declarative sentence ("x is identical to the result of joining x with `A` ") then it is false, no matter what `x` stands for. No finite string is one symbol longer than itself. But the *imperative* reading means "*change* the value of `x` : from now on, let `x` stand for the string which results from appending the string `”A”` to the end of the string that `x` stood for until now." Whatever string `x` used to stand for, make it now stand for a longer string than that. In imperative programs, the values of variables can change.

[4] I don't know if this is true, but I've heard that this conventional use of `=` rather than `:=` was settled on for an incredibly dumb reason: the language designers analyzed some code, and concluded that programmers use "let" statements *more often* than they use actual equality—and they wanted to save a keystroke.

A **program** (or *block*) is a string of statements joined together, which means to do what each of the statements says, one after another. For example, we can chain together "let" statements like this:

```
foo = ””
bar = ”A”
bar = bar + bar
result = head(bar)
```

First, this sets the variable `foo` so it denotes the empty string. Second, this sets the variable `bar` so it denotes the string `A`. Third, this changes the variable `bar` so it instead denotes `AA`. Finally, this sets the variable `result` to the value `A`.

You can think of the program as a list of instructions for someone who has a sheet of paper that lists all of the variables and their values—for example:

`foo`	(the empty string)
`bar`	`A`

The person follows the instructions one by one. When they reach a "let" statement, they write in some new value in the right-hand column, erasing whatever might already been written there. For instance, when they see the third instruction

```
bar = bar + bar
```

they will change the table to look like this:

`foo`	
`bar`	`AA`

After the final instruction, the table will then say:

`foo`	
`bar`	`AA`
`result`	`A`

The idea is that when they reach the end of the instructions, they'll tell you what is written in the `result` row of the table, which represents the “output” of the program.

7.1.1 Example

The following Py-program sets the `result` variable to the second symbol in whatever string is initially represented by `x` (if the length of `x` is at least two).

```
allButFirst = tail(x)
result = head(allButFirst)
```

7.1.2 Exercise

Write Py-programs that set the `result` variable to the following values.

(a) The third symbol in the string represented by the variable `x` (if the length of this string is at least three.)

(b) The string which has the same first two symbols as the string that `x` stands for and is followed by all but the first two symbols of the string that `y` stands for (when `x` and `y` both stand for strings with length at least two.)

Say we want to write a program that uses a specific string, such as `True`. One way to do this would be to write this:

```
trueString = ”T” + ”r” + ”u” + ”e”
```

But that’s a bit of a nuisance, so we’ll use this handy shorthand.

```
trueString = ”True”
```

Officially, `”True”` is just an abbreviation for `”T” + ”r” + ”u” + ”e”`. Similarly, `”ABC”` is an abbreviation for `”A” + ”B” + ”C”`, and so on. This is just like how we used symbols like `→` in first-order logic as abbreviations for expressions using only our “official” logical symbols. We are keeping our official language very simple, to make it easy to prove things about it, and then introducing shorthands that make the language easier to use. Programmers call this “syntactic sugar.”

Loops

Py has two basic kinds of statements. We just discussed the first kind: let statements. The second basic kind of statement is a *loop*. Loops let us write programs that do the same steps over and over again, until some “halt” condition is met.

7.1.3 Example

This program takes a string and returns the same string in reverse order. The basic idea is that we’re going to go through the symbols in the string one by one from left to right, and paste them together into a new string going from right to left.

Here’s how it works in more detail. First, set the result to the empty string. Then we do the following steps over and over until `x` stands for the empty string: remove the first symbol from the `x` -string, and add it onto the left side of the result.

Here’s the whole program:

```
result = ""
while x != "":
    result = head(x) + result
    x = tail(x)
```

The symbol `!=` is how we write “is not equal to” (that is, $\neq$) in Python syntax. Whatever value `x` starts out with, when the program reaches the end, `result` will have that same string in reverse order.

In general, for any terms a and b, and any block of statements A, we can build this kind of statement:

```
while a != b:
    A
```

This means to repeatedly do what A says as long as the values of a and b are different. We don’t stop repeating the block until a and b have the same value.

In general, we’ll think of a program as taking certain “input” variables (in this case `x`), doing some work, and finally putting the result in an “output” variable (`result`).

“Let” statements and “while” loops are the only basic kinds of statements we need for our programming language. But writing programs with just these statements can get pretty cumbersome. To write complicated programs, it's very helpful to introduce some more abbreviations for common patterns. At this point we're done with the "low-level" programming language: our basic tools. The rest of this section introduces some "higher-level" programming structures, which helps show what our programming language is capable of.

Branching

One important thing we can do is *branching*. We can write programs that can go in two different alternative directions, depending on whether two strings are the same.

```
if a == b:
    flag = ”True”
else:
    flag = ”False”
```

Again, the meaning of this is different from the conditional in first-order logic, because it is an *imperative* statement meant to change the world, rather than a *declarative* sentence meant to describe it. Here's what it means. First, evaluate whether the terms `a` and `b` denote the same string. (Note that we use a double equals sign `==` . This is because the single equals sign `=` was already taken for "let" statements.) If `a` and `b` have the same value, then we do the statements in the first block—in this case, we set the value of the variable `flag` to `True` . If `a` and `b` denote different strings, then instead we do the statements after the `else` —in this case, we set `flag` to `False` .

Here's another example:

```
if s == ””:
    result = ”It’s empty!”
else:
    result = head(s)
```

If `s` is not empty, then this statement sets the value of `result` to its first symbol. Otherwise, it just sets the result to be an error message.

We don't need to include `if` statements as basic building blocks, because we can always replace them using let statements and `while` loops. The trick is to write loops that are guaranteed to only happen at most one time. In general, if A and B are programs and a and b are terms, we can treat this

```
if a == b:
    A
else:
    B
```

as an abbreviation for this:

```
x = a
finished = ”False”

while x != b:
    B
    x = b
    finished = ”True”

while finished != ”True”:
    A
    finished = ”True”
```

Here `x` and `finished` should be variables that aren't used elsewhere in the program. The idea is that we have a loop for *B* that runs once if *a* and *b* have different values, and a second loop for *A* that runs once if the first loop didn't run.

Sometimes we don't care about the `else` part of an `if` -statement: we don't want to do anything in that case. We can indicate this by just leaving out the `else` part. That is, this program:

```
if a == b:
    A
```

means just the same thing as this one:

```
if a == b:
    A
else:
    ()
```

where the `else` block is the empty program. We can also write

```
if $a$ != $b$:
    $A$
```

as a synonym for

```
if $a$ == $b$:
    ()
else:
    $A$
```

(Remember that `!=` is Python's standard way of writing "not equal".) Sometimes it's also useful to chain together `if` statements. The Python abbreviation for this looks like this (`elif` is short for `else if`).

```
if $a_1$ == $b_1$:
    $A_1$
elif $a_2$ == $b_2$:
    $A_2$
elif $a_3$ == $b_3$:
    $A_3$
```

This means the same thing as

```
if $a_1$ == $b_1$:
    $A_1$
else:
    if $a_2$ == $b_2$:
        $A_2$
    else:
        if $a_3$ == $b_3$:
            $A_3$
```

The shorthand is nice to keep the indentation from getting out of control.

7.1.4 Exercise

Show the following questions are decidable by writing a program that returns `True` if the answer is "yes", and `False` if the answer is "no", using `if` statements.

(a) Are the values of `s` and `t` both equal to `True` ?

(b) Are either of the values of `s` or `t` equal to `True` ?

(c) Is the length of `s` at least two?

It will be useful to have names for the first two programs, to refer back to them later on: in particular, let's abbreviate them `and(s, t)` , and `or(s, t)` .

Bounded Loops

A common pattern in programs is to go through each of the symbols in a string one by one, do something with each symbol, and stop when we reach the end of the string. This is called a `for` loop.

For example, this program decides whether every symbol in a string is `A` .

```
result = ”True”
for symbol in s:
    if symbol != ”A”:
        result = ”False”
```

The `for` loop goes through the elements of the string represented by `s` one by one, and stores each symbol as the value of the variable `symbol` . This is similar to a `while` loop, but it is more specialized. One important feature of a `for` loop is that it is guaranteed to eventually stop, when it gets to the end of the string. In contrast, in principle a `while` loop might go on running forever, if the equality test is never passed.

Again, though `for` loops are very useful, we don't need to include them as an extra primitive in our programming language, because they can be eliminated using `while` loops. In general, suppose x is any variable, a is any term, and A is some program. We can understand this notation—

```
for x in a:
    A
```

—as a shorthand for this, where *y* is a variable that is not used elsewhere in the program—

```
y = a
while y != "":
    x = head(y)
    y = tail(y)
    A
```

7.1.5 Example

This program takes a string and repeats each symbol an extra time. For instance, it takes `ABC` to `AABBCC` .

```
result = ""
for symbol in s:
    result = result + symbol + symbol
```

7.1.6 Example

We can rewrite the `reverse` program a bit more concisely using a `for` loop.

```
result = ""
for symbol in x:
    result = symbol + result
```

Function Calls

There's another abbreviation which is useful for chaining programs together to make more complex programs. We have already written a program that reverses a string, and a program that repeats each symbol. We can stick these two programs together to produce a program that repeats the symbols *and* reverses their order. The obvious way to do this is to cut and paste, with one program immediately following the other:

```
result = ""
for symbol in s:
    result = result + symbol + symbol

x = result

result = ""
for symbol in x:
    result = symbol + result
```

Note that to make this work, we needed to add one extra line in between the original two programs: `x = result`. This feeds the *output* value of the first program to the *input* variable for the second program.

We can represent this program much more concisely using *function call* notation. The first step is to introduce a name for each of the two simple programs. Python has a standard notation for this. We can write the definitions of our two programs like this:

```
def reverse(x):
    result = ""
    for symbol in x:
        result = symbol + result
    return result

def repeatSymbols(s):
    result = ""
    for symbol in s:
        result = result + symbol + symbol
    return result
```

With each program, we've added an extra `def` line before it, and an extra `return` line after it. (We've also indented the whole program.) The `def` line tells us what *shorthand* we're planning to use for this program later on, and what the "input" variables are. The final `return` line tells Python where the program ends, and that the value of the `result` variable should be treated as the program's output.

Once we've done this, we can stick the two programs together using this concise shorthand:

```
finalResult = reverse(repeatSymbols(s))
```

Here we are using `repeatSymbols(s)` as a complex term, and `reverse(repeatSymbols(s))` as a more complex term. The idea is that `repeatSymbols(s)` stands for whatever final output you get by running the `repeatSymbols` program with the input `s` . Similarly `reverse(repeatSymbols(s))` means the final output of first getting the value of `repeatSymbols(s)` , then feeding that as an input to the `reverse` program. (The intuitive idea here is very similar to the idea of *substitution* for formulas in first-order logic.)

7.1.7 Example

This program returns `True` for an empty string, and `False` for a non-empty string.

```
def empty(s):
    if s == ””:
        result = ”True”
    else:
        result = ”False”
    return result
```

Then suppose we write this later:

```
x = empty(”ABC”)
```

Then this abbreviates

```
s = ”ABC”
if s == ””:
    result = ”True”
else:
    result = ”False”
x = result
```

This program has the result of assigning `False` to `x` .

Here's the general recipe for unpacking "function call" notation. Suppose we have a program A which we have called `programName` , with the input variables `x` and `y` . (That is, we have used the line `def programName(x, y):` .) Then say we have a let statement like this one:

```
z = programName(a, b)
```

We can unpack it like this:

```
x = a
y = b
A
z = result
```

(In fact, the real rule is a little trickier than this: *first*, we should modify all the variable names used in A so that we don't have any clashes.) If we use the shorthand more than once, we can just follow these rules as many times as we need to.

When you are writing programs, feel free to use all of the shorthands we have introduced: complex terms, `if … elif … else` branching, `for`-loops, and function call notation. Since we know that each of these *can* be eliminated and replaced with simple let and `while` statements, this means that for practical purposes we don't have to eliminate them from our programs.

7.1.8 Exercise
Write a program that computes the "dots" function from Exercise 2.5.10. For example, the output of the program for input `ABC` should be `•••`.

7.1.9 Exercise
Write programs to show that the following questions are decidable.

(a) Is s at least as long as t?

(b) Are s and t the same length?

7.2 Syntax and Semantics

Syntax for Programs

Here's a summary of the syntactic rules for terms and programs in the language Py. As in first-order logic, we're assuming that we have in the background some countably infinite set of variables. In Py, our convention for variables is a bit more flexible than in our first-order language: we will allow almost any string consisting entirely of letters and numbers (but beginning with a letter).[5]

We'll give two recursive definitions: one for Py-*terms*, and the other for Py-*programs*. We'll start with terms. Remember that in Section 3.2 we chose some constants for the language of strings: `””` for the empty string, and constants like `”A”` , `”B”` , `quo` , and `new` for single-symbol strings. Py is also a language for talking about strings, so we will use each of these same constants as Py-terms. In fact, the whole definition of Py-terms is almost the same as the definition of terms in the language of strings that we gave in Section 3.2, except we have two extra function symbols `head` and `tail` for "unpacking" strings.

7.2.1 Definition

$$\frac{x \text{ is a variable}}{x \text{ is a term}}$$

$$\frac{c \text{ is a constant in the language of strings}}{c \text{ is a term}}$$

$$\frac{t_1 \text{ is a term} \qquad t_2 \text{ is a term}}{\texttt{(}t_1\texttt{+}t_2\texttt{)} \text{ is a term}}$$

$$\frac{t \text{ is a term}}{\texttt{head(}t\texttt{)} \text{ is a term}} \qquad \frac{t \text{ is a term}}{\texttt{tail(}t\texttt{)} \text{ is a term}}$$

That much should look pretty familiar. Next we'll give the recursive definition of programs.

7.2.2 Definition

$$\frac{}{\text{The empty string is a program}}$$

[5] We have to ban a few special strings from being variables: `while` , `if` , `else` , `elif` , `def` , `return` `head` , `tail` , `quo` , `com` , `lpa` , `rpa` , and `new` .

$$\frac{x \text{ is a variable} \qquad t \text{ is a term} \qquad A \text{ is a program}}{\begin{array}{l} x \texttt{ = } t \\ A \end{array} \quad \text{is a program}}$$

$$\frac{t_1 \text{ and } t_2 \text{ are terms} \qquad A \text{ and } B \text{ are programs}}{\begin{array}{l} \texttt{while } t_1 \texttt{ != } t_2\texttt{:} \\ \qquad A \\ B \end{array} \qquad \text{is a program}}$$

Syntax Details*

That should get the idea across, but before we move on, let's get clear on a few details about what this definition means. (You can skip over these details, but they're important if you're going to do some of the parsing exercises in Section 7.4.)

Programs are strings. (Just like always, we can ask, is a program *really* just a string, or are programs some other structure that can be *represented* by a string? This is a fair question, but it will make things easier if we suppose that a program just *is* a certain string.) A string is just a bunch of symbols, one after another. But because programs can get pretty long, it would be a pain to write out a program in a single line of text. That's no problem, though: we have a special symbol in our alphabet that means "start a new line". So, for example, take this program:

```
y = x
z = y
```

We could spell out the sequence of symbols in this string very explicitly like this:

(y, , =, , x, new line, z, , =, , y, newline)

In general, we can spell out the syntax rule for let statements very explicitly like this: if x is a variable, t is a term, and A is a program, then

$$x \oplus \texttt{ = } \oplus t \oplus \text{newline} \oplus A$$

is also a program.

A second note is that our syntax uses *indentation* to indicate the structure of a `while` loop. Like writing programs in multiple lines, this "white space" convention makes programs easier to read. Each statement within a while loop should be moved over to the right by adding some white space to the beginning of the line (officially, the space symbol ` ` four times).

Let's be explicit. For any program A, there is a unique sequence of strings $(s_1, s_2, \dots, s_n)$ which are the *lines* of A (see Exercise 4.6.4). None of these strings contains any newline symbols, and

$$\begin{aligned} A = {} & s_1 \oplus \text{newline} \\ & \oplus s_2 \oplus \text{newline} \\ & \vdots \\ & \oplus s_n \oplus \text{newline} \end{aligned}$$

Then

$$\begin{aligned} \text{indent}(A) = {} & \texttt{~~~~} \oplus s_1 \oplus \text{newline} \\ \oplus {} & \texttt{~~~~} \oplus s_2 \oplus \text{newline} \\ & \vdots \\ \oplus {} & \texttt{~~~~} \oplus s_n \oplus \text{newline} \end{aligned}$$

(To be even *more* explicit, we could define this indent function recursively. Compare the "tokenization" function from Section 4.6.)

Now we can state the syntax rule for `while` statements more explicitly. If t_1 and t_2 are terms, and A and B are programs, then this is also a program:

$$\texttt{while } \oplus t_1 \oplus \texttt{ != } \oplus t_2 \oplus \texttt{:} \oplus \text{newline} \oplus \text{indent}(A) \oplus B$$

That's it for the syntax of programs.

As with first-order logic, an important fact about these definitions is that there is no syntactic ambiguity in our notation. Py-terms and programs also each have an Injective Property. (This can be officially proved by writing and checking a parsing function, as in Section 3.3.) But we will skip over these details.

Just like with formulas, it's helpful to know when a variable is "loose" in a program: in this context, this means that the variable is "read" without previously being "written". Typically variables like these represent the input for a program. We can start by defining what it is for a variable to **occur in** a Py *term*; this definition is basically identical to Definition 3.5.3 for terms in a first-order language, so we won't bother to spell it out. Then we can use this definition for terms to give a definition for the free variables in a *program.*

::: Definition The **free variables** in a program are defined recursively as follows.

1. No variables are free in the empty program.

2. A variable y is free in a program of the form

   ```
   $x$ = $t$
   $A$
   ```

   iff y is distinct from x and either y occurs in t or y is free in A.

3. A variable y is free in a program of the form

   ```
   while $t_1$ != $t_2$:
       $A$
   $B$
   ```

   iff either y occurs in t_1 or in t_2, or y is free in A or in B.

Semantics for Programs

So far we've been working with an intuitive sense of how programs work. Now let's give a precise account of the meaning of the programming language. Just like we did with first-order logic, we can recursively define a *denotation* function for Py-terms and programs.

Since programs involve variables, we'll want to use assignment functions for this. Just like before, an assignment is a function that assigns values (in this case, strings) to variables. We can recursively define the denotation of a term t with respect to an assignment g, again written $[\![t]\!]\,g$. This will always be a string—unless the denotation of t with respect to g is undefined. (Like the denotation function for terms using definite descriptions, the denotation function for programs is a *partial* function.)

7.2.3 Definition

$$
\begin{aligned}
[\![x]\!]\,g &= gx \quad \text{for each variable } x\\
[\![\texttt{""}]\!]\,g &= \text{the empty string}\\
[\![c]\!]\,g &= a \text{ if } c \text{ is the constant for the single symbol } a\\
[\![t_1\texttt{+}t_2]\!]\,g &= [\![t_1]\!]\,g \oplus [\![t_2]\!]\,g\\
[\![\texttt{head}(t)]\!]\,g &= \begin{cases} a & \text{if } [\![t]\!]\,g = a \oplus s \text{ for a single symbol } a \text{ and string } s\\ \text{undefined} & \text{if } [\![t]\!]\,g \text{ is empty}\end{cases}\\
[\![\texttt{tail}(t)]\!]\,g &= \begin{cases} s & \text{if } [\![t]\!]\,g = a \oplus s \text{ for a single symbol } a \text{ and string } s\\ \text{undefined} & \text{if } [\![t]\!]\,g \text{ is empty}\end{cases}
\end{aligned}
$$

This handles all of the terms. Note that one thing that can happen is that a variable might not be defined for an assignment g. In that case, the program crashes: $[\![x]\!]\,g$ is undefined. The same thing happens if we try to take the `head` or `tail` of an empty string—we crash, and get no denotation. If t doesn't denote anything, then $t\texttt{+}u$, $u\texttt{+}t$, $\texttt{head}(t)$, and $\texttt{tail}(t)$ also don't denote anything.

What should the "semantic value" of a *program* be? Remember, each statement in a program means something imperative. It is an instruction; following this instruction results in a change in the world. It doesn't make sense to ask whether a program is true or false—it is the wrong kind of linguistic expression for that. Instead, programs should have "dynamic semantics": that is to say, the "meaning" of a program should represent a way for the world to *change*. The key idea is that the "semantic value" of a program—which we also call its *denotation*—represents the effects that it has when it is run.

In Py, the effect that a program has is to change the values of variables. So, if we start with an assignment g of values to variables, we can think of a program as denoting the *new* assignment of values to variables that results from doing what the program says. If A is a program, then $[\![A]\!]\,g$—that is, the denotation of A with respect to g—should be the new assignment.

For example, consider this very simple program:

```
x = ”A”
```

The result of running it is to assign the value `A` to the variable `x`. If we start with an assignment g, then the new assignment we get after running this program is the *variant* assignment $g[\mathtt{x} \mapsto \mathtt{A}]$. Thus, for any assignment g, we can write the denotation of this program as

$$\llbracket \mathtt{x\ =\ ”A”} \rrbracket\, g = g[x \mapsto \mathtt{A}]$$

In general, a “let” statement $x\ \mathtt{=}\ t$ (for a variable x and a term t) works by taking an assignment g, and updating it to a variant assignment $g[x \mapsto d]$, where d should be the denotation of t (with respect to the original assignment g). In short, for a program that just consists of a let statement,

$$\llbracket x\ \mathtt{=}\ t \rrbracket\, g = g[x \mapsto d] \qquad \text{where } d = \llbracket t \rrbracket\, g$$

Usually the let statement will be followed by more stuff, though. In general, the rule for let statements looks like this:

$$\left\llbracket \begin{array}{l} x\ \mathtt{=}\ t \\ A \end{array} \right\rrbracket g = \llbracket A \rrbracket \left(g \left[x \mapsto \llbracket t \rrbracket\, g \right] \right)$$

The case of `while` statements is a bit trickier. Here’s another example:

```
while x != ”A”:
    x = head(y)
    y = tail(y)
```

What is the effect of this program? We can work it out in steps. Whatever assignment we start with—call it g_0—we’ll start by checking whether $g_0(\mathtt{x}) = \mathtt{A}$. If it is, then the program does nothing at all: that is, the final assignment is just g_0. If the value of `x` isn’t `A` , then we do what the inner block says to do, updating to a new assignment g_1. (In particular, $g_1 = g_0[\mathtt{x} \mapsto s, \mathtt{y} \mapsto s']$ where s is the first symbol of $g_0(\mathtt{y})$, and s' is all of $g_0(\mathtt{y})$ except the first symbol.) Then we go back and check again: this time, we check whether $g_1(\mathtt{x}) = \mathtt{A}$. And so on. If eventually we reach an assignment g_n such that $g_n\mathtt{x} = \mathtt{A}$, then assignment g_n is the final result of this program. But this might never happen. For this program, if the original value of `y` is a string that doesn’t contain the letter `A` (or it it doesn’t *have* a value) then eventually the inner block will crash, when `y` runs out of symbols. Or another thing that can happen is the loop keeps on going forever: our program hangs and we get the spinning beach ball of doom. In either of these cases, the denotation of the `while` loop is just undefined.

Here’s another way of describing this. If our `while` loop has a denotation at all for a starting assignment g, then there is a finite sequence of assignments

$(g_0, g_1, g_2, \ldots, g_n)$ such that $g_0 = g$, each step in the sequence from g_i to g_{i+1} is given by doing what the inner block

```
x = head(y)
y = tail(y)
```

says to do, and g_n is the *first* assignment in the sequence such that $g_n(\texttt{x}) = \texttt{A}$. We'll call a sequence like this a *finite loop sequence.* If there is a sequence like this, then the result of running the program is g_n.

We can put these ideas together to give an official definition for the semantics of programs.

7.2.4 Definition

The *denotation* of a program A with respect to an assignment g is defined recursively as follows.

The empty program () doesn't do anything:

$$[\![()]\!]\, g = g$$

For a "let" statement, where x is a variable, t is a term, and A is a program,

$$\left[\!\!\left[\begin{array}{l} x \texttt{ = } t \\ A \end{array}\right]\!\!\right] g = [\![A]\!]\left(g\left[x \mapsto [\![t]\!]\, g\right]\right)$$

The most complicated case is a `while` statement,

$$\begin{array}{l} \texttt{while } t_1 \texttt{ != } t_2\texttt{:} \\ \qquad A \\ B \end{array}$$

(where t_1 and t_2 are terms and A and B are programs). A **finite loop sequence** (for terms t_1 and t_2 and a program A) is a sequence of assignments $(g_0, g_1, \ldots, g_n)$ such that

1. Each step in the sequence applies the block A once. That is, for each $i < n$,

$$g_{i+1} = [\![A]\!]\, g_i$$

2. g_n is the *first* assignment in the sequence for which the values of t_1 and t_2 are the same. That is,
$$\llbracket t_1 \rrbracket\, g_n = \llbracket t_2 \rrbracket\, g_n$$
and for each $i < n$,
$$\llbracket t_1 \rrbracket\, g_i \neq \llbracket t_2 \rrbracket\, g_i$$

We can show by a simple inductive proof that, for any assignment g (and for any terms t_1 and t_2 and program A) there is *at most one* finite loop sequence whose first element is g. (But remember there might be *no* such sequence.) Thus we can define the denotation of the `while` program as follows.

$$\left\llbracket \begin{array}{l} \texttt{while } t_1 \texttt{ != } t_2\texttt{:} \\ \quad A \\ B \end{array} \right\rrbracket g \quad = \quad \begin{cases} \llbracket B \rrbracket\, h & \text{where } h \text{ is the last element of} \\ & \text{the finite loop sequence for } t_1, t_2, \\ & \text{and } A \text{ whose first element is } g \text{, if} \\ & \text{there is any such sequence} \\ \text{undefined} & \text{if there is no such sequence} \end{cases}$$

That completes the recursive definition of the semantics for programs.

Just like we did with formulas, it will be helpful to have some notational conventions to minimize the amount of assignment-wrangling we have to do. We will use the notation $A(x)$ for a program in which at most the variable x is free. Programs with more free variables are similarly written as $A(x, y)$, etc. If we have made these "input" variables clear in context, then instead of talking about an assignment $[x \mapsto s, y \mapsto t]$, we can just talk about the sequence of values (s, t). Similarly, while officially the denotation of a program gives us back a full variable assignment, usually we are only interested in the final value of the "output" variable, which we will always assume is the variable `result`. This motivates the following definition.

7.2.5 Definition

(a) Let $A(x)$ be a program, and let s be a string. Then we use the notation $\llbracket A \rrbracket\,(s)$ for the final result of running the program $A(x)$ with s as the initial value of x. That is, if g is the assignment $\llbracket A \rrbracket\,[x \mapsto s]$, then $\llbracket A \rrbracket\,(s) = g(\texttt{result})$. More briefly:
$$\llbracket A \rrbracket\,(s) \quad = \quad \big(\llbracket A(x) \rrbracket\,[x \mapsto s]\big)(\texttt{result})$$
If there is no final result (either because there is no assignment g, or because g does not have a value for the variable `result`) then $\llbracket A \rrbracket\,(s)$ is undefined.

(b) A program $A(x)$ **halts** for input s iff $\llbracket A \rrbracket(s)$ is defined.

(c) The **extension** of a program $A(x)$ is the partial function that takes each string s to $\llbracket A \rrbracket(s)$ (the final result of running the program $A(x)$ with s as its input) if $A(x)$ halts for input s, and otherwise is undefined.

We generalize these definitions to programs with more than one input variable in the obvious way. We also use a similar convention for a program A with *no* free variables: in this case the notation $\llbracket A \rrbracket$ means the result of running A with the empty input assignment. We also use similar notational shortcuts for Py-terms.

Now that we have a formal definition of the semantics of programs, we can ask: which functions can be expressed by a program? In other words, which functions are *computable* using Py programs?

7.2.6 Definition

(a) A function $f : \mathbb{S} \to \mathbb{S}$ is **Py-computable** iff it is the extension of some program.

(b) A set of strings $X \subseteq \mathbb{S}$ is **Py-decidable** iff its characteristic function is Py-computable: that is, the function that takes each string $s \in X$ to `True` and each string $s \notin X$ to `False` is the extension of some program.

The definitions are similar for n-place functions and sets of n-tuples.

Notice that these definitions are closely analogous to our earlier definitions of *definable* functions and sets. The key difference is just what kind of language we are using: then, we were talking about the extensions of terms and formulas in a first-order language, and now we are talking about the extensions of programs. In a slogan, we could say that a *computable* function is one that is definable using a *programming* language, rather than a first-order language, and likewise, a decidable set is one that is definable using a programming language.

7.2.7 Example

Prove that the following program halts, for any initial value for `x`.

```
while x != ”A”:
    x = ”A”
result = tail(x)
```

Proof
We'll work this one out in tedious detail, to show how all the pieces are working.

Let's work from the inside out. Start by looking at the inner block,

```
x = ”A”
```

Using the definition of the denotation function for "let" assignments (and for the empty program) tells us that the denotation of this block is the function that takes any assignment g to the assignment

$$g[\texttt{x} \mapsto \texttt{A}]$$

That is, this block updates the value of the variable `x` to the string `A` .

Next, let's use this to evaluate the `while` statement,

```
while x != ”A”:
    x = ”A”
```

To show that this halts, we need to show that there is some *finite loop sequence* for the terms `x` and `”A”` and the inner block. Let g be an assignment. There are two cases: either $g(\texttt{x})$ is `A` , or it is something else. If $g(\texttt{x}) \neq \texttt{A}$, then we can easily show that this length-two sequence (g_0, g_1) meets the three conditions of the definition of a finite loop sequence.

$$(g, g[\texttt{x} \mapsto \texttt{A}])$$

First, g_1 is clearly given by applying the denotation of the inner block to the assignment g_0. Second, $g_0(\texttt{x}) \neq \texttt{A}$ by assumption. Third, clearly $g_1(\texttt{x}) = \texttt{A}$.

On the other hand, if $g(\texttt{x}) = \texttt{A}$, then we can show that the length-one sequence (g) meets all the conditions. The first and second conditions are both vacuously true, since there is no number $i < 0$. The last condition is obvious: $g(\texttt{x}) = \texttt{A}$.

In either case, there is a finite loop string for the `while` loop starting with g, and so the loop halts. Note also that whether or not $g(\texttt{x}) = \texttt{A}$ for the intial assignment, for the final assignment h in the sequence, $h(\texttt{x}) = \texttt{A}$; in particular, this is not empty. Now evaluate the final "let" statement (which is followed by the empty program):

```
result = tail(x)
```

The denotation of this program, given the assignment h, is

$$h\left[\texttt{result} \mapsto [\![\texttt{tail(x)}]\!]\, h\right]$$

To ensure that this is defined, we just need to check that $[\![$`tail(x)`$]\!]\,h$ is defined. And this is true: looking at the definition for the denotation of `tail` terms and variables, we see that this is defined as long as $h($`x`$)$ is not empty, which we have already shown is true.

In short, for any string s, $[\![A]\!]\,s$ is defined, which means that A halts for every input. □

7.2.8 Exercise
Give an example of a program that does not halt for any input, and use the definition of the denotation function for programs to prove this.

7.2.9 Definition

(a) Let $A(x)$ be a program and let t be a term. Then $A(t)$ is the program that adds a let statement to the beginning of $A(x)$:

```
x = t
A(x)
```

(The idea here is similar to substitution for first-order formulas.)

(b) Similarly, if $A(x)$ and $B(y)$ are programs, then $B(A(x))$ is the program

```
A(x)
y = result
B(y)
```

(This is similar to our “function call” shorthand.)

7.2.10 Exercise

(a) For any program $A(x)$ and term t,

$$[\![A(t)]\!] = [\![A]\!]\left([\![t]\!]\right)$$

That is, the result of running the program $A(t)$ is the same as the result of running the program $A(x)$ with the denotation of t as its input.

(b) For any programs $A(x)$ and $B(y)$ and any string s,

$$[\![B(A(x))]\!](s) = [\![B]\!]\big([\![A]\!](s)\big)$$

In other words, running the "composite" program $B(A(x))$ with input s has the same result as first running $A(x)$ with the input s, and then passing that result on as the input for $B(y)$.

7.3 The Church-Turing Thesis

If we want to show that a question is decidable, we can write a program to answer it. But how would we show that a question is *undecidable*? To do this, we wouldn't just need to show that no program in our little language Py answers the question—we'd need to show that no program in *any* reasonable programming language can answer it. If a question is undecidable, then there isn't any systematic algorithm for solving it at all.

Alonzo Church and Alan Turing each hypothesized that there are *universal* programming languages: languages which are expressive enough to describe *every* systematic algorithm. In fact, they didn't just hypothesize that such languages exist: they proposed some specific candidates. (In Church's case, these consisted of a small family of operations on functions of natural numbers. In Turing's case, the "language" consisted of **Turing Machines**—hypothetical devices for reading and printing on a long tape.) These proposals amounted to giving a formal analysis of the intuitive concept of a decidable question. You might doubt whether such an analysis could succeed. (Surely any conceptual analysis like this would have counterexamples!) But in fact, we have very strong evidence that Church and Turing's proposal is right.

The key philosophical claim is called the **Church-Turing Thesis**. The first bit of evidence for it is packed right into its name. Church's and Turing's theses look *different*: they are apparently different analyses of the concept of a decidable question. But they turned out to be equivalent to one another. That is, any question which is decidable using a Turing Machine is also decidable using Church's functions, and vice versa.

Today we have hundreds more examples—formal languages like C++ or Python or Haskell and so on: these *also* turn out to be equivalent to Turing and Church's languages. This also means that we get a little bit more empirical evidence for the truth of the Church-Turing Thesis every time a programmer takes a precisely described algorithm and implements it in their favorite programming language. The Church-Turing Thesis is thus a hypothesis which is extraordinarily well-confirmed by the practice of modern programming.

(Besides this empirical evidence, there are also some very strong *philosophical* arguments for the Church-Turing Thesis. `CITE` Smith, Kripke. We have some clear and precise *sufficient* conditions for a question to be effectively decidable: it's enough to show that we can write a program, for example, in our little language Py. There are also interesting arguments for some precise *necessary* conditions on decidability. But it turns out that we can prove that these sufficient conditions and these necessary conditions are equivalent! So this argument would show that Py-decidability is a necessary and sufficient condition for decidability, just as the Church-Turing Thesis says. `TODO: explain this a bit.`)

Even so, it's worth remembering that it is a philosophical thesis—an extraordinarily successful philosphical thesis, but not officially a theorem. We can prove lots of theorems about various kinds of formal languages. But the Church-Turing Thesis is about the relationship between these formal languages and the intuitive notion of a decidable question.

In particular, our little language Py is equivalent to each of these other programming languages: a function is Py-computable if and only if it is computable using a Turing Machine, if and only if it is computable using Church's functions, if and only if it can be computed by a program in C++ or any other standard programming language. So if any of these languages is a universal programming language, so is Py. So according to the Church-Turing Thesis, whatever can be done in any systematic way—by any algorithm at all—can also be done using humble Py.

7.3.1 The Church-Turing Thesis

(a) A partial function $f : \mathbb{S} \to \mathbb{S}$ is effectively computable iff f is Py-computable.

(b) A set $X \subseteq \mathbb{S}$ is effectively decidable iff X is Py-decidable.

In what follows, we will freely appeal to the Church-Turing Thesis (though it's generally a good idea to be clear about when exactly we're relying on it). This is extremely useful in two ways.

First, this lets us deduce the existence of programs, even without formally writing them out. In order to show that a question is decidable, it's enough to informally give some reasonably careful description of a systematic procedure for answering it. But even once we've done this much, transforming an informal description of an algorithm into a formal program can still be pretty tricky. (That's what professional programmers are for.) Given the Church-Turing Thesis, we can deduce the existence of a program from the existence of an algorithm, even when we haven't worked out exactly how to write that program. We'll do this in what follows: rather than writing out fully detailed programs in our little language, we can just outline how a program ought to work, and posit that *some* program does in fact work that way, appealing to the Church-Turing Thesis.[6]

Second, this lets us prove results about undecidability. We can mathematically prove that every Py-decidable set has certain properties. Then, using the Church-Turing Thesis, we can conclude that every *decidable* set has those properties as well, or to put that the other way around, any set without those properties is undecidable.

7.3.2 Exercise
Given the Church-Turing Thesis, prove that there are uncountably many effectively undecidable sets of strings.

7.4 The Universal Program

Programs operate on strings: they take strings as input, and spit out strings as output. But a program also *is* a string of symbols itself. This means we can use programs themselves as the input or output for other programs. Programs that manipulate programs might sound recherché, but it's actually very common and practical. When we write a program in Python, what we are doing is typing in a certain string of symbols. When we then want to *run* that program, we are providing this string as an argument to a Python *interpreter*—which is some other program. Somebody wrote that program, too, in some programming language. In fact, the interpreter might be written in Python itself!

Even our little language Py can do this. We can write a "Py-interpreter" in Py. This is a program run(`program`, `inputValue`) with two input variables. The first input should be a Py-program A, and the second input is an input value s to provide to A. Then the final result of run is the same as the final result of running A with the input s. At least, it has this result if A *has* any final result. It could be that A crashes or goes into an infinite loop. In that case, the interpreter will also just crash or run forever. In short, for any program A and string s,

$$[\![\text{run}]\!](A, s) \quad = \quad [\![A]\!](s)$$

[6] Peter Smith CITE calls these "labor-saving" uses of the Church-Turing Thesis. This contrasts with

Basically, what we're doing is precisely describing the denotation function for Py, within Py! This is very close to what Tarski's Theorem showed we *couldn't* do, for sufficiently strong theories: we can represent the semantics of Py within Py. A key difference is that Py programs (unlike first-order sentences) can *crash*. We'll come back to this point in Section 7.5 and Section 7.6.

First, let's introduce some tools which are analogous to what we did in Chapter 6. Officially, our Py-programs only have one "data type": strings. But there are natural ways of using strings to represent other things—like numbers, or sequences of strings.

7.4.1 Exercise

In Section 6.2 we defined a string representation function for sequences of strings. Show that the following functions are computable:

(a) The function that takes the string representation of a non-empty sequence of strings to its first element.

(b) The function that takes a the string representation for a sequence of strings s, and a string t, and returns `True` if t is an element of the sequence s, and otherwise returns `False`.

For the universal program, we'll need string representations for one other important kind of thing: *assignment functions*. There are many ways to do this, but here's one. We have already discussed a way of representing a sequence of strings using a single string in Section 6.2. We can represent an assignment function as a sequence of strings like this one:

$$(\texttt{x:hello}, \quad \texttt{result:}, \quad \texttt{s:ABC})$$

This represents the assignment function

$$\left[\begin{array}{lcl} \texttt{x} & \mapsto & \texttt{hello} \\ \texttt{result} & \mapsto & \text{the empty string} \\ \texttt{s} & \mapsto & \texttt{ABC} \end{array}\right]$$

Each element of the sequence joins up a *variable* with its *value* string, separated by the symbol `:`. (For this to work out right, it's important that we have stipulated that the symbol `:` can't ever show up within a variable name.) Then we can use the string representation function for sequences to represent a key-value sequence like this as a single string.

(We'll only ever need to worry about assignment functions that are defined for just finitely many variables—which is a good thing, because there is no way to represent arbitrary *infinite* assignment functions using finite strings. There are too many of them.)

7.4.2 Exercise

The following functions are computable, with respect to the string representation function defined above.

(a) The function that takes a string representation of an assignment function g and a variable x to its value gx. That is, this function takes the string representation for a sequence of strings s, and a "key" string k which does not include the symbol `:`, and returns a string v such that k`:`v is an element of the sequence s, if there is any such string v.

(b) The function that takes an assignment function g, a variable x, and a string s, to the new assignment $g[x \mapsto s]$, which modifies g by setting the value of x to s.

7.4.3 Lemma

The **denotation function** takes a pair of a program A and an assignment g and returns the denotation $[\![A]\!]\,g$ (when this is defined). The denotation function is Py-computable.

Proof Sketch

We won't write out a full program for this, but we will informally describe an algorithm for doing this. By the Church-Turing Thesis, this algorithm can be implemented by some program.

The first part of this project is called *parsing*. We need to take a program (or a term), and split it up into its meaningful parts. We can write a bunch of small programs to handle basic parsing tasks. Here's our to-do list. (We won't actually do all of this: the goal here is just to make it apparent that the interpretation function is *computable*, not to actually write a complete parser and interpreter. But if you have the time and interest, it's fun to work out some of these details in front of a computer.)

1. Write a program that takes a program as input, and returns `empty` if it is the empty program, `let` if it begins with a let statement, or `while` if it begins with a `while` loop.

2. Write three programs that each take as their input a program that begins with a let statement, and return (a) the variable on the left side of the equals sign, (b) the term on the right side of the equals sign, and (c) the rest of the program after the let statement.

3. Write four programs that take a program beginning with a `while` loop, of the form

```
while a != b:
    A
B
```

   and return (a) the first term a, (b) the second term b, (c) the inner block A, and (d) B the remaining lines of the program after the while loop.

   (One slightly tricky part here is figuring out where the inner block ends. As we noted earlier, in Python this depends on the indentation.)

4. (a) Write a program that takes a Py-term as input and identifies whether it is a variable, a constant (either the constant `""` for the empty string, or else one of the constants like `"A"` for a one-symbol string) or a term of the form `head(`t`)`, `tail(`t`)`, t_1`+`t_2.
   (b) For `head` and `tail` terms we should also write programs that return the inner term t, and in in the case of `+` we should write programs that return each of the inner terms t_1 and t_2.
   (c) For constant terms, we should also write a program that tells us which string the constant stands for. (In most cases, this just means stripping off the outer quotation marks, but remember that there are a few special cases.)

The components we have described so far just analyze the syntax of programs. To calculate what a program *does*, we'll need to keep track of an assignment function, and work out how each part of a program ends up modifying it. For this purpose we'll use the programs from Exercise 7.4.2 that manipulate assignment functions: we have a program `getValue` for looking up the value of a variable in an assignment, and a program `updateAssignment` for updating the value of a variable in an assignment.

We can build our interpreter by putting all these components together. There will be two parts: a term-evaluator, and a program-interpreter.

The term-evaluator takes an assignment and a term and returns the string that the term denotes (with respect to that assignment). We start by figuring out which form the term has. If it's a variable, then we look up the value of the variable in our assignment (using function 6 above). If it's one of the constants like `""` , `”A”` , or `quo` , then we return the corresponding string—either the empty string, or a one-symbol string.

The other cases—terms built from `head` , `tail` , or `+` , are a little trickier, because these terms include other terms. The most natural way to handle this would be with a *recursive* program that can call itself (see Section 7.5). Since recursive programming isn't a part of basic Py, we need to be a little devious. Here's the trick. We can easily evaluate a term if it's simple enough—if it doesn't nest `+` or `head` or `tail` . But we can always break down a complicated term into simple terms, by introducing extra let statements. In order to evaluate a complex expression like `”A” + head(x)` , we can break it into two steps: first, set an intermediate variable `temp` to the value of `head(x)` , and then evaluate `”A” + temp` instead.

So the idea is that, before we try to interpret a program, we can start by simplifying its terms. Say we have a program that begins with this statement:

```
x = ”A” + head(y + z)
```

Then we can break this up into simpler let statements, like this:

```
temp1 = ”A”
temp2 = y + z
temp3 = head(temp2)
x = temp1 + temp3
```

In this simplified program, we never embed any term *other than variables* inside more complex terms.

Say a term is **simple** iff it has no subterms other than variables. That is to say, a simple term is either (a) a variable, (b) a constant, (c) of the form `head(`*x*`)`, `tail(`*x*`)`, or *x*`+`*y* for some *variables* *x* or *y*.

Say a *program* is simple iff all of the terms that appear in its first line are simple (or else it is the empty program). That is, a simple program is either empty, or else of the form

```
$x$ = $t$
$A$
```

for a simple term t, or else of the form

```
while $t_1$ != $t_2$:
    $A$
$B$
```

for simple terms t_1 and t_2.

Then we can add these syntactic manipulations to our to-do list.

5. Write a program that takes a program as input, and returns `True` iff it is simple.

6. Write a program that takes a program which is not simple as input, and returns an equivalent simpler program. For example, this will take a program of the form

   ```
   $x$ = $t_1$ + $t_2$
   $A$
   ```

   to a new program with this form:

   ```
   $y$ = $t_1$
   $z$ = $t_2$
   $x$ = $y$ + $z$
   $A$
   ```

   (where y and z are variables which are not already used in the original program). This result might not be simple yet: t_1 might still be another complex term. But if we do this enough times, eventually the resulting program will be simple. We'll call this program `simplify`.

   The `simplify` program is a reasonably straightforward bit of syntactic manipulation, though it would take some work to write out. (If you're going to try to write it yourself, one thing you'll need to do first is write a program that takes a program as input, and returns a "new" variable which is not used in that program.)

Now we can just write our term-evaluator for *simple* terms, which is pretty straightforward, once we have the parsing and assignment-wrangling tools from our to-do list.

```
def evaluateSimpleTerm(term, g):
    kind = kindOfTerm(term)
    if kind == ”variable”:
        result = getValue(g, term)
    elif kind == ”constant”:
        result = getStringFromConstant(term)
    elif kind == ”head”:
        x = innerTermOfHead(term)
        result = head(getValue(g, x))
    elif kind == ”tail”:
        x = innerTermOfTail(term)
        result = tail(getValue(g, x))
    elif kind == ”join”:
        x = firstTermOfJoin(term)
        y = secondTermOfJoin(term)
        result = getValue(g, x) + getValue(g, y)
    return result
```

Here's how our program-interpreter will work. First, we'll check if the program is simple or not. If it isn't simple, then our first job is to simplify it. After that we'll try again.

Once we have a simple program, we'll look at its first statement to decide what to do. If it doesn't have any first statement—the program is empty–then we're already done. If it's a let statement x `=` t, then first we use our term-interpreter to evaluate the (simple) term t, and we use our "update an assignment" program to set the value of x to whatever value t denotes.

Next, suppose it's a `while` -statement, so the program has the form

```
while $t_1$ != $t_2$:
    $A$
$B$
```

Then we'll start by evaluating the (simple) terms t_1 and t_2. If they have the same value, then we're done with the loop, so we just go on to run the rest of the program B. If they have different values, though, then we will add another copy of the subprogram A to the beginning of our program (including this `while` loop) and keep going. That is, in this case we'll run the program

```
$A$
while $t_1$ != $t_2$:
    $A$
$B$
```

Let's spell this out in more detail. The whole interpreter goes in one big `while` loop. We'll keep track of an assignment `g` as we go, and step through the statements we need to evaluate one by one, updating `g` as we go.

```
def interpretProgram(program, g):
    while program != ””:
        if simpleProgram(program) == ”False”:
            program = simplify(program)
        else:
            kind = kindOfProgram(program)
            if kind == ”let”:
                variable = variableInLetStatement(program)
                term = termInLetStatement(program)
                value = evaluateSimpleTerm(term, g)
                g = updateAssignment(g, variable, value)
                program = remainderAfterLetStatement(program)
            elif kind == ”while”:
                a = firstTermInWhileStatement(statement)
                b = secondTermInWhileStatement(statement)
                block = blockInWhileStatement(statement)
                value1 = evaluateSimpleTerm(a, g)
                value2 = evaluateSimpleTerm(b, g)
                if value1 == value2:
                    program = block + program
                else:
                    program = remainderAfterWhileStatement(program)
    return g
```

And that about finishes it up, aside from the details we skipped over. So the denotation function for Py-programs is computable. □

7.4.4 Exercise
The **Py-interpretation function** is the function that takes each pair of a program $A(x)$ and a string s to the result $[\![A]\!](s)$, whenever this is defined, and which is undefined otherwise. Use Lemma 7.4.3 to show that the Py-interpretation function is Py-computable.

7.5 The Halting Problem

Recall that a program *halts* if and only if it has some well-defined value. A program that halts is one that neither crashes with an error nor "hangs" in an infinite loop. Here is perfectly sensible question: which programs halt? The Py-interpretation function is a precisely defined partial function. The "halting problem" is the precise question of which programs are in the domain of this function. For any particular program A, either A has some final value, or it doesn't.

This is a practically important question. If you've been working through the exercises, by now you've probably accidentally written some programs that crash or hang. It would be extremely useful to have a program-checking program: a program that determines whether your program will go into a never-ending `while` loop, or not.

Unfortunately, there is no such program. The question of which programs halt—while it is a perfectly precise question with a correct answer—is effectively undecidable. There is no systematic method for determining, in general, which programs are going to eventually return a value.

This fact is very closely connected to Tarski's Theorem about the undefinability of truth. (Remember that decidability and definability are very closely related: the difference is that one uses a programming language, while the other uses a first-order language.) The proof is also very similar.

Let's introduce some notation to make the analogies more obvious. Just like in the first-order language of strings, in our programming language we have a standard term for each string, like `”A” + ”B” + ”C” + ””` . As before, let’s call this a string’s **canonical label** (in Py), and use the notation $\langle s\rangle$. (Using the abbreviation we introduced in Section 7.1, we can abbreviate this term as `”ABC”` .)

We can also plug these terms into programs: $A\langle s\rangle$ is the program that runs the program $A(x)$ with the input s. (That is, $A\langle s\rangle$ is the program consisting of the let statement x `=` $\langle s\rangle$ followed by $A(x)$.) Similarly, anything that has a standard string representation—such as sentences or programs—has a canonical label in Py, which is just the canonical label for its string representation. This is easy to check:

7.5.1 Proposition

For any string s,

$$[\![\langle s\rangle]\!] \quad = \quad s$$

7.5.2 Exercise

For any program $A(x)$ and string s,

$$[\![A\langle s\rangle]\!] = [\![A]\!](s)$$

The first step is to write some programs to do basic syntactic manipulations. First, just as the label function was *definable* in the sequence theory, similarly it is *computable* in Py. We can show this by writing a program.

7.5.3 Exercise

The function that takes each string s to its canonical label $\langle s\rangle$ is computable.

7.5.4 Proposition

The “substitution” function that takes any program A(`x`) and term t to the program $A(t)$ is computable.

Proof
Here is a program:

```
def substitution(program, term):
    result = ”x = ” + term + new + program
    return result
```

7.5.5 Exercise
The “diagonalization” function that takes any program $A(\texttt{x})$ to the program $A\langle A(\texttt{x})\rangle$ is computable. That is, there is a program Diag(y) such that, for any program $A(\texttt{x})$,

$$[\![\mathrm{Diag}]\!]\,(A(\texttt{x})) = A\langle A(\texttt{x})\rangle$$

Let’s be very clear about what this means. The program Diag(y) is a syntax-manipulating program. It takes a program as its input, and then it modifies that program to produce another program as output. The new program simply adds a line of the form `x = ` t to the beginning of $A(\texttt{x})$, where specifically the term t is the canonical label for the program $A(\texttt{x})$ itself. For example, suppose $A(\texttt{x})$ is this very simple program:

```
z = x
```

Then the result of applying the “diagonalization” function to $A(\texttt{x})$ is this slightly more complex program:

```
x = ”z = x” + new
z = x
```

(Note in particular that while `x` was a free “input” variable in $A(\texttt{x})$, the diagonalized program $A\langle A(\texttt{x})\rangle$ does not have any free variables.)

7.5.6 Exercise (Kleene’s Fixed Point Theorem)
Let $F(x)$ be any program. Then there is a program A such that

$$[\![F]\!]\,(A) = [\![A]\!]$$

That is, the result of running the program $F(x)$ with the “fixed point” program A as its input is the same as the result of running A itself.

Hint. Refer back to the proof of Gödel’s Fixed Point Theorem (Exercise 6.7.11). It may also be helpful to remember Exercise 7.2.10.

Notice in particular that if A is a “fixed point” of $F(x)$ in the sense of Kleene’s Theorem, then A halts iff $F(x)$ halts for the input A.

7.5.7 Exercise

Write a program Flip(x) which does not halt for the input `True`, and which halts for any input besides `True`.

7.5.8 Exercise (Turing's Theorem)

The set of programs that halt is undecidable (given the Church-Turing thesis).

Hint. Suppose that there is a program Halt(x) such that

$$\llbracket \text{Halt} \rrbracket\,(A) = \begin{cases} \texttt{True} & \text{if } A \text{ halts} \\ \texttt{False} & \text{otherwise} \end{cases}$$

Then you can use Kleene's Theorem and the program Flip(x) to derive a contradiction, using similar reasoning to the proof of Tarski's Theorem or the Liar Paradox.

We used Kleene's Fixed Point Theorem as a lemma on the way to proving Turing's Theorem. But this is also an important result in its own right, because it provides a foundation for *recursive* programming. It's often handy to write programs that call themselves. For example, here's another way of writing the reverse program:

```
def reverse(x):
    if x == "":
        result = ""
    else:
        reversedTail = reverse(tail(x))
        result = reversedTail + head(x)
    return result
```

This program calls the `reverse` program itself. Since each time `reverse` calls itself, the string passed along as the value of `s` gets shorter, eventually these recursive self-calls will bottom out at the empty string. So even though the program calls itself, it will always end up halting. This is very similar to the kind of recursive definitions we've given for functions on numbers and strings.

Self-calling programs like this one are not an official part of Py. But Kleene's theorem shows us how to unpack programs like this in Py, using a *fixed point*. First, we need to state a slightly more general version of Kleene's Theorem, which allows us to also pass a "side argument":

7.5.9 Proposition (Kleene's Fixed Point Theorem Version 2)
Let $F(x, y)$ be a program. Then there is a program $A(y)$ such that, for any string s,

$$[\![A]\!](s) = [\![F]\!](A(y), s)$$

This can be proved in basically the same way as Exercise 7.5.6

Now, suppose we want to write the recursive program `reverse`. Let's start by modifying it a bit. At the point where we wanted to call the `reverse` program itself, instead we can run some *arbitrary* program which is provided as an extra argument.

```
def protoReverse(program, x):
    if x == "":
        result = ""
    else:
        reversedTail = run(program, tail(x))
        result = reversedTail + head(x)
    return result
```

(Here `run` is the Universal Program from Exercise 7.4.4.) Then Proposition 7.5.9 tells us that there is a program $R(\mathtt{x})$ which has the same effect as running the protoReverse program with $R(\mathtt{x})$ itself as the first argument.

$$[\![R]\!](s) = [\![\text{protoReverse}]\!](R(\mathtt{x}), s)$$

In other words, $R(\mathtt{x})$ is equivalent to a program that calls $R(\mathtt{x})$ itself! So the simple Py-program $R(\mathtt{x})$ has the same behavior as the recursive program `reverse`. In general, we can construct a recursive program as a fixed point of a "higher-order" program like protoReverse. (For this reason, Kleene's Fixed Point Theorem is also known as Kleene's *Recursion* Theorem.)

7.6 Semi-Decidable and Effectively Enumerable Sets

Here is a point that might be a little confusing. The denotation function for programs is computable; but the question of whether a program halts is undecidable. Why can't we use the Universal Program `run` to decide whether a program halts?

We can clarify the relationship between these two facts by introducing another notion: this is something which is less demanding than decidability, but still goes a long way toward it. A *semi-decidable* set is one that can be "decided in one direction". What that means is that there is an algorithm such that, for any given d, if d *is* in the set, then the algorithm will eventually tell you so—and the algorithm won't ever tell you something is in the set which really isn't—but if d is *not* in the set, then there is no guarantee that the algorithm will tell you anything at all. The algorithm will tell you the good things are good, and it won't say any bad things are good, but the bad things might just end up crashing or hanging your program instead.

7.6.1 Definition

A **semi-decision procedure** for a set X is a program A such that, for each string $s \in \mathbb{S}$,

$$[\![A]\!](s) = \texttt{True} \quad \text{iff} \quad s \in X$$

But note that if s is *not* in X, $[\![A]\!](s)$ isn't guaranteed to return any value at all: it is just required *not* to return the value `True`. A set X is **semi-decidable** iff there is some semi-decision procedure for X.

It follows directly from the definition that every decidable set is semi-decidable. But as the next exercise shows, the converse does not hold.

7.6.2 Exercise

The set of programs that halt is semi-decidable. Thus there is a set which is semi-decidable, but not decidable.

7.6.3 Exercise

Uncountably many sets of strings are not even semi-decidable.

7.6.4 Exercise (Bounded Search)

Let $A(x, y)$ be a program which halts for every input. Use this to write another program $B(y, \texttt{bound})$ such that, for any strings t and u,

$[\![B]\!](t, u) = \texttt{True}$ if there is some string s which is no longer than u, such that $[\![A]\!](s, t) = \texttt{True}$.

$[\![B]\!](t, u) = \texttt{False}$ otherwise.

Hint. You can help yourself to a variable called `alphabet` whose value is a long string containing every symbol in the standard alphabet. Actually writing this out would require you to write a very long first line of your program:

```
alphabet = ”A” + ”B” + ”C” + ...
```

One elegant strategy for writing this program uses a *recursive* self-calling program. (We know that recursive self-calls can be eliminated in principle using Kleene's Theorem.)

7.6.5 Exercise (Unbounded Search)

Let $A(x, y)$ be a program which halts for every input. Use Exercise 7.6.4 to write another program $C(y)$ such that, for any string t,

> $\llbracket C \rrbracket(t) = \texttt{True}$ iff there is some string s (of any length) such that $\llbracket A \rrbracket(s, t) = \texttt{True}$.

If there is no such string, C does not have to return any value at all.

7.6.6 Exercise

Let $X \subseteq \mathbb{S} \times \mathbb{S}$ be a set of pairs of strings. Let

$$X^{\exists} = \{t \in \mathbb{S} \mid \text{there is some } s \in \mathbb{S} \text{ such that } (s, t) \in X\}$$

If X is decidable, then $X^{\exists}$ is semi-decidable.

Hint. Use Exercise 7.6.5.

Semi-decidability is closely linked to another idea. Some sets can be *listed*. The idea is that we can write a program that spits out each element of X one by one. One way to make this idea precise is with computable functions from the natural numbers. For a “listable” set X, we can take any number n and spit out an element of X, such that every element of X shows up for some number n. This is very similar to the idea of a *countable* set—which is a set which is the range of some function from natural numbers. But now we're not just interested in *arbitary* functions: what we want is a “counting function” which is a nice *computable* function.

If you can decide which things are in a set, then you can list it. If X is decidable, then one way to list its elements is to go through *every* string one by one in some fixed order, and for each string check whether it's in X. If it is, then spit it out, and if it isn't, then don't spit anything out, and go on to the next string.

But just because you can *list* a set doesn't guarantee that you can determine whether any particular thing is in it. You might try just going through the list looking for the thing you want. This half works. If the thing you want is in the list, then by going through the list one by one, eventually you'll find it, and you can return `True`. But if the thing you want *isn't* in the list, then you'll never find it. But at any point in your search you'll only have looked at finitely many things, so there's no point in your search where you know you never *will* find it, later on. So every effectively enumerable set is *semi*-decidable. But this doesn't mean that every effectively enumerable set is decidable.

We can make these ideas a bit more official.

7.6.7 Definition

A set of strings $X \subseteq \mathbb{S}$ is **effectively enumerable** iff X is the range of some computable total function.

7.6.8 Theorem

If X is effectively enumerable, then X is semi-decidable.

Proof

If X is effectively enumerable, then X is the range of some computable total function f. That is to say,

$$X = \{t \in \mathbb{S} \mid \text{there is some } s \in \mathbb{S} \text{ such that } f\,s = t\}$$

But also, if f is a computable total function, then the set $Y = \{(s,t) \in \mathbb{S} \mid f\,s = t\}$ is *decidable*—just calculate $f\,s$, and then check whether the result is the same string as t. Since $X = Y^{\exists}$, using Exercise 7.6.6 we can conclude that X is semi-decidable. □

We can also show that this works the other way around: every semi-decidable set is effectively enumerable. But this direction takes significantly more work to officially prove.

7.6.9 Theorem

If X is semi-decidable, then X is effectively enumerable.

Proof Sketch

Suppose that X is semi-decidable: this means we have some program that returns `True` just for inputs that are in X. We'll use this to show that X is effectively enumerable.

Here's the basic idea. We can go through strings one by one in some fixed order. The obvious thing to try is to check each string, and print it out if we get `True`. The problem with this approach is that the semi-decision program might go into an infinite loop. The first time this happens, the whole program will stop working, which means we'll never get to strings that come later in the list. So we need to make sure we never allow the semi-decision program to go on forever.

Here's how we can do this. We can run a *modified* program, which replaces each `while` loop with a `for` loop that only runs n times, for some number n, and returns `Fail` if the loop-ending condition still hasn't been met at that point. Call this the **n-bounded variant** of a program. If a program halts, then each of its `while` loops only goes through finitely many steps, which means there is *some* number n such that the n-bounded program succeeds.

So here's what we can do. We can go through the *pairs* (s, n) of a string and a number, one by one. For each pair, we'll try to run the n-bounded semi-decision program with input s. If we get `True`, then we print out `s`. If we get `False` or `Fail` then we don't print out `s` (yet) and we go on to the next pair. Because we are using bounded programs, the computation we do for each pair can only take finitely many steps. So we'll eventually reach every pair, and so eventually every string that the semi-decision program returns `True` for will get printed out. □

7.6.10 Exercise
Suppose that X is a decidable set, and Y is a subset of X. Suppose furthermore that Y and $X - Y$ (the set of strings in X but not in Y) are *both* semi-decidable. Then Y is decidable.

```
TODO. This exercise is probably too hard. I would like to refactor
this section again.
```

7.7 Decidability and Logic

Now that we have come to grips with the fundamental ideas of computability, we can apply these ideas to some important questions in logic.

Here's a common problem. You have some premises that you take to be true, and you want to know whether a certain conclusion logically follows from them. In other words, given some *axioms*, we want to know whether a certain sentence is a *theorem*. This is a task philosophers face all the time, as they are trying to figure out how certain philosophical conclusions fit together with various philosophical starting points. It's an even larger part of what mathematicians do. The question of which conclusions follow from which premises is at least somewhat important in essentially every field of inquiry, and it is often very tricky to answer.

Part of Leibniz's distinctive rationalist vision was that *all* fields of inquiry could be reduced to the problem of determining what follows from what. He wrote:

> The only way to rectify our reasonings is to make them as tangible as those of the Mathematicians, so that we can find our error at a glance, and when there are disputes among persons, we can simply say: Let us calculate, without further ado, to see who is right. [CITE "Art of Discovery" 1685, trans. Wiener.]

Leibniz imagined that "reasoning in morality, physics, medicine, or metaphysics" could be reduced to the problem of determining what logically follows from what. And he thought that solving the problem of what logically follows from what was a matter of mere calculation—and so, in principle, *every* question could be systematically answered.

In 1928, the mathematicians David Hilbert gave a challenge to the world. Can you give a general, systematic procedure that can take any statement in first-order logic, and determine whether or not it is a logical truth? If you can do this, you can also solve the more general problem: given any finite set of premises X which are formalized in first-order logic, and given any other first-order sentence A, determine whether A is a logical consequence of X. If we could do this, then we would have a general purpose tool for determining which arguments are valid, as long as we know how to formalize those arguments in first-order logic. This would be extremely handy! This problem is called Hilbert's *Entscheidungsproblem* (which is German for "decision problem").

Unfortunately, Hilbert's challenge can't be met. Like the problem of determining which programs have infinite loops, the problem of deciding which arguments are logically valid in first-order logic is *effectively undecidable*. This fact is called Church's Theorem—and we will prove it now.

The important idea is that we can link up the key concept of this chapter—*computability*—with the two key concepts of the last chapter—*definability* and *representability*. What we have to do is connect *programs* to *formulas*. For every program, there is a corresponding formula in the first-order language of strings that precisely describes what that program does. Once we've made these connections, the exciting results will basically follow as simple consequences of Tarski's Theorem from Chapter 6.

The basic idea is very similar to the idea of the Universal Program. We will explicitly represent the state of a program—that is, an assignment function—using a string. Then we will use formulas to describe what each kind of statement in our programming language does. That is, for each step of a program, we can describe the relationship between its "input" and "output" assignments using first-order logic.

We have already discussed how to represent an assignment function as a sequence of strings in Section 7.4, and also how to represent a sequence of strings with a single string in Section 6.2. One thing we'll need to do is come up with expressions in first-order logic that do the same work as some of the programs we discussed earlier.

7.7.1 Exercise

Recall from Section 7.4 that we can represent a (finite) assignment function as a sequence of key-value strings. Thus we can represent an assignment using a single string, using the idea in Section 6.2 for representing sequences of strings. Show that the following functions are *definable in* $\mathbb{S}$, with respect to this representation:

(a) The function that takes each assignment function g and variable x to its value gx.

(b) The function that takes each assignment function g, variable x, and string s, to the updated assignment function $g[x \mapsto s]$.

Hint. Back in Section 6.2 we showed that certain sequence operations are definable in $\mathbb{S}$. It will be helpful to use some of those facts.

7.7.2 Exercise

Show that for each Py-term t, the corresponding function that takes each assignment function g to its denotation $[\![t]\!]g$ is definable in $\mathbb{S}$.

By the Church-Turing Thesis, we can assume that every computable function is the extension of some Py-program. So to show that every computable function is definable in $\mathbb{S}$, we just have to show that every *Py-program* has a definable extension. And we can show *this* by induction on the structure of programs. That is, we can prove that every computable function is definable in $\mathbb{S}$ by showing three things:

1. The denotation of the empty program is definable.

2. If the denotation of A is definable, so is the denotation of

```
$t_1$ = $t_2$
$A$
```

3. If the denotations of A and B are definable, so is the denotation of

```
while $t_1$ != $t_2$:
    $A$
$B$
```

The trickiest part is step 3. Recall from Definition 7.2.4 that definition of the denotation of a `while` block uses the idea of a *finite loop sequence*. For terms t_1 and t_2 and a program A, $(g_0, \dots, g_n)$ is a finite loop sequence iff the following three conditions hold:

$$\begin{aligned} g_{i+1} &= [\![A]\!]\, g_i && \text{for each } i < n \\ [\![t_1]\!]\, g_i &\neq [\![t_2]\!]\, g_i && \text{for each } i < n \\ [\![t_1]\!]\, g_n &= [\![t_2]\!]\, g_n \end{aligned}$$

The denotation $[\![A]\!]\, g$ is the last element of a finite loop sequence whose first element is g, if there is one.

7.7.3 Exercise

Let t_1 and t_2 be Py-terms, and let A and B be programs. Suppose the denotations of A and B (that is, the functions $[g \mapsto [\![A]\!]\, g]$ and $[g \mapsto [\![B]\!]\, g]$) are each definable in $\mathbb{S}$.

(a) The set of finite loop sequences for t_1, t_2, and A is definable in $\mathbb{S}$.

(b) The function that takes each assignment g to the denotation

$$\left[\!\!\left[\begin{array}{l} \texttt{while } t_1 \texttt{ != } t_2\texttt{:} \\ \quad\; A \\ B \end{array}\right]\!\!\right] g$$

is definable in $\mathbb{S}$.

7.7.4 Exercise (The Definability Theorem)

Given the Church-Turing Thesis, every computable function is definable in the standard string structure $\mathbb{S}$.

7.7.5 Exercise
Use Exercise 7.7.4 to show that every decidable set of strings is definable in the string structure $\mathbb{S}$.

This fact has several important applications. For instance, we can use it to show the definability of certain functions we discussed in Chapter 5—like substitution, the labeling function, and translation functions. To show they're definable, we just need to show that they're *computable*. And to show this, we just need to describe some systematic algorithm for computing them. For this, their standard recursive definitions are pretty much already enough.

We can also combine the Definability Theorem with what we showed in the last chapter about *undefinable* sets, in order to derive another important result about *undecidability*.

7.7.6 Exercise
The set of true first-order sentences in the string structure, $\operatorname{Th} \mathbb{S}$, is undecidable.

7.7.7 Exercise
Show that the set of programs that halt is definable in the structure $\mathbb{S}$. So there are sets of strings which are definable but undecidable.

In fact, we can strengthen these results. We don't really need the *whole* theory of $\mathbb{S}$ to describe computable functions. Just a pretty small simple piece of this theory is enough.

Let's start by recalling some definitions from Section 6.7. Each string $s \in \mathbb{S}$ has a *canonical label*, which is a term $\langle s \rangle$ in the language of strings. If f is a partial function from $\mathbb{S}$ to $\mathbb{S}$, then a theory T **represents** f with a term $t(x)$ (possibly using definite descriptions) iff, for each s for which f is defined,

$$t\langle s \rangle \underset{T}{\equiv} \langle f\, s \rangle$$

Intuitively, this means that the theory T "knows" the correct value of f for each input string s.

The next thing to remember is the *minimal theory of strings* S. This is a finitely axiomatizable theory (Definition 5.4.3), so it is much simpler than the full theory of the string structure $\operatorname{Th} \mathbb{S}$. Even so, it is powerful enough to describe lots of things. Anything which is *definable* in $\mathbb{S}$ using a *syntactically simple enough* expression is also *representable* in S. To use the technical term introduced in Section 6.8: anything that is Σ_1-*definable* in $\mathbb{S}$ is representable in S. A Σ_1-formula consists of a formula that uses only *bounded* quantifiers— quantifiers that are restricted to strings with a certain maximum length—plus just one unbounded existential quantifier out front.

We can use this fact to prove a stronger generalization of the Definability Theorem:

7.7.8 The Representability Theorem
The minimal theory of strings S represents every computable function.

To prove the Definability Theorem, we showed that lots of different functions are definable in $\mathbb{S}$—functions that pick out elements of sequences, update assignments, and so on. We can prove the Representability Theorem by working our way back through this proof, and checking that at each step we can get by using *syntactically simple enough* expressions—that is, just Σ_1 formulas. We won't work through this in detail. (If you have worked through the "starred" section Section 6.8 and the proof of the Definability Theorem, then you have everything you need to do it yourself.) But let's think about an intuitive explanation for why this really *should* work out.

The basic idea is that, even though the theory S doesn't include *all* the truths about strings, it does include all of the "basic" truths, about what any particular string is like internally. For example, for any particular string s, the theory S knows what substrings s has, or how long it is, or whether it contains an `A` somewhere before a `B` , and so on. Furthermore, it turns out these kinds of "basic" truths about strings are enough to pin down the behavior of computable functions. Suppose A is a program that returns some value $[\![A]\!](s)$ for an input string s. Then it turns out that the value of $[\![A]\!](s)$ is determined by what *one specific string* is like. There is a string that represents the whole finite sequence of "states" (that is, assignment functions) that A steps through, starting with [`x` $\mapsto s$]. Call this sequence of assignments an *A-computation sequence*. (This is an extension of the idea of a *finite loop sequence* that we considered earlier.) We can verify that a string has all the right features to represent an A-computation sequence just by examining its internal structure—ignoring the rest of the infinite universe of alternative strings. And as we noted, these "internal" facts about a specific string are the sort of thing that the theory S can verify all on its own. More specifically, we can formalize the property of being an A-computation sequence using a *bounded* formula—we don't need to look at any strings longer than the string that represents the computation sequence itself. Similarly we can formalize the property of being the first or last element of such a sequence using another bounded formula. Then to represent the relation $[\![A]\!](s) = s'$, we can use a formula that says this:

> There is some string that represents an A-computation sequence whose first element is $[x \mapsto s]$, and whose last element is an assignment h such that $h(\texttt{result}) = s'$.

This has just one unbounded existential quantifier.

That is all we'll say here about the proof of the Representability Theorem. From here on, we will take the theorem to be established, and use it to show some other important things.

Here's one immediate application. Remember that in Chapter 6 we used the fact that the minimal theory of strings S represents *syntax*: that is, it represents the substitution function and the labeling function. We can now get these facts as corollaries of the Representability Theorem. It is clear that these functions are *effectively computable*. Indeed, the recursive definitions themselves already amount to an effective method for calculating the results of substitution and labeling. Thus, by the Church-Turing Thesis these functions are *Py*-computable, and thus by the Representability Theorem they are representable in S.

7.7.9 Exercise
Use the Representability Theorem to show that, if X is a decidable subset of $\mathbb{S}$, then the minimal theory of strings S represents X.

Recall that a "sufficiently strong" theory is one that interprets the minimal theory of strings S (or alternatively, the theory of minimal arithmetic Q).

7.7.10 Exercise
Any sufficiently strong theory represents every decidable set.

7.7.11 Exercise (The Essential Undecidability Theorem)
No sufficiently strong consistent theory is decidable.

Hint. Use Tarski's Theorem.

7.7.12 Exercise (Church's Theorem)
The set of first-order logical truths (in the language of strings) is undecidable.

Hint. Here are two useful facts to bear in mind. First, the theory S is finitely axiomatizable. Second, if $A_1, \ldots, A_n$ is some finite list of sentences, then the function that takes each sentence B to the sentence

$$(A_1 \wedge \cdots \wedge A_n) \to B$$

is computable.

Church's Theorem shows that Hilbert's general "decision problem" is impossible. There is no general systematic way to decide which statements are logical consequences of a given set of axioms.

The Essential Undecidability Theorem, which we used to prove Church's Theorem, is also going to be very important in Chapter 8, so take a bit of time to meditate on what it says. Take any theory T that is strong enough to describe some basic string operations (or a bit of basic arithmetic) but *not* so strong that it includes logical contradictions. Then there is no general systematic method, even in principle, to determine what exactly T says. To put it another way, there is no decidable theory anywhere "in between" the minimal theory of strings (or if you prefer, the minimal theory of arithmetic) and the inconsistent theory. In particular, the minimal theory of strings itself is undecidable (it is a sufficiently strong consistent theory). But furthermore, in this sense it is *essentially* undecidable.

This amounts to a refutation of Leibniz's rationalist vision. *Even if* all questions in "morality, physics, medicine, or metaphysics" can be reduced to questions of logic, this would not make answering them a matter of mere "calculation"—because questions of logic are effectively undecidable.

Chapter 8

The Unprovable

> Thus they argued, and intended to go on, but the Empress interrupted them: I have enough, said she, of your chopped logic, and will hear no more of your syllogisms; for it disorders my reason, and puts my brain on the rack; your formal argumentations are able to spoil all natural wit.
>
> Margaret Cavendish, *The Blazing World* (1668)

So far we've been thinking about logic in terms of *structures*: A is a logical consequence of X iff A is true in every structure where each sentence in X is true. To put it another way, a logically valid argument is one with no *counterexamples*, where a counterexample is a structure where the premises are true and the conclusion is false. We'll now look at a different approach to logic, which instead uses the idea of a *formal proof*. A formal proof builds up a complicated argument by chaining together very simple steps. The basic steps are chosen so that they are very closely connected to the basic roles of our logical connectives. Because of this, many people have thought that proofs are in some sense conceptually more basic than structures.

One of the central facts about first-order logic is that these two different ways of thinking about logic perfectly line up. An argument from premises X to a conclusion A has a proof if and only if it has no counterexamples. (This is called **Soundness and Completeness**.) This fact is important because it lets us go from facts which are obvious about provability to corresponding facts about structures which are less obvious, and vice versa. For instance, it will be obvious from the way we build up proofs that no proof relies on infinitely many premises. From this we can deduce the less obvious fact that no *logical consequence* essentially relies on infinitely many premises. (This is called the **Compactness Theorem**.) Similarly, we can show that a certain argument is not logically valid by coming up with a specific counterexample. From this we can deduce the less obvious fact that the argument has *no proof*.

We can also combine provability with the other ideas we've been exploring. A key fact about our proof system—and indeed, any reasonable system of proofs that a finite being could use to establish results—is that the question of what counts as a correct proof of a certain conclusion is *effectively decidable*. This basic fact, together with the things we have already established about undecidability in Chapter 7, has deep and important consequences. First, we can show that the set of logical truths is effectively enumerable—basically, because proofs are the sort of thing we can systematically list one by one. (This means, in light of Church's Theorem (Exercise 7.7.12), that the set of logical truths is another example of a set that is semi-decidable, but not decidable.) More generally, consider any "reasonably simple" theory: a theory that consists of just the logical consequences of some effectively decidable set of *axioms*. Any theory like this is also effectively enumerable. But this leads us directly to Gödel's First Incompleteness Theorem: *no* theory is "reasonably simple", sufficiently strong, consistent, and complete. Notice in particular that the set of *truths* is sufficiently strong, consistent, and complete (in all but the most impoverished languages); so it follows that the truth cannot be simple. There is no hope, for example, for a rationalist project of writing down elegant axioms from which all truths can be systematically derived. (That is—systematically derived by *finite beings*. Perhaps, as Leibniz believed, God can know some truths by way of *infinite* proofs, which are not covered by this theorem.)

8.1 Proofs

A proof is an expression in a formal language: a string of symbols built up systematically from certain basic pieces using certain rules. In this respect, proofs are just like terms, formulas, and programs. Just like we did with those other formal languages, we will give a recursive definition of the structure of proofs, which will specify some "basic" proof steps and some rules for putting them together. Since the point of a formal proof is to make it very clear and easy to check that a conclusion follows from some premises, there shouldn't be too many different proof rules, and no particular rule should be too complicated. Even so, proofs are our most complicated formal language so far: they are built up from formulas, which are already a bit complicated, and there are multiple proof rules for each one of the basic logical connectives we use to build up formulas (`∧` , `¬` , `=` , and `∀`). So we'll take it slow.

There are many different formal proof systems for first-order logic, which make different trade-offs. We'll use what's called a *natural deduction* system. The key feature of natural deduction systems is that they let us make intermediate suppositions in our proofs—the kind of step that we express in our ordinary informal proof using the word "suppose". We do this when we use the technique of proof by contradiction. Here's a classic example—the reasoning of Russell's Paradox:

> Suppose *x* is a set such that for any *y*, *y* is an element of *x* iff *y* is not an element of *y*. So, in particular, *x* is an element of *x* iff *x* is not an element of *y*. We can derive a contradiction from this claim. First, suppose that *x* is an element of *x*. In that case, by the claim, *x* is not an element of *x*. This is a contradiction, so it follows that *x* is not an element of *x*. But in that case the claim implies that *x* *is* an element of *x*. This is a contradiction again. So the claim must be false. This shows that there is no set *x* such that, for any *y*, *y* is an element of *x* iff *y* is not an element of *y*.

In a natural deduction system, the formalized version of the proof has basically the same structure as the informal proof. It's a bit more austere, a bit more repetitive, and some steps get rearranged a bit. It looks like this.

```
for arbitrary x:
    suppose ∀y (y ∈ x ↔ ¬(y ∈ y)):
        suppose x ∈ x:
            x ∈ x                   Assumption

            ∀y (y ∈ x ↔ ¬(y ∈ y))   Assumption
            x ∈ x ↔ ¬(x ∈ x)        ∀Elim
            x ∈ x                   Assumption
            ¬(x ∈ x)                ↔Elim
        ¬(x ∈ x)                    Reductio
        ∀y (y ∈ x ↔ ¬(y ∈ y))       Assumption
        x ∈ x ↔ ¬(x ∈ x)            ∀Elim
        x ∈ x                       ↔Elim

        suppose x ∈ x:
            x ∈ x                   Assumption
            ∀y (y ∈ x ↔ ¬(y ∈ y))   Assumption
            x ∈ x ↔ ¬(x ∈ x)        ∀Elim
            x ∈ x                   Assumption
            ¬(x ∈ x)                ↔Elim
        ¬(x ∈ x)                    Reductio
    ¬∀y (y ∈ x ↔ ¬(y ∈ y))          Reductio
∀x ¬∀y (y ∈ x ↔ ¬(y ∈ y))           ∀Intro
```

The main difference from the informal version is that we have formalized all of the logical connectives, and we have cut out almost all of the other words. We use indentation to help keep the structure of the proof clear without transition words like "in that case." Another difference is that our official proof syntax is a bit more restrictive than you may be used to when it comes to *order* you put the steps of the proof in. (The reason for this is just because it makes our official definitions simpler, which will make it a little easier to prove things *about* formal proofs.) Each rule has to come immediately below the parts of the proof that it relies on. This also has the effect of making our proofs extra repetitive. For example, in this proof, the subargument for `¬(x ∈ x)` is repeated twice. That's because this conclusion is really used twice in the proof: once on the way to the first conclusion `x ∈ x`, and a second time to state the contradictory conclusion `¬(x ∈ x)`. Our official proof syntax will require us to prove things over again each time we use them.

In practice, a convenient notational shorthand is to leave out redundant subproofs, and instead just refer to the line number where we originally derived that step. (We have to be careful not to cheat, though, and refer to conclusions from within `Reductio` subproofs that wouldn't actually be correct in the current context.) Using this convention a lot lets us rewrite the formal proof in a way which is much more concise, and maybe a bit less alien-looking.

```
1  for arbitrary x:
2      suppose ∀y (y ∈ x ↔ ¬(y ∈ y)):
3          suppose x ∈ x:
4              x ∈ x ↔ ¬(x ∈ x)        ∀Elim (2)
5              ¬(x ∈ x)                ↔Elim (3, 4)
6          ¬(x ∈ x)                    Reductio (3, 5)
7          x ∈ x ↔ ¬(x ∈ x)            ∀Elim (2)
8          x ∈ x                       ↔Elim (6, 7)
9      ¬∀y (y ∈ x ↔ ¬(y ∈ y))          Reductio (8, 6)
10 ∀x ¬∀y (y ∈ x ↔ ¬(y ∈ y))           ∀Intro
```

Another detail is that `↔Elim` is not really one of the *basic* rules of our system—indeed, `↔` is not officially one of our basic connectives. So what we have written here is an abbreviation of the full official proof, which would spell out the biconditional using `∧` and `¬`, and derive the rule of `↔` from the corresponding basic proof rules for those connectives. We'll see how this works very soon.

Proof systems that don't allow intermediate assumptions are called "Hilbert-style" systems. The main advantage of natural deduction proofs over Hilbert-style proofs is that they are more intuitive to read and write. The main disadvantage is that natural deduction proofs are a bit more structurally complex than Hilbert-style proofs. A natural deduction proof isn't just a "flat" list of statements: it has interesting syntactic structure. But by this point we have plenty of experience handling complex syntax.

Our proof system has twelve rules. We can group them into five families—one family for each basic logical connective (`∧` , `¬` , `=` , and `∀`) plus a few extra "structural" rules for putting pieces together. We'll start by taking a quick informal tour of these rules and how to use them, after which we'll give an official definition that summarizes them.

The main point of a proof is to show that a certain conclusion follows from certain premises—in particular, that the conclusion is *provable* from the premises. If X is set of formulas and A is a formula, the notation $X \vdash A$ means that the conclusion A is provable from premises in X. We use the same notational shortcuts for the "single turnstile" notation for provability as we have been using for the "double turnstile" notation for logical consequence. For instance, $X, A, B \vdash C$ means the same thing as $X \cup \{A, B\} \vdash C$. The official definition of provability will come later—after we have gone through all the pieces of the definition of proofs. But we will be able to show lots of things about provability before we get that far, as we build up some particular examples of formal proofs. (This is just like how we could go ahead and show certain things about *decidability* long before we had finished our full official definition of *programs*.)

Assumption

The simplest kind of proof just asserts something we already know—either because it is one of our premises, or because we have supposed it for reductio. We call this rule `Assumption` .

8.1.1 Example

For any formula A, and any set of formulas X,

$$X, A \vdash A$$

Proof

```
A    Assumption
```

Obviously we can't do very much with the `Assumption` rule all by itself. But we usually need it to get a more complicated proof started.

Something to notice right away is that a proof of A from some premises X doesn't actually have to explicitly use all of those premises. It's okay to just ignore some premises. (In this way, classical logic is unlike *relevant logic,* which does require every premise to actually play a role in the argument.) So, for example, consider the proof

```
∀x (suc x ≠ 0)  Assumption
```

This counts as a proof of `∀x (suc x ≠ 0)` from just the premise `∀x (suc x ≠ 0)` itself, but it also counts as a proof of this conclusion from *bigger* premise sets that include this particular formula—for example, the set of all the axioms of minimal arithmetic.

Conjunction Rules

Next we have some rules for reasoning about conjunction. The ideas are simple. If we have proved A and B, then we can deduce the conjunction `(`A`∧`B`)`. We call this rule Conjunction Introduction, or `∧Intro` for short. For example:

```
1 + 0 = 1                Assumption
1 ≠ 0                    Assumption
(1 + 0 = 1) ∧ (1 ≠ 0)    ∧Intro
```

In general, to prove A`∧`B, first we write down a proof of A, then we write down a proof of B, and finally we deduce the conjunction.

8.1.2 Example

For any formula A,

$$A \vdash A \wedge A$$

```
Proof
A          Assumption
A          Assumption
A ∧ A       ∧Intro
```

Something to notice is that our official proof repeats the assumption *twice*. We have a proof of A, and then *another* proof of A (though of course really it's the same proof) and finally we deduce the conjunction. In practice, this is the sort of thing we might abbreviate, like this:

```
1  A          Assumption
2  A ∧ A       ∧Intro (1, 1)
```

This indicates that we're using the same subproof (the one ending at line 1) twice over. □

Another basic proof rule is that, if we have proved $(A \wedge B)$, then we can deduce A. Likewise, if we have proved $(A \wedge B)$, we can deduce B. These two rules are called `∧Intro1` and `∧Intro2` . (Notice that we really need *two* rules here. As always, even though $(A \wedge B)$ and $(B \wedge A)$ are logically equivalent, that doesn't make them the same formula.)

8.1.3 Example

For any formulas A and B,

$$A \wedge B \vdash B \wedge A$$

```
Proof
1  A ∧ B        Assumption
2  B              ∧Elim2 (1)
3  A              ∧Elim1 (1)
4  B ∧ A        ∧Intro (2, 3)
```

If we spell out the repetitive steps explicitly, it looks like this:

```
A ∧ B       Assumption
B            ∧Elim2
A ∧ B       Assumption
A            ∧Elim1
B ∧ A       ∧Intro
```

8.1.4 Example

For any formulas A, B, and C,

$$A \land (B \land C) \vdash (A \land B) \land C$$

Proof
Notice that in the officially spelled out proof, the premise $A \land (B \land C)$ gets used three times, and so there are three repeated `Assumption` lines.

```
A ∧ (B ∧ C)   Assumption
A                ∧Elim1
A ∧ (B ∧ C)   Assumption
B ∧ C           ∧Elim2
B                ∧Elim1
A ∧ B           ∧Intro
A ∧ (B ∧ C)   Assumption
B ∧ C           ∧Elim2
C                ∧Elim2
(A ∧ B) ∧ C   ∧Intro
```

Notice the structure of this proof. Before the last line

```
(A ∧ B) ∧ C   ∧Intro
```

there are two subproofs: first, a proof of $(A \land B)$, and second, a proof of C. Similarly, the line

```
A ∧ B           ∧Intro
```

comes after two subproofs; first, a proof of A, and then a proof of B. □

The rules for conjunction follow a pattern. We have one *introduction rule*, which lets us derive a conjunction as a conclusion. We also have two *elimination rules*, which let us use a conjunction as a premise to derive something else. This pattern is typical: we will also have introduction and elimination rules for other logical connectives, like `=` and `∀x`. (Negation is a bit special, though.)

8.1.5 Exercise
For any formula A,

$$A \land A \vdash A$$

Negation Rules

Our tool for "proving a negative" is proof by contradiction, also called *reductio ad absurdum*, or `Reductio` for short. To prove not-A, suppose A, and then derive a contradiction from this supposition.

8.1.6 Example
For any formulas A and B,

$$\neg A \vdash \neg(A \land B)$$

```
Proof
suppose A ∧ B:
    A ∧ B    Assumption
    A          ∧Elim1
    ¬A         Assumption
¬(A ∧ B)    Reductio
```

In this proof, we add an extra assumption, $A \land B$. We use this assumption to derive a contradiction: that is, first we have a subproof for A, and then we have another subproof for $\neg A$. Then we can conclude that our original assumption was false.

In general, `Reductio` looks like this. To get a proof of ¬A, first we write down `suppose` A**:** . Then we write down two subproofs: a proof of some formula B (this can be any formula we want), and a proof of its negation ¬B. These subproofs can use A as an `Assumption` step wherever we want (in addition to any other assumptions we already had available). Finally, we write down our conclusion, ¬A by `Reductio` . Here is the template:

```
suppose A:
    ⋮
    B
    ⋮
    ¬B
¬A        Reductio
```

We'll state the rules more rigorously in Section 8.2.

An alternative label for `Reductio` is `¬Intro` (following the same `Intro` / `Elim` naming pattern as conjunction). Feel free to use it if you prefer. But I'll stick with the traditional medieval name.

We don't have a proof rule that lets us derive conclusions from an arbitrary negated premise. Instead, we have *double*-negation elimination, or `¬¬Elim` for short. If we have a proof of ¬¬A, then underneath it we can write the line

```
A    ¬¬Elim
```

8.1.7 Exercise (Explosion)

$A, \neg A \vdash B$.

Remember, officially our language only includes the connectives ∧ and ¬ . Formulas using other connectives, like → and ∨ , are officially considered to be abbreviations of formulas using ∧ and ¬ . Similarly, we will only officially have basic *proof rules* for the connectives ∧ and ¬ . But we can use the definitions of these other connectives to derive their standard proof rules, as well.

8.1.8 Example (Modus Ponens)

For any formulas A and B,

$$A, A \to B \vdash B$$

Proof
Recall that we defined the conditional $(A \to B)$ to be an abbreviation for $\neg(A \wedge \neg B)$. So what we want to show is

$$A, \neg(A \wedge \neg B) \vdash B$$

□

We can show this by providing a formal proof.

```
suppose ¬B:
    A                 Assumption
    ¬B                Assumption
    A ∧ ¬B           ∧Intro
    ¬(A ∧ ¬B)        Assumption
¬¬B                   Reductio
B                     ¬¬Elim
```

8.1.9 Exercise (Modus Tollens)
For any formulas A and B,

$$A, A \to \neg B \vdash \neg A$$

8.1.10 Exercise (Disjunction Introduction)
For any formulas A and B,

$$A \vdash A \vee B$$

$$B \vdash A \vee B$$

(Recall that $A \vee B$ is officially an abbreviation for $\neg(\neg A \wedge \neg B)$.)

It's also useful to show some relationships between different provability facts. For example:

8.1.11 Example (Conditional Proof)
If $X, A \vdash B$, then $X \vdash A \to B$.

(This is also sometimes called the Deduction Theorem.)

Proof
Suppose that P is a proof of B from the premises $X \cup \{A\}$. We want to use P to build up a more complex proof of $(A \to B)$ which only relies on the premises X.

Remember, $(A\to B)$ is officially an abbreviation for $\neg(A\land\neg B)$. So we can schematically put together a proof like this.

```
suppose A ∧ ¬B:
    A ∧ ¬B     Assumption
    A           ∧Elim1

    suppose A:
        P     # This is a proof of B from X and A

        A ∧ ¬B Assumption
        ¬B      ∧Elim2
    ¬A          Reductio
¬(A ∧ ¬B)      Reductio
```

Notice in particular that this proof does not rely on the assumption A: this assumption is available within the inner `Reductio` subproof, but not outside of it. So this is a proof of $\neg(A\land\neg B)$ that just relies on the premises X. (The whole proof uses these premises because the subproof P uses them and they have not been caught be any `suppose` line.) □

8.1.12 Exercise (Contraposition)
$X, A \vdash B$ iff $X, \neg B \vdash \neg A$.

Identity Rules

We also have an introduction rule and an elimination rule for the identity symbol `=`. The introduction rule says that we can always prove a thing is identical to itself (from no premises). That is, we can always write a one-line proof of $a=a$, where a is any term. The pattern-following name for this is `=Intro`, and the traditional name is simply `Identity`. (Feel free to use either one.)

The elimination rule says (putting it a bit roughly) that if we know a and b are the very same thing, and we have also proved that a has a certain property, then we can conclude that b has the property as well. Our more official version doesn't say anything about *properties*, though: instead we do it by substituting the terms a and

b into a certain formula.) If we have a proof of $a{=}b$ and a proof of $A(a)$, then we can stick those proofs together and then deduce $A(b)$. This is called either `=Elim` or `Leibniz’s Law` .

8.1.13 Example
For any terms a and b,

$$a{=}b \vdash b{=}a$$

Proof

$a = b$	`Assumption`
$a = a$	`=Intro`
$b = a$	`=Elim`

In this proof, we have a one-line subproof of $a = b$, then another one-line subproof of $a = a$. We can think of this second proof as saying that a has the property of being idential to b—so the last line concludes that b also has this property. More officially, what’s going on is that the second line is the result of plugging a into the formula $\mathtt{x} = a$, and the third line is the result of plugging b into that same formula. □

8.1.14 Exercise (Euclid’s Property)
For any terms a, b, and c,

$$a{=}b, a{=}c \vdash b{=}c$$

Quantifier Rules

Finally, the universal quantifier also has an introduction rule and an elimination rule. Let’s consider the elimination rule first, because it’s easier. If we know that *everything* has a certain property, then we also know that each particular thing has that property. Again, our official version of the rule doesn’t say anything about “properties”, and uses substitution instead. Given $\forall x\ A(x)$, we can deduce $A(a)$.

$$\texttt{∀Elim} \quad : \quad \forall x\ A \vdash A[x \mapsto a]$$

8.1.15 Exercise (Existential Generalization)
For any term a and formula $A(x)$,

$$A(a) \vdash \exists x \; A(x)$$

(Recall that $\exists x \; A(x)$ is officially an abbreviation for $\neg\forall x \; \neg A(x)$.)

The final rule is the subtlest. First, we should call attention to something that has been in the background so far. In our proof system, the steps of a proof can include *free variables*—they don't have to be whole sentences. For example, this is a perfectly fine proof.

```
x = 0 + x    Assumption
x > 0        Assumption
0 + x > 0    =Elim
```

Here we have used the free variable `x` and the open term `0 + x` as our terms a and b for an application of Leibniz's Law (`=Elim` , plugging each of these terms into the formula `y > 0`). Variables, and terms that include variables, can be used just like any other terms in our proofs.

It might seem odd to allow this, but it actually reflects an important aspect of our informal proofs. Remember the example we considered earlier—the reasoning of Russell's paradox. It looked like this.

> Suppose x is a set such that y, y is an element of x iff y is not an element of y. So [more reasoning here, where we derive a contradiction from this assumption]. This shows that there is no set x such that, for any y, y is an element of x iff y is not an element of y.

We were trying to prove a certain generalization: there is *no* set with a certain property. (We could formalize this "no" claim as a universal generalization: `∀x ¬∀y (y ∈ x ↔ ¬(y ∈ y))` .) In order to do it, we introduced an informal *variable* with the statement "Let x be a set". We then went on to prove things "about x"—that is, we made a bunch of statements that used that variable. But the variable isn't meant to stand for any particular thing, the way a name would. (Indeed, we show in the end that there *isn't* anything with the property we are supposing. It isn't as if x were a name for a non-existent Russell-set.) It's really a hard philosophical problem to say exactly what the variable `x` means in this kind of reasoning.[1] But in any case we can understand why the reasoning is correct:

[1] For example, see Breckenridge and Magidor (2012).

what we are showing is that x has certain properties, given certain assumptions, *no matter what x might be*.

Our formalization of this reasoning looks like this.

```
for arbitrary x:
    suppose ∀y (y ∈ x ↔ ¬(y ∈ y)):
        ⋮    # This is where we derived a contradiction
    ¬∀y (y ∈ x ↔ ¬(y ∈ y))          Reductio
∀x ¬∀y (y ∈ x ↔ ¬(y ∈ y))           ∀Intro
```

(The systematic name for this rule is `∀Intro`. Its traditional name is `Universal Generalization`.)

In this argument, we consider an *arbitrary* thing x. We then go on to prove that this arbitrary x does not have the Russell-set-property, and so we can conclude that *nothing* has the Russell-set-property—that is, there is no Russell set.

What does it mean for x to be "arbitrary"? In our formal proofs, what it means is that we don't rely on any special assumptions about what x is like. The key feature that lets us generalize in the last step is that the subproof within the "for arbitrary x" bit does not rely on any assumptions in which x is a free variable.

(This constraint is a little bit subtle. We *can* have x as a free variable in an `Assumption` line within that subproof, if it's an assumption we've introduced for `Reductio`. But we can't use any assumptions about x that we bring in "from outside.")

The general form of the rule looks like this:

```
for arbitrary y:
    ⋮
    A(y)
∀x A(x)      ∀Intro
```

The inner subproof should prove $A(y)$ without relying on any assumptions that include y as a free variable. The details of this rule are explained more precisely in Section 8.2.

8.1.16 Exercise

(a) $\vdash \top$.

(b) $\bot \vdash A$, for any formula A.

(Recall that $\top$ is an abbreviation for the standard truth, `∀x (x = x)`, and $\bot$ is an abbreviation for the standard falsehood $\neg\top$.)

8.1.17 Exercise (Change of Variables)

For any variables x and y, and for any formula $A(x)$ in which y does not occur free,

$$\forall x\ A(x) \vdash \forall y\ A(y)$$

8.1.18 Exercise (Existential Instantiation)

Suppose x is not free in B or in any formula in X. Then,

$$\text{If} \quad X, A(x) \vdash B \quad \text{then} \quad X, \exists x\ A(x) \vdash B$$

This fact corresponds to a kind of reasoning we've often used in our informal proofs. Suppose we know that there is something with the property A. Then we can "give it a name"—we suppose in particular that x has the property A. The sequent $X, A(x) \vdash B$ corresponds to reasoning that uses the assumption that x, in particular, is one of the A's. The name we choose had better be "arbitrary", in the sense that we haven't made any other assumptions about x already. If we can draw a conclusion B that doesn't say anything specifically about x, then that conclusion *also* follows from the mere existential claim that *something* is A.

8.2 Official Syntax

Now that we have gone over the rules for putting together proofs informally, it's time to give an official definition. The informal bits and pieces are enough when we want to show particular things are provable. But the official recursive definition is important for proving things about *all* proofs. There are three main facts about provability that we will show using the recursive definition.

1. *Compactness.* No formal proof essentially relies on infinitely many premises.
2. *Soundness.* If you can formally prove a conclusion from some premises, then the conclusion is a *logical consequence* of those premises in the sense we

defined in Chapter 5. In other words, no argument has both a proof and a counterexample.

3. *Decidability.* The question of what counts as a formal proof is effectively decidable. The question of what is *provable* from a decidable set of premises is not always decidable, but it is at least *semi-decidable*. (We'll return to this one in Section 8.5.)

8.2.1 Definition

We recursively define the relation P **is a proof of** A **from the premises** X**,** or P **proves** $X \vdash A$ for short, where X is any set of sentences and A is any sentence.

1. For any sentence A and any set of sentences X,

   ```
   A      Assumption
   ```

   proves $X, A \vdash A$.

2. If P proves $X \vdash A$, and Q proves $X \vdash B$, then

   ```
   P
   Q
   A ∧ B    ∧Intro
   ```

   proves $X \vdash A{\wedge}B$.

3. If P proves $X \vdash A \wedge B$, then

   ```
   P
   A      ∧Elim1
   ```

   proves $X \vdash A$.

4. If P proves $X \vdash A \wedge B$, then

   ```
   P
   B      ∧Elim2
   ```

   proves $X \vdash B$.

5. If P proves $X, A \vdash B$, and Q proves $X, A \vdash \neg B$, then

   ```
   suppose A:
       P
       Q
   ¬A          Reductio
   ```

   proves $X \vdash \neg A$.

6. If P proves $X \vdash \neg\neg A$, then

   ```
   P
   A      ¬¬Elim
   ```

   proves $X \vdash A$.

7. For any term a,

   ```
   a = a    =Intro
   ```

   proves $X \vdash a{=}a$.

8. For any terms a and b, and any formula A (in which x is free for both a and b), if P proves $X \vdash a{=}b$, and Q proves $X \vdash A[x \mapsto a]$, then

   ```
   P
   Q
   A[x  ↦  b]    =Elim
   ```

   proves $X \vdash A[x \mapsto b]$.

9. For any term a and formula A (in which a is free for x), if P proves $X \vdash \forall x\ A$, then

   ```
   P
   A[x  ↦  a]    ∀Elim
   ```

   proves $X \vdash A[x \mapsto a]$.

10. Suppose P proves $X \vdash A[x \mapsto y]$, where y is a variable which is not free in any formula in X, and y is free for x in A. Then

    ```
    for arbitrary y:
        P
    ∀x A      ∀Intro
    ```

    proves $X, Y \vdash$ ∀x A.

    (Typically x and y will just be the same variable, but the official rule is more flexible to help us avoid variable name clashes that come up once in a while.)

As in any recursive definition, we can say "that's all": P proves $X \vdash A$ *only* when this statement can eventually be reached by applying these rules.

Those are our official syntax rules. These rules are pretty rigid, but there are a bunch of ways in which we relax these official rules when we are writing out formal proofs in practice. (These shortcuts are analogous to the notational flexibility we have allowed ourselves for terms, formulas, and programs.)

- We are rather free with whitespace.
- We use all of our usual notational shortcuts for writing down formulas and terms.
- We use "derived rules" like Modus Ponens as if they were part of our basic repertoire.
- In rules like `∧Intro`, `=Elim`, and `Reductio` that have two subproofs P and Q, we sometimes mix up the order of the two subproofs.
- Instead of repeating a whole proof of a statement, sometimes we will just write the line number where the statement was previously proved (as long as that earlier proof didn't use any assumptions that aren't available for our subproof). When we use this style, we also can just refer to the line number of a `suppose` line to indicate that we are using a corresponding one-line proof by `Assumption`.

8.2.2 Definition

(a) We say P is a **proof** iff there are some X and A such that P proves $X \vdash A$.

(b) We say A **is provable from** X (abbreviated $X \vdash A$) iff there is some proof P such that P proves $X \vdash A$.

8.2.3 Example

$X \vdash \top$ for any set of formulas X.

Proof

Recall that $\top$ is an abbreviation for `∀x (x = x)`. We can write a formal proof:

```
for arbitrary x:
    x = x       =Intro
∀x (x = x)      ∀Intro
```

Let's explicitly check that this is a proof of $X \vdash \forall x\ (x{=}x)$, using Definition 8.2.1.

First, the subproof `x = x =Intro` is a proof of $\varnothing \vdash x{=}x$. Furthermore, the variable `x` does not appear free in any formula in the empty set. Thus the whole thing proves $X \vdash \forall x\ (x{=}x)$.

(One detail to notice is that the `∀Intro` rule lets us add in the extra premises from X. This is important, because in general those premises might include the free variable `x`, in which case we aren't allowed to use them in the inner proof.) □

8.2.4 Example

For any set of formulas X and any formula A,

$$\text{If} \quad X, \neg A \vdash \bot \quad \text{then} \quad X \vdash A$$

Proof

Remember that $\bot$ is an abbreviation for $\neg\top$. Suppose that P proves $X, \neg A \vdash \neg\top$. We can use this to put together a proof for $X \vdash A$, as follows. Let Q be the proof given by Example 8.2.3 for $X, \neg A \vdash \top$. Then the following proves $X \vdash A$.

```
suppose ¬A:
    Q
    P
¬¬A         Reductio
A           ¬¬Elim
```

To be explicit: since

$$\begin{aligned} Q &\text{ proves } X, \neg A \vdash \top \\ P &\text{ proves } X, \neg A \vdash \neg\top \end{aligned}$$

this means the `Reductio` part proves

$$X \vdash \neg\neg A$$

Thus the whole proof with the `¬¬Elim` step proves

$$X \vdash A$$

□

We have spelled out a definition of how proofs are put together, and what each proof proves. But often what we are most interested in is not the details of what *proofs* are like, but just what is *provable* somehow or other. So it's helpful to summarize the recursive definition of proofs (Definition 8.2.1) just in terms of what it tells us about what is provable, leaving out all the syntactic details about what the proof that proves it happens to look like. This straightforwardly follows from (Definition 8.2.1).

8.2.5 Proposition

Provability is closed under each of the following rules.

$$\frac{}{X, A \vdash A}\ \texttt{Assumption}$$

$$\frac{X \vdash A \qquad X \vdash B}{X \vdash A \wedge B}\ \texttt{∧Intro} \qquad \frac{X \vdash A \wedge B}{X \vdash A}\ \texttt{∧Elim1} \qquad \frac{X \vdash A \wedge B}{X \vdash B}\ \texttt{∧Elim2}$$

$$\frac{X, A \vdash B \qquad X, A \vdash \neg B}{X \vdash \neg A}\ \texttt{Reductio} \qquad \frac{X \vdash \neg\neg A}{X \vdash A}\ \texttt{¬¬Elim}$$

$$\frac{}{X \vdash a = a}\ \texttt{=Intro} \qquad \frac{X \vdash a = b \qquad X \vdash A[x \mapsto a]}{X \vdash A[x \mapsto b]}\ \texttt{=Elim}$$

$$\frac{X \vdash A[x \mapsto y]}{X, Y \vdash \forall x\ A}\ \texttt{∀Intro}\text{, where } y \text{ is not free in } X \qquad \frac{X \vdash \forall x\ A}{X \vdash A[x \mapsto a]}\ \texttt{∀Elim}$$

Taking this perspective, we can give more elegant (informal) proofs of (formal) provability facts, which abstract from the details of what particular proofs look like. For example, we can redo the proof of Example 8.2.4 much more compactly, by leaving out all of the syntactic details of proofs.

Second proof of Example 8.2.4
We want to show:

$$\text{If} \quad X, \neg A \vdash \bot \quad \text{then} \quad X \vdash A$$

We do this as follows.

$$\begin{array}{rl} X, \neg A \vdash \top & \text{Example 8.2.3} \\ X, \neg A \vdash \neg\top & \text{by assumption} \\ X \vdash \neg\neg A & \texttt{Reductio} \\ X \vdash A & \neg\neg\texttt{Elim} \end{array}$$

□

Another common presentation style for this kind of proof uses a diagram.

Third proof of Example 8.2.4

$$\dfrac{\dfrac{\dfrac{}{X, \neg A \vdash \top}\,\text{Example 8.2.3} \qquad X, \neg A \vdash \neg\top}{X \vdash \neg\neg A}\,\texttt{Reductio}}{X \vdash A}\,\neg\neg\texttt{Elim}$$

□

Each line in the diagram corresponds to some fact we know about provability—either from the definition, or from things we have already shown. This spare presentation style is common in proof theory. (But it is entirely optional.)

8.2.6 Exercise

Use Proposition 8.2.5 to show

$$A, A \to B \vdash B$$

(Recall that $A \to B$ is officially an abbreviation for $\neg(A \wedge \neg B)$.)

Proofs and provability are defined recursively. This means we can do yet another kind of proof by induction: induction on *proofs*. This works essentially the same way as induction on terms, formulas, or programs. Let's start with an example.

8.2.7 Example (Weakening)

For any sets of formulas X and Y, and any formula A, if $X \vdash A$, then $X, Y \vdash A$.

Proof
Let Y be any set. We will prove the following by induction on proofs:

For any P, X, and A such that P proves $X \vdash A$, P also proves $X, Y \vdash A$.

This proof is long and tedious, but not difficult. This is typical of induction on proofs. The proof has ten steps, corresponding to the ten rules for putting together proofs.

1. (`Assumption`) Consider a proof

   ```
   A    Assumption
   ```

   which proves $X, A \vdash A$. Then clearly this same proof also proves $X, Y, A \vdash A$. (Note that $(X \cup Y) \cup \{A\}$ and $(X \cup \{A\}) \cup Y$ are the very same set of premises—the order doesn't matter.)

2. (`∧Intro`) Let P prove $X \vdash A$ and let Q prove $X \vdash B$. We *suppose* (for the inductive hypothesis) that P also proves $X, Y \vdash A$ and Q also proves $X, Y \vdash B$. We want to *show* that the proof

   ```
   P
   Q
   A ∧ B    ∧Intro
   ```

   also proves $X, Y \vdash A \wedge B$. This immediately follows from the part of Definition 8.2.1 for `∧Intro` .

3. (`∧Elim`) Let P prove $X \vdash A \wedge B$. *Suppose* for the inductive hypothesis that P also proves $X, Y \vdash A \wedge B$. We want to *show* that

   ```
   P
   A      ∧Elim1
   ```

   proves $X, Y \vdash A$. Again, this is clear from the part of Definition 8.2.1 for `∧Elim1` .

4. (`∧Elim2`) This step goes just like `∧Elim1` .

5. (`Reductio`) Let P prove $X, A \vdash B$ and let Q prove $X, A \vdash \neg B$. *Suppose* that P also proves $X, Y, A \vdash B$ and Q also proves $X, Y, A \vdash \neg B$. (Again, note that the order of premises doesn't matter.) We want to *show* that

```
suppose A:
    P
    Q
¬A    Reductio
```

also proves $X, Y \vdash \neg A$. Again, this follows from Definition 8.2.1.

And so on. The remaining steps continue in just the same way. For each rule, we suppose for our inductive hypothesis that each component proof can be "weakened" by adding more unused premises, and then we put these "weakened" component proofs back together using the very same rule. The only step that is slightly trickier is `∀Intro`.

10. (`∀Intro`) Let P prove $X \vdash A[x \mapsto y]$ where y is not free in X. *Suppose* that P also proves $X, Y \vdash A[x \mapsto y]$. We want to *show* that

```
for arbitrary y:
    P
∀x A       ∀Intro
```

also proves $X, Z, Y \vdash \forall x\ A$. In this case, we can just ignore the inductive hypothesis: since P proves $X \vdash \forall x\ A$ and x is not free in X, the definition tells us that this `∀Intro` proof also proves $X, Z, Y \vdash \forall x\ A$. □

8.2.8 Technique (Proof by induction on proofs)

Suppose you want to prove that *every* proof of a conclusion from premises is *nice*. That is, you want to show that whenever P proves $X \vdash A$, the triple of P, X, and A is nice. You can show this in ten steps, one for each proof rule. In each step, you consider a proof put together from subproofs using one of the ten rules. *Suppose* that each of the subproofs are nice, and *show* that this implies the whole proof is nice as well. Then you are done.

Sometimes you use this technique to show something that isn't explicitly about *proofs* at all, but just about what is *provable*. The trick is that, in order to show

Whenever $X \vdash A$, ______

(fill in the blank) it is enough to show

> Whenever P proves $X \vdash A$, ______

You can show this by induction on the syntax of proofs.

8.2.9 Example (Provability is Compact)
If $X \vdash A$, then there is a finite subset $X_0 \subseteq X$ such that $X_0 \vdash A$.

The basic reason for this is that each proof has just finitely many steps, and each step of a proof only relies on finitely many premises, so the proof can only rely on finitely many premises all together. This is intuitively clear enough. But to get some practice with provability-induction, let's go ahead and show this fact in detail. It's a little trickier than you might expect.

Proof
We prove this by induction on proofs. Again, there are ten steps, and again this proof is long and tedious, but easy. We'll just spell out a few parts of it, to get the idea. We will show more specifically that whenever P proves $X \vdash A$, there is some finite subset $X_0 \subseteq X$ such that P proves $X_0 \vdash A$ as well.

1. (`Assumption`) Any proof by assumption of $X, A \vdash A$ is also a proof of $A \vdash A$, and $\{A\}$ is a finite subset of $X \cup \{A\}$.

2. (`∧Intro`) Let P prove $X \vdash A$ and let Q prove $X \vdash B$. Suppose for the inductive hypothesis that there are finite subsets $X_0 \subseteq X$ and $Y_0 \subseteq X$ such that P proves $X_0 \vdash A$ and Q proves $Y_0 \vdash B$. By Weakening, we also have

$$P \text{ proves } X_0, Y_0 \vdash A$$
$$Q \text{ proves } X_0, Y_0 \vdash B$$

   and $X_0 \cup Y_0$ is also a finite subset of X. Then the proof

```
P
Q
A ∧ B  ∧Intro
```

   proves $X_0, Y_0 \vdash$ `(A∧B)` as well.

We'll skip steps 3–8, since they all go basically the same way.

9. (`∀Elim`) Let P prove $X \vdash \forall x\ A$. Suppose for the inductive hypothesis that there is a finite subset $X_0 \subseteq X$ such that

$$P \text{ proves } X_0 \vdash \forall x\ A$$

Then

```
P
A[x ↦ t]   ∀Elim
```

proves $X_0 \vdash A[x \mapsto t]$.

10. (`∀Intro`) Let P prove $X \vdash A[x \mapsto y]$ (where y is not free in any formula in X) and suppose that there is a finite subset $X_0 \subseteq X$ such that

$$P \text{ proves } X_0 \vdash A[x \mapsto y]$$

Then

```
~~~
for arbitrary y:
    P
∀x A        ∀Intro
~~~
```

also proves $X_0 \vdash \allF\{x\}\{A\}$, and X_0 is a finite subset of $X \cup Y$.

That completes the induction. □

8.2.10 Exercise
Fill in the steps for `Reductio` , `¬¬Elim` , and `=Intro` in the inductive proof of Example 8.2.9.

8.2.11 Exercise
Let X be a set of formulas. Use the fact that Provability is Compact to show that, if every finite subset of X is consistent, then X is consistent (in the proof-theoretic sense).

Here's another important fact: you can stick two proofs together to get a proof. If you have a proof of a conclusion A, and another proof that uses A as a premise, then you can stick those together into one proof. Spelling that out a bit more, suppose you have a proof of $X \vdash A$, and another proof of $Y, A \vdash B$, which uses the conclusion of the first proof as a premise. Then by "sticking together" these proofs, we get a proof of This is a proof of $X, Y \vdash B$. That is, our conclusion is still B, but we don't rely on A as a premise anymore—since it is proved along the way. Instead, we have whatever premises the first proof used to *prove* A.

8.2.12 Example (Cut)
If $X \vdash A$ and $Y, A \vdash B$, then $X, Y \vdash B$.

Proof
We'll prove this by induction on proofs again. This time, we have to choose which of the two statements to do the induction on—the "upper proof" $X \vdash A$, or the "lower proof" $Y, A \vdash B$. The intuitive idea is that we can add the steps of the lower proof onto the upper proof, one by one. This gives us a clue that the lower proof is the one we want to do induction on. In other words, we will prove:

> Whenever P proves $Y, A \vdash B$, if $X \vdash A$, then $X, Y \vdash B$.

It might be helpful to put that even more explicitly in the "official" form for a statement to proved inductively. Let X be a set of formulas, and let A be a formula, and suppose $X \vdash A$. We want to show that every triple (P, Y', B) such that P proves $Y' \vdash B$ has the following property:

> For any set of formulas Y, if $Y' = Y \cup \{A\}$, then $X, Y \vdash B$.

What we have to show is that every proof *inherits* this property from its subproofs. As usual there are ten parts to this proof. It is still long and tedious, but not quite as straightforward as some of the others. We'll go through some of the steps, and leave the rest as exercises.

In order to avoid some messiness (in the `∀Intro` step in particular), we'll make a simplifying assumption: no variable x is both *free* in some formula in X, and also used as an *arbitrary* variable somewhere in the proof P (appearing in a line **`for arbitrary`** x**`:`** . If a clash like this did come up, we could always just switch to using a different variable in the proof P. This would still be a perfectly good proof of the same conclusion from the same premises.[2]

[2] This complication in our proof of Cut is very closely parallel to the complication that "captured" variables introduced for our definition of substitution in formulas (**??**). In general, there is a close

(But what if X is an infinite set that uses up *all* of the free variables? To avoid this case, we can start by finding a finite subset $X_0 \subseteq X$ such that $X_0 \vdash A$. This is guaranteed to only have finitely many free variables. Then we can use the same argument we give below to prove that, if $Y, A \vdash B$, then $X_0, Y \vdash A$. Finally, $X, Y \vdash A$ follows by Weakening.)

1. Consider a proof of $Y, A \vdash B$ by `Assumption`. This means B must be an element of $Y \cup \{A\}$. That is, either B is A, or else $B \in Y$.

   If $A = B$, then since we have already assumed that $X \vdash A$, we know $X \vdash B$. So by Weakening, $X, Y \vdash B$.

   If $B \in Y$, then in fact the same proof by `Assumption` also proves $Y \vdash B$. So again by Weakening, $X, Y \vdash B$.

   In either case, we have $X, Y \vdash B$, which is what we needed to show for this part.

2. Consider a proof of $Y, A \vdash B \wedge C$ by `∧Intro`. This has two subproofs:

   $$P \text{ proves } Y, A \vdash B$$
   $$Q \text{ proves } Y, A \vdash C$$

   For the inductive hypothesis, we suppose that each of these subproofs have the property. This means:

   $$X, Y \vdash B$$
   $$X, Y \vdash C$$

   Then we can put together these two new proofs into another proof using `∧Intro`: this shows

   $$X, Y \vdash B \wedge C$$

   This is what we needed to show.

The steps for `∧Elim1` and `∧Elim2` are left as an exercise.

5. Consider a proof by `Reductio` of $Y, A \vdash \neg B$. This also has two subproofs (for some formula C):

   $$P \text{ proves } Y, A, B \vdash C$$
   $$Q \text{ proves } Y, A, B \vdash \neg C$$

analogy between Cut and substitution. *Premises* in proofs are closely analogous to *free variables* in terms and formulas. This is part of a family of analogies called the "Curry-Howard correspondence."

In this case our inductive hypothesis says:

$$X, Y, B \vdash C$$
$$X, Y, B \vdash \neg C$$

Then we can put these back together with `Reductio`, to conclude:

$$X, Y \vdash \neg B$$

Again, this is what we needed to show for this part.

6. Consider a proof by `¬¬Elim` of $Y, A \vdash B$. This has one subproof:

$$P \text{ proves } Y, A \vdash \neg\neg B$$

We suppose for the inductive hypothesis:

$$X, Y \vdash \neg\neg B$$

Then we can use `¬¬Elim` again to conclude

$$X, Y \vdash B$$

The steps for `=Intro`, `=Elim`, and `∀Elim` are left as exercises. The only step with some fiddly business is `∀Intro`.

10. Consider a proof by `∀Intro` of $Y, A \vdash \forall x B$. This has one subproof. The premises of this subproof are a subset of $Y \cup \{A\}$ in which the variable y is not free. So there are two cases, depending on whether this subset includes A. There is a subset $Z \subseteq Y$, in which y is not free, such that either

$$P \text{ proves } Z \vdash B[x \mapsto y]$$

or else

$$P \text{ proves } Z, A \vdash B[x \mapsto y]$$

But the first case also implies the second case, by Weakening. So either way, our inductive hypothesis tells us

$$X, Z \vdash B[x \mapsto y]$$

We want to show that $X, Y \vdash \forall x B$. Here is where we'll use our simplifying assumption: since the variable y is used an arbitrary variable in this proof, by assumption it is not also a free variable in any formula in X. This means we can use `∀Intro` again to conclude

$$X, Y \vdash \forall x \ B$$

□

8.2.13 Exercise
Fill in the missing steps in the proof of Cut (Example 8.2.12).

Cut is handy for putting different provability facts together.

8.2.14 Example
`TODO.`

One important thing about formal proofs is that they line up with the notion of logical consequence we introduced in Section 5.3. Remember, there we said that A is a logical consequence of X iff the argument from X to A has no *counterexamples:* there is no structure in which every sentence in X is true, and A is false. In first-order logic, proofs and structures fit together neatly. First, if an argument has a *proof*, then it *has no counterexamples.* This fact is called *Soundness.* Second, if an argument *does not* have a counterexample, then it *does* have a proof. This fact is called *Completeness.* Putting the two facts together, they tell us that every argument has a proof or a counterexample, but not both.

The Completeness Theorem takes quite a bit of work to prove; this is in Section 8.3. But at this point we already have everything we need to prove the Soundness Theorem pretty straightforwardly.

8.2.15 Theorem (Soundness)
Let X be a set of formulas, and let A be a formula. If A is provable from X, then A is true in every model of X. In short:

$$\text{If} \quad X \vdash A \quad \text{then} \quad X \vDash A$$

Proof beginning
We will prove by induction that every proof P of $X \vdash A$ has the property that $X \vDash A$. To show this, we need to show that this property is inherited from subproofs.

It will be helpful to refer back to some facts about logical consequence that we showed back in Section 5.3.

1. (`Assumption`) For this step, we need to show that $X, A \vDash A$. This is clearly true: A is true in every model of $X \cup \{A\}$.
2. (`∧Intro`) For this step, we suppose that $X \vDash A$ and $X \vDash B$, and show that $X \vDash A{\wedge}B$. We already proved that this follows in Section 5.3.

3. (`∧Elim1`) For this step, we suppose that $X \vDash A \wedge B$, and prove that $X \vDash A$. Again, we proved this in Section 5.3.

Checking the steps for the remaining proof rules is left as an exercise. □

8.2.16 Exercise

Fill in the remaining steps of the proof of the Soundness Theorem, using facts about logical consequence from Section 5.3.

Before we move on, we should check a few more basic facts about provability that we will need later on.

8.2.17 Example (Contraposition)

If $X, A \vdash B$ then $X, \neg B \vdash \neg A$.

Proof

$$\dfrac{\dfrac{X, A \vdash B}{X, \neg B, A \vdash B}\,\text{Weakening} \qquad \dfrac{}{X, \neg B, A \vdash \neg B}\,\text{Assumption}}{X, \neg B \vdash \neg A}\,\text{Reductio}$$

□

8.2.18 Exercise

The following are equivalent:

$$\begin{gathered} X \vdash \bot \\ X \vdash A \text{ and } X \vdash \neg A \quad \text{for some } A \\ X \vdash A \quad \text{for every } A \end{gathered}$$

In Section 5.3 we defined “consistent” to mean “has a model”. For this section and the next, we’ll use a different definition of “consistent” instead.

8.2.19 Definition

A set X is **inconsistent** iff $X \vdash \bot$. (Exercise 8.2.18 gives us two other equivalent ways of saying this.) Otherwise X is consistent.

When we want to contrast the two meanings of “consistent”—this definition using proofs, and our earlier definition using models—we can distinguish *proof-theoretic* consistency and *model-theoretic* consistency. It is also common to call these *syntactic* consistency and *semantic* consistency, respectively. (But this terminology, while standard, is less transparent and more philosophically loaded.)

In the next section we'll show that in fact these two definitions exactly line up for first-order logic. That's why it isn't usually such a big deal to have two different definitions for the same word. But until we've proved that fact, we will need to be careful about which one we are talking about. And while we are showing things about formal proofs, it will be convenient to keep the word "consistent" reserved for the proof-theoretic notion.

8.2.20 Exercise
If X has a model, then X is proof-theoretically consistent: that is, $X \nvdash \bot$.

8.3 The Completeness Theorem

The Soundness Theorem shows that no argument has *both* a proof *and* a counterexample. There are "not too many" proofs or counterexamples, so they don't come into conflict with one another. What we'll now show is that every argument has one or the other: any argument with no countermodels has a formal proof. There are "enough" proofs and countermodels to settle the validity of every argument. The proof of this fact—the Completeness Theorem—is quite a bit trickier than the proof of the Soundness Theorem. For Soundness, we just needed to go through all the basic proof rules and make sure none of them led to trouble. For Completeness, though, we need to start with something that *doesn't* have a proof, and show that it *does* have a countermodel—and in this case induction on the structure of proofs is no help.

(Note that this is a different sense of the word "complete" from our earlier definition of a (negation-)*complete theory*—that is, a theory that includes each sentence or its negation. The two senses of "complete" are related, though. If you have a negation-complete theory, you can't add any extra sentences without introducing inconsistencies. If you have a complete proof system, you can't give proofs for any extra arguments without adding proofs for invalid arguments.)

Our strategy is to show that *any proof-theoretically consistent set of sentences has a model*. Given a set of sentences X which does not prove any contradictions, we can build up a structure in which every sentence in X is true. We'll do this in four stages: we'll start by constructing models for sets of very simple formulas, and work up to more complicated formulas little by little.

- **Stage 1.** First, suppose X is a set of formulas which don't include any logical symbols at all: X only contains *relation* formulas of the form $R(a,b)$. We'll start by constructing a model for X in this simple case.

- **Stage 2.** Next, we'll show how we can extend the idea of Stage 1 so it also works for a set X that contains *identity* formulas, of the form $a{=}b$. This is called a **canonical model**.

  (A formula which is either of the form Rab or of the form $a{=}b$ is called an **atomic formula.**)

- **Stage 3.** Next, we'll allow X to include formulas with the other logical connectives ($\neg$, $\wedge$, and $\forall$). But we'll make the further assumption that, not only is X consistent, but also X is *completely specific*, in two different senses.

  The first sense is that X has an answer to every "yes-or-no" question. For each formula A, either A or $\neg A$ is in X. (That is, X is **negation-complete**.)

  The second sense is that X has an answer to every "which" question. For each formula $A(x)$, either X names some particular example of a thing that satisfies $A(x)$—that is, X includes some substitution instance $A(t)$—or else X says that *nothing* satisifes $A(x)$—that is, X includes $\forall x\ \neg A(x)$. (In this case we say X is **witness-complete**.)

  We can show that if X is consistent and specific in both of these ways, then X has a model. (In fact, the same model we constructed in Stage 2 turns out to work.)

- **Stage 4.** We'll show that *any* consistent set of sentences X can be *extended* to a consistent set of formulas X^+ which is completely specific in those two senses. Since Stage 3 shows that this extended set X^+ has a model, this will also be a model of the smaller set X.

Stage 1: Relation Formulas

Our first job is to show how to come up with a model for a set of relational formulas. Suppose we are given a set X that just contains formulas of the form Rab. We want to come up with a model of X. We want to come up with some objects for our formal language to "talk about", and some way of interpreting each of the basic pieces of vocabulary in this language. This doesn't have to be a *plausible* interpretation of the language: it's fine for us to interpret the constant symbol `0` as denoting a fish or a mountain or whatever we want. We just have to come up with some structure or other that satisfies X.

How can we do this? We want a very general recipe, that is going to work for *any* first-order language. But this seems a bit magical. All we know about our set of formulas is that it doesn't prove any contradictions. But just given this, we have to conjure some domain of real things for the language to talk about! What sort of things are guaranteed to exist, just given an abstract formal language?

Here's the trick: we can use the *expressions of the language itself* as the domain of a structure. (Of course the existence of a consistent theory guarantees the existence of *linguistic* things!) It turns out that we can interpret the language as talking about *itself!*

8.3.1 Definition

Suppose X is some set of relational L-formulas (of the form Rab where R is an L-predicate and a and b are L-terms). Let the **simple model** for X be the pair of a structure S and an assignment function g given as follows.

1. The domain D_S is the set of all L-terms.
2. For each constant symbol c in L, the extension c_S is the constant term c itself.
3. For each one-place function symbol f, the extension f_S is the function that takes each L-term a to the L-term fa.
4. For each two-place function symbol f, the extension f_S is the function that takes each pair of L-terms a and b to the L-term $f(a,b)$.
5. For each relation symbol R in L, the extension R_S is the set of pairs (a,b) of a term a and a term b such that $X \vdash Rab$.
6. The assignment function g is the function that takes each variable x to itself.

8.3.2 Exercise

Let X be a set of relational formulas, and let S and g be the structure and assignment from Definition 8.3.1.

(a) Every L-term a denotes itself: that is, $[\![a]\!]_S g = a$.

(b) For every L-formula A, (S,g) satisfies A iff $X \vdash A$.

Stage 2: Identity Formulas

Now we'll try to come up with a model that will also work for *identity* formulas. Suppose, for example, that X includes the sentence `suc 0 = suc 0 + 0`. Notice that the simple model from Stage 1 definitely won't satisfy this sentence. On the "linguistic" interpretation, `suc 0` denotes itself, the term `suc 0`, while `suc 0 + 0` denotes the term `suc 0 + 0`, and these two terms are different. So on the "self-referential" Stage 1 interpretation, `suc 0 = suc 0 + 0` will come out false. So we'll need to modify the Stage 1 structure to make it possible for *different* terms to denote the same thing.

What we want to do is "blur together" some of the different elements of the domain of the Stage 1 structure. There is a neat general trick for doing this, called the method of *equivalence classes*. Instead of using the terms themselves as the elements of our domain, we can use special *sets* of terms. Each set will contain some terms that are *equivalent* to one another, in the sense that X says that $a = b$. The key observation here is that, even if a and b are two different terms, if a and b are equivalent, then *the set of terms that are equivalent to* a, and *the set of terms that are equivalent to* b are the very same object. So *sets* of terms can do the job of satisfying the right identity formulas.

8.3.3 Definition

Let X be a set of L-formulas.

1. Terms a and b are **equivalent** given X iff $X \vdash a = b$.

2. For any term a, the **equivalence class** of a is the set of all terms which are equivalent to a given X: that is,

$$E(a) = \{b \in L\text{-terms} \mid X \vdash a = b\}$$

   (So E is a function that takes each L-term to a set of L-terms.)

8.3.4 Exercise

For any L-terms a and b, $E(a) = E(b)$ iff a and b are equivalent in X.

8.3.5 Exercise

(a) For any L-terms a and b, if $E(a) = E(b)$, then for any one-place function symbol f, $E(f\,a) = E(f\,b)$.

(b) State the generalization of (a) for two-place function symbols. (But you don't have to prove this separately.)

8.3.6 Definition

Let X be a set of atomic formulas. The **canonical model** for X is the pair (S, g) of a structure and assignment constructed as follows.

1. The domain of S is the range of E. That is, D_S is the set of all equivalence classes of L-terms.
2. For each constant c, the value c_S is the equivalence class $E(c)$.
3. For each one-place function symbol f, the extension f_S is a function from equivalence classes to equivalence classes, defined so that for each term a:

   $$f_S(E(a)) = E(fa)$$

   This is well-defined, because if $E(a) = E(b)$, then $E(fa) = E(fb)$ as well.
4. The clause for two-place function symbols is similar.
5. For each relation symbol R, the extension R_S is the set of pairs

   $$\{(E(a), E(b)) \mid X \vdash Rab \text{ for any } L\text{-terms } a \text{ and } b\}$$

6. For each variable x, the assignment g takes the variable x to its equivalence class $E(x)$.

8.3.7 Exercise

Let X be a set of atomic formulas, and let (S, g) be the canonical model for X.

(a) Every L-term a denotes its own equivalence class:

$$[\![a]\!]_S g = E(a)$$

(b) For any two-place relation symbol R and any L-terms a and b,

$$(S, g) \text{ satisfies } Rab \quad \text{iff} \quad X \vdash Rab$$

(c) For any L-terms a and b,

$$(S, g) \text{ satisfies } (a{=}b) \quad \text{iff} \quad X \vdash (a{=}b)$$

8.3.8 Exercise
Is the domain of the canonical model countable or uncountable? Explain.

Stage 3: Negation-Complete and Witness-Complete Theories

The Stage 2 model correctly handles atomic formulas, including identity. But so far it doesn't "know about" the rest of logic.

For example, consider the set $X = \{$`∃x (f(x) = c)`$\}$. This set X doesn't imply any identities for *any* two distinct terms. So in fact, the canonical model for X has as its domain the singleton sets for every term, and in this structure the extension of the function symbol `f` takes each set $\{t\}$ to the singleton set $\{\mathtt{f}(t)\}$. This function doesn't map anything to the singleton set $\{\mathtt{c}\}$. So if we construct the canonical model for X in the same way as Stage 2, the existential claim `∃x (f(x) = c)` will turn out to be *false*, even though X "says" that it's true.

The trouble here is that X includes an "unwitnessed" generalization: it says that *something* has to satisfy a condition (getting mapped to c), but it doesn't provide any specific example of a thing that satisfies that condition. We can avoid this problem if we add an extra specificity constraint, that insists that every generalization has a specific "witness". For Stage 2, we want to consider a "completely specific" set of formulas. Here's what that means.

8.3.9 Definition
Let X be a set of L-formulas.

1. X is **negation-complete** iff for every L-formula A, either $A \in X$ or $\neg A \in X$. (This is the same as our earlier definition of "complete" from Section 5.3.)

2. X is **witness-complete** iff for every L-formula $A(x)$, either there is some L-term t such that $A(t) \in X$, or else $\forall x\ \neg A(x) \in X$.

8.3.10 Exercise
If X is consistent and negation-complete, then $X \vdash A$ iff $A \in X$.

8.3.11 Exercise

Suppose that X is consistent, negation-complete, and witness-complete.

(a) $\neg A \in X$ iff $A \notin X$.

(b) $A \wedge B \in X$ iff $A \in X$ and $B \in X$.

(c) $\forall x \;\; A(x) \in X$ iff for every term t, $A(t) \in X$.

8.3.12 Exercise

Suppose that X is consistent, negation-complete, and witness-complete. Let X_0 be the set of atomic formulas in X, and let (S, g) be the canonical model for X_0 (as in Definition 8.3.6). For any formula A,

$$(S, g) \text{ satisfies } A \quad \text{iff} \quad A \in X$$

Hint. Use induction on the complexity of A. Exercise 8.3.7 and Exercise 8.3.11 will help.

8.3.13 Lemma

Suppose X is a consistent, negation-complete, and witness-complete set of formulas. Then X has a model.

Proof

By Exercise 8.3.12, the canonical model for the set of atomic formulas in X is a model of X. □

Stage 4: Extending a Consistent Set

The last step is to get from an *arbitrary* consistent set to a bigger set which is also negation-complete and witness-complete. To do this, we'll use the following three facts about consistency.

8.3.14 Exercise

If $X \cup \{A\}$ is inconsistent, and $X \cup \{\neg A\}$ is inconsistent, then X is inconsistent.

8.3.15 Exercise

If $X_0 \subseteq X_1 \subseteq X_2 \subseteq \dots$ is a chain of consistent sets, then their union $\bigcup_n X_n$ is consistent.

Hint. Recall this fact from back in Chapter 2: if Y is a *finite* subset of $\bigcup_i X_i$, then there is some number n for which Y is a subset of X_n.

8.3.16 Lemma

Suppose that X is a consistent set of formulas. Then X has a consistent and negation-complete extension.

Proof

The idea is that we can go through all the formulas one by one, and in each case if it's consistent with what we already have we can add it in, and otherwise we can add in its negation. We can make this idea precise with an inductive argument. There are countably infinitely many formulas: so we can put them all in an infinite sequence, so each formula is A_n for some number n. Then we can recursively define a sequence of sets, as follows:

$$X_0 = X$$

$$X_{n+1} = \begin{cases} X_n \cup \{A_n\} & \text{if this is consistent} \\ X_n \cup \{\neg A_n\} & \text{otherwise} \end{cases}$$

We start with our original consistent set X, and go through all the formulas adding it or its negation. We can prove by induction that for every number n, X_n is consistent. For the base case, X_0 is consistent by assumption. For the inductive step, we need to show that if X_n is consistent, then either $X_n \cup \{A_n\}$ is consistent or else $X_n \cup \{\neg A_n\}$ is consistent. This follows from Exercise 8.3.14: this exercise showed that if both of these two sets are *inconsistent*, then X_n must also be inconsistent.

So each set X_n is consistent. Furthermore, these sets form a chain $X_0 \subseteq X_1 \subseteq X_n \subseteq \dots$. Thus, by Exercise 8.3.15, it follows that their union $X^+ = \bigcup_n X_n$ is also consistent. Furthermore, it's clear that for every formula A, either $A \in X^+$ or $\neg A \in X^+$: so X^+ is a consistent, negation-complete extension of X. □

8.3.17 Exercise

Suppose that y is not free in any formula in X or in $A(y)$. If $X \cup \{A(y)\}$ is inconsistent, and $X \cup \{\forall x\ \neg A(x)\}$ is inconsistent, then X is inconsistent.

8.3.18 Lemma

Suppose that X is a consistent set of *sentences*. Then X has a consistent witness-complete extension. That is, there is some consistent and witness-complete set of formulas Y such that $X \subseteq Y$.

Proof
The reason we start with *sentences* and end up with *formulas* in this case is that we'll use free variables in order to come up with enough terms to have a specific instance of every generalization—so we need to guarantee that we haven't already "used up" too many variables to start out with.[3]

The proof is very similar to Lemma 8.3.16. Once again, we'll list the formulas A in an infinite sequence, so each formula is $A_n(x)$ for some number n. We'll also come up with a sequence of variables: for each n, let y_n be a variable which is not free in any of the formulas $A_0(x), \dots, A_n(x)$, and which is distinct from each of the earlier variables $y_0, \dots, y_n$. There is always such a variable, because there are only finitely many free variables in each formula, and there are infinitely many variables to choose from.

Then, as before, we can recursively define a sequence of sets X_n, as follows:

$$X_0 = X$$
$$X_{n+1} = \begin{cases} X_n \cup \{A_n(y_n)\} & \text{if this is consistent} \\ X_n \cup \{\forall x \ \neg A_n(x)\} & \text{otherwise} \end{cases}$$

First, note that for each n, the variable y_n is not free in any formula in X_n. (This relies on the fact that no variables are free in X_0.) Then we can show by induction that each set X_n is consistent. For the inductive step, we need to show that for any consistent set, we can always consistently add *either* $A_n(y_n)$ (with an unused variable y_n), or else $\forall x \ \neg A_n(x)$. This follows from Exercise 8.3.17: if *both* of these additions are inconsistent, then so is the original set. Since we have assumed that X_0 is consistent to begin with, by induction every set X_n is consistent.

It then follows that the union $X^+ = \bigcup_n X_n$ is also consistent. Furthermore, it's clear from the construction that for *every* formula $A(x)$, either $A(y) \in X^+$ for some term y, or else $\forall x \ \neg A(x) \in X^+$. So X^+ is a consistent and witness-complete extension of X. □

[3] This restriction to just sets of sentences is avoidable. Instead, we could add infinitely many new *constants* to our language in order to get enough fresh terms to serve as witnesses to every generalization. But if we did things that way, we would need to prove some (easy, but tedious) facts about the relationship between consistent sets of formulas in *different languages*. Alternatively, we could start with a "relettering" step, switching around all of the free variables in a way that leaves infinitely many variables unused. But this approach also depends on proving tedious consistency facts about relettered sets of formulas.

8.3.19 Exercise (Henkin's Lemma)
If X is a consistent set of sentences, then X has a model.

Hint. Put the previous three lemmas together (in the right order).

8.3.20 Exercise (The Completeness Theorem)
If $X \vDash A$, then $X \vdash A$.

8.3.21 Exercise (The Compactness Theorem)
If $X \vDash A$, then there is a finite subset $X_0 \subseteq X$ such that $X_0 \vDash A$.

Before we move on, we should note another neat consequence of the *way* we proved the Completeness theorem. We didn't just show that every consistent set has some model or other. In fact, for any consistent set of sentences X we gave a specific recipe for a *canonical* model for a set of formulas that includes X. An important feature of this model is that it is not too big. So we can prove the following fact as well.

8.3.22 Exercise (The Downward Löwenheim-Skolem Theorem)
If X has a model, then X has a *countable* model.

As you might guess from the name, there is also an "upward" version of this theorem. Here is what it says:

8.3.23 The Upward Löwenheim-Skolem Theorem
If X has a model with an infinite domain D, then for any set D^+ with at least as many elements as D, X has a model with domain D^+.

Putting both directions together, we get this result:

8.3.24 The Löwenheim-Skolem Theorem
If X has an infinite model, then X has a model of every infinite size.

Proving the "upward" theorem uses ideas that go beyond this text. (See CITE.) The basic idea is that we can add in lots of harmless copies of the elements of the structure without affecting any of the first-order truths.

8.4 Models of Arithmetic*

UNDER CONSTRUCTION

Discuss: the Inductive Principle, first-order PA and induction schema.

Discuss: the standard model, and standard models of arithmetic more generally. (Isomorphism. Give an example: domain is $\{2, 3, 4, \ldots\}$, addition given by $(m + n - 4)$, etc.)

8.4.1 Exercise
Consider a structure S for the language of arithmetic. If S is a standard model of arithmetic, then every element of the domain of S is the denotation of some numeral:

$$0 \quad \mathrm{suc}\, 0 \quad \mathrm{suc}\, \mathrm{suc}\, 0 \quad \ldots$$

8.4.2 Exercise
Consider the signature of the language of arithmetic with one additional constant symbol c. The theory

$$\mathrm{Th}\, \mathbb{N} \cup \{c \neq 0, c \neq 1, c \neq 2, \ldots\}$$

has a model.

8.4.3 Exercise
There is a non-standard model of arithmetic: that is, there is a structure which is a model of $\mathrm{Th}\, \mathbb{N}$ and which is not isomorphic to the standard model $\mathbb{N}$.

TODO. Discuss the gap between the induction schema and the Inductive Principle.

8.5 The Incompleteness Theorem

One nice feature of formal proofs is that they are computationally tractable—much more so than structures. We can systematically check whether any particular string of symbols is a proof, and, if so, what it proves. This gives us another important connection between two of the main ideas of this course: *decidability* and *provability*. Furthermore, the Soundness and Completeness Theorems tell us that *provability* exactly lines up with *logical consequence* (in our earlier sense involving structures).

This lets us—at last!—use things we have learned about undecidable sets to find logical limits on simple theories.

What is a simple theory? Earlier (Section 5.4) we considered some theories that consisted of the logical consequences of a finite set of axioms. We also considered some theories like PA and ZFC which *aren't* finitely axiomatizable, but are still "simple" in the important sense. Now that we have the tools of computability theory at our disposal, we can describe this more carefully. Even though the set of axioms of First-Order Peano Arithmetic isn't a *finite* set, it is still a *decidable* set: there is a simple mechanical rule for answering the question "Is this an axiom of PA?". Very often that is enough.

Recall that a set of sentences X *axiomatizes* T iff T is the set of all of the logical consequences of X. Using Soundness and Completeness, we can now equivalently say that X axiomatizes T iff, for every sentence A,

$$A \in T \quad \text{iff} \quad X \vdash A$$

8.5.1 Definition

A theory T is **effectively axiomatizable** iff there is some effectively decidable set of sentences X that axiomatizes T. We usually just say "axiomatizable" for short.

So instead of our loose notion of a "simple theory", we now have the precise notion of an *axiomatizable* theory.

8.5.2 Exercise

Suppose that X is an effectively decidable set of formulas. Explain why the set of pairs (P, A) such that $P : X \vdash A$ is effectively decidable, using Definition 8.2.1.

(Officially showing this in detail—by writing a program—would be a big job. You don't have to do that: just describe the basic idea of an algorithm for checking whether P is a proof of A from X.)

For the following exercises, it will be helpful to refresh your memory of the things we showed about semi-decidable and effectively enumerable sets in Section 7.6.

8.5.3 Exercise

(a) Suppose that X is a decidable set of formulas. Show that the set of formulas A such that A is provable from X is semi-decidable.

(Thus the set of formulas which are provable from X is also effectively enumerable.)

(b) Give an example of a decidable set of formulas X such that the set of formulas that are provable from X is not decidable. Explain.

8.5.4 Exercise

(a) Any effectively axiomatizable theory is effectively enumerable.

(b) The set of logical truths is effectively enumerable.

8.5.5 Exercise

If X is a set of sentences which is effectively enumerable, consistent, and negation-complete, then X is decidable.

8.5.6 Exercise (Gödel's First Incompleteness Theorem)

No theory is sufficiently strong, axiomatizable, consistent, and complete.

8.5.7 Exercise

For each of the following theories, say (i) whether it is axiomatizable, and (ii) whether it is negation-complete. Briefly explain.

(a) The theory of strings $\mathsf{Th}\,\mathbb{S}$.

(b) The theory of arithmetic $\mathsf{Th}\,\mathbb{N}$.

(c) The minimal theory of strings S.

(d) First-order Peano Arithmetic PA.

(e) First-order set theory ZFC (supposing this is consistent and sufficiently strong, which we have not shown).

(f) The set of all logical truths.

(g) The set of all sentences.

8.6 Gödel Sentences

Lots of interesting theories are sufficiently strong, axiomatizable, and consistent. The minimal theory of strings is like this, and so is the minimal theory of arithmetic. So are lots of reasonable axiomatic theories that extend or interpret these, like Peano

Arithmetic, first-order set theory, or many formalized physical theories. Gödel's First Incompleteness Theorem tells us that no theory like this is complete: for any theory like this, there are sentences that can be neither proved nor disproved.

Our version of "Gödel's First Incompleteness Theorem" is a bit anachronistic. What we proved is a little different from what Gödel proved in 1931, and the way we proved it is also a bit different.[4] In several respects, we actually proved a bit more than Gödel did (with the benefit of hindsight). But in one important respect, we did a bit less. Consider the theory of Peano Arithmetic (PA). We know that there *exist* sentences in the first-order language of arithmetic which PA neither proves nor disproves. But so far we haven't actually given any *example* of such a sentence. In this sense, unlike Gödel's proof, our proof of the First Incompleteness Theorem was not *constructive*. Can we do better?

Let's start by trying to reverse engineer the proof we already gave. We showed, first, that if a theory T is effectively axiomatizable, then its theorems are effectively enumerable. Second, if T is also consistent and complete, then T is *decidable*. This means that if T is also sufficiently strong, then T can *represent* the set of sentences that are provable from T's axioms. In other words, there is some formula $\text{Prov}_T(x)$ that represents T within T:

$$\begin{array}{ll} T \vdash \text{Prov}_T\langle A\rangle & \text{if} \quad T \vdash A \\ T \vdash \neg\,\text{Prov}_T\langle A\rangle & \text{otherwise} \end{array}$$

Then, by Gödel's Fixed Point Theorem, we have a sentence G which is equivalent

[4] In fact, to be historically accurate, all three of the notions "sufficiently strong", "effectively axiomatizable", and "consistent" in the statement of the theorem need some qualification.

1. Gödel didn't know about the theories Q or S (in particular, he didn't know that theories quite as simple as this could represent every decidable set). So he used a different definition of "sufficiently strong", which referred to a much richer formal theory: the one given in Russell and Whitehead's *Principia Mathematica*, PM. Since PM interprets Q, Gödel's notion of "sufficiently strong" follows from ours.
2. Gödel didn't know about Church and Turing's definitions of computable functions and decidable sets, or the Church-Turing Thesis—and certainly not our programming language Py. (Church and Turing's definitions were both developed in 1936. In fact, Gödel also developed his own equivalent definition of computability in 1933.) So instead of talking about an "effectively axiomatizable" theory (which has a *decidable* set of axioms), he talked about a theory that has a *primitive recursive* set of axioms. This turns out to be equivalent to what can be expressed in Py using only `for` loops, instead of `while` loops. Every primitive recursive set is also decidable.
3. Gödel's proof turned on a stronger consistency requirement, called *ω-consistency*. We'll discuss this below.

(in T) to $\neg\,\mathrm{Prov}_T\langle G\rangle$. But this implies that T is inconsistent (by Tarski's Theorem).

But in fact, in a theory T which *is* consistent, running that last step backwards tells us that there really *isn't* any formula $\mathrm{Prov}_T(x)$ that represents T within T. This is exactly what Tarski's Theorem (Exercise 6.9.5) tells us. So of course we can't really get an example of an undecidable sentence G by taking a fixed point of the negation of this non-existent formula.

But we can still do something very similar! Here's something else we know: if T is effectively axiomatizable, then the *relation* "P is a *proof* of A from T's axioms" is decidable. For short, call this the **T-proof relation**. So, if T is sufficiently strong, we can represent this relation in T, using a formula $\mathrm{Proof}_T(\mathtt{x},\mathtt{y})$.

$T \vdash \mathrm{Proof}_T\langle P\rangle\langle A\rangle$	if P is a proof of A from T's axioms
$T \vdash \neg\,\mathrm{Proof}_T\langle P\rangle\langle A\rangle$	otherwise

Now consider the formula $\exists\mathtt{x}\ \mathrm{Proof}_T(\mathtt{x},\mathtt{y})$. It's customary to call this formula $\mathrm{Prov}_T(\mathtt{y})$—the *provability* formula for T. But we have to be very careful about this. As we just said, by Tarski's Theorem we know that this formula can't really *represent* provability in T (unless T is inconsistent). But it does still have an important close relationship to provability. In a sense, provability is "representable in one direction". (This notion of one-way representability also came up in Section 6.7. It's analogous to *semi-decidability*.)

8.6.1 Exercise

Suppose that $\mathrm{Proof}_T(\mathtt{x},\mathtt{y})$ represents the T-proof relation in T. Let $\mathrm{Prov}_T(\mathtt{y})$ be $\exists\mathtt{x}\ \mathrm{Proof}_T(\mathtt{x},\mathtt{y})$.

(a) For any sentence A, if $T \vdash A$, then $T \vdash \mathrm{Prov}_T\langle A\rangle$.

(b) Suppose furthermore that the theory T is *true* in the standard string structure $\mathbb{S}$. In that case, if $T \nvdash A$, then $T \nvdash \mathrm{Prov}_T\langle A\rangle$.

Notice the difference between clause (b) in this exercise and the definition of "represent". In a case where A isn't provable, it isn't that T says that A is *not* provable—but at least T *doesn't* incorrectly say that A *is* provable.

8.6.2 Definition

Let T be a sufficiently strong, effectively axiomatizable theory. Let $\mathrm{Proof}_T(\mathtt{x},\mathtt{y})$ be a formula that represents the T-proof relation in T. (There is such a formula, by the Representability Theorem (Principle 7.7.8).) Let $\mathrm{Prov}_T(y)$ be the provability formula $\exists\mathtt{x}\ \mathrm{Proof}_T(\mathtt{x},\mathtt{y})$.

A **Gödel sentence** for T is a fixed point of the negation of the provability formula: that is, it is a sentence G_T such that

$$G_T \underset{T}{\equiv} \neg \mathrm{Prov}_T \langle G_T \rangle$$

8.6.3 Lemma

Any sufficiently strong, effectively axiomatizable theory T has a Gödel sentence G_T.

Proof

This immediately follows from Gödel's Fixed Point Theorem (Exercise 6.7.11). □

8.6.4 Exercise

Let T be a sufficiently strong, effectively axiomatizable theory, and let G_T be a Gödel sentence for T.

(a) If T is consistent, then $T \nvdash G_T$.

(b) If T is true in the standard string structure $\mathbb{S}$, then $T \nvdash \neg G_T$.

We can improve a bit on part (b), by paying attention to exactly how *truth*—that is, truth-in-the-standard-string-structure—comes into the argument. The key thing that this heads off is the following possibility. Suppose there is no proof of G_T. Then, since T represents the T-proofs, for each particular string s, we're guaranteed that T says, "s is not a proof of G_T". But what if T *also* says "But there is a proof of G_T!" This wouldn't be a logical inconsistency: it's not logically impossible for there to be something *else*, something that isn't one of the standard finite strings, which is a proof of G_T. (But even precisely *stating* this possibility goes beyond what we can say in the first-order theory of strings.) Still, even though this wouldn't be formally inconsistent, a theory like this would would still be bad in a way. It has a kind of "infinite inconsistency". A theory like this accepts a generalization, while ruling out every possible instance. This motivates the following definition.

8.6.5 Definition

A theory T is ω-**inconsistent** (pronounced "omega inconsistent") iff there is some formula $A(x)$ such that

(a) $T \vdash \exists x \; A(x)$

(b) For every string s, $T \vdash \neg A\langle s\rangle$.

If there is no such formula, then T is **ω-consistent.**

8.6.6 Exercise (Gödel's First Incompleteness Theorem, Version 2)
Suppose T is a sufficiently strong, effectively axiomatizable theory, and let G_T be a Gödel sentence for T. If T is consistent and ω-consistent, then $T \nvdash G_T$ and $T \nvdash \neg G_T$.

So that pretty much gives us what we were hoping for. If a theory T is sufficiently strong, effectively axiomatizable, consistent, *and also ω-consistent*, not only do we know that T is incomplete, but we can give a particular example of a sentence that T neither proves nor refutes: the theory's Gödel sentence.

(A sentence which can be neither proved nor refuted is often called *undecidable*. But watch out—this meaning of "undecidable" is totally different from the notion involving programs.)

8.7 Rosser Sentences*

```
UNDER CONSTRUCTION.
```

We've considered two different proofs of Gödel's First Incompleteness Theorem. The first was non-constructive: it didn't give us a concrete example of a undecidable sentence. The second (closer to Gödel's original proof) gave us a specific example of an undecidable sentence, but it used the extra assumption of ω-consistency. It turns out that that there is a third proof of Gödel's First Incompleteness Theorem that has the advantages of *both* of the proofs we've already given. It gives us a specific example of an undecidable sentence, and it only depends on ordinary consistency, rather than ω-consistency. The main downside to this proof (the reason we didn't use it as our official proof all along) is that it is extra sneaky.

The trick is to notice that if there is a proof of A, then there is also a *shortest* proof.

8.8 Consistency is Unprovable

What we've shown is that for any sufficiently strong, consistent, axiomatizable theory, there is some true statement that it cannot prove—and we gave an example, the Gödel sentence. But Gödel showed something more: he gave another specific example of an unprovable statement which is of particularly deep importance. Any sufficiently strong axiomatizable theory has the resources to "talk about" what is provable in that very theory, using the provability formula from Section 8.6. So one of the things such a theory can talk about is whether it can prove any contradictions. That is, if T is a sufficiently strong axiomatizable theory, then it includes a sentence that says "T is consistent"—that is, a sentence which says "no contradiction is provable in T". The further thing Gödel showed is that if T really is consistent, then *this* statement is also unprovable. No reasonable theory can prove its own consistency. This is called **Gödel's Second Incompleteness Theorem.**

The basic idea of the proof is that in a sufficiently strong theory T, the proof of Gödel's *First* Incompleteness Theorem can be *formalized*. The steps we went through to justify First Incompleteness Theorem can also be carried out in a formal *proof* from the axioms of T. We won't work through all of the details of the proof of this result, but we will examine the main ideas.

Let's start with a recap of the proof of the First Incompleteness Theorem. Suppose that T is a sufficiently strong theory with a decidable set of axioms X. Then as we discussed in Section 8.6, there is a formula $\text{Proof}_T(\mathsf{x}, \mathsf{y})$ such that

$$\begin{array}{ll} T \vdash \text{Proof}_T\langle P\rangle\langle A\rangle & \text{if } P \text{ is a proof of } A \text{ from } X \\ T \vdash \neg\text{Proof}_T\langle P\rangle\langle A\rangle & \text{otherwise} \end{array}$$

We also noted that this doesn't mean that *provability* can be represented in T. (Indeed, Tarski's Theorem tells us that, if it were, then T would be inconsistent.) But that doesn't stop us from defining a *provability formula*: we can let $\text{Prov}_T(y)$ be $\exists\mathsf{x}\ \text{Proof}(\mathsf{x}, \mathsf{y})$. This doesn't fully represent provability in T, but it does "represent provability in one direction." If A is provable in T, then A has some proof P. So $T \vdash \text{Proof}_T\langle P\rangle\langle A\rangle$, and thus by existential generalization, $T \vdash \text{Prov}_T\langle A\rangle$. In short, for any sentence A,

$$\text{If} \quad T \vdash A \quad \text{then} \quad T \vdash \text{Prov}\langle A\rangle$$

But, to reiterate, we *don't* get the other half of the definition of representability: if A is *not* provable, there is no guarantee that T "knows" that fact. (Indeed, it will follow from the Second Incompleteness Theorem that T *can't* know that there is no proof of A.)

Remember that a theory T is *inconsistent* iff $\bot$ is provable in T.

8.8.1 Definition

The **consistency sentence** for T is the sentence $\neg\mathrm{Prov}_T\langle\bot\rangle$. This is abbreviated Con_T. That is, to spell this out, Con_T is the sentence

$$\neg\exists \mathsf{x}\ \mathrm{Proof}_T(\mathsf{x}, \langle\bot\rangle)$$

where $\mathrm{Proof}_T(\mathsf{x}, \mathsf{y})$ represents the T-proofs in T.

(Note that while we say "*the* consistency sentence", this is a bit loose. There are many ways for T to represent the relation "P is a proof of A". Different choices of the formula $\mathrm{Proof}_T(\mathsf{x}, \mathsf{y})$ will clearly give rise to different consistency sentences for T. In fact, it can make a difference which one we choose.)

The result we are working toward says that no consistent theory can prove its own consistency sentence. That is:

$$\text{If}\quad T \vdash \mathrm{Con}_T \quad\text{then}\quad T \vdash \bot$$

For the first step, recall from Section 8.6 that any sufficiently strong, effectively axiomatizable theory T has a **Gödel sentence** G_T, such that

$$T \vdash G \leftrightarrow \neg\mathrm{Prov}_T\langle G\rangle$$

Recall also from Exercise 8.6.4 that if T is consistent, then T does not prove its own Gödel sentence. Putting that the other way around:

$$\text{If}\quad T \vdash G_T \quad\text{then}\quad T \vdash \bot$$

(From here on out, we'll drop the T subscripts when it's clear how to fill them in.)

The second step is to show that this first step can be *formalized* in T. To do this, we need to begin by showing that T "knows" some basic facts about how proofs are put together. Here are two basic things we know about provability:

$$\begin{array}{rl} \text{If}\quad T \vdash A \to B \quad\text{and}\quad T \vdash A & \text{then}\quad T \vdash B \\ \text{If}\quad T \vdash A & \text{then}\quad T \vdash \mathrm{Prov}_T\langle A\rangle \end{array}$$

That is, provability is closed under modus ponens; and if A is provable, then it is provable that A is provable. Our proof of the Second Incompleteness Theorem relies on T also "knowing" both of these two facts.

8.8.2 Definition

A theory T satisfies the **derivability conditions** iff

$$T \vdash \mathrm{Prov}\langle A \to B\rangle \to \mathrm{Prov}\langle A\rangle \to \mathrm{Prov}\langle B\rangle$$
$$T \vdash \mathrm{Prov}\langle A\rangle \to \mathrm{Prov}\langle \mathrm{Prov}\langle A\rangle\rangle$$

The first condition formalizes the claim that provability is closed under modus ponens. The second condition formalizes the claim that if A is provable, then it is provable that A is provable.

Showing exactly which theories satisfy the derivability conditions involves some fiddly details that we are going to skip over. We are just going to take this for granted in what follows. (In particular, it can depend a bit on the details of the theory T and the way in which we define the formula $\mathrm{Proof}(x, y)$. I'm ignoring some complications here.) But here's one important example: first-order Peano Arithmetic PA satisfies the derivability conditions.

8.8.3 Notation

We are going to do some fairly intricate reasoning about proofs about provability. For this purpose it can be helpful to introduce some more concise notation, inspired by modal logic. We can use the "box" notation $\Box A$ as an abbreviation for the sentence $\mathrm{Prov}\langle A\rangle$. Using box notation, we can summarize the key facts about provability more concisely like this:

$$T \vdash G \leftrightarrow \neg\Box G$$
$$\text{If} \quad T \vdash G \quad \text{then} \quad T \vdash \bot$$

$$\text{If} \quad T \vdash A \to B \quad \text{and} \quad T \vdash A \quad \text{then} \quad T \vdash B$$
$$T \vdash \Box(A \to B) \to \Box A \to \Box B$$

$$\text{If} \quad T \vdash A \quad \text{then} \quad T \vdash \Box A$$
$$T \vdash \Box A \to \Box\Box A$$

We can also rewrite the consistency sentence Con_T as $\neg\Box\bot$.

8.8.4 Exercise

Here is a pretty basic logical fact: for any sentence A,

$$\text{If} \quad T \vdash A \quad \text{and} \quad T \vdash \neg A \quad \text{then} \quad T \text{ is inconsistent}$$

Use the facts about provability to show that T "knows" this fact. That is:

$$T \vdash \mathrm{Prov}\langle A\rangle \to \mathrm{Prov}\langle \neg A\rangle \to \mathrm{Prov}\langle \bot\rangle$$

In box notation:

$$T \vdash \Box A \to \Box \neg A \to \Box \bot$$

8.8.5 Exercise

We have already proved this fact (Exercise 8.6.4 (a)):

$$\text{If} \quad T \vdash G \quad \text{then} \quad T \vdash \bot$$

In this exercise, we'll show that the proof of this fact can be carried out within T.

(a) $T \vdash \Box G \to \Box \neg \Box G$
(b) $T \vdash \Box G \to \Box \bot$

8.8.6 Exercise (Gödel's Second Incompleteness Theorem)

Use the previous exercise and Exercise 8.6.4 (a) to show that if T proves the consistency sentence for T, then T is inconsistent. That is:

$$\text{If} \quad T \vdash \mathrm{Con}_T \quad \text{then} \quad T \vdash \bot$$

Or in other words:

$$\text{If} \quad T \vdash \neg \Box \bot \quad \text{then} \quad T \vdash \bot$$

Chapter 9

Second-Order Logic*

Ever and anon we are landed in particulars, but this is not what I want.

Plato (c. 428–c. 248 BCE), *Meno*

UNDER CONSTRUCTION

Overview:

1. The idea of second-order logic
2. Semantics for second-order logic
3. Second-order Peano Arithmetic (PA_2) does not have non-standard models
4. Thus PA_2 is negation-complete
5. Thus PA_2 is not effectively enumerable
6. Thus second-order logic has no sound and complete proof system
7. Type theory

9.1 Syntax and Semantics

- Second-order variables.
- Atomics of the form $Xt_1 \ldots t_n$.
- Second-order quantifiers.

- Second-order assignments.
- Semantic clauses for atomics and quantifiers.
- Second-order logical consequence and consistency. ($\vDash_2$)
- The second-order theory of a structure. (Th_2)
- Observe that Gödel's Fixed Point Theorem, Tarski's Theorem, the Representability Theorem, the Essential Undecidability Theorem, and Church's Theorem all still apply.

9.2 Second-order Peano Arithmetic

Contrast the Inductive Property (of numbers) and the first-order induction schema again.

Write out PA_2. The induction axiom:

$$\forall X(X0 \to \forall n(Xn \to X(\mathrm{suc}\, n)) \to \forall n\, Xn)$$

Note that PA_2 is sufficiently strong.

9.2.1 Definition
… **isomorphism** …

9.2.2 Definition
A theory is **categorical** iff all of its models are isomorphic to one another.

9.2.3 Exercise
PA_2 is categorical.

9.2.4 Exercise
Second-order consequence is not compact. That is, there is an infinite set of second-order sentences X such that every finite subset of X has a model, but X does not have a model.

9.2.5 Exercise

(a) $PA_2 = \mathrm{Th}_2\, \mathbb{N}$

(b) PA_2 is negation-complete.

9.2.6 Exercise

Note that the Essential Undecidability Theorem implies that PA_2 is undecidable (since it is sufficiently strong and consistent). Show:

(a) PA_2 is not semi-decidable.

(b) There is a decidable set X such that the set of sentences A such that $X \vDash_2 A$ is not semi-decidable.

9.2.7 Definition

Let a **proof system** be a set of triples (π, X, A)—we say that π is a *proof* of A from assumptions X—which satisfies the further principle that for any decidable set of sentences X, the set of pairs (π, A) such that π is a proof of A from X is *decidable*.

9.2.8 Definition

Let $\vDash$ be any relation between a set of formulas and a formula.

(a) A proof system is **sound** for $\vDash$ iff, for any X and A, if there is a proof of A from X, then $X \vDash A$.

(b) A proof system is **complete** for $\vDash$ iff, for any X and A, if $X \vDash A$, then there is a proof of A from X.

9.2.9 Exercise

No proof system is sound and complete for second-order logical consequence.

9.3 Further Directions

- The plural interpretation
- Type theory
- Higher-order identity. Extensionality and intensionality
- Henkin models

Chapter 10

Set Theory*

> There is no doubt in my mind that in this way we will get farther and farther ahead, never reaching an unsurmountable limit, but also attaining not even an approximate grasp of the absolute. The absolute can only be acknowledged, but never known, not even approximately.
>
> Georg Cantor, "Fundamentals of a General Theory of Manifolds: A Mathematical-Philosophical Study in the Theory of the Infinite" (1883)

UNDER CONSTRUCTION

Overview:

1. First order set theory ZFC.
2. Set theory has no *intended* model.
3. If ZFC is consistent, it has countable models. Skolem's Paradox.
4. If there are large cardinals, ZFC is consistent.
5. Thus (by Gödel's Second Incompleteness Theorem) ZFC does not prove there are large cardinals.
6. Some independence results (stated without proof): large cardinals, the Continuum Hypothesis.

6. Second-order set theory ZFC_2.
7. ZFC_2 does not have countable models
8. Zermelo's categoricity theorem.
9. Kreisel's Principle.

References

Breckenridge, Wylie, and Ofra Magidor. 2012. “Arbitrary Reference.” *Philosophical Studies* 158 (3): 377–400. https://doi.org/10.1007/s11098-010-9676-z.

Lewis, David. 1986. *On the Plurality of Worlds*. Oxford: Blackwell.

Russell, Bertrand. n.d. *The Philosophy of Logical Atomism*. 1st edition. London; New York: Routledge.

Smeding, Gideon Joachim. 2009. “An Executable Operational Semantics for Python.” PhD thesis, Utrecht, The Netherlands: Universiteit Utrecht.