The Limits of Logic

Jeffrey Sanford Russell
University of Southern California
Last revised November 19, 2018
Image: Wassily Kandinsky, *Circles in a Circle*, 1923
Contents

Preface vii
   The Big Picture ........................................ vii
   About This Text ......................................... xi
   Acknowledgments ....................................... xiv

1 Sets and Functions 1
   1.1 Sets ................................................. 2
   1.2 Functions ............................................ 8
   1.3 Ordered Pairs ....................................... 15
   1.4 Higher-Order Sets and Functions ................... 16
   1.5 Counting ............................................. 18
   1.6 The Self-Application Trick .......................... 27
   1.7 Simplifications of Set Theory* ...................... 34
   1.8 Review ............................................... 38

2 The Infinite 41
   2.1 Numbers and Induction ............................... 42
   2.2 Recursion ............................................ 48
   2.3 The Recursion Theorem* ............................. 55
   2.4 Sequences ............................................ 58
   2.5 Strings ............................................... 67
## CONTENTS

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.6</td>
<td>What the Minimal Theory of Strings Can Represent*</td>
<td>206</td>
</tr>
<tr>
<td>5.7</td>
<td>Syntax and Arithmetic</td>
<td>216</td>
</tr>
<tr>
<td>6</td>
<td>The Undecidable</td>
<td>221</td>
</tr>
<tr>
<td>6.1</td>
<td>Programs</td>
<td>222</td>
</tr>
<tr>
<td>6.2</td>
<td>Syntax and Semantics</td>
<td>238</td>
</tr>
<tr>
<td>6.3</td>
<td>The Church-Turing Thesis</td>
<td>249</td>
</tr>
<tr>
<td>6.4</td>
<td>The Universal Program</td>
<td>251</td>
</tr>
<tr>
<td>6.5</td>
<td>The Halting Problem</td>
<td>259</td>
</tr>
<tr>
<td>6.6</td>
<td>Semi-Decidable and Effectively Enumerable Sets</td>
<td>263</td>
</tr>
<tr>
<td>6.7</td>
<td>Decidability and Logic</td>
<td>267</td>
</tr>
<tr>
<td>7</td>
<td>The Unprovable</td>
<td>275</td>
</tr>
<tr>
<td>7.1</td>
<td>Proofs</td>
<td>276</td>
</tr>
<tr>
<td>7.2</td>
<td>Official Syntax</td>
<td>287</td>
</tr>
<tr>
<td>7.3</td>
<td>The Completeness Theorem</td>
<td>298</td>
</tr>
<tr>
<td>7.4</td>
<td>Models of Arithmetic*</td>
<td>307</td>
</tr>
<tr>
<td>7.5</td>
<td>The Incompleteness Theorem</td>
<td>308</td>
</tr>
<tr>
<td>7.6</td>
<td>Gödel Sentences</td>
<td>310</td>
</tr>
<tr>
<td>7.7</td>
<td>Rosser Sentences*</td>
<td>314</td>
</tr>
<tr>
<td>7.8</td>
<td>Consistency is Unprovable</td>
<td>314</td>
</tr>
<tr>
<td>8</td>
<td>Second-Order Logic*</td>
<td>319</td>
</tr>
<tr>
<td>9</td>
<td>Set Theory*</td>
<td>321</td>
</tr>
</tbody>
</table>

References 323
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 1.6.)

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 7.5) 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 undergraduates 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.”

In particular, these discoveries are not just part of the philosophy of mathematics. These Theorems are central facts in the philosophy of language and epistemology (and were clearly understood this way by their discoverers). 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, the minimal theory of arithmetic $\mathbb{Q}$ takes a back seat to the minimal theory of strings $\mathbb{S}$.) But the shift from theorems about numbers to theorems about strings is usually pretty straightforward, 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 5.7, 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 would be reasonable to skim over the first chapter quickly, and perhaps also the second. (But make sure not to skip Section 1.6!)

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 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 trivial and 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 really seem like the trivial details they are. 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 6.)

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

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.

In this chapter we’re working up to one of the most important foundational facts in set theory: every set has more subsets than it has elements (Section 1.6). This might not sound so exciting on its own, but it turns out to be a really rich idea. As we’ll see in the next chapter on infinity, it implies that if there are any infinite sets at all, then there is a whole hierarchy of different sizes of infinity. It shows that some very natural intuitive ideas about “collections” of things can’t all be true: for example, not every precise way of describing a bunch of things really picks out a set of things. And the trick we’ll use to prove this fact—the “self-application” trick—shows up again and again, throughout this text, but also in many other philosophical contexts, with surprising consequences for properties, propositions, possibility, knowledge, truth, and being.

Our first job is to get familiar with some basic techniques for working with sets. This will involve introducing some basic assumptions, or “axioms”, about what sets there are and what they are like, and some practice using these axioms to carefully show other simple facts about sets.

(It’s worth noting that 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 ??.
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

\[
\{0, \ 1 + 1, \ 2 + 3, \ 3 − 1\}
\]

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

\[
A = \{a_1, a_2, \ldots, a_n\}
\]

then this means that \( a_1, \ldots, a_n \) 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, “\( \text{blah} \) iff \( \text{zoom} \)” means the same thing as “if \( \text{blah} \), then \( \text{zoom} \), and if \( \text{zoom} \), then \( \text{blah} \)”.)
To know what a set is, you just have to know what elements it has. There are no two different sets with the very same elements.

1.1.3 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, \( A = B \).

1.1.4 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 \( A \subseteq C \).

(c) If \( A \) and \( B \) have exactly the same elements, this means that every element of \( A \) is an element of \( B \), and also every element of \( B \) is an element of \( A \). So \( A \subseteq B \) and \( B \subseteq A \). So by the Axiom of Extensionality, \( A = B \). For the other direction (the converse), suppose that \( A = B \). Then since \( A \) clearly has exactly the same elements as \( A \), and \( B \) just is \( A \), it follows that \( A \) has exactly the same elements as \( B \).

\[ \square \]

1.1.5 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.) The main tool we have for doing this is to use the Extensionality axiom: we show that every element of \( A \) is an element of \( B \), and we also show that every element of \( B \) is an element of \( A \).
CHAPTER 1. SETS AND FUNCTIONS

1.1.6 Example
If $A$ and $B$ are sets, then their **intersection** is a set $A \cap B$ whose elements are just those things $x$ such that $x \in A$ and $x \in B$. The **union** of $A$ and $B$ is a set $A \cup B$ whose elements are just those things $x$ such that $x \in A$ or $x \in B$. Show the following fact:

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

**Proof**
We will use Extensionality: in order to show that the set $A \cap (A \cup B)$ is the same set as $A$, we need to show:

(a) $A \cap (A \cup B) \subseteq A$. That is, every element of $A \cap (A \cup B)$ is an element of $A$.
(b) $A \subseteq A \cap (A \cup B)$. That is, every element of $A$ is an element of $A \cap (A \cup B)$.

For part (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.

For part (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. □

1.1.7 Exercise
Using the definitions from Example 1.1.6, 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 $C \subseteq A \cap B$.

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,

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

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 Axiom of Separation

For any set \( A \), and any property \( F \), there is a set whose elements are just those elements \( a \) of \( A \) which are \( F \). This set is labeled

\[ \{ a \in A \mid F(a) \} \]

In other words:

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 \( F(a) \).

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 4.4). 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 4).

A second way of restating the Axiom of Separation uses what’s called *second-order logic* (Chapter 8). 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.

1.1.10 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:

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

1.1.11 Example
Suppose that $a_1, \ldots, a_n$ are elements of $A$ (though $A$ may have other elements as well). Then there is a set $\{a_1, \ldots, a_n\}$. That is, there is a set $B$ whose elements are all and only the elements $a_1, \ldots, a_n$. We can show this using Separation. To put the fact another way: there is a set $B$ such that, for any object $a$, $a$ is an element of $B$ iff it is $a_1$, or it is $a_2$, or $\ldots$, or it is $a_n$. So, if we use the Axiom of Separation and let

$$B = \{a \in A \mid a = a_1 \text{ or } a = a_2 \text{ or } \ldots \text{ or } a = a_n\}$$

then $B = \{a_1, \ldots, a_n\}$.

1.1.12 Technique (Defining a Subset)
Suppose we have a set $A$, and we want to show there is a subset of $A$ that satisfies some property. The main tool we have for doing this is the Separation axiom. What
we have to do is come up with some property $F$ that the elements of the subset would have in common, and which would distinguish the elements of the subset from any other elements of $A$. Then we can define the subset to be $\{ a \in A \mid F(a) \}$.

1.1.13 Exercise
Let $U$ be a set, and let $A$ and $B$ be subsets of $U$. Use the Axiom of Separation to show that $A$ and $B$ have a union and an intersection (as defined in Example 1.1.6).

1.1.14 Empty Set Axiom
There is a set with no elements. This is called the empty set. It is labeled $\emptyset$ 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 $\{a \in A \mid a \neq a\}$.)

1.1.15 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 $A = A'$. To do this, we can use Extensionality. We know that since $A$ has no elements, each of its elements is an element of $A'$. (A counterexample to this would be an element of $A$ which is not an element of $A'$, and clearly there are no such things.) Similarly, since $A'$ has no elements, every one of its elements is an element of $A$. So by Extensionality, $A = A'$. □

1.1.16 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 prove 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 $\{0, 1, 2, \ldots\}$.

(Sometimes functions are defined to be certain special sets—for instance, as sets of ordered pairs—see Section 1.7 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 Stresjkj. 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
1.2. FUNCTIONS

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 \( f : A \rightarrow B \) is the set of elements of \( B \) that are assigned by \( f \) to some element of \( A \). That is,

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

Or in even more concise notation,

\[
\text{range } f = \{ f a \mid a \in A \}
\]

(What is the range of the successor function?)

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 \( f : A \rightarrow B \) by specifying the value of \( f \) for each element of \( A \). For instance, we could say

\[
\begin{align*}
    f(1) &= \text{Los Angeles} \\
    f(2) &= \text{San Diego} \\
    f(3) &= \text{San Jose}
\end{align*}
\]

Or we could define a function \( g : A \rightarrow B \) like this:

\[
\begin{align*}
g(n) &= \text{the } n \text{th largest city in California (in 2015)} \quad \text{for every } n \in \{1, 2, 3\}
\end{align*}
\]

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. This shows that \( f \) and \( g \) are the same function—using the following general principle about functions, which is analogous to the Axiom of Extensionality.

1.2.4 Axiom of Function Extensionality
For any functions \( f : A \rightarrow B \) and \( g : A \rightarrow B \), if \( f a = g a \) for every \( a \in A \), then \( f = g \).

1.2.5 Technique (Proving Functions are Equal)
If \( f \) and \( g \) are functions from \( A \) to \( B \), the main way to prove \( f = g \) is to use the Axiom of Function Extensionality: we show that \( f \) and \( g \) have the same “output” for each possible “input”. The proof usually goes like this:
Let $a$ be any element $A$. Then [fill in reasoning]. So $f_a = g_a$. So by Function Extensionality $f = g$.

That’s how we know $f$ and $g$ are equal in our cities example (Example 1.2.3). But how do we know that there is any such function at all? How do we know that, in addition to the numbers 1, 2, and 3, and the cities Los Angeles, San Diego, and San Jose, there is another thing, a function, which links the numbers to the cities? Here is the reasoning we implicitly relied on.

For each $n \in \{1, 2, 3\}$, there is some city which is the $n$th largest in California. So there is a function $g$ such that, for each $n \in \{1, 2, 3\}$, $gn$ is the $n$th largest city in California.

We can generalize this reasoning with the following principle.

1.2.6 Axiom of Choice

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 $f : A \rightarrow B$ 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. How should we understand the $F$? Intuitively, it stands for a relation between things in $A$ and things in $B$. 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.

Here are some examples of the Axiom of Choice at work:

- 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 $b$.

- 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 $n$, $fn$ is two more than $n$. 
These two examples each describe a unique object for each object in the codomain: a building has only one address, and a number has only one number which is two more than it. But this isn’t necessary.

- 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 $A$, $hA \in A$.

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

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, 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 second 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 some 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.

1.2.7 Technique (Defining a Function)

The most common way to prove the existence of a function $f : A \to B$ is to define $f$ explicitly, by saying precisely what the value of $fa$ is for each “input” $a$ in $A$. For example, we can define the function that takes each number to the number six more than it, by saying

$$fn = n + 6$$

for each number $n$. 
When we define a function this way, we are really using both Choice and Function Extensionality. Choice tells us that there is \textit{at least one} function that satisfies this definition. Since for each number \( n \), there is some number which is equal to \( n + 6 \), Choice tells us that there is at least one function \( f \) such that \( fn = n + 6 \) for each number \( n \). Function Extensionality tells us that there is \textit{at most one} function that satisfies this definition. For suppose that there was some other function \( f' \) such that

\[
f'n = n + 6 \quad \text{for each number } n
\]

In that case, \( fn = n + 6 = f'n \) for each number \( n \). So, by Function Extensionality, since \( f \) and \( f' \) have the same output for each input, they are the very same function.

1.2.8 Notation

If a set \( A \) is finite, then one way we can define a function from \( A \) to \( B \) is just by explicitly listing its value for each element of \( A \). For example, we did this for the function \( f \) from \( \{1, 2, 3\} \) to cities in California in Example 1.2.3. Here’s some notation which is handy for this case:

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

1.2.9 Exercise

Suppose \( 2 \) is a set with exactly two elements, which we’ll call True and False. We can think of functions to \( 2 \) 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 \( 2 \) which we call the \textbf{characteristic function} of \( X \), or \( \text{char } X : A \rightarrow 2 \) 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 \( f : A \rightarrow 2 \) is a function, we can define a subset of \( A \) that includes just the things that pass the \( f \)-test. This is called the \textbf{kernel} of \( f \), or \( \text{ker } f \).

\[
\text{ker } f = \{ a \in A \mid fa = \text{True} \} 
\]
Show that for any set \( X \subseteq A \),

\[
\ker(\text{char } X) = X
\]

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 \( b = \text{address } b' \), 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 \( \texttt{alfkj/404.html} \). We say that this function is not onto: its range does not completely “cover” the set of strings of symbols.

**1.2.10 Definition**

(a) A function \( f : A \to B \) is one-to-one (or injective) iff for each \( a, a' \in A \), if \( f a = f a' \) 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 \( f a = b \)

(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.

**1.2.11 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.

1.2.12 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 $B$.

1.2.13 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:

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

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,

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

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

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

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.

1.2.14 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 $C$. 

□
1.3. ORDERED PAIRS

1.2.15 Exercise
If \( f : A \to B \) is a one-to-one correspondence, then there is a one-to-one correspondence from \( B \) to \( A \).

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 0, 1, and 2 be sets with 0, 1, and 2 elements, respectively. How many elements do the following sets have? Explain your answers.

(a) \(1 \times 2\)
(b) \(2 \times 0\)
(c) \((2 \times 2) \times 2\)
1.3.3 Exercise
Show that there is a one-to-one correspondence between $A \times B$ and $B \times A$.

1.3.4 Exercise
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 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 even 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 $A \rightarrow B$.

1.4.5 Exercise
For each function $f : A \rightarrow B$, 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

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

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 1.2.9 we defined the characteristic function for a subset $X \subseteq A$ to be a certain function $\text{char } X : A \rightarrow 2$. So this defines a higher-order function:

$$\text{char} : PA \rightarrow 2^A$$

Show that this function is one-to-one and onto.
1.5 Counting

Consider this function (from Example 1.2.3)

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

This function is a one-to-one correspondence between the sets

\[ A = \{1, 2, 3\} \]

\[ B = \{\text{Los Angeles, San Diego, San Jose}\} \]

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.

1.5.1 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 \( A \sim B \).

Notice that this definition doesn’t actually say anything about numbers. You can think of 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” will even make sense when we apply it to infinite sets, which don’t have any finite number of elements. (We will look at ways of counting the elements of infinite sets in Section 2.9.)

1.5.2 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 to say,

\[ A \times B \sim B \times A \]

(b) 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 to say,

\[ A \sim \{(a_1, a_2) \in A \times A \mid a_1 = a_2\} \]
1.5. **COUNTING**

(c) 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 to say,

$$ \mathcal{P} A \sim 2^A $$

1.5.3 Example
For any set $A$,

$$ A^2 \sim A \times A $$

That is, there are just as many ordered pairs of elements of $A$ as there are functions from a two-element set to $A$.

*Proof*
Let’s call the two elements of 2 “1” and “2”. The 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.

To be very precise, we can define a function from $A^2$ to $A \times A$, and then show that it is one-to-one and onto. We can define this function $f$ as follows:

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

We’ll show that $f$ is one-to-one an onto.

To show that $f$ is one-to-one, suppose that $h$ and $h'$ are each functions from 2 to $A$, and $f h = f h'$. 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 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 $A^2$ such that $f h = (a, a')$. And there is: we can let

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

Then $f h = (h1, h2) = (a, a')$, which is what we wanted. \qed
1.5.4 Exercise
Let 1 be a set with exactly one element. For any set $A$,

$$A \sim A^1$$

1.5.5 Exercise
For any sets $A$, $B$, and $C$,

$$(A \times B) \times C \sim A \times (B \times C)$$

1.5.6 Exercise
For any sets $A$, $B$, and $C$:

$$C^{A \times B} \sim (C^B)^A$$

That is, there is a one-to-one correspondence between \textit{two-place} functions from $A \times B$ to $C$, and \textit{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.)

1.5.7 Exercise
For any sets $A$, $B$, and $C$, if $A \sim B$ then $C^A \sim C^B$ and $A^C \sim B^C$.

(The main tricky point about this exercise is handling higher-order functions carefully.)

We should check that our definition of “same number of elements” has some general properties we would reasonably expect.

1.5.8 Exercise
Let $A$, $B$, and $C$ be sets.

(a) $A \sim A$.
(b) If $A \sim B$ and $B \sim C$ then $A \sim C$.
(c) If $A \sim B$ then $B \sim A$.

\textit{Hint.} Use facts from Section 1.2.
The set 
\[ A = \{1, 2, 3\} \]
has the same number of elements as the set 
\[ B = \{\text{Los Angeles, San Jose, San Diego}\} \]

The set 
\[ C = \{1, 2\} \]
has fewer elements than \( B \). The set of cities \( B \) contains a distinct element for each element of \( C \), and more besides.

In general, sets can be ordered based on their sizes: some sets are smaller, and some are bigger.

1.5.9 **Definition**

For any sets \( A \) and \( B \), \( A \) has no more elements than \( B \) (abbreviated \( A \leq B \)) iff \( A \) has the same number of elements as some subset of \( B \).

We also say \( B \) has at least as many elements as \( A \) (\( B \geq A \)) in this case.

Here is another way of putting this.

1.5.10 **Proposition**

For any sets \( A \) and \( B \), \( A \leq B \) iff there is a one-to-one function from \( A \) to \( B \).

**Proof**

This immediately follows from Exercise 1.2.12, which says that there is a one-to-one function from \( A \) to \( B \) iff there is a one-to-one correspondence from \( A \) to some subset of \( B \). \( \square \)

In practice, you can use whichever of these two ways of thinking about the “no more elements relation” happens to be more convenient.

1.5.11 **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,

\[ A \leq A \times A \]
(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 \( A \),

\[ A \leq P A \]

### 1.5.12 Exercise

Let \( A \) and \( B \) be sets. If \( B \) is not empty, then

\[ A \leq A \times B \]

Let’s also look at a couple more equivalent ways of describing the “no more elements” relation, which can be useful.

### 1.5.13 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 “\( f \) has a value for \( a \)”.)

### 1.5.14 Proposition

There is a one-to-one function from \( A \) to \( B \) iff there is a partial function from \( B \) to \( A \) which is onto.

**Proof**

First we’ll prove the left-to-right direction:

If there is a one-to-one function from \( A \) to \( B \), then there is a partial function from \( B \) to \( A \) which is onto.

Suppose that \( f : A \to B \) is a one-to-one function. Then \( f \) is a one-to-one correspondence from \( A \) to the range of \( f \), which is a subset of \( B \) (Exercise 1.2.12). We also know that if there is a one-to-one correspondence from \( A \) to the range of \( f \), then there is also a one-to-one correspondence \( g \) from the range of \( f \) to \( A \) (Exercise 1.2.15). In particular, \( g \) is onto. So \( g \) is a partial function from \( B \) to \( A \) which is onto, which is what we needed to show.

Next we’ll prove the right-to-left direction:
If there is a partial function from $B$ to $A$ which is onto, then there is a one-to-one function from $A$ to $B$.

Suppose that $g$ is a partial function from $B$ to $A$ which is onto. That means:

For each $a \in A$, there is some $b \in B$ such that $b$ is in the domain of $g$ and $gb = a$.

Then, using the Axiom of Choice, we can conclude:

There is a function $f : A \to B$ such that for each $a \in A$, $fa$ is in the domain of $g$ and $g(fa) = a$

Now we just have to check that this function $f$ is one-to-one. Let $a$ and $a'$ be any elements of $A$, and suppose that

$$fa = fa'$$

Then we can apply $g$ to both sides:

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

We chose $f$ so that $g(fa) = a$ and $g(fa') = a'$. So this tells us $a = a'$. Thus $f$ is a one-to-one function from $A$ to $B$. □

1.5.15 Proposition

Suppose that $f$ is a partial function from $A$ to $B$. If $B$ is not empty, then there is a total function $f^+$ from $A$ to $B$ that extends $f$: that is, for each element $a$ which is in the domain of $f$, $f^+a = fa$.

Proof

We can define the extended function $f^+$ piecewise. We already know what $f^+$ should do on the piece of $A$ where $f$ is defined. So we just have to come up with something for $f^+$ to do on the piece of $A$ where $f$ is not defined. Since $B$ is not empty, it has at least one element. So we can let $b_0$ be an element of $B$. Now we can define a function $f^+$ like this: for each $a \in A$,

$$f^+a = \begin{cases} fa & \text{if } a \text{ is in the domain of } f \\ b_0 & \text{otherwise} \end{cases}$$

This defines a total function from $A$ to $B$, and it is clear from the definition that if $a$ is in the domain of $f$, then $f^+a = fa$. □

Now let’s put these facts together.
1.5.16 Exercise

For any sets $A$ and $B$, the following are equivalent:

(a) $A$ has no more elements than $B$: that is, $A \leq B$.
(b) There is a one-to-one correspondence between $A$ and some subset of $B$.
(c) There is a one-to-one function from $A$ to $B$.
(d) Either there is an onto function from $B$ to $A$, or $A$ is empty.
(e) There is a partial function from $B$ to $A$ which is onto.

*Hint.* The fact that (a), (b), (c), (d), and (e) are equivalent means that twenty different “if” statements are true: (a) $\Rightarrow$ (b), (b) $\Rightarrow$ (a), (a) $\Rightarrow$ (c), and so on. But you don’t have to show this with a twenty part proof! You just need to prove enough of the “if” statements so that the rest of them obviously follow from the ones you’ve done. For this exercise, one reasonable approach is to show the following four things (using facts we’ve already shown):

1. (a) $\Leftrightarrow$ (b)
2. (b) $\Leftrightarrow$ (c)
3. (c) $\Leftrightarrow$ (e)
4. (d) $\Leftrightarrow$ (e)

Exercise 1.5.16 gives us five different equivalent ways of saying that $B$ is at least as big as $A$, or $A$ is no bigger than $B$. This equivalence is a very useful thing to know. It means that whenever you know one of these five facts about some sets $A$ and $B$, you can immediately conclude any of the other four. You can use whichever version of the “at least as big” property is most useful for the job you are doing.

Now that we know these different properties are all equivalent, in practice you might as well just forget which of them was the official definition of “$A \leq B$”. You can think of the definition of $A \leq B$ as being all of these three properties—or whichever one of them happens to be easiest to show. It doesn’t matter, since they are equivalent.

Again, we should check that this notion of “no more elements” has some properties we would reasonably expect (and which our $\leq$ notation strongly suggests).
1.5. COUNTING

1.5.17 Exercise
Let $A$, $B$, and $C$ be sets.

(a) $A \subseteq A$.
(b) If $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$.
(c) If $A \sim B$, then $A \leq B$ and $B \leq A$.
(d) If $A \sim B$ and $A \leq C$, then $B \leq C$.

*Hint.* Use facts from Section 1.2 again.

1.5.18 Exercise

(a) For any function $f$, the range of $f$ has no more elements than the domain of $f$.

\[ \text{range } f \leq \text{domain } f \]

(b) For any sets $A$ and $B$, $A \leq B$ 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.

1.5.19 Definition

For any sets $A$ and $B$, $A$ has (strictly) fewer elements than $B$ (abbreviated $A < B$) iff $A$ has no more elements than $B$ ($A \leq B$), but $A$ does not have at least as many elements as $B$ ($A \not\leq B$).

We also say $B$ has strictly more elements than $A$ ($B > A$) in this case.

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.

- **Wrong.** If there is a function from $A$ to $B$ which is one-to-one but *not* onto, then $A < B$. 
• **Wrong.** If there is a one-to-one function from \( A \) to a proper subset of \( B \), 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. We’ll discuss this in detail in Section 2.7 and Section 2.9. For now, just be careful!

Again, we should check that this notion of “bigger” has some properties that we would reasonably expect.

**1.5.20 Exercise**

Let \( A \), \( B \), and \( C \) be sets.

(a) If \( A \leq B \) and \( B < C \), then \( A < C \).

(b) If \( A < B \) and \( B \leq C \), then \( A < C \).

Here are a couple more facts about comparing sizes of sets. These 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.

**1.5.21 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 \( A \leq B \) and \( B \leq A \), then \( A \sim B \).

**1.5.22 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, \( A \leq B \) or \( B \leq A \).

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 2.10, if you are interested.

1.6 The Self-Application Trick

We know how to show that two sets are the same size: we just have to define a one-to-one correspondence between them. But how would we ever show that two sets are different sizes? To show this we would have to show that no function from one to the other is a one-to-one correspondence. For particular finite sets, like \{1, 2\} and \{Los Angeles, San Jose, San Diego\}, we can do this by just checking each function between them one by one. (There are nine of them in this case.) This is tedious, but it would work. But we will also be working with infinite sets. So we will need some more general way of working out whether two sets are the same size or different sizes.

(Historically it was often thought that, either the notion of size makes no sense at all for infinite sets, or else they are all of the same size—\textit{infinite}. But as we’ll see in Section 2.9, this was wrong: there are many different sizes of infinity. The theorem we’re about to prove is our main tool for showing this.)

Our main tool for showing that two sets are different sizes is called \textit{Cantor’s Theorem}. What it says is that every set has strictly more subsets than elements. In other words, every set is smaller than its power set. Or more briefly, for any set \(A\),

\[ A < P A \]

This fact and the way it is proved are both very important. The fact is important, because it underlies a fundamental technique called “counting arguments” which we will use many times. We’ll see some examples of this technique at the end of this section. The way it is proved is important, because this same trick—the \textit{self-application} trick, which is often called “diagonalization”—is also used for almost all of the central theorems in this course, about what is inexpressible, undecidable, and unprovable. So it’s worth going slowly to make sure we really understand what is going on here. The argument is short, but it is devious. Let’s look at it from several different perspectives.
We’ve already shown that \( A \leq PA \), that is, that the power set of \( A \) has at least as many elements as \( A \) (this was Exercise 1.4.3). The important additional step here is to show that \( PA \) has strictly more elements than \( A \), which means we need to show that \( PA \negleq A \). You should remember that this means that there is no onto function from \( A \) to \( PA \). (Exercise 1.5.16. This is using the fact that \( PA \) can’t be empty, since every set has at least one subset—for example, \( A \) is a subset of itself.) So we will let \( f : A \to PA \) be any function from \( A \) to its power set, and we will try to show that \( f \) is not onto. In other words, what we want to show is that, whatever function \( f \) may be, there is some subset of \( A \) which is not in the range of \( f \).

Here’s a puzzle.\(^1\) 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 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.

What does this have to do with our problem? 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 \( A \),

\[ a_2 \in f a_1 \iff a_1 \text{ shaves } a_2 \]

Is the shaving function onto? If it is onto, this means that for any set of people in the community, there is somebody who shaves exactly those people. But the “barber paradox” gives us an example of a set that is not in the range of the shaving function: namely, the set of people who do not shave themselves. We can represent this set in symbols like this:

\[ X = \{ a \in A \mid a \notin f a \} \]

For any function \( f \), this set \( X \) is a perfectly fine set (using the Axiom of Separation). But anybody who shaved exactly the people in \( X \) would be like the paradoxical barber, which is impossible. So the set \( X \) is not in the range of the shaving function. Thus the shaving function is not onto.

\(^1\)Russell (n.d., 101).
1.6. THE SELF-APPLICATION TRICK

Furthermore, this argument didn’t really depend on anything special about the set of people or the shaving function. The same kind of argument shows that for any set $A$, and any function $f : A \to PA$, there is some set $X \in PA$ which is not in the range of $f$. Your task in Exercise 1.6.1 is to clearly spell out the general version of this argument.

Here is a second perspective on Cantor’s argument, using the picture that gives the technique the name “diagonalization.” Take a simple example where the elements of $A$ are Al, Bea, and Cece. (In this simple case we could show that $PA > A$ just by counting up the subsets—there are eight, which is clearly more than three—but we want to do things in a way that doesn’t really depend on the set being so small, so that we can generalize the argument to arbitrary sets.) Here is an example of a function like this:

- $Al \mapsto \{ Al, Bea \}$
- $Bea \mapsto \{ \}$
- $Cece \mapsto \{ Al, Cece \}$

(The unusual spacing is just to keep the same elements visually lined up for each set.) Our goal is to come up with a rule for finding a subset of $A$ which is not in the range of $f$, in a way which will work not just for this example, but for any choice of a set $A$ and a function $f : A \to PA$. That’s what we’ll try to do now.

First, it’s helpful to represent this function in a slightly different way. As we showed in Exercise 1.2.9, 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:

<table>
<thead>
<tr>
<th></th>
<th>Al</th>
<th>Bea</th>
<th>Cece</th>
</tr>
</thead>
<tbody>
<tr>
<td>$Al$</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
<tr>
<td>$Bea$</td>
<td>False</td>
<td>False</td>
<td>False</td>
</tr>
<tr>
<td>$Cece$</td>
<td>True</td>
<td>False</td>
<td>True</td>
</tr>
</tbody>
</table>

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 $f(Al)$. Row $Cece$, column $Al$ says True, because Al is an element of $f(Cece)$. 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 $A_l$, column $A_l$; row $B_{ea}$, column $B_{ea}$; row $C_{ce}$, column $C_{ce}$. 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 $A_l$-row about $A_l$, and disagrees with the $B_{ea}$-row about $B_{ea}$, and disagrees with the $C_{ce}$-row about $C_{ce}$. 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 $A_l$, which does contain $B_{ea}$, and which does not contain $C_{ce}$—that is, in this case it’s the set $\{B_{ea}\}$.

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 $f a$. So this picture has brought us back to the very same set as the barber paradox: the set $\{a \in A \mid a \not\in f a\}$.

Here’s a third perspective on Cantor’s argument, which connects it to another idea we will explore more deeply later on (Section 5.4). 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

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

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.

The idea of Cantor’s theorem is based on 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 $f : A \to P D$ 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
1.6. THE SELF-APPLICATION TRICK

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 \textit{non-self-applying} applies just to those adjectives which do not self-apply. For all adjectives \(a\):

\[
\text{non-self-applying applies to } a \iff a \text{ does not apply to } a \quad (**)
\]

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, \textit{non-self-applying} is an expression we can understand, that applies to some adjectives, like \textit{short} and not others, like \textit{long}. So what else could its extension be, if not the set of adjectives which do not self-apply?)

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

\[
X = \{a \in A \mid a \notin f(a)\}
\]

The reasoning we just went through shows that there is no adjective \(a\) such that \(fa = X\). So \(f\) is not onto.

Again, this reasoning did not really depend on which set \(A\) was, or which function \(f\) was. We can generalize this argument to show that for any set \(A\), and for any function \(f : A \to PA\), there is some set \(X \in PA\) which is not in the range of \(f\). This general reasoning shows that there is no onto function from \(A\) to \(PA\), so \(PA \not\subseteq A\). Again, your task in Exercise 1.6.1 is to spell out the general version of this argument.

1.6.1 Exercise (Cantor’s Theorem)
For any set \(A\), \(A < PA\).

1.6.2 Exercise
For any set \(A\), if 2 is a set with two elements, then \(A < 2^A\).
1.6.3 Exercise
Use Cantor’s Theorem to show that there is no set of all sets.

1.6.4 Example
Let $W$ be some set of words, and let $D$ be some set of objects. Sets of words are a kind of object, so let’s suppose in particular that that each set of words is one of the objects in $D$. Let $i : W \to D$ be an interpretation function from words to objects; for each $w \in W$, $iw$ is the object that $w$ stands for, the interpretation of $w$. Let $I \subseteq D$ be the set of objects that are the interpretation of some word. That is, $I$ is the range of the interpretation function. Show that there is some object that is not the interpretation of any word. In other words, $I \neq D$.

Proof
Since every set is the same size as itself, to show that $I \neq D$, it’s enough to show that $I$ and $D$ are different sizes. In particular, we can show that $I < D$.

Since $I$ is the range of the interpretation function, the interpretation function $i$ is an onto function from $W$ to $I$. So $I \leq W$: there are no more interpretations of words than there are words.

By Cantor’s Theorem, $W < PW$. Furthermore, $PW$ is a subset of $D$, so $PW \leq D$. So putting this together:

$$I \leq W < PW \leq D$$

So $I < D$, and thus $I$ and $D$ are different sets, which means the interpretation function is not onto.

1.6.5 Technique (Counting Arguments)
Cantor’s Theorem is a powerful tool for showing that two sets are of different “sizes”—in the sense that there is no one-to-one correspondence between them. Sometimes this is useful as a step on the way to an even simpler fact: that two sets are distinct. This is called a counting argument.

Here is the standard shape of this kind of argument, in the abstract. Suppose you want to show that there is something which is in a set $B$, but not in another set $A$. One way to show this is to show that $A$ is strictly smaller than $B$. If you can show that $A < B$, then it follows that $B$ is not a subset of $A$, which means that $B$ has an element that is not in $A$, which was the goal.

1.6.6 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. 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**
We will use a counting argument: in particular, we will show that the set of definable sets of strings is strictly smaller than the set of all sets of strings, and so there is at least one undefinable set of strings.

Let \( D \) be the set of definable sets of strings. For every definable set of strings \( X \in D \), there is some description \( d \in A \) such that, for every string \( s \),

\[
\text{\( d \) is true of \( s \) \quad \text{iff} \quad s \in X}
\]

(That’s just what it means for a set to be definable.) Thus there is a function \( f : D \rightarrow A \) from definable sets to descriptions such that, for each definable set of strings \( X \in D \),

\[
\text{\( f(X) \) is true of \( s \) \quad \text{iff} \quad s \in X}
\]

We can also show that this function \( f \) is one-to-one. If \( X \) and \( Y \) are both sets of strings, and \( f(X) = f(Y) \), then for any string \( s \),

\[
\text{\( s \in X \) iff the description \( f(X) \) is true of \( s \)}
\]

\[
\text{iff the description \( f(Y) \) is true of \( s \) (because \( f(X) = f(Y) \)}
\]

\[
\text{iff \( s \in Y \)}
\]

This shows that \( X = Y \). So \( f \) is one-to-one.

Since there is a one-to-one function from \( D \) to \( A \), this tells us that \( D \leq A \). But also, each description is a string, which means \( A \subseteq S \). So we have

\[
D \leq A \leq S < P S
\]

That is, the set of definable strings has strictly fewer elements than the set of all strings, which is what we wanted to show.

\[\square\]

### 1.6.7 Exercise (Undecidable Sets)
Let \( S \) be a set of strings. Suppose that \( S \) includes two different strings True and False. Let \( P \) be a set of programs, and suppose that each program is a string. For each program \( A \), there is a partial function from strings to strings, which we call the denotation of \( A \), or \( \llbracket A \rrbracket \) for short. If the function \( \llbracket A \rrbracket \) is defined for a
string \( s \), then its value \( \llbracket A \rrbracket(s) \) is called the \textit{result of running} \( A \) \textit{with input} \( s \).

If \( X \) is a set of strings, then we say \( X \) is \textit{decidable} iff there is some program \( A \) such that the result of running \( A \) with input \( s \) is \texttt{True} for each string \( s \in X \), and the result of running \( A \) with input \( s \) is \texttt{False} for each string \( s \notin X \). To put that more succinctly, \( X \) is decidable iff there is some program \( A \in \mathcal{P} \) such that

\[
\llbracket A \rrbracket(s) = \begin{cases} \texttt{True} & \text{if } s \in X \\ \texttt{False} & \text{if } s \notin X \end{cases}
\]

If there is no program \( A \) like this, then \( X \) is called \textit{undecidable}.

Given these assumptions, use a counting argument to show that there is at least one undecidable set of strings.

\textit{Hint.} Let \( D \) be the set of all decidable sets, and prove that \( D < \mathcal{P} \mathcal{S} \).

\subsection*{1.6.8 Exercise (Kaplan’s Paradox)}

Let \( P \) be a set of \textit{propositions}, and let \( W \) be a set of \textit{possible worlds}. We’ll consider two relations between propositions and possible worlds. First, a proposition can be \textit{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 \( w \) \textit{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, use a counting argument to 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.

\subsection*{1.7 Simplifications of Set Theory*}

\textbf{UNDER CONSTRUCTION.}

We have introduced many different principles about sets as “axioms”. But these principles are not all \textit{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 Extensionality 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 1.3.5: 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:

1.7.1 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 $a = b$).
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 $(a, a)$.

### 1.7.2 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.

1.8 Review

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 \(f : A \rightarrow B\)) then \(f\) maps each element \(a\) in \(A\) to some element \(fa\) in \(B\).
1.8. REVIEW

- 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 \( A \).

- Sets \( A \) and \( B \) have the same number of elements \((A \sim B)\) iff there is a one-to-one correspondence between them.

- \( A \) has no more elements than \( B \) \((A \leq B)\) iff any of the following equivalent conditions hold:
  
  \(A\) 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:
  
  \(A \leq B\) and \( B \nleq A \).
  
  (b) \( A \leq B \) and \( A \napprox B \).

- Cantor’s Theorem. Every set has strictly more subsets than it has elements. That is, \( A < PA \).

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 \( fa = ga \) for every element \( a \in A \).

• You can define 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 define 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 \( B \).

• 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.

• You can prove existence facts using a counting argument. You can show that \( B \) has elements that are not in \( A \) by showing that \( A < B \).
Chapter 2

The Infinite

In this chapter we’ll explore some important infinite sets. 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, because human languages are productive. Here’s a very simple example:

(1) Snow is white.
(2) Snow is white, and snow is white.
(3) 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 productive, just like English: there are infinitely many 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. We will also encounter the striking fact that there are infinitely many different sizes of infinity. This fact is deep and beautiful, but also surprisingly practical.

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, … and so on. (In these notes, 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. Sometimes for convenience we’ll look at sequences of numbers starting from one, instead.)

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 \( \text{suc } n \) for the successor of the number \( n \). (Some people use the notation \( n' \) 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 \( \text{suc } n \) as \( n + 1 \) would be circular.)

For every number \( n \), \( \text{suc } n \) is a number. This means we have a function \( \text{suc} : \mathbb{N} \to \mathbb{N} \). 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.

\[
1 = \text{suc} 0 \\
2 = \text{suc} 1 = \text{suc} \text{suc} 0 \\
3 = \text{suc} 2 = \text{suc} \text{suc} \text{suc} 0 \\
\vdots
\]

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.

\[
\begin{align*}
0 \text{ is a number} & \quad & n \text{ is a number} \\
\text{suc } n \text{ is a number} & & \\
\end{align*}
\]

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 \( \text{suc } n \) 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 \( \text{suc } n = n \).

Proof

We’ll prove this by induction. What we want to show is that every number \( n \) has a certain property: namely, the property that \( \text{suc } n \neq n \). Let’s call a number nice if 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. We can show this in two steps.

The first step is easy: we’ll show that zero is nice. That is, we’ll show that \( \text{suc } 0 \neq 0 \). This is guaranteed by the Injective Property, which says that zero is not the successor of any number—including zero itself.

The second step is a little trickier: we’ll show that niceness is inherited by successors. That is, we’ll show that whenever any number \( n \) is nice, the next number after \( n \) is also nice. Let \( n \) be an arbitrary number, and suppose that \( n \) is nice: that is, \( \text{suc } n \neq n \). We want to show that \( \text{suc } n \) is nice. That is, we want to show:

\[
\text{suc(suc } n) \neq \text{suc } n
\]

The Injective Property says that the successor function is one-to-one. Furthermore, we have assumed that \( \text{suc } n \) and \( n \) are different numbers. So by the Injective Property, \( \text{suc } n \) and \( n \) must also have different successors. This is exactly what we wanted to show: \( \text{suc } n \) is nice.

This shows two things.

1. 0 is nice.
2. For every number \( n \), if \( n \) is nice, then \( \text{suc } n \) 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.

0 is nice

Then the second step tells us:

If 0 is nice, then 1 is nice

(since $1 = \text{suc } 0$). Putting these two facts together:

1 is nice

The second step also tells us:

If 1 is nice, then 2 is nice

(since $2 = \text{suc } 1$). So, putting these two facts together, we can conclude:

2 is nice

And the second step also tells us:

If 2 is nice, then 3 is nice.

And thus:

3 is nice

And obviously we can keep going. In the same way, we can use the 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 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. \qed

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 $n = \text{suc } m$.

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 \textit{good} iff either \( n = 0 \) or else there is some number \( m \) such that \( n = \text{suc} m \). We want to show that every number is good. Again, we can do this in two steps.

For the first step, 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 second 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 \( \text{suc} n \) is either zero or a successor. Again, this is obvious, because obviously \( \text{suc} n \) is the successor of some number (namely \( n \)).

Like before, these two steps guarantee that every number is good. The first step tells us that zero is good. The second step tells us that, if zero is good, so is one. The second 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 reach every number. So every number is good.

\[\square\]

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, \textit{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 \( \text{suc} n \) 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.
If you aren’t 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

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**.\(^1\)

---

\(^1\)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. Later on (in Section 4.4) we’ll encounter some other principles that are also sometimes called “the Peano Axioms”, but which don’t say anything about sets.
2.1.7 Axiom of Numbers

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

(a) \textbf{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) \textbf{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 \textbf{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 \textbf{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.

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

2.2 Recursion

Another fundamental technique we’ll use when working with inductive structures such as numbers and sequences is \textbf{recursive definition}. This is very closely related
2.2. RECURSION

to inductive proof. Proof by induction is a way of showing that a certain property applies to every number. 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 a 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 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 double \( \text{suc } n \). That is, we suppose that we know double \( n \), and we say what 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:

\[
\text{double(\text{suc } n)} = \text{suc suc double } n
\]

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 double 3 using these rules. We know 3 = suc 2, and 2 = suc 1, and 1 = suc 0. So we can work it out like this:

\[
\begin{align*}
\text{double } 0 &= 0 \\
\text{double } 1 &= \text{double(\text{suc } 0)} \\
&= \text{suc suc(\text{double } 0)} \\
&= \text{suc suc } 0 \\
\text{double } 2 &= \text{double(\text{suc } 1)} \\
&= \text{suc suc double } 1 \\
&= \text{suc suc suc } 0 \\
\text{double } 3 &= \text{double(\text{suc } 2)} \\
&= \text{suc suc double } 2 \\
&= \text{suc suc suc suc suc } 0 \\
&= 6
\end{align*}
\]
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 \( n \), \( \text{suc } n = n + 1 \).

Proof

Remember that \( 1 = \text{suc } 0 \). So:

\[
n + 1 = n + \text{suc } 0 \quad \text{by the definition of } 1
\]
\[
= \text{suc}(n + 0) \quad \text{by the recursive step of the definition of } +
\]
\[
= \text{suc } n \quad \text{by the base case of the definition of } +
\]

So from now on, we can go ahead and use either the notation \( \text{suc } n \) 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:

\[
\begin{align*}
    k + 0 &= k \\
    k + (n + 1) &= (k + n) + 1
\end{align*}
\]

2.2.3 Exercise

Use the definition of addition to explicitly show the following:

(a) \(1 + 1 = 2\).

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

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

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 \(\text{suc } n\) has the property as well. That is, for an arbitrary number \(n\), we’ll suppose that \(0 + n = n\), and try to show that \(0 + \text{suc } n = \text{suc } n\). We can show this using the recursive step of the definition of addition.

\[
0 + \text{suc } n = \text{suc}(0 + n) = \text{suc } n
\]

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

\(\square\)
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 \(1 + \text{suc } n = (\text{suc } n) + 1\).

\[
1 + \text{suc } n = \text{suc}(1 + n) \quad \text{by the definition of addition}
\]
\[
= \text{suc}(n + 1) \quad \text{by the inductive hypothesis}
\]
\[
= \text{suc suc } n
\]

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 \(\text{suc suc } n = (\text{suc } n) + 1\). That finishes the proof.

\[\square\]

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 \((k + m) + \text{suc } n = k + (m + \text{suc } n)\).

\[
(k + m) + \text{suc } n = \text{suc}((k + m) + n) \quad \text{definition of +}
\]
\[
= \text{suc}(k + (m + n)) \quad \text{inductive hypothesis}
\]
\[
= k + \text{suc}(m + n) \quad \text{definition of +}
\]
\[
= k + (m + \text{suc } n) \quad \text{definition of +}
\]

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. The conceptually 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 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:

\[
\begin{align*}
m \cdot 0 &= 0 \\
m \cdot \text{suc } n &= m \cdot n + m
\end{align*}
\]

For example, let’s work out \( 3 \cdot 2 \).

\[
\begin{align*}
3 \cdot 0 &= 0 \\
3 \cdot 1 &= 3 \cdot \text{suc } 0 \\
&= (3 \cdot 0) + 3 \\
&= 0 + 3 \\
&= 3 \\
3 \cdot 2 &= 3 \cdot \text{suc } 1 \\
&= (3 \cdot 1) + 3 \\
&= 3 + 3
\end{align*}
\]

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 \(1 \cdot \text{suc } n = \text{suc } n\). In fact, by the definition of multiplication,

\[
1 \cdot \text{suc } n = 1 \cdot n + 1 \quad \text{by the definition of } \cdot \\
= n + 1 \quad \text{by the inductive hypothesis} \\
= \text{suc } n \quad \text{by a fact we proved earlier}
\]

\[\square\]

2.2.10 Exercise
Show that \(\text{double } n = 2 \cdot n\), 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: that is, you have some other set \(A\), and you want to come up with an example of a 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 \(\text{double}\) 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:

\[f0 = \text{______________}\]

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:

\[f(n + 1) = \text{______________}\]

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 “\(f n\)”. When you
describe the value of $f$ for $n + 1$, you get to assume that you already know $f(n)$, 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 the next section 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.

2.3 The Recursion Theorem*

We have given an intuitive justification for recursive definition, in terms of our intuitive understanding of the inductive structure of numbers. In this section we’ll back up this intuition by providing a more precise proof 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.

As an example, recall the recursive definition we gave for the doubling function.

\[
\begin{align*}
\text{double} 0 &= 0 \\
\text{double}(\text{suc } n) &= \text{suc suc}(\text{double } n) \quad \text{for each number } n
\end{align*}
\]

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:

\[
\begin{align*}
\text{double} 0 &= z \\
\text{double}(\text{suc } n) &= s(\text{double } n) \quad \text{for each number } n
\end{align*}
\]

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 $s : A \rightarrow A$ 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.

2.3.1 The Recursion Theorem

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

\[
\begin{align*}
f(0) &= z \\
f(s(n)) &= s(f(n)) \quad \text{for each number } n
\end{align*}
\]

Call these the Recursive Properties.

In the rest of this section we’ll prove the Recursion Theorem. We’ll assume we are given some set \( A \), some element \( z \in A \), and some function \( s : A \rightarrow A \), and we will show that there is exactly one function with the Recursive Properties for \( z \) and \( s \).

The basic idea of the proof is that we can build up a total function \( f : \mathbb{N} \rightarrow A \) out of little pieces which are partial functions from \( \mathbb{N} \) to \( A \). We can use induction to show that for each number \( n \), there is some partial function that has the Recursive Properties for the numbers where it is defined, and which is defined as far as \( n \). This will give us infinitely many functions defined on bigger and bigger domains. Then we can join up all these functions into one function which is defined for every number, and we can show that this function has the Recursive Properties without restriction.

We will be building up our recursively defined function \( f \) out of pieces like these partial functions:

\[
[0 \mapsto z] \\
[0 \mapsto z, 1 \mapsto sz] \\
[0 \mapsto z, 1 \mapsto sz, 2 \mapsto s(sz)]
\]

And so on. Our first job is to precisely say what these “special” partial functions have in common: the idea is that they satisfy the Recursive Properties when they are defined.

2.3.2 Definition

If \( g \) is a partial function from \( \mathbb{N} \) to \( A \), then say \( g \) is special iff

(a) \( g0 = z \), and
2.3. **THE RECURSION THEOREM**

(b) For any number \(n\), if \(\text{suc } n\) is in the domain of \(g\), then \(n\) is also in the domain of \(g\), and \(g(\text{suc } n) = s(gn)\).

Our goal is to combine all the special partial functions into one big function. The idea is that the value of this big function at \(n\) should be whatever value *any* one of the special functions assigns to \(n\). We will need to check that every number gets one and only one value this way.

2.3.3 **Definition**

Say a number \(n \in \mathbb{N}\) *selects* a value \(a \in A\) iff there is some special function \(g\) such that \(n\) is in the domain of \(g\) and \(gn = a\).

We can start by checking the “only one” part: special functions all agree with each other, for the numbers where they are defined.

2.3.4 **Exercise**

For every number \(n\), if \(g\) and \(g'\) are both special functions which have \(n\) in their domains, then \(gn = g'n\). Thus for each number \(n\), there is *at most one* \(a \in A\) such that \(n\) selects \(a\).

*Hint.* Use induction.

Note in particular that if a *total* function \(f : \mathbb{N} \to A\) has the Recursive Properties, then it counts as special. So this exercise also implies that there is *at most one* total function \(f : \mathbb{N} \to A\) with the Recursive Properties. (Any two such functions would have to have the same value for *every* number \(n \in \mathbb{N}\).) This proves the uniqueness part of the Recursion Theorem. We still need to show the existence part: that there is *at least one* total function with the Recursive Properties.

2.3.5 **Exercise**

Suppose \(f : \mathbb{N} \to A\) is a function such that for each \(n \in \mathbb{N}\), \(n\) selects \(fn\). Then \(f\) has the Recursive Properties.

*Hint.* You can show (i) if 0 selects \(f0\) then \(f0 = z\), and (ii) if \(n\) selects \(fn\) and \(\text{suc } n\) selects \(f(\text{suc } n)\), then \(f(\text{suc } n) = s(fn)\).

2.3.6 **Exercise**

For every number \(n \in \mathbb{N}\), there is some value \(a \in A\) such that \(n\) selects \(a\). Thus there is a function \(f : \mathbb{N} \to A\) such that, for every number \(n\), \(n\) selects \(fn\).
CHAPTER 2. THE INFINITE

Hint. Induction.

For the base case, consider the function \([0 \mapsto z]\).

For the inductive step, suppose that \(n\) is in the domain of \(g\). Then we can define another function \(g'\) such that \(\text{suc } n\) is in the domain of \(g'\), like this:

\[
g'k = \begin{cases} 
gk & \text{if } k \text{ is in the domain of } g \text{ and } k \neq \text{suc } n \\
\text{s}(gn) & \text{if } k = \text{suc } n 
\end{cases}
\]

Check that if \(g\) is special, then so is \(g'\).

(If \(\text{suc } n\) is already in the domain of \(g\), then since \(g\) is special it follows that \(g'\) is the very same function as \(g\). So the important case here is when \(n\) happens to be the last element of the domain of \(g\)—\(n\) is in the domain of \(g\), but \(\text{suc } n\) is not. In this case, \(g'\) extends \(g\) by adding one more value to it.)

This function \(g'\) is called a variant of \(g\). Later on (in Section 3.6) we will use a special notation for variant functions; in this notation, the function \(g'\) is labeled

\[g[\text{suc } n \mapsto \text{s}(gn)]\]

(You can read this as saying “\(g\) modified so \(\text{suc } n\) goes to \(\text{s}(gn)\”).

2.3.7 Exercise
Put these facts together to finish the proof of the Recursion Theorem.

2.4 Sequences

These notes consist (mainly) of sentences. Each sentence consists (mainly) of words, and each word consists (mainly) of letters. But a sentence isn’t just a set of words, and a word isn’t just a set of letters. In each case, the order matters. “Dog bites man” and “man bites dog” are different sentences involving the very same set of words \{“bites”, “dog”, “man”\}. A sentence is better represented as an ordered sequence of words than as a set.

(But is a sentence really just a sequence of words? Perhaps not. Sentences have syntactic structure—but the very same sequence of words can have different syntactic structures. “Everyone loves someone” is one sequence of words that might encode two different sentences, with different meanings. We’ll return to syntax in Chapter 4. For now, we’ll just be looking at “flat” unstructured sequences.)
2.4. SEQUENCES

When we express ideas, we almost always do it by stringing together symbols in some order. So the theory of finite sequences of symbols is centrally important for studying language, philosophy, and logic.

One reason sequences are so useful is because they 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.

Consider a sequence of letters

\[(A, B, C, B, A)\]

Like a set, this sequence has elements. But unlike a set, the elements come in a certain order, and they can repeat. If we call this sequence \(s\), we use the notation \(s_0, s_1, s_2, s_3, s_4\) to pick out its elements in order. In this case, \(s_0\) is \(A\), \(s_1\) is \(B\), \(s_2\) is \(C\), \(s_3\) is \(B\) again, and \(s_4\) is \(A\) again. (We’ll usually start counting elements from zero rather than one, in order to line up the elements of sequences with the natural numbers, which start from zero.)

We’ll use the notation \(A^*\) for the set of all finite sequences of elements of a set \(A\).

Let’s describe finite sequences more precisely, along the same lines as our precise description of the finite numbers. Finite sequences can be built up by repeatedly applying some basic steps. In this case, our natural starting point is the very simplest finite sequence—the empty sequence, which is a sequence of length zero. We’ll use the notation \(()\) for the empty sequence. Starting from \(()\), rather than one-element sequences, is convenient in some of the same ways that it’s convenient to include zero as a finite number, rather than starting from one.

With numbers, each number has a unique next number, its successor. But given a finite sequence, there isn’t just one sequence that comes next. Instead of just adding one, we can make a sequence longer by adding any element \(a \in A\). So instead of a successor function, we have a function which takes an element \(a \in A\), and a length \(n\) sequence \(s\), and gives us a length \(n+1\) sequence that sticks \(a\) onto the beginning of \(s\). We’ll use the notation \((a : s)\) for this operation. (Computer scientists standardly call this function “cons”, which is short for “construct”.) If the elements of \(s\) are \(s_0, s_1, \ldots, s_{n-1}\), then for any \(a \in A\),

\[
(a : s) = (a, s_0, \ldots, s_{n-1})
\]

We can build up any finite sequence by starting from the empty sequence, and adding symbols one by one. For example, the sequence \((a, b, c)\) can be produced by starting with the empty sequence \(()\), then sticking \(c\) in front of it, then sticking
b in front of that, and finally sticking a in front of that. So we can understand the notation \((a, b, c)\) as an alternative notation:

\[
(a, b, c) = a : b : c : () \\
= (a : (b : (c : ())))
\]

(If we leave out the parentheses, they are understood to be added in this way, “associating to the right,” because this is what makes sense.)

Furthermore, this is the only way to produce this sequence \((a, b, c)\) by adding elements 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 sequence. In general, every finite sequence can be reached in exactly one way by starting with the empty sequence and adding symbols to the front one by one.

For each symbol \(a \in A\) and sequence \(s \in A^*\), there is an element \((a : s) \in A^*\). This means that the \((:)\) operation is a function from the set of ordered pairs \(A \times A^*\) to \(A^*\).

We can summarize this fact using “formation rule” notation, similar to what we did for numbers. There are two ways of building up finite sequences of elements of a set \(A\), which can be described with the following rules:

\[
\begin{align*}
() & \text{ is a sequence in } A^* \\
\text{a is an element of } A & \implies (a : s) \text{ is a sequence in } A^* \\
\text{s is a sequence in } A^* &
\end{align*}
\]

Every finite sequence in \(A^*\) can be reached in exactly one way using these two rules.

This means that, just like with numbers, we can do proofs by induction for finite sequences. If we want to prove that every finite sequence has a certain property, it’s enough to show two things: (a) The empty sequence 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 finite sequences 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 finite sequences, too. Every finite sequence can be reached in exactly one way, by starting with the empty sequence and repeatedly appending new elements. So we can define an “output” of a function \(f\) for every finite sequence in \(A^*\) in two steps.

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

### 2.4.1 Definition
Let’s recursively define the length of a finite sequence. This is a function \( \text{length} : A^* \rightarrow \mathbb{N} \) that takes each finite sequence in \( A^* \) to a number. The definition involves two steps. For the base case, we define the length of the empty sequence:

\[
\text{length}(\varepsilon) = 0
\]

For the recursive step, we suppose that we already know the length of \( s \), and we use this to define the length of the sequence that results from appending one symbol to the end of \( s \). 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 \( s \).

\[
\text{length}(a : s) = \text{suc}(\text{length}(s))
\]

### 2.4.2 Definition
For any set \( A \), the set of all length-\( n \) sequences of elements of \( A \) is called \( A^n \). That is,

\[
A^n = \{ s \in A^* \mid \text{length}(s) = n \}
\]

Here’s another example. The \((:)\) function lets us add one symbol to a sequence. But another thing we sometimes want to do is add a whole sequence of symbols to a sequence. That is, sometimes we’ll want to stick sequences together, end to end. If \( s \) and \( t \) are both sequences in \( A^* \), 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.

### 2.4.3 Definition
For any sequence \( t \in A^* \), we define the function that takes a sequence \( s \in A^* \) to \( s \oplus t \) recursively, as follows.

For the base case, we say how to add the empty sequence to the beginning of \( t \). This is easy:

\[
() \oplus t = t
\]

For the recursive step, we suppose that we already know how to add \( s \) to the beginning of \( t \), and then use this to define the result for the longer sequence \( (a : s) \). The
idea is that we can do this by first adding all the elements of $s$ to $t$, and then finally adding $a$ as well.

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

### 2.4.4 Example

Show explicitly using the definition:

$$(a, b) \oplus (c, b, a) = (a, b, c, b, a)$$

**Proof**

Remember that $(a, b)$ is shorthand for $a : b : ()$

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

$$(()) \oplus (c, b, a) = (c, b, a)$$

Using the recursive step,

$$(b : ()) \oplus (c, b, a) = b : (c, b, a)$$

$$= (b, c, b, a)$$

Using the recursive step again,

$$(a, b) \oplus (c, b, a) = (a : b : ()) \oplus (c, b, a)$$

$$= a : ((b : ()) \oplus (c, b, a)))$$

$$= a : (b, c, b, a)$$

$$= (a, b, c, b, a)$$

□

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

### 2.4.5 Example

For any finite sequences $s$ and $t$,

$$\text{length}(s \oplus t) = \text{length } t + \text{length } s \quad (2.1)$$

**Proof**

Let $t$ be any finite sequence. We’ll use induction to prove that every finite sequence $s$ has the property (2.1).
2.4. **SEQUENCES**

**Base case.** Consider the empty sequence. By definition, () ⊕ t = t. So:

\[
\begin{align*}
\text{length}(() \oplus t) &= \text{length } t & \text{by the definition of } \oplus \\
&= \text{length } t + 0 & \text{by the definition of } + \\
&= \text{length } t + \text{length } () & \text{by the definition of length}
\end{align*}
\]

**Inductive step.** Suppose that \( s \) has the property (2.1). (This assumption is the inductive hypothesis.) We want to show that, for any symbol \( a \), the sequence \( (a : s) \) also has the property (2.1).

\[
\begin{align*}
\text{length}((a : s) \oplus t) &= \text{length}(a : (s \oplus t)) & \text{by the definition of } \oplus \\
&= \text{suc(length}(s \oplus t)) & \text{by the definition of length} \\
&= \text{suc(length } t + \text{length } s) & \text{by the inductive hypothesis} \\
&= \text{length } t + \text{suc(length } s) & \text{by the definition of } + \\
&= \text{length } t + \text{length}(a : s) & \text{by the definition of length}
\end{align*}
\]

\[\square\]

2.4.6 **Definition**

The sequence \((a)\) is the length-one sequence whose only element is \( a \). To be explicit, \((a) = (a : ())\). This is called the **singleton sequence** of \( a \), or the **unit sequence** of \( a \).

2.4.7 **Exercise**

Show that, for any element \( a \in A \) and sequence \( s \in A^* \),

\[(a : s) = (a) \oplus s\]

2.4.8 **Exercise**

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

\[s \oplus t = t \oplus s\]

for all sequences \( s, t \in A^* \)? If so, give a proof by induction; otherwise, give a counterexample.

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

\[s \oplus (t \oplus u) = (s \oplus t) \oplus u\]
for all sequences \( s, t, u \in A^* \)? If so, give a proof by induction; otherwise, give a counterexample.

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

### 2.4.9 Definition

Suppose \( s \) is a finite sequence in \( A^* \). We can recursively define the set of elements of \( s \) as follows.

\[
\text{elements}(\cdot) = \emptyset \\
\text{elements}(a : s) = \{a\} \cup \text{elements}(s)
\]

This recursively defines a function

\[
\text{elements} : A^* \rightarrow \mathcal{P} A
\]

### 2.4.10 Exercise

Use the definition to show explicitly:

\[
\text{elements}(1, 2, 1) = \{1, 2\}
\]

### 2.4.11 Exercise

If \( s \) and \( t \) are finite sequences, then

\[
\text{elements}(s \oplus t) = \text{elements}(s) \cup \text{elements}(t)
\]

### 2.4.12 Exercise

Let \( A \) be any set. Prove by induction that, for any sequence \( s \), there is a finite sequence \( t \) such that

\[
\text{elements}(s) = A \cap \text{elements}(s)
\]

We can call this the **restriction** of \( s \) to \( A \).

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

### 2.4.13 Technique (proof by induction for sequences)

Suppose you have some set \( A \), and you want to show:

For every sequence \( s \) of elements of \( A \), \( s \) is nice.

Here "\( s \) is nice" is a placeholder for any statement about \( s \). You can do this in two steps.
1. Show that the empty sequence is nice.
2. Show that for any sequence $s$, and any element $a \in A$, if $s$ is nice, then the longer sequence $(a : s)$ is also nice.

Then you’re done: this is enough to show that every sequence (of elements of $A$) is nice.

::: {.Technique title="recursively defining a function on sequences" tech:recursion-sequences} Suppose you have a set $A$, and you want to come up with a function whose domain is the set of all sequences of elements of $A$. That is, for some other set $B$, you’re trying to come up with a function $f : A^* \to B$. You can do this in two steps.

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

\[
    f() = \underline{\text{________________}}
\]

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

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

\[
    \text{For any } a \in A, \ f(a : s) = \underline{\text{________________}}
\]

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 $A^*$ to $B$. :::

The set of numbers and the set of sequences are both inductive structures. In this course we’ll encounter many other inductive structures: they play a central role throughout logic. (For example, we’ll see later that the formulas of first-order logic make up an inductive structure, and so do formal proofs.) So proofs by induction and recursive definitions are two of the fundamental skills of logic.

Just like we did with numbers, we can describe the inductive structure of sequences 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. Similarly, we can more precisely state, and more carefully justify, the technique of recursively defining a function on sequences with a theorem, which is closely
analogous to the Recursion Theorem for numbers. The idea is basically the same, 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 Sequences or the Recursion Theorem for Sequences. The Axiom of Sequences is just a way of precisely spelling out the main intuitive idea:

Every finite sequence can be reached in exactly one way, by starting from the empty sequence, and appending symbols one by one.

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

2.4.14 Axiom of Sequences
Let $A$ be a set. There is a set $A^*$, an element $()$ in $A^*$, and a function $(:): A \times A^* \to A^*$, which have the following properties.

(a) **Injective Property.**

(i) The empty sequence $()$ is not in the range of the cons function. That is, there is no element $a$ in $A$ and sequence $s$ in $A^*$ such that $\text{cons}(a, s) = ()$.

(ii) The “cons” function $(:) is one-to-one. That is, suppose $a$ and $a'$ are elements of $A$ and $s$ and $s'$ are sequences in $A^*$. If $(a : s) = (a' : s')$, then $a = a'$ and $s = s'$.

(b) **Inductive Property.** Let $X$ be a set. Suppose (i) the empty sequence $()$ is in $X$, and (ii) for each $a \in A$ and finite sequence $s \in X$, $(a : s)$ is also in $X$. Then $X$ includes every sequence in $A^*$.

2.4.15 The Recursion Theorem for Sequences
Let $A$ and $B$ be sets. $A^*$ is the set of finite sequences of elements of $A$. Suppose that we have some element $e \in B$, and some function $c: A \times B \to B$. Then there is a unique function $f: A^* \to B$ with the following two Recursive Properties:

\[
\begin{align*}
f() &= e \\
f(a : s) &= c(a, fs) & \text{for each element } a \in A \text{ and sequence } s \in A^*
\end{align*}
\]
Proof Sketch
We can use the same idea we used for numbers: we’ll build up the big function $f$ from little partially-defined functions, which have the Recursive Property wherever they are defined. If $g$ is a partial function from $A^*$ to $B$, then say $g$ is special iff

(a) $g() = e$, and
(b) For any $a \in A$ and $s \in A^*$, if $(a : s)$ is in the domain of $g$, then $s$ is also in the domain of $g$, and $g(a : s) = c(a, gs)$.

Then say that $s \in A^*$ selects $b \in B$ iff there is some special function $g$ such that $gs = b$. As in the proof of the Recursion Theorem for numbers, we can show the following:

1. Any two special functions have the same value whenever they are both defined. In particular, there is at most one total function with the Recursive Properties.
2. If $s$ selects $f s$ for every sequence $s$, then $f$ has the Recursive Properties.
3. Every sequence $s$ selects some value. So there is a function $f$ such that $s$ selects $f s$ for every sequence $s$. Thus, by step 2, there is at least one total function with the Recursive Properties.

As we’ll see later, induction and recursion make sense not just for numbers and sequences, 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.

2.5 Strings

One of the main applications we’ll use sequences for is 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 just use 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 used 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.) So, our standard alphabet consists 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. A symbol is any element of the standard alphabet, and a string is any finite sequence of symbols.

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.

### 2.5.1 Notation

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

In the case of a single symbol, the notation A is unfortunately ambiguous: it can denote the symbol A, which is an element of the standard alphabet A, or it can denote the length-one string A, which is an element of A*. We rely on context to determine which one we mean. But this will rarely be an issue.

If we try to use this notation to talk about the empty string, then it’s very hard to see. (It would just look like this: .) So we’ll continue to use the notation ( ) to stand for the empty string (since this is just the empty sequence of symbols).

It will also be convenient to have an alternative notation for joining strings together: instead of using the join symbol ⊕, we can just write two strings next to each other, so st is the same as s ⊕ t. Likewise, As is the same as A ⊕ s, and ABCsDEF is the same as ABC ⊕ s ⊕ 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 xy instead of x · y for multiplication.)

In principle we can always expand this string notation explicitly, using the “cons” operation (∶), instead. For instance,

\[ ABC = A : B : C : () \]
2.5.2 Exercise
Let $s = tu$. Which of these strings are the same?

(a) $stu$
(b) $s \oplus tu$
(c) $s \oplus tu$
(d) $s \odot tu$
(e) $s tu$
(f) $s \oplus s$
(g) $s s$
(h) $t \oplus u \oplus t \oplus u$
(i) $s \odot tu$
(j) $s \odot \odot tu$

2.5.3 Exercise
When you log into a website, to protect your privacy your password usually isn’t shown directly on your screen: instead, a sequence 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 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
   \[
   \text{length}(\text{dots } s) = \text{length } s
   \]
(c) Use your definition to show
   \[
   \text{elements}(\text{dots } s) = \{•\} \]
(d) Use your definition to show
\[ \text{dots}(s \oplus t) = \text{dots}s \oplus \text{dots}t \]

(e) Show that
\[ \text{length } s = \text{length } t \iff \text{dots } s = \text{dots } t \]

2.6 Properties of Numbers and Sequences

At this point, we have stated the Axiom of Numbers and the Axiom of Sequences: these describe the fundamental structure of finite numbers and finite sequences, 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 sequences 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 sequences are true, but also that we can prove all of these facts from our basic axioms and definitions.

It will be helpful to refer back to these facts as we go.

2.6.1 Definition
For numbers \( m \) and \( n \), we say \( n \) is at least \( m \) (abbreviated \( m \leq n \)) iff there is some number \( k \) such that \( m + k = n \). We say \( m \) is (strictly) less than \( n \) (abbreviated \( m < n \)) iff \( m \leq n \) and \( m \neq n \).

2.6.2 Exercise
Use facts about addition and the definition of the ordering of numbers to show the following, for any numbers \( m, n, k \):

(a) \( n \leq n \). (\( \leq \) is reflexive.)
(b) If \( m \leq n \) and \( n \leq k \), then \( m \leq k \). (\( \leq \) is transitive.)
2.6. PROPERTIES OF NUMBERS AND SEQUENCES

(c) If \( m \leq n \) and \( n \leq m \), then \( m = n \). (\( \leq \) is anti-symmetric.)

(d) For any numbers \( m \) and \( n \), either \( m \leq n \) or \( n \leq m \). (\( \leq \) 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 the previous exercise shows that the natural numbers are totally ordered.

2.6.3 Exercise
There is no natural number \( n < 0 \). (Hint. Suppose \( n + k = 0 \), and consider the case where \( k = 0 \) and the case where \( k \) is a successor.)

2.6.4 Exercise
\( m \leq n \) iff \( m < \text{suc } n \), for any numbers \( m \) and \( n \).

2.6.5 Exercise
For any numbers \( m \) and \( n \), either \( m \leq n \) or \( n \leq m \).

2.6.6 Exercise
(a) If \( m \leq n \), then either \( m = n \), or \( \text{suc } m \leq n \).

(b) If \( m \leq \text{suc } n \), then either \( m \leq n \) or \( m = \text{suc } n \).

(c) If \( m < \text{suc } n \), then either \( m < n \) or \( m = n \).

2.6.7 Exercise
For any number \( n \), there is a length-\( n \) finite sequence that includes each number \( k < n \) as an element.

Hint. Give a recursive definition of a function \( f : \mathbb{N} \rightarrow \mathbb{N}^* \), then use this definition to show that for each number \( n \), the sequence \( f n \) has the properties we want, namely:

\[
\text{length}(f n) = n \\
\text{elements}(f n) = \{ k \in \mathbb{N} \mid k < n \}
\]

2.6.8 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.
2.6.9 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 $n \in X$, $m \leq n$.

2.6.10 Exercise
For any number $n$, there is a length-$n$ sequence $\bar{n}$ such that elements $n = \{k \in \mathbb{N} \mid k < n\}$.

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.

2.6.11 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. $m + \text{suc } n = \text{suc}(m + n)$
5. $n \cdot 0 = 0$
6. $m \cdot \text{suc } n = (m \cdot n) + m$
7. $n$ is not less than 0
8. $m \leq n$ iff $m < \text{suc } n$
9. $m \leq n$ or $n \leq m$

We can do some similar things for sequences.

2.6.12 Definition
For sequences $s$ and $t$ in $A^*$, we say $s$ is an initial subsequence of $t$ (abbreviated $s \leq t$) iff there is some sequence $u \in A^*$ such that $s \oplus u = t$. We say $s$ is a proper initial subsequence of $t$ (abbreviated $s < t$) iff $s \leq t$ and $s \neq t$.

2.6.13 Exercise
$s \leq t$ iff either $s$ is empty, or for some $a$, $s = (a : s')$, $t = (a : t')$, and $s \leq t'$.

2.6.14 Exercise
If $s \leq t$ then $\text{length } s \leq \text{length } t$. 
2.7. THE FINITE AND THE INFINITE

2.6.15 Exercise (Cancellation Property)
If \( s \oplus t = s \oplus t' \), then \( t = t' \).

2.6.16 Exercise
If \( s \preceq t \) and \( s' \preceq t \), then either \( s \preceq s' \) or \( s' \preceq s \).

In what follows, our most important kind of sequences will be sequences of symbols from our standard alphabet: that is, *strings*. As with numbers, it will be useful to collect together a few particularly important facts about strings to refer back to later.

2.6.17 Definition
For each symbol \( a \) in the standard alphabet, there is a length-one string \( (a) \) whose only element is \( a \). We call this \( a \)'s *singleton string* (or *unit string*).

2.6.18 The Minimal Theory of Strings
Let \( s \) and \( t \) be strings, and let \( a \) and \( b \) both be single symbols.

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 symbol \( a \) and string \( t \) such that \( s = (a) \oplus t \).

2.7 The Finite and the Infinite

We have encountered some examples of finite sets (such as \{ Silver Lake, Echo Park \}) and some examples of infinite sets (such as the set of natural numbers). In this section we’ll look more closely 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.
One way of understanding finite sets appeals to finite sequences. We have already described finite sequences explicitly (in terms of their inductive property). A finite set is like a finite sequence, except that we don’t need to pay attention to the order of elements, or how many times they are repeated. We know that \{Silver Lake, Echo Park\} is a finite set, because there is a corresponding finite sequence, namely

(Silver Lake, Echo Park)

Of course, there are also other finite sequences with the same elements as this set. For example:

(Echo Park, Silver Lake)

or

(Echo Park, Silver Lake, Echo Park, Silver Lake)

There are infinitely many other options as well. But any one of these finite sequences is enough to show us that the set is finite. We also can tell more precisely how big the set is: its elements can be enumerated in a list of length two, but not with any shorter sequence than this. That is a precise way of saying that the set has exactly two elements.

2.7.1 Definition
A set \(A\) is finite iff there is some finite sequence \(s\) such that every element of \(A\) is an element of \(s\). In other words,

There is some finite sequence \(s\) such that \(A = \text{elements } s\).

In this case we say that \(s\) (finitely) enumerates \(A\). A set is infinite iff it is not finite.

To put it another way, a set \(A\) is finite iff it is in the range of the function elements : \(A^* \rightarrow PA\).

2.7.2 Exercise
If \(A\) is finite and \(B\) is finite, then \(A \cup B\) is finite.

2.7.3 Exercise
Any finite union of finite sets is finite. In other words: suppose \(A_1, \ldots, A_n\) are each finite sets. Their union \(U = \bigcup_j A_j\) is the set of just those things which are in \(A_j\) for some \(i\). Show that \(U\) is finite.
2.7. THE FINITE AND THE INFINITE

2.7.4 Definition

If $A$ is a finite set, then the number of elements of $A$ is the smallest number $n$ such that some length-$n$ sequence enumerates $A$.

That gives us a way of understanding finiteness and infinity in terms of finite sequences. There is another way of understanding the distinction, in terms of the natural numbers, instead. Recall that we can use one-to-one functions as a way of comparing the “sizes” of sets. A set $B$ is at least as big as the set $A$ iff there is some one-to-one function from $A$ to $B$. So 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.

(Why say an infinite set has \textit{at least as many} elements as there are numbers, rather than \textit{exactly} as many? This will become clear in the next section: some infinite sets have even \textit{more} elements than there are numbers. The set $\mathbb{N}$ is the \textit{smallest} infinite set.)

There is also a third way of thinking about infinity. This way doesn’t depend on either sequences or numbers, so it is, in a way, “purer” and more abstract than the first two.

Suppose you have an ordinary hotel, which, like most ordinary hotels, has finitely many rooms. There is one person in each room. Now you rearrange people by moving them to different rooms. After the rearrangement, if nobody is sharing a room, then the hotel is still full: there aren’t any empty rooms left over. To put it another way: for any function that takes each room to a room, if the function is one-to-one, then it is onto. This is true for ordinary hotels—because ordinary hotels have only \textit{finitely many rooms}. But an \textit{infinite} hotel isn’t like this. If you have an infinite hotel, with one person in each room, then you can move people to different rooms in a way that leaves some rooms empty, without making anyone double up.

2.7.5 Definition

A set $A$ is \textbf{Dedekind-infinite} iff there is some function $f : A \to A$ which is one-to-one but not onto. Otherwise $A$ is \textbf{Dedekind-finite}.

2.7.6 Exercise (Hilbert’s Hotel)

The set $\mathbb{N}$ of all natural numbers is Dedekind-infinite.
2.7.7 Exercise

A is Dedekind-infinite iff $A$ is the same size as one of its proper subsets: that is, for some $B \subsetneq A$, $A \sim B$.

2.7.8 Exercise

Suppose that $A$ is Dedekind-infinite.

(a) If $A \sim B$, then $B$ is Dedekind-infinite.

(b) If $A \subseteq B$, then $B$ is Dedekind-infinite.

(c) If $A \leq B$, then $B$ is Dedekind-infinite.

An important fact is that all three of these ways of thinking about infinity are equivalent. For any set $A$, if any of the following statements is true, so are the other two:

1. The elements of $A$ cannot be listed in a finite sequence
2. $A$ has at least as many elements as $\mathbb{N}$
3. There is a “Hilbert’s hotel” function for $A$ (that is, $A$ is Dedekind-infinite).

In the rest of this section we will carefully prove that these three things are equivalent. (But for the purposes of the rest of this text, it is perfectly fine if you prefer to take this fact on faith, and move on to the next section.)

First, let’s consider a useful technique for working with finite sets. Finite sets can be built up one element at a time. So there is a version of induction that works for finite sets, which is very similar to the version of induction that works for finite sequences.

::: {.Technique title="Induction on Finite Sets" tech:induction-finite-sets} Let $A$ be a set. Suppose we want to show that every finite subset of $A$ is nice. We can do this in two steps.

1. **Base case.** Show that the empty set is nice.

2. **Inductive step.** Suppose that $B$ is any finite subset of $A$. Show that if $B$ is nice, then for any $a \in A$, the union $\{a\} \cup B$ is also nice.

Then we’re done. :::
2.7. THE FINITE AND THE INFINITE

Why does this work? The basic reason is that every finite set can be built up by starting with the empty set and adding elements one at a time. This means that the finite sets have their own Inductive Property.

2.7.9 The Inductive Property of Finite Sets

Let $A$ be any set, and let $X$ be any set. Suppose that

- (a) the empty set is in $X$, and
- (b) for every finite subset $B \subseteq A$, if $B$ is in $X$, then for any $a \in A$, $\{a\} \cup B$ is in $X$ as well.

Then every finite subset of $A$ is in $X$.

**Proof**

By definition, a finite set is the set of elements of some finite sequence. So in order to show that every finite set is in $X$, it’s enough to show that for every finite sequence $s$, its set of elements $\text{elements}(s)$ is in $X$. We can do this by induction on sequences.

**Base case.** $\text{elements}()$ is in $X$. That is, the empty set is in $X$. This was given as assumption (a).

**Inductive step.** Let $s$ be any sequence, and suppose $\text{elements}(s)$ is in $X$. We want to show that, for any $a \in A$, $\text{elements}(a : s)$ is in $X$. That is to say, $\{a\} \cup \text{elements}(s)$ is in $X$. This follows immediately from assumption (b).

(Notice that the finite sets *don’t* have their own Injective Property, because there isn’t just one way to build up a finite set by adding elements one at a time. You can have two different finite sets $A$ and $A'$, and two different elements $a$ and $a'$, such that $A \cup \{a\} = A' \cup \{a'\}$.)

2.7.10 Example

An upper bound of a set of numbers $A$ is a number $n$ such that every number in $A$ is at most $n$. Any finite set of numbers has an upper bound.

**Proof**

We can show this by induction on the finite subsets of $\mathbb{N}$.

**Base case.** The empty set has an upper bound. In fact, any number at all is an upper bound of the empty set. For example, $0$ is an upper bound: no element of the empty set is greater than $0$, because the empty set has no elements.
**Inductive step.** Suppose that \( A \) has an upper bound \( n \), and let \( k \) be any number. Then either \( n \geq k \), or else \( k > n \). In the first case, \( n \) is an upper bound of \( \{ k \} \cup A \), and in the second case, \( k \) is an upper bound of \( \{ k \} \cup A \). So in either case, \( \{ k \} \cup A \) has an upper bound.

Notice in particular that the set of all numbers \( \mathbb{N} \) has no upper bound: for any number \( n \), there is a number bigger than \( n \). So \( \mathbb{N} \) is infinite.

**2.7.11 Lemma**
Every finite set is Dedekind-finite.

**Proof**
We can prove this by induction on finite sets.

**Base case.** The empty set is Dedekind-finite: the only function from the empty set to itself is the empty function, and this is onto.

**Inductive step.** We will show that if \( A \) is Dedekind-finite, then \( A \cup \{ b \} \) is also Dedekind-finite. Putting that the other way around, we will show that if \( A \cup \{ b \} \) is Dedekind-infinite, then \( A \) is also Dedekind-infinite.

Suppose that we have a Hilbert’s hotel function for \( A \cup \{ b \} \). That is, some function \( f : A \cup \{ b \} \to A \cup \{ b \} \) is one-to-one, but not onto. We’ll show that we can “squash \( f \) down” to get a Hilbert’s hotel function on \( A \), as well. We assume that \( b \notin A \). (If \( b \in A \), then this is trivial, because in that case \( A \cup \{ b \} \) and \( A \) are the very same set).

The rough idea is that we can just cut \( f \) down to the smaller set \( A \), and get a function from \( A \) to \( A \) which is one-to-one but not onto. But that idea doesn’t quite work. The issue is that \( f \) might take some element \( a \in A \) outside of \( A \), to \( b \). In that case just restricting the domain of \( f \) wouldn’t give us a function to \( A \). We’ll need to find somewhere else to take \( a \). Fortunately, we also have a new room opening up, which has been vacated by \( b \). So here’s what we can do. For each \( a \in A \), we can let
\[
\begin{align*}
g_a &= \begin{cases} 
fa & \text{if } fa \neq b \\
f_b & \text{if } fa = b 
\end{cases}
\end{align*}
\]
First, we should check that this really does define a function from \( A \) to \( A \). For any \( a \in A \), if \( fa \neq b \), then we must have \( fa \in A \). Otherwise, if \( fa = b \), then since \( f \) is one-to-one, \( fb \neq b \), and so \( fb \in A \).

Next, we can check that \( g \) is one-to-one. This is left as an exercise. (Hint. Assume \( ga = ga' \). There four different cases to check, depending on whether \( fa = b \) and
whether \( f a' = b \). Finally, we can also check that \( g \) is onto. This is also left as an exercise. \( \text{(Hint. Suppose that} \ a \in A. \ \text{Since} \ f \ \text{is onto, we know that there is some} \ x \in A \cup \{b\} \ \text{such that} \ fx = a. \ \text{This time there are two different cases to consider: either} \ x \in A, \ \text{or else} \ x = b. \) \)

\[ \square \]

2.7.12 Exercise

Fill in the remaining details in the proof of Lemma 2.7.11.

2.7.13 Lemma

For any infinite set \( A, \mathbb{N} \leq A \).

\textbf{Proof}

Let \( A \) be an infinite set. We’ll show that there is a one-to-one function \( h \) from numbers to \( A \).

The idea is that we can let \( h0 \) be any element we want, and then let \( h1 \) be any element of \( A \) other than \( h0 \), and then let \( h2 \) be any element of \( A \) other than \( h0 \) or \( h1 \), and so on. Since we will only have used up finitely many elements of \( A \) at any step, we can always keep extending this function, until we have picked a unique value of \( h \) for every number. Making this idea precise is a little tricky.

Since \( A \) is infinite, for any finite sequence \( s \) of elements of \( A \), there is some \( a \) which is not an element of \( s \). Thus, by the Axiom of Choice, there is a function \( f \) that takes each sequence \( s \in A^* \) to an element of \( A \) that is not an element of \( s \).

We can use \( f \) to recursively define a function \( g : \mathbb{N} \rightarrow A^* \) from numbers to finite sequences of elements of \( A \).

\[
g0 = \emptyset \\
g(n + 1) = (f(gn) : gn)
\]

For each number \( n \), the \( g(n + 1) \) adds one new element to \( gn \).

Finally, we can define the sequence we wanted: for each number \( n \), let \( hn \) be \( f(gn) \), which is the first element of \( g(n + 1) \). We just need to check that \( h \) is one-to-one.

We can easily show by induction that, for any number \( k \), \( hn \) is an element of the sequence \( g(n + 1 + k) \). \( \text{(Base case.} \ hn = f(gn) \ \text{is an element of} \ g(n + 1). \ \text{Inductive step.} \ \text{If} \ hn \ \text{is an element of} \ g(n + 1 + k), \ \text{then it is still an element of} \ g(n + 1 + k + 1). \) So if \( n < m \), \( hn \) is an element of \( gm \). Since \( hm = f(gm) \) was chosen \textit{not} to be an element of \( gm, hn \) and \( hm \) must be distinct. Thus \( h \) is one-to-one. \( \square \)
Now we just have to put all these facts together.

**2.7.14 Exercise**
Let $A$ be any set. The following are equivalent:

(a) $A$ is infinite.
(b) $A \geq \mathbb{N}$.
(c) $A$ is Dedekind-infinite.

This means that we can go back and forth between these three notions of infinity, depending on which one is more useful for any particular purpose.

**2.7.15 Exercise**
For any non-empty set $A$, the set of finite sequences $A^*$ is infinite.

**2.7.16 Exercise**
If $A$ is finite, then for any number $n$ there are finitely many length-$n$ sequences: that is, the set $A^n$ is finite.

**2.8 Induction and Infinity**

We’ve given three different characterizations of infinity: in terms of finite sequences, in terms of natural numbers, and in terms of one-to-one functions (Dedekind-infinity). The first two ways correspond to “axioms” we’ve assumed: the Axiom of Numbers (there is a set of natural numbers), and the Axiom of Sequences (for any set $A$, there is a set of all finite sequences of elements of $A$). There is also a natural axiom corresponding to the third view of infinity:

**2.8.1 Axiom of Infinity**
There is a Dedekind-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 others as consequences.

**2.8.2 Exercise**
Explain why the Axiom of Numbers implies the Axiom of Infinity, and why the
2.8. **INDUCTION AND INFINITY**

Axiom of Sequences implies the Axiom of Infinity.

### 2.8.3 Theorem

The Axiom of Numbers, the Axiom of Sequences, and the Axiom of Infinity are equivalent.

**Proof**

Given Exercise 2.8.2, 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 Sequences.

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 $fa \in X$. 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 $\text{suc}$ 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 Sequences. We can do this by finding a way to “encode” finite sequences with numbers. There are many different ways to do this: here is one. Consider the sequence

\[(A, B, C, B, A)\]

We can completely describe this sequence by saying “Element 0 is \(A\), element 1 is \(B\), element 2 is \(C\), element 3 is \(B\), and element 4 is \(A\)” (We’re counting from zero, because zero is the first natural number, and this is convenient for some purposes. But it doesn’t matter very much.) So the sequence is completely described by specifying a certain *function* from the first five numbers \(\{0, 1, 2, 3, 4\}\) to letters, which says which letter appears at each position in the sequence. In other words, we can represent the sequence with this function:

\[
\begin{align*}
0 & \mapsto A, \\
1 & \mapsto B, \\
2 & \mapsto C, \\
3 & \mapsto B, \\
4 & \mapsto A
\end{align*}
\]

Call this function \(a\). So element 0 of the sequence is \(a0\), element 1 is \(a1\), and so on. And it’s clear that this will work for every sequence.

To prove the Axiom of Sequences from the Axiom of Numbers, we need to show that for any set \(A\), there exists some set \(A^*\), an element \(() \in A^*\), and for each \(a \in A\) and \(s \in A^*\) we have some element \((a : s) \in A^*\), where these have the Injective Property and Inductive Property for sequences. So, using the idea we just described, we can let \(A^*\) be a certain set of functions. Let \(()\) be the empty function from \(\emptyset\) to \(A\). For any partial function \(s\) from \(\mathbb{N}\) to \(A\), and for any \(a \in A\), let \((a : s)\) be the function

\[
\begin{align*}
0 & \mapsto a, \\
\quad n + 1 & \mapsto \text{sn} \quad \text{if } n \text{ is in the domain of } s
\end{align*}
\]

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 \(A^*\) 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 \(A^*\) has the Inductive Property for sequences.

The last thing to check is that \(A^*\) also has the Injective Property. First, it’s clear that for any \(a\) and \(s\), \((a : s)\) 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.
2.9. **THE COUNTABLE AND THE UNCOUNTABLE**

(The same philosophical question arises for sequences: is this really what a finite sequence is—a certain function? 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 Sequences for technical purposes—because again, all we will really care about is that for each set $A$ there is some $A^*$, $(),$ and “cons” operation ($:$) with the right structural features. It won’t really matter what that set’s elements really are, as far as our formal 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.)

### 2.9 The Countable and the Uncountable

Infinite sets are not all alike. Just as finite sets come in many different sizes, there are also infinite sets which have different sizes, in the sense we have been talking about since Section 1.6: there is no way of putting their elements in one-to-one correspondence. Indeed, 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 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.

#### 2.9.1 Definition

A set $A$ is **countable** (also called *enumerable* or *denumerable*) iff $A \leq \mathbb{N}$. Remember that these are all equivalent ways of saying this (see Exercise 1.5.16):

1. There is a one-to-one function from $A$ to $\mathbb{N}$.
2. There is a one-to-one correspondence between $A$ and some set of numbers.
3. There is an onto function from $\mathbb{N}$ to $A$, or else $A$ is empty.
4. There is an onto partial function from $\mathbb{N}$ to $A$.

Here’s another way of thinking about this. We can count the elements of a set by listing them, and thus assigning numbers to each of them. If the set is finite, we can do this with just finitely many numbers: for a finite set $X$, there is a function from the first $n$ numbers onto $X$. But we can similarly count up the elements of an *infinite* set by going on through *all* of the counting numbers, without any finite limit. A *countable* set is one that can be counted up using either some or all of the
 CHAPTER 2. THE INFINITE

counting numbers. Many important sets can be “infinitely counted” this way. But
as we’ll see, other infinite sets are too big even for that.

We can represent infinite sequences using functions, in a similar way to how we
represented finite sequences with functions in Section 2.8. For example, consider
this infinite sequence:

\[(A, B, C, A, B, C, \ldots)\]

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

\[\{0 \mapsto A, \ 1 \mapsto B, \ 2 \mapsto C, \ \ldots\}\]

2.9.2 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 \(\omega\)-sequence.)

So another way of putting Definition 2.9.1 is that a countable set is one whose
elements can be listed in an infinite sequence (or else the empty set).

2.9.3 Proposition

A set \(A\) is countable iff there is some finite or infinite sequence that includes every
element of \(A\).

Proof

Suppose \(A\) is countable. Then either there is an onto function from \(\mathbb{N}\) to \(A\), or else \(A\)
is empty. The first case is just what it means for there to be an infinite sequence that
includes every element of \(A\). In the second case, clearly there is a finite sequence
that includes every element of \(A\)—the empty sequence.

For the other direction, again if there is an infinite sequence that includes every
element of \(A\), then there is an onto function from \(\mathbb{N}\) to \(A\), so \(A\) is countable. On the
other hand, if there is a finite sequence that includes every element of \(A\), then there
is a partial function from \(\mathbb{N}\) to \(A\) which is onto, a function which is defined for just
the first \(n\) numbers. So again \(A\) is countable.

2.9.4 Proposition

Every finite set is countable.
2.9. THE COUNTABLE AND THE UNGOUNTABLE

Proof
If $A$ is finite, then (by definition) there is a finite sequence whose elements include every element of $A$. So $A$ is countable. □

2.9.5 Example
The even numbers are countable. We can show this using the enumeration
e = (0, 2, 4, 6, 8, …)
This infinite sequence is represented by the function
\[ [0 \mapsto 0, \ 1 \mapsto 2, \ 2 \mapsto 4, \ 3 \mapsto 6, \ \ldots ] \]
This is just the doubling function. It is an onto function from $\mathbb{N}$ to the even numbers.

2.9.6 Exercise
Any subset of a countable set is countable.

2.9.7 Theorem
$A$ is countably infinite iff $A \sim \mathbb{N}$.

Proof
$A$ is countable iff $A \leq \mathbb{N}$. $A$ is infinite iff $\mathbb{N} \leq A$. So $A$ is countably infinite iff $A \leq \mathbb{N}$ and $\mathbb{N} \leq A$. By the Schröder-Bernstein Theorem (Theorem 1.5.21) this holds iff $A \sim \mathbb{N}$. □

Notice that we have learned something a bit counterintuitive here. The set of even numbers $E$ is a proper subset of the set of all natural numbers $\mathbb{N}$, and in fact it 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! Since $E$ is countably infinite, by Theorem 2.9.7 we know $E \sim \mathbb{N}$. That is, there are just as many even numbers as natural numbers.

This counterintuitive result is another manifestation of the basic way that infinite sets are counterintuitive: “Hilbert’s hotel”. As we discussed in Section 2.7, one of the basic properties that distinguishes infinite sets from finite sets is that if $A$ is an infinite set, then $A$ is the same size as some of its proper subsets. You can throw out some elements of $A$, and still have just as many as you started with. In fact, as we see here, it’s not just that you can throw out one element or a few of them: you can throw out infinitely many elements, and still have just as many as you started with.
CHAPTER 2. THE INFINITE

To put it another way, not only can a full Hilbert’s hotel accommodate a few extra guests by moving people around, it can accommodate infinitely many extra guests.

More generally, you should be careful about your intuitions about which sets are “bigger” than others. In general, 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 $A < B$. Infinite sets don’t work that way! Just because there is a one-to-one function that isn’t onto, this doesn’t rule out the existence of a different one-to-one function which is onto. To put it another way, finding a way of mapping $A$ into $B$ with some stuff left over does tell you that $A \leq B$—that $B$ has at least as many elements as $A$—but it could still turn out that $B \leq A$ as well, in which case they would be the same size after all.

Here is another example where this kind of intuition often leads astray. It’s obvious that there are at least as many ordered pairs of numbers as there are numbers. For example, if we map each number $n$ to the ordered pair $(n, 0)$, this is clearly a one-to-one function. This obvious mapping leaves out infinitely many pairs—so it’s tempting to conclude that there are more pairs of numbers than there are numbers (that is, that $\mathbb{N} < \mathbb{N} \times \mathbb{N}$). But this is also wrong! In fact, there is a less obvious way of mapping numbers to pairs of numbers that catches all of them.

2.9.8 Theorem
The set of ordered pairs of numbers is countable. That is, $\mathbb{N} \times \mathbb{N} \sim \mathbb{N}$.

Proof
Our goal is to come up with a way of listing all of the pairs of numbers $(m, n) \in \mathbb{N} \times \mathbb{N}$ in one infinite list which doesn’t leave out any pairs. We can visualize pairs of numbers as an infinite grid. What we want to do is find some infinite route that eventually reaches every pair in this grid. We could try going through the grid row by row, like in Fig. 2.1.

But this is no good: we’ll never reach the second row at this rate! Going down the columns has the same problem. So we need to be a bit more devious. There’s more than one way to do it, but here’s a trick that works (Fig. 2.2).

First we’re visiting all the pairs that add up to 0, then all the pairs that add up to 1, then all the pairs that add up to 2, and so on. (The set of pairs that add up to 2 corresponds to the second diagonal sequence in the diagram, for example.) The trick is that for any particular number $k$, there are only finitely many different pairs $(m, n)$ such that $m + n = k$. So we have divided up the set $\mathbb{N} \times \mathbb{N}$ of all pairs into a sequence of finite sets.
2.9. THE COUNTABLE AND THE UNE Countable

\[(0, 0) \rightarrow (1, 0) \rightarrow (2, 0) \rightarrow (3, 0) \rightarrow \cdots\]

\[(0, 1) \quad (1, 1) \quad (2, 1)\]

\[(0, 2) \quad (1, 2) \quad \cdots\]

\[(0, 3)\]

\[\vdots\]

Figure 2.1: A bad strategy for enumerating all the pairs

\[(0, 0) \quad (1, 0) \quad (2, 0) \quad (3, 0) \quad \cdots\]

\[(0, 1) \quad (1, 1) \quad (2, 1)\]

\[(0, 2) \quad (1, 2) \quad \cdots\]

\[(0, 3)\]

\[\vdots\]

Figure 2.2: A better strategy for enumerating all the pairs
To be explicit, for each number \( k \), we can define the \( k \)th diagonal to be the set

\[
A_k = \{(m, n) \in \mathbb{N} \times \mathbb{N} \mid m + n = k\}
\]

For each \( k \), this set \( A_k \) is a finite set. Furthermore, every pair of numbers is in one of these sets \( A_k \). That is,

\[
\mathbb{N} \times \mathbb{N} = \bigcup_k A_k
\]

We can go through each of these finite sets, one by one, and eventually we’ll reach them all, and thus we’ll list every element of every one of these finite sets. So to finish the proof, it’s enough to prove the following Lemma.

\[\square\]

2.9.9 Lemma

Suppose that \( A \) is a countable union of finite sets. That is, for each number \( i \in \mathbb{N} \), \( A_i \) is a finite set, and \( A = \bigcup_i A_i \). Then \( A \) is countable.

Proof

The idea is that we can list out each finite set one after another, and eventually we’ll reach each element of each set \( A_i \). That probably is already enough to make the idea intuitively clear. But for completeness, let’s spell it out a bit more precisely, by explicitly defining a function from \( A \) to \( \mathbb{N} \) and showing that it is one-to-one.

For each number \( i \in \mathbb{N} \), let \( n_i \) be the number of elements of the \( i \)th set \( A_i \). We know there is a one-to-one function \( f_i \) from \( A_i \) to numbers less than \( n_i \).

We can also define a function which counts up how much room we’ll need for the sets that come before the \( i \)th one. We’ll use a recursive definition:

\[
s(0) = 0
\]

\[
s(i + 1) = s(i) + n_i
\]

Then the idea is that, to assign unique numbers to the elements of the \( i \)th set \( A_i \), first we’ll skip up to the number \( s(i) \), to make sure we don’t clash with any earlier sets.

For each \( a \) in the union \( A \), there is some smallest number \( i \) such that \( a \in A_i \). Then we can define

\[
g(a) = s(i) + f_i(a)
\]

Now we just need to show that \( g \) is one-to-one.

Suppose that \( a, a' \in A \) and \( g(a) = g(a') \). Let \( i \) and \( j \) be the first numbers such that \( a \in A_i \) and \( a' \in A_j \). Then there are three cases to consider.
First, suppose $i = j$. So
\[ s(i) + f_i(a) = g(a) = g(a') = s(i) + f_i(a') \]
By cancelling the left-hand term on both sides,
\[ f_i(a) = f_i(a') \]
Since $f_i$ is one-to-one, $a = a'$.

Second, suppose $i < j$. So $i + 1 \leq j$, which also means that $s(i + 1) \leq s(j)$ (since $s$ is an increasing function). In this case,
\[
\begin{align*}
g(a) &= s(i) + f_i(a) \quad \text{by the definition of } g \\
&< s(i) + n_i \quad \text{because } f_i(a) < n_i \\
&= s(i + 1) \quad \text{by the definition of } s \\
&\leq s(j) \\
&\leq s(j) + f_j(a') \\
&= g(a') \quad \text{by the definition of } g
\end{align*}
\]
So $g(a) \neq g(a')$.

In the third case, where $i > j$, we can reason similarly to the second case. So in any case, if $g(a) = g(a')$ then $a = a'$, which means $g$ is one-to-one, and so $A$ is countable.

\[ \square \]

Theorem 2.9.8 is also very striking. Not only can we fit $\mathbb{N}$ within $\mathbb{N} \times \mathbb{N}$ with a bit of room left over, but in fact we can fit infinitely many copies of $\mathbb{N}$ in $\mathbb{N} \times \mathbb{N}$. One copy of $\mathbb{N}$ is the set of pairs whose first coordinate is zero:
\[ B_0 = \{(0,n) \mid n \in \mathbb{N}\} \]
Another copy is the set of pairs whose first coordinate is one,
\[ B_1 = \{(1,n) \mid n \in \mathbb{N}\} \]
And so on, giving us a different complete copy of the natural numbers for every single number.
\[ B_i = \{(i,n) \mid n \in \mathbb{N}\} \]
So we have packed infinitely many copies of the natural numbers into a set which is the same size as the set of natural numbers. To put it another way, 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.

We can generalize this fact. As we just saw, we can slice up \( \mathbb{N} \times \mathbb{N} \) into countably many pieces, each of which looks like \( \mathbb{N} \). That is,

\[
\mathbb{N} \times \mathbb{N} = B_0 \cup B_1 \cup \ldots = \bigcup_i B_i
\]

where each set \( B_i \) is countably infinite. In other words, \( \mathbb{N} \times \mathbb{N} \) is a countably infinite union of countably infinite sets. We just proved that \( \mathbb{N} \times \mathbb{N} \) is countable. Now we can use what we know about the particular example of \( \mathbb{N} \times \mathbb{N} \) to show that any set that can be similarly “sliced up” is countable.

2.9.10 Exercise
A countably infinite union of countable sets is countable. In other words, suppose that we have an infinite sequence \( A_0, A_1, A_2, \ldots \) of countable sets. Then their union \( \bigcup_i A_i \) is also countable. (Remember, this union is the set of all things \( a \) such that \( a \in A_i \) for some number \( i \in \mathbb{N} \).)

*Hint.* We know that there is an onto function \( f_0 : \mathbb{N} \to A_0 \), another onto function \( f_1 : \mathbb{N} \to A_1 \), another onto function \( f_2 : \mathbb{N} \to A_2 \), and so on. We can use this infinite sequence of functions to define an onto function from the set of pairs of numbers \( \mathbb{N} \times \mathbb{N} \) to the union \( A \).

Notice that Exercise 2.9.10 is a generalization of Lemma 2.9.9. It applies to a union of countable sets instead of just a union of finite sets.

2.9.11 Technique (Proving a Set is Countable)
The previous exercise provides one of our main tricks for showing that a set is countable. Say we want to show that \( A \) is countable. The strategy is to find a way of building \( A \) up from pieces. If we only use countably many pieces, and we can show that each piece along the way is countable, then \( A \) is countable.

2.9.12 Example
Let \( n \) be a number, and let \( A \) be a countable set. The set \( A^n \) of all length \( n \) sequences of elements of \( A \) is countable.

*Proof*
We’ll prove this by induction. For the base case, clearly the set of length-zero sequences is countable, because there is only one empty sequence. For the inductive
2.9. THE COUNTABLE AND THE UNCOUNTABLE

step, suppose that $A^n$ is countable. We’ll prove that the set of length-$(n + 1)$ sequences is also countable.

Each length-$(n + 1)$ sequence is just a length-$n$ sequence with an extra element added to the end. Since $A$ is countable, we can list its elements: $a_0, a_1, a_2, \ldots$. Then let $B_i$ be the set of length-$(n + 1)$ sequences whose last element is $a_i$. Each of these sets is countable: consider the function that takes each sequence $s \in A^n$ to $(a_i : s)$. This is an onto function from $A^n$ to $B_i$, so $B_i \leq A^n$, and $A^n$ is countable by the inductive hypothesis. Furthermore, every sequence in $A^{n+1}$ is in one of these sets $B_i$, because each of these sequences begins with $a_i$ for some number $i$. This tells us:

$$A^{n+1} = \bigcup_i B_i$$

That is, $A^{n+1}$ is a countable union of countable sets. So $A^{n+1}$ is also countable. □

2.9.13 Exercise

If $A$ is a non-empty countable set, then the set $A^n$ of all finite sequences of elements of $A$ is countably infinite.

2.9.14 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.

2.9.15 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 $A$ is uncountable.

2.9.16 Exercise

For each of the following sets, say whether it is countable or uncountable. Explain briefly.

(a) The set of all strings.

(b) The set of all finite sequences of strings.
(c) The set of all sets of strings.

2.9.17 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 1.6.)

2.9.18 Exercise
Suppose that \( L \) is some set of strings. We’ll call \( L \) a language, and we’ll call the elements of \( L \) 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 \( ds \) the set described by \( s \). Show that there are infinitely many sets of numbers which are not described by any description in \( L \).

2.9.19 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, \( D = \{0, 1, \ldots, 9\} \). 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 \( s \oplus \bar{9} \) be the result of adding an infinite sequence of 9’s to the end of \( s \). Then

\[
X = \{ t \in D^N \mid t = s \oplus \bar{9} \text{ for some } s \in D^* \}
\]

In other words, \( X \) is the range of the function \( [s \mapsto s \oplus \bar{9}] \) from \( D^* \) to \( D^N \). This is the key fact that you can take for granted about the decimal representation of
2.10. MORE COUNTING* 93

real numbers: there is a one-to-one correspondence between $D^N - X$ 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 irrational numbers. (In fact, you will prove that there are uncountably many of them.)

2.10 More Counting*

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 1.6. 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.

2.11 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 sequences to prove that every finite sequence (or every string) has a certain property. (Technique 2.4.13)
• You can use recursion on sequences to come up with a function whose domain is $A^*$ the set of all finite sequences of elements of $A$. In particular, you can do this to come up with a function whose domain is the set of all strings $\mathcal{S}$. (Technique 2.4.13)

• 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 $A$ is uncountable, and $B$ is countable. (Technique 2.9.17)

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 Sequences** intuitively says that every finite sequence can be reached in *at most one way* by starting from the empty string and adding elements one at a time.

• The **Inductive Property of Sequences** intuitively says that every finite sequence can be reached in *at least one way* by starting from the empty string and adding elements one at a time.

• The **Recursion Theorem for Sequences** intuitively says that recursive definitions on sequences work. If you give a value for the empty sequence, and a rule for going from the value for a string $s$ to a value for any string $(a) \oplus s$ which is one element longer, then this pins down exactly one function which is defined for every sequence.
2.11. REVIEW

- The Injective Property and Inductive Property for sequences tells us everything we need to know about the structure of sequences: we can use these basic properties to prove facts about operations like joining sequences 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 2.5.1)

- The set $\mathbb{N}$ of all numbers is infinite. So is the set $\mathcal{S}$ 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
  - $\mathcal{S}$, 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
  - $P\mathcal{S}$, 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:
  - $P A$, 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
Chapter 3

Structures

3.1 Signatures and Structures

A set is just some things. 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 any old countable set. 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 sequence, 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 $0$, $\text{suc}$, $+$, $\cdot$, and $\leq$, 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. For instance, it’s fine to use the length-three string $\text{suc}$ as a function symbol.

Note that since there are only countably many different strings, we are only considering signatures that have countably many different constants, function symbols, and relation symbols. These are called countable signatures. In fact, most of the languages we will consider are even more restricted, and have finite signatures. (To be explicit: a finite signature includes only finitely many constants, finitely many function symbols, and finitely many predicates.)

(In other contexts sometimes it’s nice to 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 counts as a constant symbol for
For a language like that you might want an *uncountable* signature. But for the purposes of this text it’s convenient to be more restrictive: we’ll be focusing just 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, \( \land (x) \) 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 3.4.

### 3.1.2 Example

The **signature of the language of arithmetic** has one constant symbol \( 0 \), one one-place function symbol \( \text{suc} \), two two-place function symbols \( + \) and \( \cdot \), and one relation symbol \( \leq \).

### 3.1.3 Example

The **signature of the language of strings** has a two-place function symbol \( \oplus \), a relation symbol \( \preccurlyeq \) (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. An exception to this pattern is the quotation mark \" itself. It would be confusing and potentially ambiguous to use \"\" as a constant. So we’ll use the constant **quote**, instead.

---

\(^1\)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]
CHAPTER 3. STRUCTURES

(There is another exception, though it won’t matter until much later on in the course. In Section 6.1 and Chapter 7, we will use multi-line strings to write down programs and proofs. For this purpose, we have a symbol in our alphabet that represents the start of a new line. This symbol is difficult to write down on its own. Our constant that stands for the new line symbol is newline.)

(This is a pretty large signature, since our standard alphabet has a lot of different symbols. But it is still finite.)

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 $\mathbb{N}(0, \text{suc}, +, \cdot, \leq)$, which we just call $\mathbb{N}$ for short, has the following components:
3.1. SIGNATURES AND STRUCTURES

1. The domain \( D_\mathbb{N} \) is the set of natural numbers \( \mathbb{N} \).

2. The extension of the constant \( 0 \) (that is, \( 0_\mathbb{N} \)) is the number zero.

3. The extension of the function symbol \( \text{suc} \) is the successor function.

4. The extension of the function symbol \( + \) is the addition function.

5. The extension of the function symbol \( \cdot \) is the multiplication function.

6. The extension of the relation symbol \( \leq \) 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:

\[
\begin{align*}
D_\mathbb{N} &= \mathbb{N} \\
0_\mathbb{N} &= 0 \\
\text{suc}_\mathbb{N} &= \text{suc} \\
+_{\mathbb{N}}(m, n) &= m + n \quad \text{for each } m, n \in \mathbb{N} \\
\cdot_{\mathbb{N}}(m, n) &= m \cdot n \quad \text{for each } m, n \in \mathbb{N} \\
\leq_\mathbb{N} &= \{(m, n) \in \mathbb{N} \times \mathbb{N} \mid m \leq n\}
\end{align*}
\]

Note that we need to be careful about use and mention here as well. The word \textbf{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.

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 \( \text{suc} \) stands for the successor function. (But unlike \( \mathbb{N}(0, \text{suc}, +, \cdot, \leq) \), this structure doesn’t have the symbols \( +, \cdot \), or \( \leq \) 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 \( S \) 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 \( \oplus \) is the function that joins two strings together. The extension of the relation symbol \( \preceq \) is the set of pairs of strings \((s, t)\) such that \( \text{length } s \leq \text{length } t \).

In other words:

\[
D_S = S \\
\oplus_S (s, t) = s \oplus t \quad \text{for each } s, t \in S \\
\preceq_S = \{(s, t) \mid \text{length } s \leq \text{length } t\} \\
""_S = () \quad \text{(that is, the empty string)}
\]

And for each symbol \( a \) in the standard alphabet, if \( c \) is its corresponding singleton constant, then

\[
c_S = (a)
\]

For example, "A" \( _S = A \), and quote \( _S = " \).

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 \( \emptyset, +, \) 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 \((\emptyset, \text{suc}, (+))\) given as follows:

1. The domain of \( S \) consists of all of the buildings in Los Angeles.
2. The extension of \( \emptyset \) in \( S \) is the Natural History Museum of Los Angeles County.
3. The extension of $\text{suc}$ 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 $\oplus$ 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 $\mathbb{N}(0, \text{suc}, +, \cdot, \leq)$ 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 suc 0, so we can use the expression $\text{suc } 0$ to pick it out. Similarly, we can use $\text{suc suc } 0$ to stand for the number two, and so on. Here $\text{suc } 0$ is a complex symbol, built out of two basic symbols $\text{suc}$ and $0$. The things we get by putting these symbols together are called numerals: they are labels for numbers. The numerals are these expressions:

$$0, \ \text{suc } 0, \ \text{suc suc } 0, \ \text{suc suc suc } 0, \ \ldots$$

We can also describe a number in other ways: for example, the number two isn’t just suc suc 0, but it’s also $(\text{suc } 0)+(\text{suc } 0)$ (that is, $1+1$). We also have $\oplus$ in the language
of arithmetic, so we can also build up the expression $(\text{suc } 0) + (\text{suc } 0)$ 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 4.)

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 $\text{suc suc } 0 \cdot (\text{suc } 0 + \text{suc } 0)$. 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 shows how each stage is built up using one of the basic symbols.

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.

Here are the rules for forming terms in the language of arithmetic. Each rule means: given the facts written above the line, we can derive the fact written below the line.
3.2. TERMS

A list of rules like this is sometimes called a grammar. These formation rules show us the inductive structure of terms—along the same lines as the Axiom of Numbers and the Axiom of Sequences.

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 $\cdot$, look best in between the two things they apply to (“infix” notation). Other two-place function symbols, like $\text{cons}$ 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 $\cdot$ and $\oplus$. 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 \( \text{suc} \ 0 + 0 \)—officially we are really talking about a different, closely related string—in this case, \( +\{\text{suc}(0), 0\} \). In practice, this shouldn’t really be a big deal. (There will be other notational issues like this that come up later on.)

### 3.2.1 Definition

The set of **closed terms** for a signature \( L \) are given inductively by the following rules:

- \( c \) is a constant
- \( c \) is a term
- \( f \) is a one-place function symbol
- \( t \) is a term
  - \( f(t) \) is a term
- \( f \) is a two-place function symbol
- \( t_1 \) and \( t_2 \) are terms
  - \( f(t_1, t_2) \) 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 3.6 why the definition says “closed terms.”

It’s worth pausing here on a philosophical question. Are terms really just strings of symbols? This is similar to some questions we encountered before: whether sequences are really functions from numbers, and whether functions are really sets of ordered pairs, and 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 \( +00 \) 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.

But one important thing to check is that strings at least have the right structural features to play the role of terms. When we say that the terms are “given inductively” by the formation rules for constants and function symbols, what we mean is that every term can be formed in exactly one way using these rules. Just like with numbers and finite sequences, we can spell this idea out using an Inductive Property and an Injective Property. We’ll do this carefully in Section 3.3. In this
section, though, we’ll start by focusing on the most important practical upshots of our inductive definition. We can do a new kind of \textit{inductive proof}: induction on the syntactic structure of terms. This works very similarly to induction on numbers or sequences. And we can also give a new kind of \textit{recursive definition}, which also uses the syntactic structure of terms.

3.2.2 \textbf{Technique (Induction on Closed Terms)}
Suppose that we want to show that every closed term is \textit{nice}. We can do this in three steps.

1. Let \(c\) be any constant. \textit{Show} that \(c\) is nice.

2. Let \(f\) be any one-place function symbol, and let \(t\) be any closed term. \textit{Suppose} that \(t\) is nice. (This is the \textit{inductive hypothesis}.) Then \textit{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. \textit{Suppose} that \(t_1\) is nice and \(t_2\) is nice. (Again, this is the \textit{inductive hypothesis}.) Then \textit{show} that the term \(f(t_1,t_2)\) is also nice.

3.2.3 \textbf{Example}
Every closed term contains at least one constant.

\textit{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.

\[\square\]

3.2.4 \textbf{Exercise}
Say a string is \textbf{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 inductive definition of closed terms also gives us a new kind of recursive definition. In order to define a function that assigns a value to every term, you can assume that you have already defined the function for each subterm.

### 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.

\[
\begin{align*}
\emptyset & \mapsto 1 \\
\text{suc } 0 & \mapsto 2 \\
\text{suc } 0 + 0 & \mapsto 4 \\
\text{suc } 0 \cdot (\text{suc } 0 + 0) & \mapsto 7
\end{align*}
\]

Here is the recursive definition:

\[
\begin{align*}
\text{complex } c &= 1 \\
\text{complex}(f(t)) &= 1 + \text{complex } t \\
\text{complex}(f(t_1, t_2)) &= 1 + \text{complex } t_1 + \text{complex } t_2
\end{align*}
\]

As usual, recursive definitions work hand in hand with inductive proofs.

### 3.2.6 Example

For any closed term \( t \),

\[
\text{complex } t \leq \text{length } t
\]

**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:

\[
\text{complex } t \leq \text{length } t
\]
3.2. TERMS

Then
\[
\text{complex } f(t) = 1 + \text{complex } t \\
\leq 1 + \text{length } t \\
< \text{length } f + 1 + \text{length } t + 1 \\
= \text{length } f(t)
\]

(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:
\[
\text{complex } t_1 \leq \text{length } t_1 \\
\text{complex } t_2 \leq \text{length } t_2
\]

Then
\[
\text{complex } f(t_1, t_2) = 1 + \text{complex } t_1 + \text{complex } t_2 \\
\leq 1 + \text{length } t_1 + \text{length } t_2 \\
< \text{length } f + 1 + \text{length } t_1 + 1 + \text{length } t_2 + 1 \\
= \text{length } f(t_1, t_2)
\]

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 \( \text{suc 0 + 0} \) 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 \( \text{suc 0 + 0} \) 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 $f_Sd$, 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 $⟦·⟧_S$, with a dot indicating where to write the function’s argument.) Using this notation, we can rewrite the recursive definition more concisely.

\[
\begin{align*}
⟦c⟧_S &= c_S & \text{for each constant } c \\
⟦f(t)⟧_S &= f_S⟦t⟧_S & \text{for each one-place function symbol } f \text{ and term } t \\
⟦f(t_1, t_2)⟧_S &= f_S(⟦t_1⟧_S, ⟦t_2⟧_S) & \text{for each two-place function symbol } f \text{ and terms } t_1 \text{ and } t_2
\end{align*}
\]

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 $\text{suc suc 0 + suc 0}$ denotes the number three, in the standard model of arithmetic $\mathbb{N}$. That is,

\[
⟦\text{suc suc 0 + suc 0}⟧_{\mathbb{N}} = 3
\]

(In our totally official notation, this term would be written

\[+(\text{suc(suc(0))), suc(0))\]

But you don’t have to bother with this, unless you really want to.)
3.2. TERMS

Proof

\[\llbracket \text{suc suc 0 + suc 0} \rrbracket = \llbracket \text{suc suc 0} \rrbracket + \llbracket \text{suc 0} \rrbracket\]
by the clause for the function symbol \(+\)

\[= \text{suc} \llbracket \text{suc 0} \rrbracket + \llbracket \text{suc 0} \rrbracket\]
by the \text{suc} clause (twice)

\[= \text{suc suc} [\text{0}] + \llbracket [\text{0}] \rrbracket\]
by the \text{suc} clause again

\[= \text{suc suc 0 + suc 0}\]
by the clause for the constant symbol \([\text{0}]\)

\[= 2 + 1 = 3\]
□

3.2.9 Exercise

Use the definition of the denotation function in the standard string structure \(\mathcal{S}\) to show the following:

(a) The term \("" \oplus "A") \oplus "B"\) denotes the string \(AB\) in \(\mathcal{S}\). That is,

\[\llbracket ('' \oplus "A") \oplus "B" \rrbracket_{\mathcal{S}} = AB\]

(b) For any term \(t\), the term \(t \oplus ""\) has the same denotation in \(\mathcal{S}\) as \(t\). That is,

\[\llbracket t \oplus "" \rrbracket_{\mathcal{S}} = \llbracket t \rrbracket_{\mathcal{S}}\]

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 \(\langle 0 \rangle\), the numeral for the number one is the term \(\text{suc } 0\), the numeral for the number two is the term \(\text{suc suc } 0\), 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.

\[\langle 0 \rangle = 0\]

\[\langle \text{suc } n \rangle = \text{suc} \langle n \rangle \quad \text{for every } n \in \mathbb{N}\]

(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 \text{suc} on the left side is a function on numbers, while the \text{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:

$$\llbracket \langle n \rangle \rrbracket_\mathbb{N} = n \quad \text{for every } n \in \mathbb{N}$$

(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.

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 $\mathbb{N}(0, \text{succ})$ is explicit.

(c) Show that the string structure $\mathcal{S}$ is explicit, by recursively defining a function that takes each string $s \in \mathcal{S}$ to some term $\langle s \rangle$ in the standard language of strings such that $\llbracket \langle s \rangle \rrbracket_\mathcal{S} = s$ (as in Exercise 3.2.11).

3.3 More Officially*

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. 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. These are very closely analogous to the properties with the same names for numbers and sequences. In this section we’ll state these properties more carefully. We’ll also see how, in the case of terms, we don’t have to build these properties into an axiom, the way we did with numbers and
sequences. Instead we can get them as *theorems*, using a carefully chosen *definition* of closed terms.

The idea of the Inductive Property for terms is very similar to our official Inductive Properties for numbers and strings. 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. What this amounts to is that *proof by induction on the structure of terms works*.

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 $f(t_1, t_2)$ 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 $X$.

### 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.

For numbers and strings, we built the Inductive Property into certain basic *axioms*. But we are taking terms to just be certain strings, rather than their own kind of thing obeying their own basic principles. Because of this, we can get their inductive
property not from an axiom, but rather from a more official definition of the set of terms. It goes like this.

3.3.3 Definition (Official Version)
For any signature $L$, the set of closed $L$-terms is the set of strings

$$\{ t \in \mathbb{S} \mid t \in X \text{ for every } L\text{-hereditary set } X \}$$

It is straightforward to check that the Inductive Property for Terms follows from this definition.

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 last 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.)

3.3.4Injective 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. Consider three “term-forming” functions:

- $T_0$ takes each constant to itself;
- $T_1$ takes each pair of a one-place function symbol $f$ and a term $t$ to the term $f(t)$;
- $T_2$ takes each pair of a two-place function symbol $g$ and a pair of terms $(t_1, t_2)$ to the term $g(t_1, t_2)$. 

3.4. PARSING TERMS*

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.

It’s important that this Injective Property is true, but proving it is surprisingly fiddly and not especially illuminating (unless, for example, you are interested in writing a computer program to interpret syntactic structures). A proof is included in Section 3.4 for completeness, but feel free to skip over it unless you are curious.

3.4 Parsing Terms*

In this section we’ll work through the proof that the string representations for terms described in the previous section really do uniquely pick out the syntactic structure of terms. There is only one way of “parsing” a term. In other words, what we will prove is the Injective Property for Terms.

To prove this, 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 $\text{suc}(x)$ as one of your constants, you could get ambiguities. To keep things simple, we’ll just say that you aren’t allowed to use any parentheses or commas in your constants or function symbols. Call a signature like this simple.

Really, this is a bit more restrictive than we really want: the most straightforward notation for the language of strings uses constants like “(“ and “)”, and this doesn’t have to introduce any ambiguity if we’re careful about it. But it’s a nuisance to handle this special case correctly, so let’s just ignore this complication. (To avoid the issue, we could always officially make our signatures simple by just using boring constants instead, like $\text{symbol1}$, $\text{symbol2}$, and so on.)

(Another issue that comes up later is that we want to make sure that constants can be distinguished from variables, and also later on 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.)

One key fact that we will use is Exercise 3.2.4: any string which is a term is balanced, meaning that it has the same number of left and right parentheses.

3.4.1 Definition

A string $s$ is a delimited initial substring of $t$ iff $s$ is an initial substring of $t$ which is followed by a comma or right parenthesis: that is, either $s \leq t$ or $s \leq t$. 


3.4.2 Definition

A string $s$ is **left-heavy** iff $s$ contains strictly more left parentheses than right parentheses.

3.4.3 Lemma

Every delimited initial substring of a term is left-heavy.

*Proof*

We prove this by induction on the structure of terms.

If $c$ is a constant, then $c$ does not include any parentheses or commas, so it doesn’t have any delimited proper substrings.

Suppose $s$ is a delimited initial substring of $f(t)$. Since $f$ doesn’t include any commas or parentheses, $s$ must be of the form

$$f(s')$$

where $s'$ is a delimited initial substring of $t$. So either $s' = t$, or else $s'$ is a delimited initial substring of $t$. In the first case, $s'$ is balanced. In the second case, $s'$ is left-heavy by the inductive hypothesis. So $s$ is also left-heavy, since it includes all the parentheses in $s'$ plus one extra $()$.

Suppose $s$ is a delimited initial substring of $f(t_1,t_2)$. Then there are two possible cases:

1. $s$ is

$$f(s')$$

where $s'$ is a delimited substring of $t_1$. In this case, either $s'$ is $t_1$—in which case $s'$ is balanced—or else $s'$ is a delimited initial substring of $t_1$, in which case by the inductive hypothesis $s'$ is left-heavy. In either case, $s$ is left-heavy, since it includes the parentheses from $s'$ plus one extra $()$.

2. $s$ is

$$f(t_1,s')$$

where $s'$ is a delimited initial substring of $t_2$. By similar reasoning, $s'$ is balanced or left-heavy, and so $s$ is left-heavy as well.

□
3.4. **PARSING TERMS**

### 3.4.4 Exercise
No term is a delimited initial substring of another term.

### 3.4.5 The Injective Property for Terms (The Unique Readability Theorem)
Let $L$ be a simple signature, whose set of constants, one-place function symbols, and two-place function symbols are $C_0$, $C_1$, and $C_2$, respectively. Let $T$ be the set of $L$-terms. Recall that we have three term-building functions:

$$
T_0 : C_0 \rightarrow T \\
T_1 : C_1 \times T \rightarrow T \\
T_2 : C_2 \times T^2 \rightarrow T
$$

To be explicit, $T_0$ takes each constant symbol to itself, $T_1$ takes each pair of a function symbol $f$ and a term $t$ to the string $f(t)$, and $T_2$ takes each pair of a function symbol $f$ and a pair of terms $t_1$ and $t_2$ to the string $f(t_1, t_2)$.

$T_0$, $T_1$, and $T_2$ are each one-to-one functions, and their ranges have no elements in common.

**Proof**

It’s obvious that $T_0$ is one-to-one. It’s also clear that the range of $T_0$ is disjoint from the ranges of $T_1$ and $T_2$, since no string in the range of $T_0$ includes any parentheses.

Suppose that for some function symbols $f$ and $f'$ and terms $t$ and $t'$,

$$s = f(t) = f'(t')$$

Since function symbols don’t include parentheses, we know that $f$ is the initial substring of $s$ that includes everything before the first $()$. Likewise, we know the same thing about $f'$. So $f$ and $f'$ must be the same string. Then by the Cancellation Property, $t = t'$, and then by Cancellation again (on the other side), $t = t'$. So $T_1$ is one-to-one.

By similar reasoning, if

$$s = f(t) = f'(t_1, t_2)$$

then $f$ and $f'$ are the same string, and thus $t$ is the same as $t_1, t_2$. But in that case, $t_1$ would be a delimited initial substring of $t$, which is impossible. So $T_1$ and $T_2$ have disjoint ranges.

Finally, suppose

$$s = f(t_1, t_2) = f'(t'_1, t'_2)$$
In that case, we can use similar reasoning to deduce that
\[ t_1, t_2 = t'_1, t'_2 \]
Since \( t_1 \) and \( t'_1 \), are both initial substrings of the same string, one of them must be an initial substring of the other. Thus either \( t_1 = t'_1 \), or else \( t_1 \) is a delimited initial substring of \( t'_1 \), or else \( t'_1 \) is a delimited initial substring of \( t_1 \). But Exercise 3.4.4 rules out the second and third options, so \( t_1 = t'_1 \). Then by cancellation, \( t_2 = t'_2 \) as well.

\[ \square \]

3.5 The Recursion Theorem 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 is 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.

3.5.1 The Recursion Theorem for Terms

Let \( L \) be a signature. Let \( C_0, C_1, \) and \( C_2 \) 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:
\[
T_0 : C_0 \rightarrow T \\
T_1 : C_1 \times T \rightarrow T \\
T_2 : C_2 \times T^2 \rightarrow T
\]
Now, let \( A \) be any set, and consider any three functions with the same shape:
\[
f_0 : C_0 \rightarrow A \\
f_1 : C_1 \times A \rightarrow A \\
f_2 : C_2 \times A^2 \rightarrow A
\]
3.6. VARIABLES

Then there is a unique function $r : T \rightarrow A$ with the following Recursive Properties:

$$
\begin{align*}
rc &= f_0 c & \text{for each constant } c \\
r(f(\mathit{t})) &= f_1(f, rt) & \text{for each one-place function symbol } f \text{ and term } \mathit{t} \\
r(f(\mathit{t}_1, \mathit{t}_2)) &= f_2(g, (rt_1, rt_2)) & \text{for each two-place function symbol } g \text{ and terms } \mathit{t}_1 \text{ and } \mathit{t}_2
\end{align*}
$$

This theorem can be proved using the Injective Property and Inductive Property for terms in a similar way to the proof of the Recursion Theorem for numbers. But we won’t go into this.

TODO. Add a very short proof sketch.

TODO. At this point it would be cool to discuss the abstract version, and the universal properties of inductive structures.

3.6 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 $\text{suc } 0 + \text{suc suc } 0$ to label the number three; and we can use $+$ to label the addition function, or $\text{suc}$ to label the successor function. But how about the “add two” function?

$$
\begin{bmatrix}
0 & 2, \\
1 & 3, \\
2 & 4, \\
\vdots
\end{bmatrix}
$$

Or how about the doubling function?

$$
\begin{bmatrix}
0 & 0, \\
1 & 2, \\
2 & 4, \\
\vdots
\end{bmatrix}
$$

We can represent these functions using a language with variables. For instance, the “add two” function can be represented by the term $\text{suc suc } x$ . (“For each $x$, take the successor of the successor of $x$.”) Similarly, the doubling function can be represented by the term $x \cdot \text{suc suc } 0$ . (“For each $x$, multiply $x$ by 2.”) Of course, these aren’t the only options. We could also use $x + \text{suc suc } 0$ 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 countably 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 $x_0$, $x_{12}$, 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.

3.6.1 Definition

The terms for a signature $L$ are given inductively by the following four rules:

\[
\begin{align*}
\text{$x$ is a variable} & \quad \text{$x$ is a term} \\
\text{$c$ is a constant} & \quad \text{$c$ is a term} \\
\text{$f$ is a one-place function symbol} & \quad \text{$t$ is a term} \\
\text{$f(t)$ is a term} \\
\text{$g$ is a two-place function symbol} & \quad \text{$t_1$ and $t_2$ are terms} \\
\text{$g(t_1, t_2)$ is a term}
\end{align*}
\]

(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 $z_2$ is a variable, $z_2$ 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 $x$!)

Officially, this definition can be unpacked in terms of another Inductive Property and Injective Property, where we have to add on 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.
3.6. VARIABLES

3.6.2 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.

3.6.3 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 $\text{Var}$ that takes each term to the set of variables that occur in that term:

$$\text{Var } x = \{x\} \quad \text{for each variable } x$$

$$\text{Var } c = \emptyset \quad \text{for each constant } c$$

$$\text{Var } f(t) = \text{Var } t \quad \text{for each one-place function symbol } f \text{ and term } t$$

$$\text{Var } f(t_1, t_2) = \text{Var } t_1 \cup \text{Var } t_2 \quad \text{for each two-place function symbol } f \text{ and terms } t_1 \text{ and } t_2$$

Then, finally, we say $x$ occurs in $t$ iff $x \in \text{Var } t$. 
CHAPTER 3. STRUCTURES

3.6.4 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.

3.6.5 Technique (Recursively defining 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 \( r(t_2) \).

3.6.6 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:

<table>
<thead>
<tr>
<th>Term</th>
<th>Plugged In</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{succ succ } x )</td>
<td>( \text{succ succ } (0 + 0) )</td>
</tr>
<tr>
<td>( x + \text{succ } x )</td>
<td>( (0 + 0) + \text{succ } (0 + 0) )</td>
</tr>
<tr>
<td>( x + \text{succ } y )</td>
<td>( (0 + 0) + \text{succ } y )</td>
</tr>
<tr>
<td>( y + y )</td>
<td>( y + y )</td>
</tr>
</tbody>
</table>
3.6. VARIABLES

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.

3.6.7 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 \( y \),

\[
y[x \mapsto a] = \begin{cases} 
a & \text{if } y \text{ is } x \\
y & \text{otherwise}
\end{cases}
\]

2. For each constant \( c \),

\[
c[x \mapsto a] = c
\]

3. For each one-place function symbol \( f \) and term \( t \),

\[
f \left( t \right)[x \mapsto a] = f \left( t[x \mapsto a] \right)
\]

4. For each two-place function symbol \( f \) and terms \( t_1 \) and \( t_2 \),

\[
f \left( t_1, t_2 \right)[x \mapsto a] = f \left( t_1[x \mapsto a], t_2[x \mapsto a] \right)
\]

3.6.8 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.
CHAPTER 3. STRUCTURES

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.

3.6.9 Proposition
Suppose \( t(x) \) and \( u(y) \) are terms of one variable, and \( a \) is a term. Then these are the very same term:

\[
t[x \mapsto u][y \mapsto a] = t[x \mapsto u[y \mapsto a]]
\]

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 \( t \),

\[
t[u][a] = t[u[a]] \tag{3.1}
\]

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.

\[
\begin{align*}
f(t)[u][a] &= f(t)[u][a] \quad \text{definition of substitution} \\
&= f(t[u][a]) \quad \text{definition of substitution} \\
&= f(t[u][a]) \quad \text{inductive hypothesis} \\
&= f(t)[u[a]] \quad \text{definition of substitution}
\end{align*}
\]

4. The step for two-place function symbols is similar.

This completes the induction.

\[\square\]

### 3.6.10 Exercise

Prove by induction that if \( x \) doesn’t occur in \( t \), then \( t[x \mapsto a] = t \).

### 3.6.11 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 \( a \).

### 3.6.12 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,

\[
\text{If } \llbracket a \rrbracket_S = \llbracket a' \rrbracket_S \quad \text{then } \llbracket t(a) \rrbracket_S = \llbracket t(a') \rrbracket_S
\]

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 \( \text{suc } x \), 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 its variables*. First, let’s define what we mean by a “choice of values.”
3.6.13 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.

3.6.14 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 $g(x)$, 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 $f_Sd$ 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 $[t]_Sg$. So we can restate the definition more concisely.

\[
\begin{align*}
[x]_Sg &= g(x) \\
[c]_Sg &= c_S \\
[f(t)]_Sg &= f_S([t]_Sg) \\
[f(t_1,t_2)]_Sg &= f_S([t_1]_Sg,[t_2]_Sg)
\end{align*}
\]

We often drop the $S$ subscript when it’s clear in context which structure we’re talk-
ing about. So here it is again, a bit tidier:

\[
\begin{align*}
\langle x \rangle_g &= g(x) \\
\langle c \rangle_g &= c_S \\
\langle f(t) \rangle_g &= f_S(\langle t \rangle_g) \\
\langle f(t_1, t_2) \rangle_g &= f_S(\langle t_1 \rangle_g, \langle t_2 \rangle_g)
\end{align*}
\]

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.

### 3.6.15 Example
Recall that \(S\) is the standard string structure. Let \(g\) be the assignment \([x \mapsto ABC]\). The term \(x \circ "D"\) denotes \(ABCD\) with respect to \(g\) in \(S\). We can show this explicitly using the definition.

\[
\langle x \circ "D" \rangle_g = \langle x \rangle_g \oplus \langle "D" \rangle_g \quad \text{since } \circ_S \oplus = \langle x \rangle_g \oplus D \quad \text{since } "D"_S = D \\
= ABC \oplus D \quad \text{since } g(x) = ABC \\
= ABCD
\]

### 3.6.16 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:

\[
\langle t \rangle_g = \langle t \rangle_h
\]

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.

### 3.6.17 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.
CHAPTER 3. STRUCTURES

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.

3.6.18 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 \textbf{denotation of} \( t(x) \) \textbf{at} \( d \) \textbf{in} \( S \) \textbf{is} the denotation of \( t \) with respect to the variable assignment \([x \mapsto d]\) \textbf{in} \( S \).

Similarly, suppose \( t(x, y) \) is a term of two variables. Then for any \( d_1, d_2 \in D \), the \textbf{denotation of} \( t \) \textbf{at} \((d_1, d_2)\) \textbf{in} \( S \) \textbf{is} the denotation of \( t \) with respect to the assignment \([x_1 \mapsto d_1, x_2 \mapsto d_2]\) \textbf{in} \( S \).

This generalizes straightforwardly to terms of \( n \) variables and sequences of \( n \) objects.

3.6.19 Definition
Let \( S \) be a structure and let \( t(x) \) be a term of one variable. The \textbf{extension} of \( t(x) \) \textbf{in} \( S \) \textbf{is} the function that takes each \( d \in D_S \) to the denotation of \( t(x) \) \textbf{at} \( d \) \textbf{in} \( S \). We use the notation \( \llbracket t \rrbracket_S \) \textbf{for} the extension of \( t \) \textbf{in} \( S \). That is,

\[
\llbracket t \rrbracket_S(d) = \llbracket t \rrbracket_S[x \mapsto d]
\]

(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 \textbf{extension} of \( t \) \textbf{in} \( S \) \textbf{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 \textit{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.

3.6.20 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 \( \text{succ} \), \( + \), or \( \oplus \), 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 \( t(x) \).

3.6.21 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 5.1, when we have introduced more expressive languages.)
3.6.22 Exercise
Show that the doubling function is simply definable in the standard model of arithmetic.

3.6.23 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 $S$.

3.6.24 Exercise
Let $L$ be any signature. Is the set of $L$-terms finite, countably infinite, or uncountable? Explain.

3.6.25 Exercise
Let $L$ be a signature, and let $S$ be an $L$-structure with an infinite domain $D$. Show that there is some function $D \rightarrow D$ which is not simply definable in $S$.

*Hint.* Use a counting argument.

3.7 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 3.6.2)
- You can recursively define a function whose domain is the set of all terms. (Technique 3.6.5)

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 3.6.1)
• You can plug a term into another term using substitution. (Definition 3.6.7)

• In any structure, each term denotes an object, with respect to an assignment of values for its variables. (Definition 3.6.18)

• 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 3.6.19)

• These ideas (substitution, denotation, and extension) are closely related: for any term \( t(x) \) and any term \( a \),

\[
\llbracket t \rrbracket [x \mapsto \llbracket a \rrbracket] = \llbracket t[x \mapsto a] \rrbracket
\]

(in any structure). (Exercise 3.6.20)

• In an infinite structure, not every function is the extension of a term. (Exercise 3.6.25)
Chapter 4

First-Order Logic

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.

So far, our language has been 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 8.)

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.

4.1 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.

\[\forall x \forall y \forall z \ (x + (y + z) = (x + y) + z)\]

\[\exists x \ (\forall y \ (x + y = y) \land \forall y \ (y + x = y))\]

\[\forall x \forall y \exists z \ (x + z = y)\]

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

\[\forall y \ (x + y = y)\]

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.

4.1.1 Definition

Let \(L\) be a signature. We have already defined the structure of \(L\)-terms. Just to
refresh our memory, terms are inductively defined by the following rules:

\[
\begin{align*}
\text{\(x\) is a variable} & \quad \Rightarrow \quad \text{\(x\) is a term} \\
\text{\(c\) is a constant} & \quad \Rightarrow \quad \text{\(c\) is a term}
\end{align*}
\]
4.1. SYNTAX

\[ f \text{ is a one-place function symbol} \quad a \text{ is a term} \]
\[ f(a) \text{ is a term} \]

\[ f \text{ is a two-place function symbol} \quad a \text{ and } b \text{ are terms} \]
\[ f(a, b) \text{ is a term} \]

The first-order formulas in the language \( L \), or \( L \)-formulas, for short, are defined inductively 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.

\[ a \text{ is a term} \quad b \text{ is a term} \]
\[ (a = b) \text{ is a formula} \]

\[ F \text{ is a one-place predicate} \quad a \text{ is a term} \]
\[ F(a) \text{ is a formula} \]

\[ R \text{ is a two-place predicate (relation symbol)} \quad a \text{ and } b \text{ are terms} \]
\[ R(a, b) \text{ is a formula} \]

(It’s easy to extend first-order languages to include 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 \( \leq \), in between the two terms it applies to, as in \( 0 \leq \text{suc } 0 \). 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 \( \leq(0, \text{suc}(0)) \). 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.

\[ A \text{ is a formula} \]
\[ \neg A \text{ is a formula} \]

\[ A \text{ is a formula} \quad B \text{ is a formula} \]
\[ (A \land B) \text{ is a formula} \]

\[ A \text{ is a formula} \quad x \text{ is a variable} \]
\[ \forall x \quad A \text{ is a formula} \]
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 $\rightarrow$, or the existential quantifier $\exists$. That’s because we can define them up from the basic materials in the definition. For example, later on we’ll define $\exists x \ (x + x = x)$ to be just a notational shortcut for the official formula $\forall x \ \neg(x + x = x)$. 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, sequences, and terms, this inductive definition of formulas encapsulates 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. Again, we won’t bother being totally official about this. 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.1

4.1.2 Exercise
Write out the Inductive Property and the Injective Property for Formulas, based on the formation rules in Definition 4.1.1. (You can use the Inductive Property and the Injective Property for terms as a model.)

4.1.3 Example
The first-order language of arithmetic consists of the first-order formulas with the signature of the language of arithmetic: $0$, $\text{suc}$, $+$, $\cdot$, and $\leq$.

The first-order language of strings consists of the first-order formulas with the signature of the language of strings: $\emptyset$, $\subseteq$, the empty-string constant $\text{""}$, and the constant for each symbol in the standard alphabet.

---

1Note that, just like we did with terms in Section 3.3, since we are defining formulas to be certain strings, we can guarantee the Inductive Property by choosing a suitable definition for the set of formulas. To check that this set of strings has the Injective Property, technically we need to prove another parsing theorem, like the one for terms in Section 3.4. 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.
4.1.4 Example
Let $A$ be this formula in the language of arithmetic:

$$\neg((x = y) \land \forall z \neg((x + y = x + z) \land \neg(x = z)))$$

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 $\forall z$, so it is not a free variable.

4.1.5 Definition
A variable $x$ is **free** in a formula $A$ iff it satisfies the following recursive definition.

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 3.6.3).
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 $(A \land B)$ 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 $A$.

4.1.6 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

$$\neg(x = 0) \land \forall y \neg(x + y = y)$$

This says, “$x$ is not zero, and adding $x$ to anything never gives you the same thing back”. We can plug the term $\text{suc } \emptyset$ into the $x$ slot, to get the sentence
\[
\neg(suc\ 0 = 0) \land \forall y \neg(suc\ 0 + y = y)
\]

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 \(\forall x\) appears in the formula? For instance, what happens when we plug \(suc\ 0\) in for \(x\) in this formula, instead?

\[
\neg(x = 0) \land \neg\forall x \ (x + x = x)
\]

Only the first occurrence of \(x\) is free here—the other occurrences are “bound” by the quantifier \(\forall x\). So when we plug in \(suc\ 0\), we get

\[
\neg(suc\ 0 = 0) \land \neg\forall x \ (suc\ 0 + x = x)
\]

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

\[
\forall y \ (x = y)
\]

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

\[
\forall y \ (suc\ y = y)
\]

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 \(\forall y\).

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 \(\forall y \ (x = y)\), we could plug it into \(\forall z \ (x = z)\) instead.
Then the result is \( \neg \forall z \ \text{succ} \ y = z \), 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, so you don’t really have to worry about it.)

4.1.7 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 for each subformula of \( A \) which has the form \( \forall y \ B \), if \( y \) occurs in \( t \), then \( x \) is not free in \( B \).\(^2\)

4.1.8 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 \( A[x \leftrightarrow t] \) is defined recursively as follows.

\[
\begin{align*}
(a = b)[x \leftrightarrow t] & \text{ is } (a[x \leftrightarrow t] = b[x \leftrightarrow t]) \\
R(a, b)[x \leftrightarrow t] & \text{ is } R(a[x \leftrightarrow t], b[x \leftrightarrow t]) \\
(\neg A)[x \leftrightarrow t] & \text{ is } \neg(A[x \leftrightarrow t]) \\
(A \land B)[x \leftrightarrow t] & \text{ is } (A[x \leftrightarrow t] \land B[x \leftrightarrow t]) \\
(\forall y \ A)[x \leftrightarrow t] & \text{ is } \begin{cases} \forall y \ A & \text{ if } x \text{ and } y \text{ are the same variable} \\ \forall y \ A[x \leftrightarrow t] & \text{ otherwise, as long as } y \text{ does not occur in } t \end{cases}
\end{align*}
\]

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 \( A \), and the substitution instance is undefined. (But we won’t always bother stating this qualification explicitly.)

4.1.9 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 \( A[x \leftrightarrow a] \). Similarly if \( B(x, y) \) is a formula of two variables, then \( B(a, y) \) means the same thing as \( B[x \leftrightarrow a] \), and \( B(x, a) \) means the same thing as \( B[y \leftrightarrow a] \).

\(^{2}\)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.
4.1.10 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 $A$.

4.2 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:

$$\forall x \exists y \ (y = \text{suc} \ x) \land \neg \forall x \exists y \ (\text{suc} \ y = x)$$

(We haven’t officially introduced the existential quantifier $\exists$ 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 $\mathbb{N}$:

$$\forall x \forall y \exists z \ (x + z = y)$$

As we noted in Section 4.1, 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:

$$\forall x \forall y \exists z \ ((x + z = y) \lor (y + z = x))$$

(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.
4.2. SEMANTICS

But what would it even mean to say that an open formula like \( \text{suc} \ y = x \) 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 3.6. Here is a reminder:

4.2.1 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 \( A \).

4.2.2 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 \( y \),

\[
g[x \mapsto d](y) = \begin{cases} d & \text{if } x = y \\ g(y) & \text{if } x \text{ is distinct from } y \text{ and } y \text{ is in the domain of } g \end{cases}
\]

4.2.3 Definition
If \( S \) be a structure, \( A \) is a formula, and \( g \) is a variable assignment which is adequate for \( A \), we’ll recursively define the relation “\( g \) satisfies \( A \) (in \( S \))” as follows.

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 \( \llbracket a \rrbracket g = \llbracket b \rrbracket g \).

2. Suppose \( F \) is a one-place predicate and \( a \) is a term. Then \( g \) satisfies the formula \( F(a) \) iff \( \llbracket a \rrbracket g \) 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 \( (\llbracket a \rrbracket g, \llbracket b \rrbracket g) \) 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 $(A \land B)$ 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 $A$.

4.2.4 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 $S$.

4.2.5 Definition
As we discussed in Section 3.6, 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 4.2.3 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.

4.2.6 Proposition
Let $S$ be a structure.

1. Suppose $a$ and $b$ are closed terms. Then $[\bar{a} \equiv b]$ is true in $S$ iff $a$ and $b$ have the same denotation in $S$; that is, $\llbracket a \rrbracket_S = \llbracket b \rrbracket_S$.

2. Suppose $F$ is a one-place predicate and $a$ is a closed term. Then $F(\bar{a})$ is true in $S$ iff $\llbracket a \rrbracket_S \in F_S$.

3. Suppose $R$ is a two-place predicate and $a$ and $b$ are closed terms. Then $R(\bar{a}, \bar{b})$ is true in $S$ iff $\llbracket a \rrbracket_S, \llbracket b \rrbracket_S \in R_S$.

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 $(A \land B)$ 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.

4.2.7 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 $\llbracket A \rrbracket_S g$ for short, is True if $g$ satisfies $A$ in $S$, and otherwise it is False. (Usually we leave off the $S$ subscript from $\llbracket A \rrbracket_S$ for simplicity.) That is:

$$\llbracket A \rrbracket g = \begin{cases} 
\text{True} & \text{if } g \text{ satisfies } A \text{ in } S \\
\text{False} & \text{otherwise}
\end{cases}$$

4.2.8 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 $\llbracket A \rrbracket g = \llbracket A \rrbracket h$.

**Proof**
We can prove this using a straightforward but tedious inductive proof, using the inductive 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 3.6.16 (which showed the analogous
fact for terms), we know that $\llbracket a \rrbracket g = \llbracket a \rrbracket h$, and likewise that $\llbracket b \rrbracket g = \llbracket b \rrbracket h$. So:

\[
\begin{align*}
g \text{ satisfies } (a = b) & \iff \llbracket a \rrbracket g = \llbracket b \rrbracket g \\
& \iff \llbracket a \rrbracket h = \llbracket b \rrbracket h \\
& \iff h \text{ satisfies } (a = b)
\end{align*}
\]

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 3.6.16,

\[
\llbracket a \rrbracket g = \llbracket a \rrbracket h \quad \text{and} \quad \llbracket b \rrbracket g = \llbracket b \rrbracket h
\]

Thus:

\[
\begin{align*}
g \text{ satisfies } R(a,b) & \iff (\llbracket a \rrbracket g, \llbracket b \rrbracket g) \in R_S \\
& \iff (\llbracket a \rrbracket h, \llbracket b \rrbracket h) \in R_S \\
& \iff h \text{ satisfies } R(a,b)
\end{align*}
\]

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:

\[
\begin{align*}
g \text{ satisfies } \neg A & \iff g \text{ does not satisfy } A \\
& \iff h \text{ does not satisfy } A \\
& \iff h \text{ satisfies } \neg A
\end{align*}
\]

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 \((A \land B)\). 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 \((A \land B)\)). 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:

\[
g \text{ satisfies } (A \land B) \iff g \text{ satisfies } A \text{ and } g \text{ satisfies } B \iff h \text{ satisfies } A \text{ and } h \text{ satisfies } B \iff h \text{ satisfies } (A \land B)
\]

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 4.1.5, 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 \( d \in D_S \),

\[
g[x \mapsto d] \text{ satisfies } A \iff h[x \mapsto d] \text{ satisfies } A
\]

Thus:

\[
g \text{ satisfies } \forall x \ A \iff g[x \mapsto d] \text{ satisfies } A \text{ for every } d \in D_S
\]

\[
\text{ iff } h[x \mapsto d] \text{ satisfies } A \text{ for every } d \in D_S
\]

\[
\text{ iff } h \text{ satisfies } \forall x \ A
\]

That completes the inductive proof. \( \square \)
4.2.9 Exercise

If $A$ is a sentence which is true in a structure $S$, then every variable assignment $g$ satisfies $A$ in $S$.

4.2.10 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, $\llbracket t \rrbracket g = d$. Then:

$$g \text{ satisfies } A[x \mapsto t] \iff g[x \mapsto d] \text{ satisfies } A$$

Or in our alternative notation:

$$\llbracket A[x \mapsto t] \rrbracket g = \llbracket A \rrbracket(g[x \mapsto d])$$

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:

$$A(t) \text{ is true in } S \iff A(x) \text{ is true of the denotation of } t \text{ in } S$$

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 3.6.20, 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

$$\llbracket a[x \mapsto t] \rrbracket = \llbracket a \rrbracket(x \mapsto d)$$

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 3.6.20 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)

If $\llbracket t \rrbracket g = d$ then $\llbracket a[x \mapsto t] \rrbracket g = \llbracket a \rrbracket(g[x \mapsto d])$
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 3.6.20.

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 3.6.20 tells us:

\[
\llbracket a[x \mapsto t] \rrbracket g = \llbracket a \rrbracket (g[x \mapsto d])
\]

\[
\llbracket b[x \mapsto t] \rrbracket g = \llbracket b \rrbracket (g[x \mapsto d])
\]

Thus:

\[
g \text{ satisfies } (a=b)[x \mapsto t] \iff g \text{ satisfies } \llbracket a[x \mapsto t] = b[x \mapsto t] \rrbracket
\]

\[
\iff \llbracket a[x \mapsto t] \rrbracket g = \llbracket b[x \mapsto t] \rrbracket g
\]

\[
\iff \llbracket a \rrbracket (g[x \mapsto d]) = \llbracket b \rrbracket (g[x \mapsto d])
\]

\[
\iff g[x \mapsto d] \text{ satisfies } (a=b)
\]

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 \((\forall x \ A)[x \mapsto t] \rrbracket \) 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:

\[
g \text{ satisfies } (\forall x \ A)[x \mapsto t] \iff g \text{ satisfies } \forall x \ A
\]

\[
\iff g[x \mapsto d] \text{ satisfies } \forall x \ A
\]

Finally we consider the case where \( y \) is a distinct variable from \( x \). In this case, the substitution instance \((\forall y \ A)[x \mapsto t] \rrbracket \) is the formula \( \forall y \ A[x \mapsto t] \). For the inductive hypothesis, we assume that for any adequate assignment \( g' \),

\[
g' \text{ satisfies } A[x \mapsto t] \iff g'[x \mapsto d] \text{ satisfies } A
\]
Thus,:

\[ g \text{ satisfies } (\forall y \ A)[x \mapsto t] \]
\[ \iff g \text{ satisfies } \forall y \ A[x \mapsto t] \]
\[ \iff g[y \mapsto d'] \text{ satisfies } A[x \mapsto t] \text{ for every } d' \in D_S \]
\[ \iff g[x \mapsto d'][y \mapsto d'] \text{ satisfies } A \text{ for every } d' \in D_S \]
\[ \iff g[x \mapsto d] \text{ satisfies } \forall y A \]

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. \[ \square \]

4.2.11 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:

\[ \text{If } \llbracket t \rrbracket = \llbracket t' \rrbracket \text{ then } \llbracket A(t) \rrbracket = \llbracket A(t') \rrbracket \]

So far we’ve just been working with our “primitive” logical symbols: \( \forall, \neg, \land, \text{ and } = \). 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”:

\[ \neg(\neg(x = 0) \land \neg(x = 1)) \]

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.
4.2. Definition
For any formulas $A$ and $B$, terms $a$ and $b$, and variable $x$:

(a) The **material conditional** $A \rightarrow B$ abbreviates the formula $\neg (A \land \neg B)$.

(b) The **biconditional** $A \leftrightarrow B$ abbreviates the formula $(A \rightarrow B) \land (B \rightarrow A)$.

(c) The **disjunction** $A \lor B$ abbreviates the formula $\neg A \rightarrow B$.

(d) The **standard truth** $\top$ is the formula $\forall x \ (x = x)$.

(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

$$\exists x \ \forall y \ ((x = y) \rightarrow A[x \mapsto y])$$

(where $y$ is a distinct variable from $x$).

(h) $a \neq b$ abbreviates the formula $\neg (a = b)$.

4.2.13 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) $A \rightarrow B \rightarrow C$ means $A \rightarrow (B \rightarrow C)$. (In this case it does matter!)

In informal proofs, sometimes you’ll see people write things like

**∀A**: A is a set $\rightarrow A \subseteq A$

or

A is consistent $\leftrightarrow \exists S \ (S \ is \ a \ structure \ \land \ A \ is \ true \ in \ S)$

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 5.) So I recommend that you just don’t do it. When you want to make a statement about what all things are like, write the word \textit{all}, not the abbreviation \( \forall \). Only use the symbol \( \forall \) when this symbol is part of a \textit{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 \textit{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 \textit{all} and \( \forall \)) 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.

\textbf{Example}

Let \( S \) be any structure.

(a) For any sentences \( A \) and \( B \), the sentence \( A \rightarrow B \) 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 \( d \).

\textit{Proof of (a)}

Let \( A \) and \( B \) be sentences. By Definition 4.2.12, \( A \rightarrow B \) abbreviates the sentence

\[ \neg (A \land \neg B) \]

Using Proposition 4.2.6,

\[ \neg (A \land \neg B) \] is true in \( S \)

iff \( (A \land \neg B) \) is not true in \( S \)

iff \( A \) and \( \neg B \) are not both true in \( S \)

iff either \( A \) is false in \( S \) or \( B \) is true in \( S \)

\textit{Proof of (b)}

By Definition 4.2.12, \( \exists x \ A(x) \) abbreviates

\[ \neg \forall x \ \neg A(x) \]
Using Proposition 4.2.6 again,

\[ \neg \forall x \neg A(x) \text{ is true in } S \]

iff \[ \forall x \neg A(x) \text{ is not true in } S \]

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. \( \square \)

4.2.15 Exercise

Prove the following, using Definition 4.2.12 and Proposition 4.2.6.

(a) \( A \leftrightarrow B \) is true in \( S \) iff \( A \) and \( B \) have the same truth-value in \( S \).

(b) \( A \lor B \) 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 \( d \).

4.3 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 7 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

\textbf{Snow is white}

is true. But the sentence

\textbf{If snow is white, snow is white}

is not only true, but \textit{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 \textit{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 \textit{non-logical} expressions. But this raises the difficult question of what is supposed to count as a \textit{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 \textit{structures}. That means we are only looking at reinterpretations of the basic symbols in the \textit{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 \textit{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: \textit{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.

### 4.3.1 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.

### 4.3.2 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**.

(c) $A$ is a **logical consequence** of $X$ (for short, $X \models A$) iff $A$ is true in every model of $X$.

### 4.3.3 Example
Consider a signature with one constant $c$, and one one-place function symbol $f$.

This set of sentences is consistent:

$$\{ \forall x \neg(f \; x = x), \; (f \; c = c) \}$$

**Proof**
We can show this by explicitly providing a model, like the one in Fig. 4.1.

The domain of this structure $S$ has two elements—for concreteness, say the domain is $\{0, 1\}$. The value of $c_S$ (the extension of the constant $c$) is 0, and $f_S$ 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.

4.3.4 Example

This set of sentences is inconsistent:

\[
\{ \forall x \neg(f \ x = x), \ (f(f\ c) = f\ c) \}
\]

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, \( \forall x \neg(f \ x = x) \) is true in \( S \). This means that for each \( d \) in the domain of \( S \), \( \neg(f \ x = x) \) is true of \( d \) in \( S \). In particular, let \( d \) be the element that is denoted by \( f\ c \) in \( S \). Since \( \neg(f \ x = x) \) is true of \( d \) in \( S \), and \( f\ c \) denotes \( d \), it follows (from the Satisfaction Lemma Lemma 4.2.10) that \( \neg(f(f\ c) = f\ c) \) 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 \( \forall x \neg(f \ x = x) \) and \( f(f\ c) = f\ c \).

4.3.5 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) \( \{ \exists x (f \ x = c), \ \exists x \neg(f \ x = c) \} \)

(b) \( \{ f \ c = c, \ \forall x \neg(f \ x = c) \} \)

(c) \( \{ \forall x \neg(f \ c = x) \} \)
4.3. LOGIC

4.3.6 Notation
When we use the “turnstile” notation $X \vdash A$ 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—

$$\{ A, B \} \vdash C \quad X \cup \{ A \} \vdash B \quad X \cup Y \cup \{ A, B \} \vdash C \quad \emptyset \vdash A$$

—we’ll usually write these simplified versions:

$$A, B \vdash C \quad X, A \vdash B \quad X, Y, A, B \vdash C \quad \vdash A$$

(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.)

4.3.7 Example
Let $X$ be a set of sentences and let $A$ be a sentence. $X \vdash A$ 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 $X \not\vdash A$ means. 

4.3.8 Exercise
(a) $\{ \bot \}$ is inconsistent.

(b) $X$ is inconsistent iff $X \vdash \bot$.

(c) $A$ is a logical truth iff $\vdash A$.

4.3.9 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) Identity

$$A \vdash A$$
(b) **Weakening**

If $X \models A$ then $X, Y \models A$

(c) **Conjunction Introduction**

$$A, B \models A \land B$$

(d) **Modus Ponens**

$$A, A \rightarrow B \models B$$

**Proof of (a)**

We want to show that $A$ is true in every model of $\{A\}$. This is obvious: if $S$ is a model of $\{A\}$, that means that $A$ is true in $S$ (since obviously $A$ is an element of $\{A\}$). So we’re done. □

**Proof of (b)**

Suppose that $X \models A$: that is, $A$ is true in every model of $X$. We want to show that $X, Y \models A$: that is, $A$ 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 $S$ is a model of $\{A, B\}$. Then $A$ is true in $S$ and $B$ is true in $S$. By Proposition 4.2.6, this means that $A \land B$ is true in $S$. So $A \land B$ is true in every model of $\{A, B\}$, which is what we wanted to show. □

**Proof of (d)**

Suppose that $S$ is a model of $\{A, A \rightarrow B\}$: that is, $A$ is true in $S$, and $A \rightarrow B$ is true in $S$. By Exercise 4.2.15, 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 $\{A, A \rightarrow B\}$. □

**4.3.10 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.
4.3. **LOGIC**

(a) **Cut**

\[ \text{If } X \models A \text{ and } Y, A \models B \text{ then } X, Y \models B \]

(b) **Conjunction Elimination**

\[ A \land B \models A \text{ and } A \land B \models B \]

(c) **Double Negation Elimination**

\[ \neg \neg A \models A \]

(d) **Explosion**

\[ A, \neg A \models B \]

(e) **Proof by Contradiction** (Reductio)

\[ \text{If } X, A \models \bot \text{ then } X \models \neg A \]

(f) **Conditional Proof**

\[ \text{If } X, A \models B \text{ then } X \models A \rightarrow B \]

4.3.11 **Exercise**

Let \( X \) and \( Y \) be sets of sentences, and let \( B \) be a sentence. Suppose:

\[ X \models A \text{ for each sentence } A \in Y \]

\[ Y \models B \]

Then \( X \models B \).

4.3.12 **Example (Leibniz’s Law)**

Let \( a \) and \( b \) be terms, and let \( A(x) \) be a formula of one variable.

\[ a = b, \ A(a) \models A(b) \]

**Proof**

Let \( S \) be any model of \( \{ a = b, A(a) \} \). That is, \( a = b \) is true in \( S \), and \( A(a) \) is true in \( S \). The first part tells us that \( \llbracket a \rrbracket_S = \llbracket b \rrbracket_S \). The second part tells us, using the Satisfaction Lemma, Lemma 4.2.10, that \( \llbracket a \rrbracket_S \) satisfies \( A(x) \) in \( S \). Thus \( \llbracket b \rrbracket_S \) satisfies \( A(x) \) in \( S \), and so (using the Satisfaction Lemma a second time) \( A(b) \) is true in \( S \). \( \square \)
4.3.13 Exercise
Let $a$ be a closed term, and let $A(x)$ be a formula of one variable.

(a) **Reflexive Property**

\[
\models a = a
\]

(b) **Universal Instantiation.**

\[
\forall x \ A(x) \models A(a)
\]

4.3.14 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,

\[
a \equiv b \iff \llbracket a \rrbracket_S = \llbracket b \rrbracket_S \text{ for every structure } S
\]

(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,

\[
A \equiv B \iff \llbracket A \rrbracket_S = \llbracket B \rrbracket_S \text{ for every structure } S
\]

(c) $a$ and $b$ are **logically equivalent given $X$** (abbreviated $a \equiv^X b$) iff $a$ and $b$ denote the same thing in every model of $X$.

(d) $A$ and $B$ are **logically equivalent given $X$** (abbreviated $A \equiv^X B$) 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 $A \equiv^\emptyset B$, since every structure is trivially a model of $\emptyset$. 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.

4.3.15 Exercise
(a) Show:

\[
a \equiv^X b \iff X \models a \equiv^X b
\]
(b) Show:

\[ A \equiv \frac{X}{X} B \text{ iff } X \models A \equiv B \]

4.3.16 Exercise
(a) Show that if \( X \models A \equiv B \), then \( X \models A \text{ iff } X \models B \).
(b) Is the converse true? That is, suppose \( X \models A \text{ iff } X \not\models B \). Does it follow that \( X \not\models A \equiv B \)?

4.3.17 Exercise
(a) The relation \( \equiv \) 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 \( a \equiv c \).

4.3.18 Exercise
The following are equivalent:

\[ X \models A \]
\[ \frac{A}{X} \]
\[ A \Rightarrow B \equiv \frac{X}{X} B \text{ for every sentence } B \]

4.3.19 Exercise
\[ A \equiv B \text{ iff } \frac{X}{X} \]
\[ \neg A \equiv \frac{X}{X} B \]
4.3.20 Exercise
The following are equivalent:

- $X$ is inconsistent
- $A \equiv B$ for all sentences $A$ and $B$
- $A \equiv \neg A$ for some sentence $A$
- $\top \equiv \bot$

4.3.21 Exercise
Let $X$ be a set of sentences, let $A(x)$ be a formula of one variable, and let $a$ and $b$ be terms.

If $a \equiv b$ then $A(a) \equiv A(b)$

4.3.22 Exercise
$A \equiv (B \rightarrow C)$ and $(A \land B) \rightarrow C$ are logically equivalent, for any sentences $A$, $B$, and $C$.

4.3.23 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.
4.3.24 Proposition (Universal Instantiation)
For any formula $A$ and any term $a$,

$$\forall x A \models A[x \mapsto a]$$

4.3.25 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 $X \models A$, then $X \models \forall x A$.

*Hint.* Let $S$ be a structure and $g$ be an assignment. If $X \models A$, 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”.

4.4 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 6.7 and Section 7.5).

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.

4.4.1 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,

$$T = \{ A \mid X \models A \}$$

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.

4.4.2 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.)

\[
\begin{align*}
\emptyset \neq \text{suc } x \\
\text{suc } x = \text{suc } y \rightarrow x = y \\
x + \emptyset = x \\
x + \text{suc } y = \text{suc } (x + y) \\
x \cdot \emptyset = \emptyset \\
x \cdot \text{suc } y = (x \cdot y) + x \\
\neg (x < \emptyset) \\
x < \text{suc } y \iff (x < y \lor x = y) \\
x < y \lor x = y \lor y < x \\
x = \emptyset \lor \exists y (x = \text{suc } y)
\end{align*}
\]

(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-than, respectively. The last two axioms give us a kind of exhaustiveness condition. (But they are not nearly as powerful as our full-fledged exhaustiveness condition, the Inductive Property of numbers.)

4.4.3 Definition
The minimal theory of strings, or S for short, has the following axioms.

First, we have axioms corresponding to the Injective Property of strings. Remember that the language of strings includes a constant for the singleton string of each symbol in the alphabet. These are called the “singleton constants”. For each singleton constant \(c\), we have these axioms:

\[
\begin{align*}
c \cdot x \neq \"\" \\
c \cdot x = c \cdot y \rightarrow x = y
\end{align*}
\]

For each pair of distinct singleton constants \(c_1\) and \(c_2\), we have an axiom of this form:
Next, we have some axioms which capture the recursive definition of the “join” function. For the base case:

\[ c_1 \oplus x \neq c_2 \oplus x \]

For the recursive step, for each singleton constant \( c \):

\[ c \oplus x = c \oplus x \]

We also have a “special case” axiom for each singleton string:

\[ c = c \oplus "x" \]

Next, we have some axioms for the “no-longer-than” relation \( \preceq \).

\[ "x" \preceq x \]
\[ x \preceq "x" \iff x = "x" \]
\[ c_1 \oplus x \preceq c_2 \oplus y \iff x \preceq y \]
\[ x \preceq y \lor y \preceq x \]

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. Let \( c_1, \ldots, c_n \) be all of the singleton constants in the language of strings.

\[ x = "x" \lor \exists y \ (x = c_1 \oplus y \lor \cdots \lor x = c_n \oplus y) \]

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,\(^3\) at the same time they also turn out to be strong enough to

---

\(^3\)The list of axioms for \( S \) is not short: because we are using such an extravagantly large alphabet with over 120,000 basic symbols, the full list of axioms would take somewhere around 15 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.
represent lots of interesting structure. They will be important players in Chapter 5 and Chapter 6.

4.4.4 Exercise
The following are equivalent:

(a) \( T \) is a theory.
(b) \( T \) axiomatizes \( T \).
(c) For each sentence \( A \), if \( T \models A \) then \( A \in T \).
(d) For each sentence \( A \), \( A \in T \) iff \( T \models A \).

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.

4.4.5 Definition
A theory \( T \) is finitely axiomatizable iff there is some finite set of sentences \( X \) that axiomatizes \( T \).

4.4.6 Example
The minimal theory of arithmetic \( \mathbb{Q} \) and the minimal theory of strings \( \mathcal{S} \) are each finitely axiomatizable.

4.4.7 Exercise
Suppose that \( T \) is a finitely axiomatizable theory with axioms \( A_1, \ldots, A_n \). Then for any sentence \( B \), \( B \) is a theorem of \( T \) iff

\[
(A_1 \land \cdots \land A_n) \rightarrow B
\]

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:

4.4.8 Definition
First-order Peano arithmetic \( \text{PA} \) is the theory with the following axioms. The
first part:

\[
\begin{align*}
\text{suc } x & \neq 0 \\
\text{suc } x = \text{suc } y & \rightarrow x = y \\
x + 0 & = x \\
x + \text{suc } y & = \text{suc } (x + y) \\
x \cdot 0 & = 0 \\
x \cdot \text{suc } y & = (x \cdot y) + x \\
\neg (x < 0) & \\
x < \text{suc } y & \leftrightarrow (x < y \lor x = y)
\end{align*}
\]

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:

\[
A(0) \land \forall x (A(x) \rightarrow A(\text{suc } x)) \rightarrow \forall x \ A(x)
\]

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 6.

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.)

### 4.4.9 Definition

The **first-order language of pure set theory** is a first-order language with just one relation symbol \(\in\). **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 \( z \subseteq x \) as an abbreviation for \( \forall w (w \in z \rightarrow w \in x) \).

(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.)

\[
\begin{align*}
\text{# Extensionality} & \quad \forall z (z \in x \leftrightarrow z \in y) \rightarrow x = y \\
\text{# Separation} & \quad \exists y \forall z (z \in y \leftrightarrow (z \in x \land A)) \\
\text{# Power Set} & \quad \exists y \forall z (z \in y \leftrightarrow z \subseteq x) \\
\text{# Union} & \quad \exists y \forall z (z \in y \leftrightarrow \exists w (w \in x \land z \in w)) \\
\text{# Choice} & \quad \forall y (y \in x \rightarrow \exists z (z \in y)) \rightarrow \\
& \quad \exists w \forall y (y \in x \rightarrow \exists! z (z \in y \land z \in w)) \\
\text{# Infinity} & \quad \exists x (\exists y (y \in x) \land \\
& \quad \forall y (y \in x \rightarrow \exists z (z \in x \land y \subseteq z \land y \neq z))) \\
\text{# Foundation} & \quad \exists y (y \in x) \rightarrow \exists y (y \in x \land \neg \exists z (z \in y \land z \in x)) \\
\text{# Replacement} & \quad \forall y (y \in x \rightarrow \exists! z A) \rightarrow \exists w \forall y (y \in x \rightarrow \exists z (z \in w \land A))
\end{align*}
\]

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.
4.4.10 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 $\text{Th } S$ for short.

4.4.11 Example
(a) The **first-order theory of arithmetic** is $\text{Th } \mathbb{N}$, the set of all sentences that are true in the standard model of arithmetic.

(b) The **first-order theory of strings** is $\text{Th } \mathbb{S}$, 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. Is the theory of a structure really a theory in the sense of **Definition 4.4.1**—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.

4.4.12 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).

4.4.13 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 $X = \text{Th } S$.

4.4.14 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.
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 Th ℤ.) 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 7.5. But here’s a preliminary result.

4.4.15 Exercise

If \( S \) is a finite structure, then \( \text{Th } S \) is finitely axiomatizable.

*Note.* This exercise is harder than most.

4.5 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.

4.5.1 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:

\[
\begin{align*}
\text{A is a formula} & \quad x \text{ is a variable} \\
\text{the } x & \quad A \text{ is a term} \\
\hline
\text{Definite Description}
\end{align*}
\]

The definite description language is called Def \( L \), for short.
(Definite descriptions are sometimes instead written $\xi x A$ 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.

4.5.2 Definition

(a) If $x$ is a distinct variable from $y$, then $x$ is free in $\text{the } y A$ iff $x$ is free in $A$.

(b) The variable $y$ is not free in $\text{the } y A$.

The rest of the clauses of the definition of free variables are exactly the same as for ordinary terms and first-order formulas (Definition 3.6.3 and Definition 4.1.5). (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 3.6.7 and Definition 4.1.8).

4.5.3 Definition

(a) If $x$ is distinct from $y$, then the substitution instance

$$(\text{the } y A)[x \mapsto t]$$

is $\text{the } y (A[x \mapsto t])$.

(b) The substitution instance $(\text{the } y A)[y \mapsto t]$ is just $\text{the } y A$ 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

$\text{the } x ("A" + x = "ABC")$
should denote \( BC \) in the standard string structure: this is the only thing that satisfies the formula \("A" + x = "ABC"\) in \( S \). More generally, in the context of a structure where exactly one thing that satisfies \( A(x) \), then the definite description \( \text{the } x \mid A(x) \) 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

\[ "A" \oplus x = "B" \]

So there is no obvious denotation for \( \text{the } x \ ("A" \oplus x = "B") \). The most natural thing to say is that this term just doesn’t denote anything at all in \( S \). It is an empty term. But this is a perfectly well-formed sentence in the definite description language:

\[ \text{the } x \ ("A" \oplus x = "B") \preceq "C" \]

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?

\[ \neg(\text{the } x \ ("A" \oplus x = "B") \preceq "C") \]

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.)

### 4.5.4 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 \( \text{the } x \mid A \) 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 \( \text{the } x \ A \) doesn’t denote anything (for $S, g$).

\[ \text{the } x \ A \models_{S, g} d \iff d \text{ is the unique element of } D_S \text{ such that } g[x \mapsto d] \text{ satisfies } A \text{ in } S \]

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 $f, fa$ doesn’t denote anything either (for $S, g$). (And similarly for $n$-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

\[ s = \text{the } x \ W(x) \]

to a sentence of the form

\[ \exists! x \ W(x) \land W(s) \]
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:

\[ \forall x \ (W(x) \leftrightarrow s = x) \]

“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.

**4.5.5 Exercise**

For any formula \( B \),

\[ (x = \text{the } y \ B) \equiv \forall y \ (B = x = y) \]

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

\[ \text{the } y \ ("A" \oplus y = "B") \leq "C" \]

is equivalent to

\[ \exists x \ (x = \text{the } y \ ("A" \oplus y = "B") \land x \leq "C" ) \]

More generally, suppose that \( A \) is an atomic formula which contains at most \( n + 1 \) occurrences of \( \text{the} \). Then \( A \) is equivalent to a formula of the form

\[ \exists x \ (x = \text{the } y \ B \land C) \]

where \( B \) and \( C \) each contain fewer occurrences of \( \text{the} \) than \( A \)—that is, they each have at most \( n \) occurrences of \( \text{the} \). (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

\[ \exists x \ (x = \text{the } y \ B^* \land C^* ) \]

Furthermore, by applying Exercise 4.5.5, this is equivalent to

\[ \exists x \ (\forall y \ (B^* \leftrightarrow x = y) \land C^* ) \]

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 \( \land \) 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
4.5. DEFINITE DESCRIPTIONS

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:

4.5.6 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 \( \lor \) as abbreviations for formulas just containing \( \neg \) and \( \land \). 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 4.5.5.

4.5.7 Exercise

For every definite description term \( t(x) \), there is an ordinary first-order formula \( A(x, y) \) such that

\[
(t(x) = y) \equiv \forall z (A(x, z) \leftrightarrow z = y)
\]

(4.1)

(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 (4.1) holds.
Chapter 5

The Inexpressible

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?

5.1 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, if you prefer, \( 2 \cdot x \), 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
CHAPTER 5. THE INEXPRESSIBLE

\[ \exists y \ (y + y = x) \]

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:

\[ \forall y \ (\exists z \ (y \cdot z = x) \rightarrow (y = 1 \lor y = x)) \]

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 \( y \)”: \[ \exists z \ (y \cdot z = x) \]

Let’s abbreviate this formula \( \text{Div}(x, y) \).

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 \( \text{Div}(x, y) \) for divisibility):

\[ \text{the} \ z \ (\text{Div}(x, z) \land \text{Div}(y, z) \land \forall z' \ ((\text{Div}(x, z') \land \text{Div}(y, z')) \rightarrow z' \leq z)) \]

(“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 singleton (length one) strings is definable in \( \mathbb{S} \) using this formula:

\[ x \leq "\cdot" \land x \neq "" \]

This says that \( x \) is no longer than a one-symbol string \( \bullet \), but not empty. Abbreviate this formula \( \text{Singleton}(x) \). We can use this to define the function that takes each
5.1. DEFINABLE SETS AND FUNCTIONS

non-empty string to its first symbol. We can use this definite description (with a free variable $x$):

\[
\text{the } y \ (\text{Singleton}(y) \land \exists z \ (x = y \oplus z))
\]

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 $\text{head}(x)$. Note that if we plug the empty string in for $x$ in this term, the resulting term $\text{head}(\text{""})$ 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 $\text{head}(\text{""})$ 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.

5.1.1 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 $d \in D$,

\[
A(x) \text{ is true of } d \text{ in } S \iff d \in X
\]

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 $(d_1, d_2) \in D \times D$,

\[
A(x, y) \text{ is true of } (d_1, d_2) \text{ in } S \iff (d_1, d_2) \in X
\]

The same goes for sets of $n$-tuples of elements of $D$, for any number $n$.

A term $t(x)$ defines a (partial) function $f : D \rightarrow D$ in $S$ iff

\[
\llbracket t \rrbracket_S(d) = f d \quad \text{for each element } d \text{ in the domain of } f
\]

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 $\llbracket A \rrbracket_S(d)$ is True if $A(x)$ is true of $d$ in $S$, and False otherwise. Similarly, recall that
the characteristic function \( \text{char } X \) is the function that takes \( d \) to True if \( d \in X \), 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 5.5.) 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.

5.1.2 Exercise
Show that in the standard string structure \( S \), 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 2.5.3.

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.

5.1.3 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 \( S \):

\[
\forall z \ (A(x, z) \leftrightarrow y = z)
\]
5.1. **DEFINABLE SETS AND FUNCTIONS**

(where \( z \) is a distinct variable from \( x \) and \( y \)).

*Hint.* We proved a useful fact in Section 4.5.

5.1.4 **Exercise**

Recall that the *graph* of a function \( f \) is the set of pairs \((d, fd)\) 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 \( f \).

5.1.5 **Definition**

A set of numbers \( X \) is **arithmetically definable** iff \( X \) is definable in the standard model of arithmetic, \( \mathbb{N}(0, \text{succ}, +, \cdot) \). Similarly for sets of tuples of numbers and functions.

5.1.6 **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.

5.1.7 **Exercise**

Suppose that \( S \) is an infinite structure. Show that infinitely many subsets of the domain of \( S \) are undefinable.

5.1.8 **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 \( 0, \text{succ } 0, \text{succ succ } 0 \), 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:

\[
\langle 0 \rangle = 0 \\
\langle \text{suc } n \rangle = \text{suc } \langle n \rangle \quad \text{for every } n \in \mathbb{N}
\]

(b) Similarly, in Exercise 3.2.13 we showed that every string has a label in the standard string structure $\mathcal{S}$. Here’s one standard way of doing it. The string $\text{ABC}$ is the same as

\[\text{A } \oplus \text{B } \oplus \text{C } \oplus ()\]

which is built up by joining together singleton strings and the empty string. So we can label this string with the term

\[\text{("A" } \oplus \text{("B" } \oplus \text{("C" } \oplus \text{""}))}\]

in the language of strings. We call this the canonical label (or quotation name) for $\text{ABC}$. 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).

\[
\langle () \rangle = \text{""} \\
\langle (a : s) \rangle = \{c \oplus \langle s \rangle\} \quad \text{where } c \text{ is the constant for the symbol } a
\]

(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).

5.1.9 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 $A(d)$. Similarly, instead of $A(\langle d_1 \rangle, \langle d_2 \rangle)$ I’ll write the simplified version $A(d_1, d_2)$. I’ll try to put the parentheses back in when it would otherwise be confusing what something means.

5.1.10 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
5.2. STRING REPRESENTATIONS

each element \( d \) in the domain of \( f \),

\[
t(d) \equiv _{\text{Th } S} (f d)
\]

5.1.11 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 \( S \),

If \( d \in X \) then \( \text{Th } S \models A(d) \)
If \( d \not\in X \) then \( \text{Th } S \models \neg A(d) \)

5.1.12 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

\[
\langle d \in X \rangle = \begin{cases} 
\top & \text{if } d \in X \\
\bot & \text{if } d \not\in X 
\end{cases}
\]

(where \( \top \) is the standard truth and \( \bot \) is the standard falsehood). Then we can rewrite the conclusion of the previous exercise like this:

\[
A(d) \equiv _{\text{Th } S} \langle d \in X \rangle
\]

Exercise 5.1.10 and Exercise 5.1.11 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 \( A(d) \) 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 5.5.

5.2 String Representations

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 them to represent numbers, or formulas and terms, or computer programs, or
many other things. Once we have chosen a way of representing some things using
strings, we can then consider whether operations on those other things—operations
like addition for numbers, or substitution for formulas and terms—are definable in
the string structure.

Let’s start with the case of numbers. There are many different notation systems
for numbers, such as Arabic numerals, Roman numerals, binary code, and so on.
We’ll use an especially simply “tally” notation, called unary notation. We simply
represent the number one with a single dot •, two with two dots ••, three with
••• and so on. In this system zero is represented by the empty string.

5.2.1 Definition
The (unary) string representation for a number \( n \) is given by the following recur-
sive definition.

\[
\begin{align*}
\text{rep } 0 &= () \\
\text{rep}(n + 1) &= \cdot \oplus \text{rep } n
\end{align*}
\]

5.2.2 Exercise
The string representation function for numbers is one-to-one. That is, if \( \text{rep } m = \text{rep } n \), then \( m = n \).

5.2.3 Exercise
Consider the (partial) function that takes the string representation for a number
\( m \) and the string representation for a number \( n \) to the string representation for
\( m + n \). Show that this function is definable in \( S \).

5.2.4 Exercise
The range of \( \text{rep} \), which is the set of all strings that are string representations for
numbers, is definable in \( S \).

5.2.5 Definition
(a) If \( f : \mathbb{N} \to \mathbb{N} \) is a (partial) function from numbers to numbers, we say that
\( f \) is definable in \( S \) iff the partial function that takes the string representation
for a number \( n \) to the string representation for \( fn \) is definable in \( S \).

(b) If \( X \subseteq \mathbb{N} \) is a set of numbers, we say \( X \) is definable in \( S \) iff the set of string
representations for elements of \( X \) is definable in \( S \).

The definition is similar for \( n \)-place functions and relations.
We can generalize this idea. Once we have picked a notation for describing some objects—numbers, or formulas, or computer programs, or whatever—we can then talk about definability for those objects in terms of the string structure.

5.2.6 Definition
Let $\text{rep} : D \to \mathbb{S}$ be a string representation function for a domain of objects $D$. We assume that $\text{rep}$ is one-to-one.

(a) For any (partial) function $f : D \to D$, we say that $f$ is definable in $\mathbb{S}$ (with respect to $\text{rep}$) iff the partial function that takes the string representation for any element $d \in D$ to the string representation for $f(d)$ is definable in $\mathbb{S}$. In other words, $f$ is definable with respect to $\text{rep}$ iff the function

$$\text{rep } d \mapsto \text{rep } f(d)$$

is definable in $\mathbb{S}$.

(b) If $X \subseteq D$ is a subset of the domain $D$, we say $X$ is definable in $\mathbb{S}$ (with respect to $\text{rep}$) iff the set of string representations for elements of $X$ is definable in $\mathbb{S}$. In other words, $X$ is definable with respect to $\text{rep}$ iff the set

$$\{ \text{rep } d \mid d \in X \}$$

is definable in $\mathbb{S}$.

The definition is similar for $n$-place functions and relations.

Something that will be particularly useful later on is to represent a sequence of strings using a single string. If we choose these string representations well, then we can show that important features of sequences are definable in the standard string structure $\mathbb{S}$.

How should we represent a finite sequence of strings using a single string? One natural thought is to simply stick all the strings together, one after another. But this won’t quite work. Consider the two sequences $(A, BC)$ and $(AB, C)$. If we just stick the strings together end-to-end, both sequences would be represented by the string $ABC$. But we want our string representations to be unique; that is, the representation function $\text{rep}$ should be one-to-one.

There are many ways to solve this problem. We want to choose a way that is well-suited to being described in our minimal string-language. If we have the joined
together string $\text{ABC}$, what other information do we need to help us decide whether this represents the sequence $(\text{A}, \text{BC})$ or $(\text{AB}, \text{C})$? We need to know how to split the string up into pieces.

One idea is to add a “delimiter” symbol. Instead of just using the string $\text{ABC}$, which is ambiguous between the two sequences $(\text{A}, \text{BC})$ and $(\text{AB}, \text{C})$, we can use the strings $|\text{A}|\text{BC}$ and $|\text{AB}|\text{C}$ to represent them: we add a $|$ symbol in each place where a new element of the sequence of strings begins. But this won’t quite work. The problem is that the delimiting symbol $|$ might itself show up in our original strings. For example, consider how we would represent the two different sequences $(\text{A}|, \text{B})$ and $(\text{A}, |\text{B})$. Both of these two sequences would end up ambiguously represented by the string $|\text{A}||\text{B}$.

Here’s a different idea. We need to know where to put the “breaks” between the elements of the sequence: this means we need to know how long each string in the sequence is. So what we can do is use a separate “control” string that just keeps track of the lengths of the strings in the sequence. We have already decided on a way to represent numbers, using strings of dots. We can mark the divisions between these number representations using another symbol, $|$. This won’t be ambiguous, because the delimiter symbol $|$ never shows up in a number representation. For example, the control string for the sequence $(\text{AB}, \text{C})$ is $|••|•$, and the control string for $(\text{A}, \text{BC})$ is $|•|••$.

It turns out that the most convenient way to do things combines both of these two ideas. To represent the sequence $(\text{A}, \text{BC})$, we’ll use both the “delimited content” string $|\text{A}|\text{BC}$ and also the “control” string $|•|••$. (This is more convenient than using just the joined-up content string without delimiters, because this way the two strings have the same length, and we can more easily “line them up”.) Our string representation for the sequence will just join these two strings up into a single string.

Here are some examples of the string representations for some sequences of strings:

- $(\text{A}, \text{BC}) \mapsto |•|••|\text{A}|\text{BC}$
- $(\text{AB}, \text{C}) \mapsto |••|•|\text{A}|\text{BC}$
- $(\text{A}, |\text{BC}) \mapsto |••|••|\text{A}||\text{B}$
- $(\text{A}, |\text{BC}) \mapsto |•|••|\text{A}||\text{BC}$
- $(|, (), •, |) \mapsto |•|•|•|•|•|•|•|

5.2.7 Definition

For any string $s$, let

$\text{delimit } s = | \oplus s$
5.2. STRING REPRESENTATIONS

The **string representation** of any sequence of strings \( s = (s_1, \ldots, s_n) \) is the string

\[
\text{delimit}(\text{dots } s_1) \oplus \cdots \oplus (\text{dots } s_n) \oplus \text{delimit } s_1 \oplus \cdots \oplus s_n
\]

We say that this string **represents** the sequence \( s \), and we call it \( \text{rep } s \).

(We could spell this definition out a bit more explicitly using a recursive definition, but we won’t bother.)

5.2.8 Exercise

The function that takes each sequence of strings to its string representation is one-to-one. Thus, if \( s \) is a string representation for a sequence, we can talk about the sequence represented by \( s \).

5.2.9 Exercise

The set of all string representations for sequences of strings is definable in \( \mathcal{S} \).

*Hint*. A string represents a sequence iff it can be split into two parts \( x \) and \( y \) such that (i) \( x \) and \( y \) both have the same length, (ii) \( x \) consists entirely of \( \text{•} \) and \( | \) (and does not begin with a \( \text{•} \)), and (iii) for each \( n \), if the \( n \)th symbol of \( x \) is \( | \), then the \( n \)th symbol of \( y \) is also \( | \). With a bit of trickery, the third part can be restated in terms of same-length substrings, instead of referring to a number \( n \).

5.2.10 Exercise

The following sets and functions are definable in \( \mathcal{S} \). (It will be helpful to use facts from Exercise 5.1.2.)

(a) The join function for sequences of strings. That is, the partial function that takes two strings \( s \) and \( t \) that represent sequences \( x \) and \( y \) to the string representation for \( x \oplus y \) is definable.

(b) The function that takes a string \( s \) to the string representation for the length-one sequence \( (s) \).

(c) The set of pairs \( (s, t) \) where \( t \) is the string representation for a sequence that has the string \( s \) as an element.

(d) The function that takes a string that represents a non-empty sequence of strings \( (s_1, s_2, \ldots, s_n) \) to its first element \( s_1 \).
(e) The function that takes a string that represents a sequence of strings 
\((s_1, s_2, \ldots, s_n)\) with at least two elements to its second element \(s_2\).

We can use these operations on sequences to do something cool: we can effectively 
give recursive definitions within the first-order language of strings. Let’s start by 
looking at the case of numbers. Remember how recursive definitions work in gen-
eral (Section 2.3): we pick a starting place \(z \in D\), and we pick a way of stepping 
from one value to the next, \(s : D \to D\). Then we know that there is a unique 
function \(f : \mathbb{N} \to D\) such that

\[
\begin{align*}
f0 &= z \\
f(n + 1) &= s(fn)
\end{align*}
\]

The fact that a function like this exists (and is unique) was called the Recursion 
Theorem. The way we proved the Recursion Theorem was by building up \(f\) from 
“special” partial functions, like the function

\[
[0 \mapsto z, 1 \mapsto sz, 2 \mapsto s(sz), 3 \mapsto s(s(sz))]
\]

The basic idea of the proof of the Recursion Theorem was that we eventually reach 
a value for every number by building up bigger and bigger partial functions like 
this.

We can also think of a “special” partial functions as just a finite sequence of pairs— 
and this means we can represent special partial functions with strings. For any 
number \(n\), there is a “step sequence” which lists all of the pairs \((k, f k)\) for \(k\) running 
from 0 up to \(n\):

\[
((0, f0), (1, f1), (2, f2), \ldots, (n, fn))
\]

Furthermore, if \(f\) is defined recursively, then we can describe this “step sequence” 
with simple rules.

5.2.11 Exercise

Say a finite sequence \(x\) is special (for \(z\) and \(s\)) iff it has the following properties:

(a) Every element of \(x\) is an ordered pair.
(b) No two elements of \(x\) have the same first element.
(c) The pair \((0, z)\) is an element of \(x\).
(d) For every element \((n + 1, a)\) in \(x\), there is an element \((n, b)\) in \(x\) such that 
\(a = sb\).

Suppose that \(f : \mathbb{N} \to \mathbb{S}\) is a recursively defined function: \(f0 = z\) and \(f(n+1) = \)
5.2. STRING REPRESENTATIONS

$s(fn)$ for each number $n$. Then for each number $n$ and string $a$, $fn = a$ iff the ordered pair $(n, a)$ is an element of some special sequence.

Furthermore, if the step function $s$ is definable, then we can translate this description of “special sequences” into the language of strings, and thereby show that the recursively defined function is in fact definable in the string structure $\mathbb{S}$.

5.2.12 Exercise

Let $z \in \mathbb{S}$, and suppose that $s : \mathbb{S} \to \mathbb{S}$ is definable in $\mathbb{S}$. By the Recursion Theorem (Principle 2.3.1), there is a unique function $f : \mathbb{N} \to \mathbb{S}$ such that

$$f 0 = z$$
$$f (n + 1) = s(fn) \quad \text{for each number } n$$

Then this recursively defined function $f$ is definable in $\mathbb{S}$. That is, more precisely, the partial function that takes the string representation of each number $n$ to the string $fn$ is definable in $\mathbb{S}$.

We can use the same “special sequence” trick for functions that are defined recursively on strings, or formulas, or any other inductively defined structure.

5.2.13 Exercise

Let $A^1$ be the set of singleton strings. Let $e \in \mathbb{S}$, and let $c : A^1 \times \mathbb{S} \to \mathbb{S}$ be a function which is definable in $\mathbb{S}$. By the Recursion Theorem for Sequences (Principle 2.4.15), there is a unique function $f : \mathbb{S} \to \mathbb{S}$ such that

$$f () = e$$
$$f (a \oplus s) = c(a, fs) \quad \text{for each singleton string } a \text{ and string } s$$

Then this recursively defined function $f$ is definable in $\mathbb{S}$.

5.2.14 Exercise

The following functions are definable in $\mathbb{S}$.

(a) The function that takes a number $i$ and a sequence of strings $(s_1, s_2, \ldots, s_n)$ to the $i$th element $s_i$ (if there is one).

(b) The length function for sequences of strings. That is, the partial function that takes each string representation for a sequence of strings to the string representation for its length.
5.2.15 Exercise
The function that takes each string \( s \in \mathcal{S} \) to its canonical label \( \langle s \rangle \) in the language of strings (Definition 5.1.8) is definable in \( \mathcal{S} \).

5.3 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 finite strings of symbols. These are all elements of the domain of \( \mathcal{S} \), the standard string structure. So the string structure \( \mathcal{S} \) 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.

5.3.1 Exercise
The set of variables is definable in \( \mathcal{S} \). (Remember that officially each variable is the symbol \( x, y, \) or \( z \) followed by some finite sequence of subscripted numerals \( 0, 1, 2 \), etc.)

In fact, we can do much more than this. It turns out that many syntactic operations are definable in \( \mathcal{S} \). For example:

5.3.2 Lemma
Let \( L \) be a finite signature. The following sets are definable in \( \mathcal{S} \).

(a) The set of \( L \)-terms.
(b) The set of \( L \)-formulas.
5.3. REPRESENTING LANGUAGE

5.3.3 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 \( S \). That is to say, there is a term \( \text{sub}(x, y) \) in the language of strings (with definite descriptions), such that

\[ \llbracket \text{sub} \rrbracket_S(A(x), b) = A(b) \]

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 \( S \).)

The proofs of Lemma 5.3.2 and Lemma 5.3.3 are kind of tricky: our definition of substitution is recursive, and we don’t have any direct way to write out recursive definitions in first-order logic. But in Exercise 5.2.12 we discussed an indirect way to do this. We could generalize that idea here (from numbers and strings to terms and formulas), but instead we’ll postpone the proof of Lemma 5.3.3 until Chapter 6. At that point we’ll have the resources to prove something much more general about the expressive power of \( S \), which has Lemma 5.3.3 as a special case. In fact, any operation on strings which can be systematically worked out step by step is definable in \( S \). (This is called the Definability Theorem (Exercise 6.7.5).) Since the substitution function is systematic in this way, the fact that it is definable in \( S \) 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 5.1.8). 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 \( S \)). 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 5.1.10, 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 \( S \):

\[ \text{sub}(A(x))\langle b \rangle = \langle A(b) \rangle \]

(Or, equivalently, the ordinary first-order formula that results from eliminating definite descriptions from this sentence is true \( S \).)

We can describe the syntax of any language \( L \) that has finitely many primitive symbols in the first-order theory of the string structure \( S \). In particular, then, we can
apply all of these ideas to the language of strings itself. This language only has finitely many primitives ($\oplus$, $\preccurlyeq$, "", 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 $A$.

5.3.4 Example

Consider the formula $x = x$: in its completely official form, this is $\langle x=x \rangle$. The canonical label for this in the language of strings, $\langle (x=x) \rangle$, is

$$\langle "\langle x \oplus "=\oplus "x \oplus "\rangle \rangle \rangle \rangle \rangle \rangle \rangle \rangle \rangle \rangle \rangle \rangle \rangle \rangle$$

5.3.5 Exercise

Use the definition of the canonical label function $\langle \cdot \rangle$ for strings (from Definition 5.1.8) to explicitly write out each of the following expressions.

(a) $\langle \neg\forall x (x = 0) \rangle$, which is the canonical label (in the language of strings) of the formula $\neg\forall x (x = 0)$ (in the language of arithmetic).

(b) $\langle \langle \langle \rangle \rangle \rangle$, 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 $\langle x = x \rangle$.

(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 $\langle \cdot \rangle$ 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 $\mathbb{S}$:
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:

\[
\langle A \rangle \oplus "" = \langle A \rangle
\]

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.

\[
\forall x (x \oplus "" = x) \\
\forall x (x \oplus \langle()\rangle = x) \\
\forall x (x \oplus \langle""\rangle = x)
\]

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

\[
\forall x (x \oplus (\text{quote} \oplus \text{quote} \oplus "") = x)
\]

This is false in $\mathbb{S}$: for example, the string $\text{ABC""}$ is not the same as the string $\text{ABC}$. 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:

\[
\exists x (x \leq \langle A \rangle \land \neg (\langle A \rangle \leq x))
\]
Chapter 5. The Inexpressible

(“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 \( B(A) \). For example, suppose \( B(x) \) is the formula \( (x = x) \), and \( A \) is the sentence \( (0 = 0) \). Then the substitution instance \( B(A) \) is \( (\langle A \rangle = \langle A \rangle) \). Since \( \langle A \rangle \) is the term

\[
\text{“} (\text{”} \oplus \text{“}0\text{”} \oplus \text{“}=” \oplus \text{“}0\text{”} \oplus \text{“}) \text{”} \oplus \text{“}
\]

the fully spelled out sentence \( B(A) \) is this monstrosity:

\[
\text{“} (\text{”} \oplus \text{“}0\text{”} \oplus \text{“}=” \oplus \text{“}0\text{”} \oplus \text{“}) \text{”} \oplus \text{“}
\]

(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 \( A(x) \) is a formula of one variable, we can substitute the term \( \langle A(x) \rangle \) into the formula \( A(x) \), to get the formula \( A(\langle A(x) \rangle) \). For an example in English, if \( A(x) \) is the formula \( x \text{ is great} \), and \( \langle A(x) \rangle \) is its quotation-name \( ”x \text{ is great”} \), then the substitution instance \( A(\langle A(x) \rangle) \) is the sentence \( ”x \text{ is great”} \text{ is great} \). 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 \( \mathbb{S} \).

5.3.6 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 \( \text{label}(x) \) in the language of strings (with definite descriptions) such that

\[
\llbracket \text{label} \rrbracket_{\mathbb{S}}(A) = \langle A \rangle
\]

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 \( \mathbb{S} \):

\[
\text{label}(A) = \langle \langle A \rangle \rangle
\]

Like Lemma 5.3.3, this is something we could prove now, but it will be more convenient to take it on faith for the time being, since it also follows from the more general Definability Theorem that we will prove in Chapter 6 (Exercise 6.7.4).
5.4. SELF-REFERENCE AND PARADOX

5.3.7 Exercise

Let the application function be the function that takes a pair of formulas $A(x)$ and $B(x)$ to the sentence $A(B(x))$, 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 $\text{apply}(x, y)$ in the language of strings (with definite descriptions) such that,

$$\llbracket\text{apply}\rrbracket(\mathbb{S}, A(x), B(x)) = A(B(x))$$

for any formulas $A(x)$ and $B(x)$. Equivalently, for any formulas $A(x)$ and $B(x)$, the following sentence is true in $\mathbb{S}$:

$$\text{apply}(A(x))(B(x)) = A(B(x))$$

Hint. Use Lemma 5.3.3 and Lemma 5.3.6.

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.

5.4 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 $L = \text{L is not true}$
- $\text{L is not true}$ 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:

- $\text{Snow is white}$ 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 $⟨A⟩$ is a label for $A$, then the schema says:

- $⟨A⟩$ 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 $L = \textit{L is not true}$. 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 1.6**, 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 $A⟨d⟩$ is true, where $⟨d⟩$ 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 $A(B(x))$. As we discussed in the previous section, this syntactic operation is definable in the string structure $S$. There is a term $\text{apply}(x, y)$ in the language of strings such that

$$\llbracket\text{apply}\rrbracket(A(x), B(x)) = A\langle B(x)\rangle$$

in the string structure $S$. In particular, we can consider self-application—which is commonly called diagonalization. Let $\text{diag}(x)$ be the term $\text{apply}(x, x)$. Then

$$\llbracket\text{diag}\rrbracket(A(x)) = A\langle A(x)\rangle$$

Now suppose we also had a formula $\text{True}(x)$ that only applies to the true sentences (in $S$). 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:

$$H\langle H(x)\rangle \text{ is } \neg \text{True}(\text{diag}(H(x)))$$

Furthermore, since the term $\langle H(x)\rangle$ denotes the formula $H(x)$, the term $\text{diag}(H(x))$ denotes $H\langle H(x)\rangle$. So the right-hand-side is true in $S$ iff $\neg \text{True}(x)$ 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 $\neg \text{True}(x)$. 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.

5.4.1 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

$A$ is true in $S$ iff $F(x)$ is true of $A$ in $S$
Or in more concise notation,

\[ \llbracket A \rrbracket_S = \llbracket F \rrbracket_S(A) \]

(Recall that \( \llbracket A \rrbracket_S \) is the truth-value of \( A \) in \( S \).)

(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 \( \text{diag}(x) \). But we don’t really need definite descriptions, because we have Russell’s Elimination Theorem. In particular, the formula \( F(\text{diag}(x)) \), which uses definite descriptions, is logically equivalent to some formula \( H(x) \) without any definite descriptions.)

5.4.2 Exercise (Tarski’s Theorem Version 1)

The set of sentences which are true in the standard string structure \( S \) is not definable in \( S \).

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, preliminary versions of Gödel’s Fixed Point Theorem and Tarski’s Theorem. Next we’ll generalize these ideas.

5.5 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 \( S \). As we’ll see in Section 6.7, 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 \( S \) 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 $\mathcal{S}$, 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 5.1.8). So here is another way of saying that $X$ is definable (using Exercise 5.1.11): there is a formula $A(x)$ such that, for every string $s$,

\[
\begin{align*}
\text{If } & s \in X \quad \text{then} \quad \text{Th } \mathcal{S} \models A(s) \\
\text{If } & s \not\in X \quad \text{then} \quad \text{Th } \mathcal{S} \models \neg A(s)
\end{align*}
\]

Likewise, if a function $f$ is definable in $\mathcal{S}$, this means there is a term $t(x)$ (possibly using definite descriptions) such that for every string $s$ in the domain of $f$,

\[\text{Th } \mathcal{S} \models t(s) = \langle f s \rangle\]

This way of putting things suggests a natural way of generalizing the idea of definability. Instead of using the full theory of strings Th $\mathcal{S}$, we can try to do something similar 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 $s$,

\[
\begin{align*}
\text{If } & s \in X \quad \text{then} \quad T \models A(s) \\
\text{If } & s \not\in X \quad \text{then} \quad T \models \neg A(s)
\end{align*}
\]

Similarly, we say that $T$ represents a function $f$ iff there is a term $t(x)$ such that, for every string $s$,

\[T \models t(s) = \langle f s \rangle\]

Or in other words,

\[t(s) \equiv_f \langle f s \rangle\]

In Section 4.4 we introduced the minimal theory of strings $\mathcal{S}$ (Definition 4.4.3). 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 $\mathcal{S}$. For this example, we just need to recall that $\mathcal{S}$ includes an axiom of this form, where $c$ is the singleton constant for any particular symbol in our alphabet:

\[
\forall x \left( c \odot x \not= "" \right)
\]

5.5.1 Example
The minimal theory of strings $\mathcal{S}$ represents the set of all non-empty strings.
CHAPTER 5. THE INEXPRESSIBLE

Proof
We can use the obvious formula

\[ x \neq "" \]

Call this formula \( A(x) \), and suppose \( s \) is any string. We need to show two things. First:

If \( s \) is non-empty then \( \mathcal{S} \models A(s) \)

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 \( c \bullet \langle t \rangle \), where \( c \) is the singleton constant for \( a \). So \( A(s) \) is the formula

\[ c \bullet \langle t \rangle \neq "" \]

This immediately follows by universal instantiation from this axiom of \( \mathcal{S} \):

\[ \forall x (c \bullet x \neq "") \]

Second:

If \( s \) is empty then \( \mathcal{S} \models \neg A(s) \)

This is true because \( s \) is empty, then the label \( \langle s \rangle \) is the term \( "" \). So \( \neg A(s) \) is the formula \( \neg(\langle "\rangle = \langle "\rangle) \), which is a logical truth—and thus of course it is a logical consequence of \( \mathcal{S} \).

So what we’ve shown is that if \( X \) is the set of non-empty strings, then for any \( s \in X \), the theory \( \mathcal{S} \) implies \( A(s) \), and for any string \( s \notin X \), \( \mathcal{S} \) implies \( \neg A(s) \). This is the sense in which \( \mathcal{S} \) 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 5.1.8 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.
5.5. REPRESENTING SETS AND FUNCTIONS IN A THEORY

5.5.2 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 \( \langle d \rangle \).

5.5.3 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 \( d \in D \),

\[
\begin{align*}
\text{If } & \ d \in X \ \text{ then } \ T \models A\langle d \rangle \\
\text{If } & \ d \notin X \ \text{ then } \ T \models \neg A\langle d \rangle
\end{align*}
\]

(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 ??, we can put this more succinctly:

\[ A\langle d \rangle \equiv_T \langle d \in X \rangle \]

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 \( D \),

\[
\begin{align*}
\text{If } & \ (d_1, d_2) \in X \ \text{ then } \ T \models A\langle d_1 \rangle \langle d_2 \rangle \\
\text{If } & \ (d_1, d_2) \notin X \ \text{ then } \ T \models \neg A\langle d_1 \rangle \langle d_2 \rangle
\end{align*}
\]

Or more succinctly:

\[ A\langle d_1 \rangle \langle d_2 \rangle \equiv_T \langle (d_1, d_2) \in X \rangle \]

A set \( X \) is representable in \( T \) (or \( T \) represents \( X \)) iff there is some \( L \)-formula that represents \( X \) in \( T \).

5.5.4 Exercise
What should the definition be for a representable function from \( D \) to \( D \) (in a theory \( T \) with a labeling for \( D \))?  

5.5.5 Proposition
If \( X \) is any set of strings, then \( X \) is definable in \( \mathfrak{S} \) iff \( X \) is representable in \( \text{Th } \mathfrak{S} \).

Proof
This follows from the definition of a representable set, using Exercise 5.1.11. \( \square \)
5.5.6 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 \( \mathbb{S} \). The term that represents this function is the obvious one: "|" \( \oplus x \). To show that this really does represent the function in question, what we need to show is that for any string \( s \),

\[ "|" \oplus \langle s \rangle = \langle | \oplus s \rangle \]

is a theorem of \( \mathbb{S} \). In fact, the right-hand side \( \langle | \oplus s \rangle \) is defined to be "|" \( \oplus \langle s \rangle \) (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 \( \mathbb{S} \).

5.5.7 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 \( X - Y \).

5.5.8 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 5.3, 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 \( \mathbb{S} \) 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. ?? 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 $\mathbb{S}$. (The officially
stated version of this fact is $\text{#ex:sigma-one-definable}$.) This includes the expres-
sions we use to define syntactic operations. It also includes enough formulas to
define lots of other interesting operations, as we will discuss in Section 6.7.

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 $\mathbb{S}$.

(b) The label function, which takes a formula $A$ to its canonical label $\langle A \rangle$, is
representable in $\mathbb{S}$.

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 $\mathbb{S}$.

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 $\mathbb{S}$. Let’s restate it:

5.5.9 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
$\text{Def } L$, such that, for any $L$-formulas $A(x)$ and $B(x)$,

$$\text{apply}\langle A(x) \rangle \langle B(x) \rangle \equiv T \langle A\langle B(x) \rangle \rangle$$

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:

5.5.10 Theorem
The minimal theory of strings $\mathbb{S}$ 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

\[
\text{apply}(A(x))\langle B(x)\rangle = \langle A\langle B(x)\rangle \rangle
\]

(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 \(\text{Apply}(x, y, z)\) such that each sentence of the form

\[
\forall z \left( \text{Apply}((A(x)), (B(x)), z) \equiv (z = \langle A\langle B(x)\rangle \rangle) \right)
\]

is a theorem of \(T\).)

Again, we haven’t proved Theorem 5.5.10 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 ??.

We can use this to prove powerful generalizations of the theorems from Section 5.4. If we have a term \(\text{apply}(x, y)\) that represents application in a theory \(T\), then as before we can let \(\text{diag}(x)\) be the term \(\text{apply}(x, x)\). So

\[
\text{diag}(A(x)) \equiv (A(A(x)))
\]

Then we can plug this into a formula in order to come up with a “self-referential” sentence, basically the same way as before.

**5.5.11 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

\[
A \equiv_T F(A)
\]

**5.5.12 Exercise (Tarski’s Theorem Version 2)**

Suppose \(T\) represents syntax. Let \(\text{True}(x)\) be a formula, and suppose that for
5.5. REPRESENTING SETS AND FUNCTIONS IN A THEORY

each sentence $A$,

$$\text{True}(A) \equiv A$$

Then $T$ is inconsistent.

5.5.13 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 5.4.2), follows from this generalized version. We can let $T$ be $\text{Th } S$, the set of all sentences which are true in $S$. This theory is consistent (it has $S$ as a model). It also represents syntax (as we discussed in Section 5.4). So $\text{Th } S$ must not be representable in $\text{Th } S$, which means that $\text{Th } S$ is not definable in the string structure $S$.

Before we move on, here’s one more “bonus” version of the Fixed Point Theorem. It allows you to more directly formalize statements like

$\text{L = \text{L is not true}}$

in the language of strings—with definite descriptions.

5.5.14 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

$$a \equiv T\langle F(a) \rangle$$

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 $F(F(a))$ 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 \( F(G) \) in \( T \)—that is, \( G \leftrightarrow F(G) \) is a theorem of \( T \)—that doesn’t mean that the identity \( \langle G \rangle = \langle F(G) \rangle \) 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.

5.6 What the Minimal Theory of Strings Can Represent*

The full theory of strings (\( \text{Th } \mathcal{S} \)) is very complicated, and the minimal theory of strings (\( \mathcal{S} \)) is relatively simple—it has just finitely many axioms. (In Chapter 6 and Chapter 7 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 \( \mathcal{S} \). But as it turns out, the minimal theory of strings includes all of the truths about strings which are syntactically simple enough. For example, \( \mathcal{S} \) 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 \( \mathcal{S} \), 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, \( A(s) \) is true in \( \mathcal{S} \) for each string \( s \in X \), and \( \neg A(s) \) is true in \( \mathcal{S} \) 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
5.6. WHAT THE MINIMAL THEORY OF STRINGS CAN REPRESENT*

will define later in this section, they are $\Sigma_1$-definable.) This tells us the main fact we took on faith in Section 5.5: the minimal theory of strings represents syntax. In Section 6.7 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 $\mathcal{S}$ includes all the “syntactically simple enough” truths about strings? Let’s start small. We’ll start by showing that the minimal theory $\mathcal{S}$ knows enough to “unpack” each closed term in the language of strings. Remember that a term in this language is either "", a singleton constant for some symbol $a$, or else a term $(t_1 \oplus t_2)$ for some terms $t_1$ and $t_2$. Remember also that each string $s$ has a canonical label $\langle s \rangle$ (Definition 5.1.8). For example, the canonical label for $\text{ABC}$ is the term

$$\text{"A" } \oplus (\text{"B" } \oplus (\text{"C" } \oplus \text{""}))$$

We can repeatedly apply the axioms of $\mathcal{S}$ to convert an arbitrary term to its canonical form. In particular, $\mathcal{S}$ has these axioms, which correspond to the recursive definition of the join function. First, the base case:

$$\text{"" } \oplus x = x$$

Remember that each symbol in the alphabet has a corresponding “singleton constant”. For each singleton constant $c$, we have these axioms as well:

$$(c \oplus x) \oplus y = (c \oplus (x \oplus y))$$

For example, we can apply these axioms to “normalize” the term $(\text{"A" } \oplus \text{""}) \oplus \text{"B"}$. This immediately follows from one instance of these axioms (using the singleton constant $\text{"A"}$, and instantiating $x$ with $\text{""}$ and $y$ with $\text{"B"}$):

$$(\text{"A" } \oplus \text{""}) \oplus \text{"B"} = \text{"A" } \oplus (\text{"" } \oplus \text{"B"})$$

Then, since $\text{"" } \oplus \text{"B"} = \text{"B"}$ follows from an instance of the first axiom, and $\text{"B" } = \text{"B" } \oplus \text{""}$ is an instance of the third axiom, by Leibniz’s Law we have this as a theorem of $\mathcal{S}$:
The right-hand term is the canonical label for the string denoted by the left-hand term.

5.6.1 Exercise

(a) If $c$ is the constant for the symbol $a$, then

\[
\langle c \rangle = \langle (a) \rangle
\]

is a theorem of $\mathcal{S}$. (To be explicit: the right-hand side of this identity sentence is the canonical label of the singleton string for the symbol $a$.)

(b) 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 $\mathcal{S}$.

(c) Let $t$ be any term in the language of strings, and let $s = \|t\|_\mathcal{S}$ be the denotation of $t$ in $\mathcal{S}$. Show by induction on the structure of the term $t$ that

\[
t = \langle s \rangle
\]

is a theorem of $\mathcal{S}$.

(d) Let $t_1$ and $t_2$ be any terms in the language of strings. If

\[
t_1 = t_2
\]

is true in $\mathcal{S}$, then it is a theorem of $\mathcal{S}$.

We can also show similar things about distinctness. We have these axioms of $\mathcal{S}$ (which correspond to the Injective Property for strings). For each singleton constant $c$:

\[
(\langle A \rangle \oplus \langle \rangle) \oplus \langle B \rangle = \langle A \rangle \oplus (\langle B \rangle \oplus \langle \rangle)
\]
5.6. WHAT THE MINIMAL THEORY OF STRINGS CAN REPRESENT

\[ c \oplus x \neq "" \]
\[ c \oplus x = c \oplus y \rightarrow x = y \]

And for each pair of distinct singleton constants \( c_1 \) and \( c_2 \),

\[ c_1 \oplus x \neq c_2 \oplus x \]

We can use these axioms to show the following.

5.6.2 Exercise

(a) Show by induction that for any string \( s_1 \), if \( s_2 \) is a distinct string from \( s_1 \), then

\[ \langle s_1 \rangle \neq \langle s_2 \rangle \]

is a theorem of \( \mathcal{S} \).

(b) Let \( t_1 \) and \( t_2 \) be any terms in the language of strings. If

\[ t_1 \neq t_2 \]

is true in \( \mathcal{S} \), then it is a theorem of \( \mathcal{S} \).

We can do similar things with our other basic kind of formulas. The theory \( \mathcal{S} \) also has some axioms that say how the “no-longer-than” relation should work:

\[ "" \leq x \]
\[ x \leq "" \leftrightarrow x = "" \]

And for each pair of singleton constants \( c_1 \) and \( c_2 \) (not necessarily distinct),

\[ c_1 \oplus x \leq c_2 \oplus x \leftrightarrow x \leq y \]
5.6.3 Exercise
Show by induction that for any strings $s_1$ and $s_2$:

(a) If $s_1$ is no longer than $s_2$, then

$$\langle s_1 \rangle \leq \langle s_2 \rangle$$

is a theorem of $S$.

(b) If $s_1$ is longer than $s_2$, then

$$\neg (\langle s_1 \rangle \leq \langle s_2 \rangle)$$

is a theorem of $S$.

(c) Let $t_1$ and $t_2$ be terms in the language of strings. If the sentence

$$t_1 \leq t_2$$

is true in $S$, then it is a theorem of $S$. If it is false in $S$, then

$$\neg (t_1 \leq t_2)$$

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 $\neg$ and $\wedge$.

5.6.4 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 $\leq$), negation, and conjunction. If $A$ is true in $S$, then $S \not\models A$, and if $A$ is false in $S$, then $S \models \neg A$.

Hint. Use induction.

This means $S$ knows the truth-value of every sentence in the first-order language of
strings that don’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 $\mathbb{S}$. (We won’t prove this now, but it will turn out to be a consequence of Gödel’s First Incompleteness Theorem, Exercise 7.5.6.)

But not all sentences using quantifiers are out of reach. For example, consider this sentence:

$$\forall x ((x \leq "\cdot\cdot") \rightarrow ((x \leq "A") \lor ("BB" \leq x)))$$

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 $\cdot\cdot$. It turns out that the minimal theory $\mathbb{S}$ 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):

$$s_1, s_2, \ldots, s_n$$

Then, if we abbreviate the right-hand side $((x \leq "A") \lor ("BB" \leq x))$ as $A(x)$, we can rewrite the quantified sentence as a long conjunction, like this:

$$A(s_1) \land A(s_2) \land \ldots \land A(s_n)$$

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 $\mathbb{S}$ “knows” this equivalence. And since the conjunction doesn’t have any quantifiers, we have already shown that $\mathbb{S}$ knows its truth-value, too. Thus this particular quantified sentence is also a theorem of $\mathbb{S}$.

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 $\mathbb{S}$.

5.6.5 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

$$\forall x \ (x \not\equiv t \rightarrow A)$$
Similarly,

$$\exists (x \lessdot t) \ A$$

abbreviates the **bounded existential generalization**

$$\exists x \left( (x \lessdot t) \land A \right)$$

Call a formula in the language in the language of strings (without definite descriptions) **bounded** if 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 $S$.

$$x = "" \lor \exists y \ (x = c_1 \oplus y \lor \cdots \lor x = c_n \oplus y)$$

where $c_1, \ldots, c_n$ are all of the constants for single symbols. 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.

**5.6.6 Exercise**

(a) Let $s$ be any string, and let $s_1, \ldots, s_n$ be all of the strings which are no longer than $s$. Prove by induction on $s$ that

$$\forall x \ (x \lessdot \langle s \rangle \leftrightarrow (x = \langle s_1 \rangle \lor \cdots \lor x = \langle s_n \rangle))$$

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

$$B \leftrightarrow \forall (x \lessdot t) \ A(x)$$

is a theorem of $S$.

\(^1\)Other standard names for bounded formulas include $\Delta_0$-formulas, $\Sigma_0$-formulas, and $\Pi_0$-formulas.
5.6.7 Exercise
Let $A$ be any bounded sentence. If $A$ is true in $S$, then $S \models A$, and if $A$ is false in $S$, then $S \models \neg A$.

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.)

5.6.8 Exercise
Suppose $A(x)$ is a bounded formula. If

\[ \exists x \ A(x) \]

is true in $S$, 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.

5.6.9 Definition
A formula is $\Sigma_1$ (pronounced “sigma-one”) iff it has the form $\exists x \ A$, for some bounded formula $A$. That is, a $\Sigma_1$ formula is a bounded formula with an unrestricted existential quantifier in front. \(^2\)

So in other words, what Exercise 5.6.8 tells us is that, if $A$ is $\Sigma_1$, and $A$ is true in $S$, then $A$ is a theorem of $S$. But, to reiterate, in general if $A$ is false in $S$, we don’t know that $\neg A$ is a theorem of $S$.

\(^2\)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 $\Pi$ is often used to represent universal quantification. So similarly, a $\Pi_1$-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 $\Sigma_1$-formula is what you get by adding an existential quantifier to a $\Pi_1$ formula. A $\Pi_2$-formula is what you get by adding a universal quantifier to a $\Sigma_1$-formula. And you can go on this way to recursively define $\Sigma_n$ and $\Pi_n$ 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”.

---

5.6. WHAT THE MINIMAL THEORY OF STRINGS CAN REPRESENT* 213
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.”

5.6.10 Exercise
Say a formula is \( \Sigma_1 \)-equivalent iff it has the same extension in \( S \) as some \( \Sigma_1 \)-formula. If \( A \) and \( B \) are \( \Sigma_1 \) formulas, and \( t \) is a term, then the following are \( \Sigma_1 \)-equivalent.

(a) \( A \lor B \)
(b) \( A \land B \)
(c) \( \exists x \ A \)
(d) \( \forall (x \in t) \ A \)

5.6.11 Exercise
(a) Let \( X \) be a set of strings which is definable in \( S \) using a bounded formula. 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 \( S \)). Then \( A(x) \) also represents \( X \) in \( S \).

(b) Let \( X \) be a set of strings which is definable in \( S \) using a \( \Sigma_1 \)-formula \( A(x) \). Then \( A(x) \) also represents \( X \) in \( S \) “in one direction”. That is, for each string \( s \in X \), \( A(s) \) is a theorem of \( S \), and for each string \( s \not\in X \), \( A(s) \) 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.

5.6.12 Definition
Let \( f \) be a partial function from strings to strings. Say that \( f \) is \( \Sigma_1 \)-definable iff there is a \( \Sigma_1 \) 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 \( S \).

---

3In Section 6.6 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 \( \Sigma_1 \) 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.)
5.6. WHAT THE MINIMAL THEORY OF STRINGS CAN REPRESENT*

We’ll need to use one more axiom of $S$:

\[ x \leq y \lor y \leq x \]

5.6.13 Exercise

Let $A(x)$ be the formula

\[ B(x) \land \forall(x' \leq x) (B(x') \rightarrow x = x') \]

Then

\[ \forall x \forall y (A(x) \rightarrow A(y) \rightarrow x = y) \]

is a theorem of $S$.

5.6.14 Exercise

If $f$ is $\Sigma_1$-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 $s$, $A(s)(fs)$ and

\[ \forall y \forall z (A(\langle s \rangle, y) \rightarrow A(\langle s \rangle, z) \rightarrow y = z) \]

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 $\Sigma_1$-formula that defines $f$. If we did that, the uniqueness condition wouldn’t be a $\Sigma_1$ 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 $\Sigma_1$-definable, this means that there is a bounded formula $B(x, y, z)$ such that, for each $s$, $fs$ is the unique value such that $(s, fs)$ 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 $\Sigma_1$-formula:

\[ \exists z (B(x, y, z) \land \forall(y' \leq y) \forall(z' \leq z) (B(x, y', z') \rightarrow y = y')) \]
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 $S$: in fact, it is $\Sigma_1$-definable. The same goes for the labeling function. Thus by Exercise 5.6.14, 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 6.7, which is much more general.

### 5.7 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 6). 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

\[
\text{apply} \langle A(x) \rangle \langle B(x) \rangle = \langle A(B(x)) \rangle
\]

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 $Th S$ 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.)
5.7. SYNTAX AND ARITHMETIC

5.7.1 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 \( f'(x_1, \ldots, x_n) \) in the language \( L' \), and which assigns each primitive \( n \)-place relation symbol \( R \) in the language \( L \) some formula \( R'(x_1, \ldots, x_n) \) in the language \( L' \).

Given a translation manual, the translation of an \( L \)-formula is the result of replacing each occurrence of \( f(a_1, \ldots, a_n) \) with the corresponding term \( f'(a_1, \ldots, a_n) \), and each occurrence of \( R(a_1, \ldots, a_n) \) with the corresponding formula \( R'(a_1, \ldots, a_n) \).

(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.

5.7.2 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 \( \phi \) iff, for each sentence \( A \) in \( T \), its translation \( \phi(A) \) is in \( T' \).

5.7.3 Lemma
Let \( T \) be an \( L \)-theory and let \( T' \) be an \( L' \)-theory. Suppose that \( T' \) interprets \( T \) with respect to a translation function \( \phi \) 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 \( \phi(d) \) 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

\[
\begin{align*}
\text{If } & d \in X \quad \text{then } \quad T \models A(d) \\
\text{If } & d \notin X \quad \text{then } \quad T \models \neg A(d)
\end{align*}
\]

Since \( T' \) interprets \( T \) with respect to the translation function \( \phi \), we know:

\[
\begin{align*}
\text{If } & d \in X \quad \text{then } \quad T' \models \phi(A(d)) \\
\text{If } & d \notin X \quad \text{then } \quad T' \models \phi(\neg A(d))
\end{align*}
\]

Now, \( \phi(A(d)) \) is the result of systematically replacing each symbol in \( A(d) \) 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

\[
\begin{align*}
\text{If } \ d \in X & \text{ then } T' \models A'(\varphi(d)) \\
\text{If } \ d \not\in X & \text{ then } T' \models \neg A'(\varphi(d))
\end{align*}
\]

So, since $\varphi(d)$ 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.

\[\square\]

5.7.4 Definition

A theory $T$ is **sufficiently strong** iff it interprets the minimal string theory $S$.

5.7.5 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 4.4.2** we presented the **minimal theory of arithmetic** $Q$, which is a very simple theory with just ten axioms. As it turns out:

5.7.6 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.

5.7.7 Exercise

Any theory that interprets \(Q\) is sufficiently strong. In particular, the theory of arithmetic \(\text{Th } \mathbb{N}\) is sufficiently strong.

5.7.8 Definition

Let \(\varphi\) 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 \(\varphi(A)\).

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 \(\langle \varphi(A) \rangle\).

5.7.9 Exercise (Tarski’s Theorem Version 5)

The set of Gödel numbers of true first-order sentences of arithmetic is not arithmetically definable.
Chapter 6

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. Those who are accustomed to some of the more modern views of the above subject, will know that 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!)

6.1 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. 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:

---

1For 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 (for example, you can use this one: https://repl.it/languages/python3). This is a useful way to check your work.

(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:

```python
def head(x): return x[0]
def tail(x): return x[1:]
newline = "\n"
quote = "\"
```

After that, everything should work ok. You can also 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 by typing four spaces—not “tabs”—and watch out that your `while`, `for`, and `if` blocks are correctly lined up.)

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 \(\rightarrow\), \(\lor\), and \(\exists\)) 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).

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 `⊕` 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`, `y₁`, `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 \texttt{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 two exceptions. The first exception is the quotation mark itself, ", whose term is `quote`. (Using "" would be confusing, and it has a different meaning in Python.) The second is the symbol that represents the start of a new line, whose term is `newline`. (This will matter a bit now, because Py-programs are represented by multi-line strings. In fact, there are a few other exceptions for how Python interpreters handle some other special symbols, but we can ignore these.)

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 \texttt{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 \texttt{x}, except the first symbol—again, unless the string denoted by \texttt{x} is empty, in which case we crash with an error message.

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" + newline + "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:
You should read this as an \textit{imperative} sentence—“let $x$ be $a$ from now on”—and \textit{not} as a \textit{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.)\footnote{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 \textit{more often} than they use actual equality—and they wanted to save a keystroke.} So we can write things like

\begin{verbatim}
x = x + "A"
\end{verbatim}

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 \textit{imperative} reading means “\textit{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.

A \textit{program} (or \textit{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:

\begin{verbatim}
firstValue = ""
secondValue = "A"
secondValue = secondValue + secondValue
result = head(secondValue)
\end{verbatim}

First, this sets the variable \texttt{firstValue} so it denotes the empty string. Second, this sets the variable \texttt{secondValue} so it denotes the string $A$. Third, this changes the variable \texttt{secondValue} so it instead denotes $AA$. Finally, this sets the variable \texttt{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:
CHAPTER 6. THE UNDECIDABLE

<table>
<thead>
<tr>
<th>firstValue</th>
<th>(the empty string)</th>
</tr>
</thead>
<tbody>
<tr>
<td>secondValue</td>
<td>A</td>
</tr>
</tbody>
</table>

The person follows the instructions one by one. When they reach a “let” statement, they erase one of the values in the right-hand column and write in some new value. For instance, when they see the third instruction

```python
secondValue = secondValue + secondValue
```

they will change the table to look like this:

| firstValue | secondValue | AA |
|------------|-------------|
|             | AA          |

After the final instruction, the table will then say:

| firstValue | secondValue | AA |
|------------|-------------|
|             | 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.

### 6.1.1 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).

```python
callButFirst = tail(x)
result = head(allButFirst)
```

### 6.1.2 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 \( \text{True} \). One way to do this would be to write this:

```python
trueString = "T" + "r" + "u" + "e"
```

But that’s a bit of a nuisance, so we’ll use this handy shorthand.

```python
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 \( \rightarrow \) 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 to make the language easier to use.

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.

6.1.3 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, ≠) 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 `C`, we can build this kind of statement:

```python
while a != b:
    C
```

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. What it means is to 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:

```python
if s == "":
  result = "It’s empty!"
else:
  result = head(s)
  s = tail(s)
```

If \( s \) is not empty, then this statement sets the value of result to its first symbol, and modifies the value of \( s \) by removing the first element from the sequence. 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

```python
if a == b:
  A
else:
  B
```

as an abbreviation for this:
Here \( x \) and \( y \) 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 \texttt{else} part of an \texttt{if} -statement: we don’t want to do anything in that case. We can indicate this by just leaving out the \texttt{else} part. That is, this program:

\[
\text{if } a == b:\n\quad A
\]

means just the same thing as this one:

\[
\text{if } a == b:\n\quad A\n\quad \text{else:}\n\quad ()
\]

where the \texttt{else} block is the empty program. We can also write

\[
\text{if } a != b:\n\quad A
\]

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.

6.1.4 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 $\text{True}$?

(c) Does $s$ have at least two elements?

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 $\text{and}(s, t)$, and $\text{or}(s, t)$.

Bounded Loops

Another common pattern in programs is to go through each of the elements of a string one by one, do something with each one, and stop when we reach the end of the string. This is called a $\text{for}$ loop.

For example, this program decides whether every symbol in a string is $A$.

```python
result = "True"
for symbol in s:
    if symbol != "A":
        result = "False"
```

The $\text{for}$ loop goes through the elements of the string represented by $s$ one by one, and stores each symbol as the value of the variable $\text{symbol}$. This is similar to a $\text{while}$ loop, but it is more specialized. One important feature of a $\text{for}$ loop is that it is guaranteed to eventually stop, when it gets to the end of the string. In contrast, in principle a $\text{while}$ loop might go on running forever, if the equality test is never passed.

Again, though $\text{for}$ 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 $\text{while}$ loops. In general, suppose $x$ is any variable, $a$ is any term, and $A$ is some program. We can understand this notation—

```python
for x in a:
    A
```

—as a shorthand for this, where $y$ is a variable that is not used elsewhere in the program—
6.1. PROGRAMS

\[ y = a \]
\[
\text{while } y != "": \\
\quad x = \text{head}(y) \\
\quad y = \text{tail}(y)
\]

6.1.5 Example
This program takes a string and repeats each symbol an extra time. For instance, it takes ABC to AABBCC.

```python
result = ""
for symbol in s:
    result = result + symbol + symbol
```

6.1.6 Example
We can rewrite the reverse program a bit more concisely using a for loop.

```python
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:

```python
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: \( x = \text{result} \). 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:

```python
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:

```python
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.)
6.1.7 Example

This program returns `True` for an empty string, and `False` for a non-empty string.

```python
def empty(s):
    if s == "":
        result = "True"
    else:
        result = "False"
    return result
```

Then suppose we write this later:

```python
x = empty("ABC")
```

Then this abbreviates

```python
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:

```python
z = programName(a, b)
```

We can unpack it like this:

```python
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 - else branching (and not and elif), 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.

6.1.8 Exercise

Write a program that computes the “dots” function from Exercise 2.5.3. For example, the output of the program for input ABC should be •••.

6.1.9 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?

6.2 Syntax and Semantics

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 $V$ of variables. In Py, our official 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).3

We’ll give two inductive definitions: one for Py-terms, and the other for Py-programs. We’ll start with terms. The definition 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.

3We’ll ban a few special strings from being variables: while, if, else, elif, def, return head, tail, quote, and newline.
6.2. Definition

\[ x \text{ is a variable} \]
\[ x \text{ is a term} \]

Remember that in Section 3.2 we chose some constants: "" for the empty string, and constants like "A", "B", quote, and newline for single-symbol strings. Each of these constants is also a Py-term.

\[ c \text{ is a constant in the language of strings} \]
\[ c \text{ is a term} \]

\[ t_1 \text{ is a term} \]
\[ t_2 \text{ is a term} \]
\[ (t_1 + t_2) \text{ is a term} \]

\[ t \text{ is a term} \]
\[ \text{head}(t) \text{ is a term} \]
\[ \text{tail}(t) \text{ is a term} \]

Besides some variant notation, that much should look pretty familiar, because it’s very similar to the definition for terms in the first-order language of strings. Next we’ll give the inductive definition of programs.

6.2.2 Definition

The empty string is a program

\[ x \text{ is a variable} \]
\[ t \text{ is a term} \]
\[ A \text{ is a program} \]
\[ x = t \]
\[ A \text{ is a program} \]

\[ t_1 \text{ and } t_2 \text{ are terms} \]
\[ A \text{ and } B \text{ are programs} \]
\[ \text{while } t_1 \neq t_2; \]
\[ A \]
\[ B \text{ is a program} \]
That should get the idea across, but before we move on, let’s get clear on a few details about what this means. (You can skip over these details, but they’re important if you’re going to do some of the parsing exercises in Section 6.4.)

Programs are strings. (Just like always we can ask, is a program really just a string, or does they have some other structure that can be represented by a string? But it will make things easier if we suppose that a program just is a certain string.) A string is just a sequence of symbols. But because programs can get pretty long, it would be a huge pain to write out a program in a single line of text, the way we usually write sequences. 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:

\[
\begin{align*}
y &= x \\
z &= y
\end{align*}
\]

We could spell out the sequence of symbols in this string very explicitly like this:

\[
(\text{\texttt{y}, \texttt{, =, , x, new line}}, \text{\texttt{z}, \texttt{, =, , y, new line}})
\]

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

\[
x \oplus \texttt{=} \oplus t \oplus \texttt{new line} \oplus A
\]

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 four spaces to the beginning of the line.

To be totally explicit, then: for any program \(A\), there is a unique sequence of strings \((s_1, s_2, \ldots, s_n)\) which are the lines of \(A\): none of them contains any newline symbols, and

\[
A = s_1 \oplus \text{new line} \\
\oplus s_2 \oplus \text{new line} \\
\vdots \\
\oplus s_n \oplus \text{new line}
\]

Then

\[
\text{indent}(A) = \Box \oplus s_1 \oplus \text{new line} \\
\Box \oplus s_2 \oplus \text{new line} \\
\vdots \\
\Box \oplus s_n \oplus \text{new line}
\]
Now we can state the syntax rule for \texttt{while} statements more explicitly. If $t_1$ and $t_2$ are terms, and $A$ and $B$ are programs, then this is also a program:

\[
\texttt{while } t_1 \neq t_2 : \texttt{new line indent(A) B}
\]

That’s it for the syntax of programs.

Just like with formulas, it’s helpful to know when a variable is “loose” in a program: in this context, this means that it 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 \texttt{occur} in a \texttt{Py-term}; this definition is basically identical to \texttt{Definition 3.6.3}, so we won’t bother to spell it out.

\textbf{6.2.3 Definition}

The \texttt{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

   \[
   \texttt{while } t_1 \neq t_2 : \\
   A \\
   B
   \]

   iff either $y$ occurs in $t_1$ or in $t_2$, or $y$ is free in $A$ or in $B$.

In other words, the function that takes a program $A$ to its set of free variables $\text{Var } A$ is recursively defined as follows:

\[
\text{Var}(\emptyset) = \emptyset \\
\text{Var}\left( x = t \bigg\vert A \right) = (\text{Var } t \cup \text{Var } A) - \{x\} \\
\text{Var}\left( \texttt{while } t_1 \neq t_2 : A \bigg\vert B \right) = \text{Var } t_1 \cup \text{Var } t_2 \cup \text{Var } A \cup \text{Var } B
\]
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 \( \llbracket t \rrbracket g \). 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.)

### 6.2.4 Definition

\[
\begin{align*}
\llbracket x \rrbracket g &= gx \quad \text{for each variable } x \\
\llbracket "" \rrbracket g &= \text{the empty string} \\
\llbracket c \rrbracket g &= (a) \text{ if } c \text{ is the constant for the symbol } a \text{ in the alphabet} \\
\llbracket t_1 + t_2 \rrbracket g &= \llbracket t_1 \rrbracket g \oplus \llbracket t_2 \rrbracket g \\
\llbracket \text{head}(t) \rrbracket g &= \begin{cases} (a) & \text{if } \llbracket t \rrbracket g = (a : s) \\ \text{undefined} & \text{if } \llbracket t \rrbracket g \text{ is empty} \end{cases} \\
\llbracket \text{tail}(t) \rrbracket g &= \begin{cases} s & \text{if } \llbracket t \rrbracket g = (a : s) \\ \text{undefined} & \text{if } \llbracket t \rrbracket g \text{ is empty} \end{cases}
\end{align*}
\]

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: \( \llbracket x \rrbracket g \) 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 \( t + u, u + t, \text{head}(t) \), and \( \text{tail}(t) \) 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 $\llbracket A \rrbracket g$—that is, the denotation of $A$ with respect to $g$—should be the new assignment.

For example, consider this very simple program:

```python
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 $g[x \mapsto A]$. Thus, for any assignment $g$, we can write the denotation of this program as

$$\llbracket x = "A" \rrbracket g = g[x \mapsto A]$$

In general, a “let” statement $x = t$ (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,

$$\llbracket x = t \rrbracket A g = \llbracket A \rrbracket (g[x \mapsto \llbracket t \rrbracket g])$$

Usually the let statement will be followed by more stuff, though. In general, the rule for let statements looks like this:

$$\llbracket x = t \rrbracket A g = \llbracket A \rrbracket (g[x \mapsto \llbracket t \rrbracket g])$$

The case of while statements is a bit trickier. Here’s another example:

```python
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 $g_0(x) = A$. 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, \( g_1 = g_0[x \mapsto s, \ y \mapsto s'] \) where \( s \) is the first symbol of \( g_0(y) \), and \( s' \) is all of \( g_0(y) \) except the first symbol.) Then we go back and check again: this time, we check whether \( g_1(x) = A \). And so on. If eventually we reach an assignment \( g_n \) such that \( g_n(x) = A \), then that 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 \texttt{while} loop is just undefined.

Here’s another way of describing this. If our \texttt{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

\[
x = \texttt{head}(y)
\]
\[
y = \texttt{tail}(y)
\]

says to do, and \( g_n \) is the first assignment in the sequence such that \( g_n(x) = A \). We’ll call a sequence like this a \textit{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.

\textbf{6.2.5 Definition}

The \textit{denotation} of a program \( A \) with respect to an assignment \( g \) is defined recursively as follows.

The empty program \( () \) doesn’t do anything:

\[
\llbracket () \rrbracket g = g
\]

For a “let” statement, where \( x \) is a variable, \( t \) is a term, and \( A \) is a program,

\[
\llbracket x = t \ A \rrbracket g = \llbracket A \rrbracket (g[x \mapsto \llbracket t \rrbracket g])
\]

The most complicated case is a \texttt{while} statement,

\[
\texttt{while} \ t_1 \ != \ t_2 ; \ A
\]

\[
\llbracket \texttt{while} \ t_1 \ != \ t_2 ; \ A \rrbracket g
\]
(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 \(i < n\),
   \[
g_{i+1} = \llbracket A \rrbracket g_i
   \]

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.

\[
\llbracket \text{while } t_1 \neq t_2; \quad A; \quad B \rrbracket \ g = \begin{cases} 
\llbracket B \rrbracket h & \text{where } h \text{ is the last element of the finite loop sequence for } t_1, t_2, \\
\text{undefined} & \text{if there is no such sequence}
\end{cases}
\]

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 \(\text{result}\). This motivates the following definition.

6.2.6 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
That is, if \( g \) is the assignment \( \llbracket A \rrbracket [x \mapsto s] \), then \( \llbracket A \rrbracket (s) = g(\text{result}) \).

More briefly:

\[
\llbracket A \rrbracket (s) = (\llbracket A(x) \rrbracket [x \mapsto s])(\text{result})
\]

If there is no final result (either because there is no assignment \( g \), or because \( g \) does not have a value for the variable \( \text{result} \)) then \( \llbracket A \rrbracket (s) \) is undefined.

(b) A program \( A(x) \) \textbf{halts} for input \( s \) iff \( \llbracket A \rrbracket (s) \) is defined.

(c) The \textbf{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 \textit{computable} using Py programs?

\textbf{6.2.7 Definition}

(a) A function \( f : \mathbb{S} \rightarrow \mathbb{S} \) is \textbf{Py-computable} iff it is the extension of some program.

(b) A set of strings \( X \subseteq \mathbb{S} \) is \textbf{Py-decidable} iff its characteristic function is Py-computable: that is, the function that takes each string \( s \in X \) to \textbf{True} and each string \( s \notin X \) to \textbf{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 \textit{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 \textit{computable} function is one that is definable using a \textit{programming} language, rather than a first-order language, and likewise, a decidable set is one that is definable using a programming language.
6.2.8 Example

Prove that the following program halts, for any initial value for $x$.

```python
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,

```python
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

$$g[x \mapsto A]$$

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,

```python
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 $g(x)$ is $A$, or it is something else. If $g(x) \neq A$, 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.

$$(g, g[x \mapsto A])$$

First, $g_1$ is clearly given by applying the denotation of the inner block to the assignment $g_0$. Second, $g_0(x) \neq A$ by assumption. Third, clearly $g_1(x) = A$.

On the other hand, if $g(x) = A$, 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: $g(x) = A$.

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 $g(x) = A$ for the initial assignment, for the final assignment $h$ in the sequence, $h(x) = A$; in particular, this is not empty. Now evaluate the final “let” statement (which is followed by the empty program):
result = \texttt{tail}(x)

The denotation of this program, given the assignment $h$, is

$$h[\texttt{result} \mapsto \llbracket \texttt{tail}(x) \rrbracket h]$$

To ensure that this is defined, we just need to check that $\llbracket \texttt{tail}(x) \rrbracket h$ is defined. And this is true: looking at the definition for the denotation of \texttt{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 $s$, $\llbracket A \rrbracket s$ is defined, which means that $A$ halts for every input. □

6.2.9 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.

6.2.10 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 $A(x)$:

\begin{verbatim}
x  =  t
A(x)
\end{verbatim}

(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

\begin{verbatim}
A(x)
y  =  result
B(y)
\end{verbatim}

(This is similar to our “function call” shorthand.)
6.3. THE CHURCH-TURING THESIS

6.2.11 Exercise

(a) For any program $A(x)$ and term $t$,

$$\llbracket A(t) \rrbracket = \llbracket A \rrbracket (\llbracket t \rrbracket)$$

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 $s$,

$$\llbracket B(A(x)) \rrbracket (s) = \llbracket B \rrbracket (\llbracket A \rrbracket (s))$$

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 $B(y)$.

6.3 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. 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. TDOO: explain this a bit.

Even so, it’s worth remembering that it is a philosophical thesis—an extraordinarily
successful philosophical 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.

6.3.1 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.

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.

6.3.2 Exercise
Given the Church-Turing Thesis, prove that there are uncountably many effectively undecidable sets of strings.

6.4 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!

4Peter Smith CITE calls these “labor-saving” uses of the Church-Turing Thesis. This contrasts with
Even our little language Py can do this. We can write a “Py-interpreter” in Py. This is a program \( \text{run}(\text{program}, \text{inputValue}) \) 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 \( \text{run} \) 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 \( s \),

\[
[\text{run}](A, s) = [A](s)
\]

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 6.5 and Section 6.6.

First, let’s introduce some tools which are analogous to what we did in Chapter 5. 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.

6.4.1 Exercise

In Section 5.2 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 \( \text{True} \) if \( t \) is an element of the sequence \( s \), and otherwise returns \( \text{False} \).

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 5.2. We can represent an assignment function as a sequence of strings like this one:

\[(x: \text{hello}, \ result:, \ s: \text{ABC})\]
This represents the assignment function

\[
\begin{bmatrix}
    x & \mapsto & \text{hello} \\
    \text{result} & \mapsto & \text{the empty string} \\
    s & \mapsto & \text{ABC}
\end{bmatrix}
\]

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.)

### 6.4.2 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 \( s \).

### 6.4.3 Lemma

The **denotation function** takes a pair of a program \( A \) and an assignment \( g \) and returns the denotation \( \llbracket A \rrbracket g \) (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 \ \neq \ 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(\(t\)), \(t_1 \cdot t_2\).

(b) For head and tail terms we should also write programs that return
the inner term \(t\), and in in the case of \(\cdot\) 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 fromExercise 6.4.2 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 quote, 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 6.5). 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(y + z), 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:

\[
x = "A" + \text{head}(y + z)
\]

Then we can break this up into simpler let statements, like this:

```
temp1 = "A"
temp2 = y + z
temp3 = \text{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 $\text{head}(x)$, $\text{tail}(x)$, or $x \cdot y$ 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

```plaintext
x = t \\
A
```

for a simple term $t$, or else of the form

```plaintext
while \( t_1 \neq t_2 \):
   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

```plaintext
x = t_1 + t_2 \\
A
```

to a new program with this form:

```plaintext
y = t_1 \\
z = t_2 \\
x = y + z \\
A
```
6.4. THE UNIVERSAL PROGRAM

(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 \texttt{simplify}.

The \texttt{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 \textit{simple} terms, which is pretty straightforward, once we have the parsing and assignment-wrangling tools from our to-do list.

```python
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 \texttt{while}-statement, so the program has the form

\[
\texttt{while } t_1 \neq t_2 : \\
\quad A \\
\quad B
\]

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 \texttt{while} loop) and keep going. That is, in this case we’ll run the program

\[
A \\
\texttt{while } t_1 \neq t_2 : \\
\quad A \\
\quad B
\]

Let’s spell this out in more detail. The whole interpreter goes in one big \texttt{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.

```python
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":
```

6.5. **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 a 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 unde-
cidable. 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 ⟨s⟩. (Using the abbreviation we introduced in Section 6.1, we can abbreviate this term as "ABC".)

We can also plug these terms into programs: A⟨s⟩ is the program that runs the program A(x) with the input s. (That is, A⟨s⟩ is the program consisting of the let statement x = ⟨s⟩ 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:

6.5.1 Proposition
For any string s,

∥⟨s⟩∥ = s

6.5.2 Exercise
For any program A(x) and string s,

∥A⟨s⟩∥ = ∥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.

6.5.3 Exercise
The function that takes each string s to its canonical label ⟨s⟩ is computable.

6.5.4 Proposition
The “substitution” function that takes any program A⟨x⟩ and term t to the program A⟨t⟩ is computable.
6.5. THE HALTING PROBLEM

Proof

Here is a program:

```python
def substitution(program, term):
    result = "x = " + term + newline + program
    return result
```

6.5.5 Exercise

The “diagonalization” function that takes any program $A(x)$ to the program $A(A(x))$ is computable. That is, there is a program $\text{Diag}(y)$ such that, for any program $A(x)$,

$$\langle\text{Diag}\rangle(A(x)) = A(A(x))$$

Let’s be very clear about what this means. The program $\text{Diag}(y)$ 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 $A(x)$, where specifically the term $t$ is the canonical label for the program $A(x)$ itself. For example, suppose $A(x)$ is this very simple program:

```python
z = x
```

Then the result of applying the “diagonalization” function to $A(x)$ is this slightly more complex program:

```python
x = "z = x" + newline
z = x
```

(Note in particular that while $x$ was a free “input” variable in $A(x)$, the diagonalized program $A(A(x))$ does not have any free variables.)

6.5.6 Exercise (Kleene’s Fixed Point Theorem)

Let $F(x)$ be any program. Then there is a program $A$ such that

$$\langle F\rangle(A) = \langle A\rangle$$

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 5.5.11).
It may also be helpful to remember Exercise 6.2.11.

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 \( A \).

6.5.7 Exercise
Write a program \( \text{Flip}(x) \) which does not halt for the input \( \text{True} \), and which halts for any input besides \( \text{True} \).

6.5.8 Exercise (Turing’s Theorem)
The set of programs that halt is undecidable (given the Church-Turing thesis).

\[ [\text{Halt}](A) = \begin{cases} \text{True} & \text{if } A \text{ halts} \\ \text{False} & \text{otherwise} \end{cases} \]

Then you can use Kleene’s Theorem and the program \( \text{Flip}(x) \) 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:

```python
def reverse(x):
    if x == "":
        result = ""
    else:
        reversedTail = reverse(tail(x))
        result = reversedTail + head(x)
    return result
```

This program calls the \texttt{reverse} program itself. Since each time \texttt{reverse} calls itself, the string passed along as the value of \$ x \$ 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 \textit{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”:

6.5.9 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 $s$,

$$\langle A \rangle(s) = \langle F \rangle(A(y), s)$$

This can be proved in basically the same way as Exercise 6.5.6

Now, suppose we want to write the recursive program \texttt{reverse}. Let’s start by modifying it a bit. At the point where we wanted to call the \texttt{reverse} program itself, instead we can run some arbitrary program which is provided as an extra argument.

```python
def protoReverse(program, x):
    if x == "":
        result = ""
    else:
        reversedTail = run(program, tail(x))
        result = reversedTail + head(x)
    return result
```

(Here \texttt{run} is the Universal Program from Exercise 6.4.4.) Then Proposition 6.5.9 tells us that there is a program $R(x)$ which has the same effect as running the protoReverse program with $R(x)$ itself as the first argument.

$$\langle R \rangle(s) = \langle \text{protoReverse} \rangle(R(x), s)$$

In other words, $R(x)$ is equivalent to a program that calls $R(x)$ itself! So the simple Py-program $R(x)$ has the same behavior as the recursive program \texttt{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.)

6.6 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 \texttt{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.

6.6.1 Definition

A **semi-decision procedure** for a set \( X \) is a program \( A \) such that, for each string \( s \in \Sigma \),

\[
\llbracket A \rrbracket(s) = \text{True} \quad \text{iff} \quad s \in X.
\]

But note that if \( s \) is not in \( X \), \( \llbracket A \rrbracket(s) \) isn’t guaranteed to return any value at all: it is just required not to return the value \( \text{True} \). 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.

6.6.2 Exercise

The set of programs that halt is semi-decidable. Thus there is a set which is semi-decidable, but not decidable.

6.6.3 Exercise

Uncountably many sets of strings are not even semi-decidable.

6.6.4 Exercise (Bounded Search)

Let \( A(x, y) \) be a program which halts for every input. Use this to write another program \( B(y, \text{bound}) \) such that, for any strings \( t \) and \( u \),

\[
\llbracket B \rrbracket(t, u) = \text{True} \quad \text{if} \quad \text{there is some string} \ s \quad \text{which is no longer than} \ u, \quad \text{such} \quad \text{that} \ \llbracket A \rrbracket(s, t) = \text{True}.
\]

\[
\llbracket B \rrbracket(t, u) = \text{False} \quad \text{otherwise}.
\]

*Hint.* You can help yourself to a variable called \( \text{alphabet} \) 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.)

### 6.6.5 Exercise (Unbounded Search)

Let $A(x, y)$ be a program which halts for every input. Use Exercise 6.6.4 to write another program $C(y)$ such that, for any string $t$,

$$
\langle C \rangle(t) = \text{True} \text{ iff there is some string } s \text{ (of any length) such that } \langle A \rangle(s, t) = \text{True}.
$$

If there is no such string, $C$ does not have to return any value at all.

### 6.6.6 Exercise

Let $X \subseteq S \times S$ be a set of pairs of strings. Let

$$
X^3 = \{ t \in S \mid \text{there is some } s \in S \text{ such that } (s, t) \in X \}
$$

If $X$ is decidable, then $X^3$ is semi-decidable.

*Hint.* Use Exercise 6.6.5.

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 arbitrary 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 \texttt{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 \emph{semi}-decidable. But this doesn’t mean that every effectively enumerable set is decidable.

We can make these ideas a bit more official.

\textbf{6.6.7 Definition}

A set of strings $X \subseteq \mathbb{S}$ is \textbf{effectively enumerable} iff $X$ is the range of some computable total function.

\textbf{6.6.8 Theorem}

If $X$ is effectively enumerable, then $X$ is semi-decidable.

\textit{Proof}

If $X$ is effectively enumerable, then $X$ is the range of some computable total function $f$. That is to say,

$$X = \{t \in \mathbb{S} \mid \text{there is some } s \in \mathbb{S} \text{ such that } fs = t\}$$

But also, if $f$ is a computable total function, then the set $Y = \{(s, t) \in \mathbb{S} \mid fs = t\}$ is \emph{decidable}—just calculate $fs$, and then check whether the result is the same string as $t$. Since $X = Y^\exists$, using Exercise 6.6.6 we can conclude that $X$ is semi-decidable.\hfill\Box

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.

\textbf{6.6.9 Theorem}

If $X$ is semi-decidable, then $X$ is effectively enumerable.

\textit{Proof Sketch}

Suppose that $X$ is semi-decidable: this means we have some program that returns \texttt{True} just for inputs that are in $X$. We’ll use this to show that $X$ is effectively enumerable.
6.7. DECIDABILITY AND LOGIC

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. □

6.6.10 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.

6.7 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
CHAPTER 6. THE UNDECIDABLE

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 5.

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 6.4, and also how to represent a sequence of strings with a single string in Section 5.2. 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.

6.7.1 Exercise
Recall from Section 6.4 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 5.2 for representing sequences of strings. Show that the following functions are definable in $\mathcal{S}$, 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 5.2 we showed that certain sequence operations are definable in $\mathcal{S}$. It will be helpful to use some of those facts.

6.7.2 Exercise
Show that for each Py-term $t$, the corresponding function that takes each assignment function $g$ to its denotation $\llbracket t \rrbracket g$ is definable in $\mathcal{S}$.

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 $\mathcal{S}$, 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 $\mathcal{S}$ 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

$$\text{while } t_1 \neq t_2^*
A
B$$

The trickiest part is step 3. Recall from Definition 6.2.5 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 $A$, $(g_0, \ldots, g_n)$ is a finite loop sequence iff the following three conditions hold:

- $g_{i+1} = \llbracket A \rrbracket g_i$ for each $i < n$
- $\llbracket t_1 \rrbracket g_i \neq \llbracket t_2 \rrbracket g_i$ for each $i < n$
- $\llbracket t_1 \rrbracket g_n = \llbracket t_2 \rrbracket g_n$

The denotation $\llbracket A \rrbracket g$ is the last element of a finite loop sequence whose first element is $g$, if there is one.

6.7.3 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 $[g \mapsto \llbracket A \rrbracket g]$ and $[g \mapsto \llbracket B \rrbracket g]$) are each definable in $S$.

(a) The set of finite loop sequences for $t_1$, $t_2$, and $A$ is definable in $S$.

(b) The function that takes each assignment $g$ to the denotation

$$\llbracket \text{while } t_1 \neq t_2^* 
A
B \rrbracket g$$

is definable in $S$. 


6.7.4 Exercise (The Definability Theorem)

Given the Church-Turing Thesis, every computable function is definable in the standard string structure $𝕊$.

6.7.5 Exercise

Use Exercise 6.7.4 to show that every decidable set of strings is definable in the string structure $𝕊$.

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.

6.7.6 Exercise

The set of true first-order sentences in the string structure, $\text{Th} 𝕊$, is undecidable.

6.7.7 Exercise

Show that the set of programs that halt is definable in the structure $𝕊$. 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 $𝕊$ to describe computable functions. Just a pretty small simple piece of this theory is enough.

Let’s start by recalling some definitions from Section 5.5. Each string $s \in 𝕊$ has a canonical label, which is a term $⟨s⟩$ in the language of strings. If $f$ is a partial function from $𝕊$ to $𝕊$, then a theory $T$ represents $f$ with a term $t(x)$ (possibly using definite descriptions) iff, for each $s$ for which $f$ is defined,

$$t(s) \equiv_f (f s)$$

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 4.4.3), so it is much simpler than the full theory of
the string structure $\text{Th } S$. Even so, it is powerful enough to describe lots of things. Anything which is definable in $S$ using a syntactically simple enough expression is also representable in $S$. To use the technical term introduced in Section 5.6: anything that is $\Sigma_1$-definable in $S$ is representable in $S$. A $\Sigma_1$-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:

6.7.8 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 $S$—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 $\Sigma_1$ formulas. We won’t work through this in detail. (If you have worked through the “starred” section Section 5.6 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 $\llbracket A \rrbracket(s) = s'$, 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 $h(\text{result}) = s'$.

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 5 we used the fact that the minimal theory of strings $\mathcal{S}$ 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 $\mathcal{S}$.

**6.7.9 Exercise**
Use the Representability Theorem to show that, if $X$ is a decidable subset of $\mathcal{S}$, then the minimal theory of strings $\mathcal{S}$ represents $X$.

Recall that a “sufficiently strong” theory is one that interprets the minimal theory of strings $\mathcal{S}$ (or alternatively, the theory of minimal arithmetic $Q$).

**6.7.10 Exercise**
Any sufficiently strong theory represents every decidable set.

**6.7.11 Exercise (The Essential Undecidability Theorem)**
No sufficiently strong consistent theory is decidable.

*Hint.* Use Tarski’s Theorem.

**6.7.12 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

\[
(A_1 \land \cdots \land A_n) \rightarrow B
\]

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 7, 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 7

The Unprovable

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 6, 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 6.7.12), 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.)

7.1 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 an inductive 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 (\(\land\), \(\lor\), \(\equiv\), and \(\forall\)). 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 \( x \) if and only if \( y \) is not an element of \( y \). So, in particular, \( x \) is an element of \( x \) if and only if \( x \) 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 \( x \) if and only if \( y \) 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 just a bit more austere. It looks like this.

```
for arbitrary \( x \):
  suppose:
    \( \forall y \ (y \in x \leftrightarrow \neg(y \in y)) \) Assumption
    \( x \in x \leftrightarrow \neg(x \in x) \) Universal Instantiation (3)
  suppose:
    \( x \in x \) Assumption
    \( \neg(x \in x) \) \( \neg \)Elim (4, 6)
    \( x \in x \) Reductio
    \( \neg \forall y \ (y \in x \leftrightarrow \neg(y \in y)) \) \( \neg \)Elim (4, 7)
\( \forall x \neg \forall y \ (y \in x \leftrightarrow \neg(y \in y)) \) Universal Generalization
```

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”.

(One detail is that \( \leftrightarrow \) Elim is not really one of the basic rules of our system—indeed, \( \leftrightarrow \) 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 \( \land \) and \( \neg \), and derive the rule of \( \leftrightarrow \) from the corresponding 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 (\(\land\), \(\neg\), \(\equiv\), and \(\forall\)) 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 a 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, or because it is something we have already proved from our premises and suppositions. We call this rule Assumption. (This is because the point of this rule is usually to explicitly state a premise or supposition: but occasionally we also use it to restate something we proved earlier, rather than an assumption. When it’s used this way, the rule is commonly called Reiteration instead.)

7.1.1 Example

For any formula \(A\),

\[ A \vdash A \]

Proof

\(A\) Assumption
7.1. PROOFS

Obviously we can’t do very much with the \textit{Assumption} rule all by itself. But we’ll often use it to get a proof going.

**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 \((A \land B)\). We call this rule Conjunction Introduction, or \textit{\&Intro} for short. For example:

1. \( 1 + 0 = 1 \) \hspace{1cm} \text{Assumption}
2. \( 1 \neq 0 \) \hspace{1cm} \text{Assumption}
3. \((1 + 0 = 1) \land (1 \neq 0)\) \hspace{1cm} \text{\&Intro} (1, 2)

**7.1.2 Example**

For any formula \( A \),

\[ A \vdash A \land A \]

**Proof**

1. \( A \) \hspace{1cm} \text{Assumption}
2. \( A \land A \) \hspace{1cm} \text{\&Intro} (1, 1)

Likewise, if we have proved \((A \land B)\), then we can deduce \( A \); and in that case we can also deduce \( B \). These two rules are called \textit{\&Intro1} and \textit{\&Intro2}.

**7.1.3 Example**

For any formulas \( A \) and \( B \),

\[ A \land B \vdash (B \land A) \]

**Proof**

1. \( A \land B \) \hspace{1cm} \text{Assumption}
2. \( A \) \hspace{1cm} \text{\&Elim1} (1)
3. \( B \) \hspace{1cm} \text{\&Elim2} (1)
4. \( B \land A \) \hspace{1cm} \text{\&Intro} (2, 3)
7.1.4 Example
For any formulas \( A, B, \) and \( C \),

\[
A \land (B \land C) \vdash (A \land B) \land C
\]

Proof

<table>
<thead>
<tr>
<th>Step</th>
<th>Formula</th>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( A \land (B \land C) )</td>
<td>Assumption</td>
</tr>
<tr>
<td>2</td>
<td>( A )</td>
<td>&amp;Elim1 (1)</td>
</tr>
<tr>
<td>3</td>
<td>( B \land C )</td>
<td>&amp;Elim2 (1)</td>
</tr>
<tr>
<td>4</td>
<td>( B )</td>
<td>&amp;Elim1 (3)</td>
</tr>
<tr>
<td>5</td>
<td>( A \land B )</td>
<td>&amp;Intro (2, 4)</td>
</tr>
<tr>
<td>6</td>
<td>( C )</td>
<td>&amp;Elim2 (3)</td>
</tr>
<tr>
<td>7</td>
<td>((A \land B) \land C)</td>
<td>&amp;Intro (5, 6)</td>
</tr>
</tbody>
</table>

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 \( \forall x \). (Negation is a bit special, though.)

The rule \&Intro lets us prove the conclusion \( A \land B \), given the premises \( A \) and \( B \). Similarly, \&Elim1 lets us prove \( A \) from \( A \land B \), and \&Elim2 lets us prove \( B \) from \( A \land B \). So we can concisely summarize these three rules like this:

\[
\begin{align*}
\&\text{Intro} & : A, B \vdash A \land B \\
\&\text{Elim1} & : A \land B \vdash A \\
\&\text{Elim2} & : A \land B \vdash B
\end{align*}
\]

We can summarize the rule of Assumption in the same style:

\[
\text{Assumption} : A \vdash A
\]

7.1.5 Exercise
For any formula \( A \),

\[
A \land A \vdash A
\]
Negation Rules

Our main 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.

7.1.6 Example

For any formulas \(A\) and \(B\),

\[
\neg A \vdash \neg (A \land B)
\]

Proof

<table>
<thead>
<tr>
<th>Step</th>
<th>Formula</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>(\neg A)</td>
<td>Assumption</td>
</tr>
<tr>
<td>2</td>
<td>Suppose:</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>(A \land B)</td>
<td>Assumption</td>
</tr>
<tr>
<td>4</td>
<td>(A)</td>
<td>(\land) Elim 1 (3)</td>
</tr>
<tr>
<td>5</td>
<td>(\neg A)</td>
<td>Assumption</td>
</tr>
<tr>
<td>6</td>
<td>(\neg (A \land B))</td>
<td>Reductio</td>
</tr>
</tbody>
</table>

In this proof, we add an extra assumption, \((A \land B)\), derive a contradiction, and conclude that this assumption is false.

In general, Reductio looks like this. Suppose that \(P\) is a proof from certain premises \(X\) together with the extra assumption \(A\), and which shows both \(B\) and \(\neg B\). Then

\[
\text{suppose:} \quad P
\]

\[
\neg A \quad \neg\text{Intro}
\]

is a proof of \(\neg A\). So the rule of Reductio lets us make this inference about what is provable:

\[
\frac{X, A \vdash B \quad X, A \vdash \neg B}{X \vdash \neg A} \quad \text{Reductio}
\]

An alternative label for Reductio is \(\neg\text{Intro}\) (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 \(\neg\neg\)Elim for short. Given
As a premise, we can simplify this to just $A$.

\[ \neg \text{Elim} : \neg A \vdash A \]

### 7.1.7 Exercise (Explosion)

$A, \neg A \vdash B$.

Remember, officially our language only includes the connectives $\land$ and $\neg$. Formulas using other connectives, like $\rightarrow$ and $\lor$, are officially considered to be abbreviations of formulas using $\land$ and $\neg$. Similarly, we will only officially have basic proof rules for the connectives $\land$ and $\neg$. But we can use the definitions of these other connectives to derive their standard proof rules, as well.

### 7.1.8 Example (Modus Ponens)

For any formulas $A$ and $B$,

\[ A, A \rightarrow B \vdash B \]

**Proof**

Recall that we defined the conditional $(A \rightarrow B)$ to be an abbreviation for $\neg(A \land \neg B)$. So what we want to show is

\[ A, \neg(A \land \neg B) \vdash B \]

We can show this by providing a formal proof.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$A$</td>
</tr>
<tr>
<td>2</td>
<td>$\neg(A \land \neg B)$</td>
</tr>
<tr>
<td><strong>suppose:</strong></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$\neg B$</td>
</tr>
<tr>
<td>4</td>
<td>$A \land \neg B$</td>
</tr>
<tr>
<td>5</td>
<td>$\neg(A \land \neg B)$</td>
</tr>
<tr>
<td>6</td>
<td>$\neg B$</td>
</tr>
<tr>
<td>7</td>
<td>$B$</td>
</tr>
</tbody>
</table>
7.1.9 Exercise (Modus Tollens)
For any formulas $A$ and $B$,

$$A, A \rightarrow \neg B \vdash \neg A$$

7.1.10 Exercise (Disjunction Introduction)
For any formulas $A$ and $B$,

$$A \vdash A \lor B$$
$$B \vdash A \lor B$$

(Recall that $A \lor B$ is officially an abbreviation for $\neg(\neg A \land \neg B)$.)

It’s also useful to show some relationships between different provability facts. For example:

7.1.11 Example (Conditional Proof)
If $X, A \vdash B$, then $X \vdash A \rightarrow B$.

(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 $(A \rightarrow B)$ which only relies on the premises $X$. Remember, $(A \rightarrow B)$ is officially an abbreviation for $\neg(A \land \neg B)$. So we can schematically put together a proof like this.

\[
\begin{align*}
suppose: & \\ A \land \neg B & \text{Assumption} \\
suppose: & \\ A & \text{Assumption} \\
P & \# \text{This is a proof of } B \text{ from } X \text{ and } A \\
\neg B & \land\text{Elim2 (2)} \\
\neg A & \land\text{Elim (2)} \\
A & \\
\neg(A \land \neg B) & \land\text{Elim1 (2)} \\
\end{align*}
\]

Notice in particular that this proof does not rely on the assumption $A$: this assumption is available within the inner $\land\text{Elim}$ subproof, but not outside of it. So this is a proof of $\neg(A \land \neg B)$ that relies on the premises $X$. \qed
7.1.12 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 add a line to our proof of the form \( a = a \), where \( a \) is any term. The pattern-following name for this is \( \text{Identity} \), and the traditional name is simply \( \text{Identity} \). (Feel free to use either one.)

\[
\text{Identity} : \vdash a = a
\]

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 proved both \( a = b \) and \( A(a) \), then we can deduce \( A(b) \). This is called either \( \text{=Elim} \) or \( \text{Leibniz’s Law} \).

\[
\text{=Elim} : a = b, A[x \mapsto a] \vdash A[x \mapsto b]
\]

7.1.13 Example

For any terms \( a \) and \( b \),

\[ a = b \vdash b = a \]

\[
\text{Proof} \\
1. a = b \quad \text{Assumption} \\
2. a = a \quad \text{Identity} \\
3. b = a \quad \text{Leibniz’s Law (1 and 2, using the formula } x = a) \\
\]

7.1.14 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 $A(a)$.

$$\forall Elim \ : \ \forall x \ A \vdash A[x \mapsto a]$$

7.1.15 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.

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$x = 0 + x$</td>
</tr>
<tr>
<td>2</td>
<td>$x &gt; 0$</td>
</tr>
<tr>
<td>3</td>
<td>$0 + x &gt; 0$</td>
</tr>
</tbody>
</table>

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 (with $y > 0$ as the formula $A(y)$). 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: 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.

<table>
<thead>
<tr>
<th></th>
<th>for arbitrary x:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>suppose:</td>
</tr>
<tr>
<td>2</td>
<td>∀y (y ∈ x ↔ ∼(y ∈ y))     Assumption</td>
</tr>
<tr>
<td>3</td>
<td>;  # This is where we derived a contradiction</td>
</tr>
<tr>
<td>4</td>
<td>∀x ∼∀y (y ∈ x ↔ ∼(y ∈ y))     Reductio</td>
</tr>
<tr>
<td>5</td>
<td>∀x ∼∀y (y ∈ x ↔ ∼(y ∈ y))     Universal Generalization</td>
</tr>
</tbody>
</table>

( Universal Generalization is the traditional name for this rule. The systematic name is ∀Intro.)

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 “from outside”.)

Here’s the general rule. Like the Reductio rule, we get to make an inference from one fact about provability to another. Given that we can prove A from premises that don’t say anything special about x, then we can also prove ∀x A using the rule of

---

1For example, see Breckenridge and Magidor (2012).
Universal Generalization. We can summarize the effect of the rule like this:

\[
\frac{X \vdash A(x)}{X \vdash \forall x \ A(x)} \quad \text{Universal Generalization}
\]

if \(x\) is not free in any formula in \(X\)

7.1.16 Exercise
(a) \(\vdash \top\).

(b) \(\bot \vdash A\), for any formula \(A\).

(Recall that \(\top\) is an abbreviation for the standard truth, \(\forall x \ (x = x)\), and \(\bot\) is an abbreviation for the standard falsehood \(\neg \top\).

7.1.17 Exercise (Change of Variables)
For any variables \(x\) and \(y\), and for any formula \(A(x)\) in which \(y\) does not occur free,

\[
\forall x \ A(x) \vdash \forall y \ A(y)
\]

7.1.18 Exercise (Existential Instantiation)
Suppose \(x\) is not free in \(B\) or in any formula in \(X\). Then,

If \(X, A(x) \vdash B\) then \(X, \exists x A(x) \vdash B\)

This fact corresponds to a kind of reasoning we’ve often used in our informal proofs. Suppose we know that there is some \(A\). Then we can “give it a name”—we suppose in particular that \(x\) is \(A\). 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 \(A\).

7.2 Official Syntax

Now that we have gone over the rules for putting together proofs informally, it’s time to give an official inductive definition. The informal bits and pieces are enough when we want to show particular things are provable. But the official inductive definition is important for proving things about all proofs, and in particular, for
(informally) proving things about everything that is (formally) provable. There are three main facts about provability that we can show from the inductive definition.

1. **Compactness.** No formal proof essentially relies on more than finitely 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 4. 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 7.5.)

A proof is working up to a main conclusion, but along the way it also establishes lots of intermediate results. It’s convenient for us to also count the intermediate results as things that the proof proves. So in general, a single proof \( P \) can prove more than one thing. We’ll use the notation \( P : X \vdash A \) to say that \( A \) is one of the things \( P \) proves, from the premises \( X \).

### 7.2.1 Definition

The relation *\( P \) proves \( A \) from premises \( X \)*, or \( P : X \vdash A \) for short, is defined inductively. We have eight simple proof rules, two complex proof rules, and two extra “structural” rules that tell us how to put the rules together.

1. Each simple rule corresponds to a one-step proof, as follows:

   \[
   \begin{align*}
   A & \quad \text{Assumption} & : & A \vdash A \\
   A \land B & \quad \land\text{Intro} & : & A, B \vdash A \land B \\
   A & \quad \land\text{Elim1} & : & A \land B \vdash A \\
   B & \quad \land\text{Elim2} & : & A \land B \vdash B \\
   A & \quad \neg\neg\text{Elim} & : & \neg\neg A \vdash A \\
   a=a & \quad \equiv\text{Intro} & : & \vdash a=a \\
   A[x \mapsto b] & \quad =\text{Elim} & : & a=b, A[x \mapsto a] \vdash A[x \mapsto b] \\
   A[x \mapsto a] & \quad \forall\text{Elim} & : & \forall x A \vdash A[x \mapsto a]
   \end{align*}
   \]

(We call these *simple proofs.*)
2. (Reductio) Suppose $P : X, A \vdash B$ and $P : X, A \vdash \neg B$. Then

\[
\begin{align*}
\text{suppose:} & \quad P \quad 
\neg A \quad \neg \text{Intro} \\
\end{align*}
\]

\[
\Rightarrow 
X \vdash \neg A
\]

3. (Universal Generalization) Suppose $P : X \vdash A$, and $x$ is not free in any formula in $X$. Then

\[
\begin{align*}
\text{for arbitrary } x: & \quad P \\
\forall x \ A \quad \text{Universal Generalization} \\
\end{align*}
\]

\[
\Rightarrow 
X \vdash \forall x \ A
\]

We also have a rule for sticking simple proofs together to make more complex proofs. The idea is that if we have a proof $P$ that proves all of the premises for another proof $Q$, then we can stick them together to make up a bigger proof. This bigger proof proves everything that either $P$ or $Q$ proves, but it only relies on $P$’s premises—since $P$ already took care of proving all of $Q$’s premises.

4. (Cut) Suppose:

\[
\begin{align*}
P & : X \vdash A_1 \\
& \vdots \\
P & : X \vdash A_n \\
Q & : A_1, \ldots, A_n \vdash B
\end{align*}
\]

In this case we say $P$ provides a context for $Q$. Then

\[
P \\
Q
\]

is a proof from the premises $X$. Call this proof $R$. Then in particular, $R : X \vdash B$, and also $R : X \vdash A$ for each $A$ such that $P : X \vdash A$.

The last part of the definition doesn’t correspond to any part of a proof, but rather it has to do with how we interpret proofs—what we treat a proof as showing. If we have used a proof to show that a conclusion $A$ follows from certain premises $X$, then this same proof also shows that $A$ follows from those premises $X$ plus some extra premises $Y$. (Classical logic doesn’t require that every premise actually “shows up” somewhere in the proof.) This is called the rule of Weakening.
5. (Weakening) If \( P : X \vdash A \), then \( P : X, Y \vdash A \), for any set of formulas \( Y \).

As in any inductive definition, we can say “that’s all”: if we don’t eventually reach \( P, X \), and \( A \) by applying these five rules, then it is not the case that \( P : X \vdash A \).

7.2.2 Definition
(a) We say \( P \) is a proof iff there are some \( X \) and \( A \) such that \( P : X \vdash A \).
(b) We say \( A \) is provable from \( X \) (abbreviated \( X \vdash A \)) iff there is some proof \( P \) such that \( P : X \vdash A \).

We have spelled out a definition of how proofs are put together, and also what a proof with any particular structure 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 inductive definition of proofs (Definition 7.2.1) just in terms of what it tells us about what is provable, leaving out the details of what the proof that proves it happens to look like. This straightforwardly follows from (Definition 7.2.1).

7.2.3 Proposition
For any set of formulas \( X \) and any formulas \( A \) and \( B \),

\[
\begin{align*}
A & \vdash A \\
A, B & \vdash A \land B \\
A \land B & \vdash A \\
A \land B & \vdash B \\
\neg A & \vdash a = a \\
\neg a = b, A[x \mapsto a] & \vdash A[x \mapsto b] \\
\forall x A & \vdash A[x \mapsto a] \\
X, A & \vdash B \\
X, A & \vdash \neg B \\
X & \vdash \neg A \\
\end{align*}
\]

if \( x \) is not free in any formula in \( X \)

\[
\begin{align*}
X & \vdash A_1 \\
\ldots & \\
X & \vdash A_n \\
A_1, \ldots, A_n & \vdash B \\
X & \vdash B \\
X & \vdash A \\
X, Y & \vdash A
\end{align*}
\]
We can use this perspective to give some more elegant proofs of provability facts, which abstract from the details of what a particular proof looks like.

### 7.2.4 Example

\[ X \vdash A \text{ iff } X, \neg A \vdash \bot \]

#### Proof

Suppose \( X \vdash A \). By Weakening, \( X, \neg A \vdash A \), and by Assumption and Weakening \( X, \neg A \vdash \neg A \). We showed earlier that by Explosion, \( A, \neg A \vdash B \) for any \( B \), so in particular \( A, \neg A \vdash \bot \). By Cut, \( X, \neg A \vdash \bot \).

For the other direction, suppose \( X, \neg A \vdash \bot \). Since everything is provable from \( \bot \) (Exercise 7.1.16), in particular \( \bot \vdash \neg \bot \). So, by Reductio, \( X \vdash \neg \neg A \). By \( \neg \neg \text{Elim} \), \( X \vdash A \).

Alternatively, we can present these two arguments using diagrams. Each line in the diagram corresponds to some fact we know about provability. (This diagrammatic style of argument is kind of elegant, but it is entirely optional.) For the first part:

\[
\begin{align*}
X & \vdash A \\
X, \neg A & \vdash A
\end{align*}
\]

\[
\begin{array}{c}
\text{Assumption} \\
\hline
\vdash A \vdash A \\
\hline
\neg A \vdash \neg A \\
\hline
\vdash A, \neg A \vdash \bot \\
\end{array}
\]

For the second part:

\[
\begin{align*}
X, \neg A & \vdash \bot \\
\bot & \vdash \bot
\end{align*}
\]

\[
\begin{array}{c}
\text{Exercise} \\
\hline
\vdash \bot \vdash \neg \bot \\
\hline
\neg A \vdash \neg A
\end{array}
\]

\[
\begin{array}{c}
\text{Cut} \\
\hline
X, \neg A \vdash \neg \bot \\
\hline
X \vdash \neg A
\end{array}
\]

\[
\begin{array}{c}
\text{Reductio} \\
\hline
X \vdash \neg \neg A
\end{array}
\]

\[
\begin{array}{c}
\text{\( \neg \neg \text{Elim} \)} \\
\hline
X \vdash A
\end{array}
\]

\[ \square \]

### 7.2.5 Exercise

The following are equivalent:

\[
\begin{align*}
X & \vdash \bot \\
X & \vdash A \text{ and } X \vdash \neg A \text{ for some } A \\
X & \vdash A \text{ for every } A
\end{align*}
\]

In Section 4.3 we defined “consistent” to mean “has a model”. For this section and the next, we’ll use a different definition of “consistent” instead.
7.2.6 Definition
A set $X$ is \textbf{inconsistent} iff $X \vdash \bot$. (Exercise 7.2.5 gives us two other equivalent ways of saying this.) Otherwise $X$ is \textbf{consistent}.

When we want to contrast the two meanings of “consistent”—this definition using proofs, and our earlier definition using models—we can distinguish \textit{proof-theoretic} consistency and \textit{model-theoretic} consistency. It is also common to call these \textit{syntactic} consistency and \textit{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.

In Proposition 7.2.3 we listed twelve principles about provability. Taken together, these twelve principles generate \textit{all} of the provability facts. Whenever $A$ is provable from $X$, we can show this using these twelve rules. This is because whenever $X \vdash A$, there is a proof $P : X \vdash A$, and this proof is built up by some finite combination of these twelve rules. We can make this more precise by stating yet another \textit{inductive property}. This one is a bit elaborate, because it has a part for each part of the definition of proofs.

Here’s the idea. An \textit{argument} (whether it is valid or not) consists of some premises and a conclusion. So, in general, let an \textit{argument} be simply an ordered pair $(X, A)$ of a set of formulas $X$ and a formula $A$. (Such a pair is also called a \textit{sequent}, from the Latin for “follows”.)\footnote{The notation $X \vdash A$ is also sometimes used for arbitrary sequents, but since this is a confusing double-use of the $\vdash$ symbol, we won’t use this notation in this text.} Call an argument $(X, A)$ \textbf{provable} iff $A$ is provable from $X$; that is, $X \vdash A$.

Suppose we want to show that every provable argument $(X, A)$, is \textit{nice}. We can show this in twelve steps. First, we show that if $X$ and $A$ fit the pattern of any one of the simple proof rules, then $(X, A)$ is nice. That is, we start by showing that each argument of the form

$$(\{A\}, A)$$

is nice (for \textit{Assumption}), next that each argument

$$(\{A, B\}, A \& B)$$

...
is nice (for $\land$Intro), and so on. There are eight steps like this, one for each simple proof rule. Next we show that each of the two complex proof rules—Reductio and Universal Generalization—preserves niceness. Finally we show that the “structural rules” Cut and Weakening also preserve niceness. Given all this, it follows that every argument with a formal proof is nice.

Here’s what this looks like when we spell it out officially.

### 7.2.7 The Inductive Property of Provability

Suppose that $S$ is a set of pairs $(X, A)$ where $X$ is a set of formulas and $A$ is a formula. Suppose also that $S$ has the following twelve properties:

1. (Assumption) For any formula $A$,
   $$(\{A\}, A) \in S$$

2. ($\land$Intro) For any formulas $A$ and $B$,
   $$({A, B}, A \land B) \in S$$

You should be able to fill in properties 3–7 yourself, by looking at the corresponding simple proof rules in Definition 7.2.1.

8. (Universal Instantiation) For any formula $A$, variable $x$, and term $a$,
   $$({\forall x \ A, A[x \mapsto a]}, a) \in S$$

9. (Reductio) For any set of formulas $X$ and any formulas $A$ and $B$, suppose:
   $$(X \cup \{A\}, B) \in S$$
   $$(X \cup \{A\}, \lnot B) \in S$$

   Then:
   $$(X, \lnot A) \in S$$

10. (Universal Generalization) For any set of formulas $X$ and any variable $x$ which is not free in any formula in $X$, if $(X, A) \in S$, then
    $$(X, \forall x \ A) \in S$$
11. (Cut) For any set of formulas $X$ and any formulas $A_1, \ldots, A_n$ and $B$, suppose:

$$(X, A_1) \in S$$

$$\vdots$$

$$(X, A_n) \in S$$

$$(\{A_1, \ldots, A_n\}, B) \in S$$

Then $(X, B) \in S$.

12. (Weakening) For any sets of formulas $X$ and $Y$ and any formula $A$, if $(X, A) \in S$, then $(X \cup Y, A) \in S$.

If these twelve conditions all hold, then $S$ contains all pairs $(X, A)$ such that $A$ is provable from $X$.

7.2.8 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 bit trickier than you might expect.

**Proof**

We will prove by induction that every pair $(X, A)$ such that $X \vdash A$ has the following property:

There is some finite subset $X_0 \subseteq X$ such that $X_0 \vdash A$.

Call a pair $(X, A)$ with this property *compact*. The proof has twelve parts. But many of them are very similar to each other.

1. (Assumption) Consider any pair of the form $(\{A\}, A)$. Since $\{A\}$ itself is a finite subset of $\{A\}$ such that $\{A\} \vdash A$, this pair is clearly compact.

2. (AIntro) Similarly, since $\{A, B\}$ is finite and $\{A, B\} \vdash A \land B$, any pair of the form $(\{A, B\}, A \land B)$ is compact.

Things go exactly the same way for steps 3–8, since each of these proof rules only involves finitely many premises.
9. (Reductio) This step is more complicated. For this step, we want to show that, for any set of formulas \( X \) and formulas \( A \) and \( B \), if \( (X \cup \{A\}, B) \) and \( (X \cup \{A\}, \neg B) \) are both compact, then \( (X, \neg A) \) is also compact. So we can suppose this for our inductive hypothesis:

There is a finite subset \( X_0 \subseteq X \cup \{A\} \) such that \( X_0 \vdash B \), and there is a finite subset \( Y_0 \subseteq X \cup \{A\} \) such that \( Y_0 \vdash \neg B \).

We want to prove that there is a finite subset of \( X \) from which \( \neg A \) is provable. Notice that \( X_0 - \{A\} \) and \( Y_0 - \{A\} \) are both finite subsets of \( X \). So

\[
Z_0 = (X_0 - \{A\}) \cup (Y_0 - \{A\})
\]

is another finite subset of \( X_0 \). Furthermore, \( Z_0 \cup \{A\} \) extends both \( X_0 \) and \( Y_0 \). So by Weakening,

\[
Z_0, A \vdash B \\
Z_0, A \vdash \neg B
\]

Then by Reductio,

\[
Z_0 \vdash \neg A
\]

That is, \( Z_0 \) is a finite subset of \( X \) such that \( Z_0 \vdash \neg A \), which is what we wanted.

10. (Universal Generalization) Let \( X \) be a set of formulas, let \( A \) be a formula, and let \( x \) be a variable which is not free in any formula in \( X \). We can suppose for our inductive hypothesis that \( (X, A) \) is compact: that is,

There is a finite subset \( X_0 \subseteq X \) such that \( X_0 \vdash A \).

Notice that this means \( x \) is not free in any formula in \( X_0 \), either. So by Universal Generalization,

\[
X_0 \vdash \forall x \ A
\]

This is just what we wanted to show for this step: the pair \( (X, \forall x \ A) \) is also compact.

11. (Cut) For this step, our inductive hypothesis says that each of the pairs \( (X, A_1), \ldots, (X, A_n) \) and \( (\{A_1, \ldots, A_n\}, B) \) is compact. That is to say, there are finite subsets \( X_1 \subseteq X, \ldots, X_n \subseteq X \) and \( Y \subseteq \{A_1, \ldots, A_n\} \) such that:

\[
X_1 \vdash A_1 \\
\vdots \\
X_n \vdash A_n \\
Y \vdash B
\]
Let \( Z = X_1 \cup \cdots \cup X_n \), which is another finite subset of \( X \), which extends each of \( X_1, \ldots, X_n \). So by Weakening,

\[
\begin{align*}
Z \vdash A_1 \\
\vdots \\
Z \vdash A_n \\
A_1, \ldots, A_n \vdash B
\end{align*}
\]

Then by the Cut rule,

\( Z \vdash B \)

This is what we wanted to show for this step.

12. (Weakening) The last step is an easy one. Finally, suppose for our inductive hypothesis that \((X, A)\) is compact:

There is a finite subset \( X_0 \subseteq X \) such that \( X_0 \vdash A \).

Then \( X_0 \) is also a finite subset of \( X \cup Y \), so it immediately follows that \((X \cup Y, A)\) is also compact.

\( \square \)

### 7.2.9 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).

### 7.2.10 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:

If \( X \vdash A \) then \( X \models A \)

**Proof Sketch**

We will prove by induction that every pair \((X, A)\) such that \( X \vdash A \) has the following property:

\( X \models A \)

It will be helpful to refer back to some facts about logical consequence that we showed back in Section 4.3.
1. (Assumption) For this step, we need to show that $A \models A$. This is clearly true: $A$ is true in every model of $\{A\}$.

2. (∧Intro) We showed that $A, B \models A \land B$ in Section 4.3.

Checking steps 3–8 for the remaining simple rules (\land Elim1, \land Elim2, \neg\neg Elim, =Intro, =Elim, and Universal Instantiation) is left as an exercise.

9. (Reductio) For this step, we want to show that if

$$X, A \models B$$
$$X, A \models \neg B$$

then

$$X \models \neg A$$

We showed this in Section 4.3 as well.

Step 10, Universal Generalization, is also left as an exercise.

11. (Cut) Suppose for our inductive hypothesis:

$$X \models A_1$$
$$\vdots$$
$$X \models A_n$$

$A_1, \ldots, A_n \models B$

We want to show $X \models B$. This is left as an exercise.

Step 12 (Weakening) is another exercise.

\[\square\]

7.2.11 Exercise
Fill in the remaining steps of the proof of the Soundness Theorem, using facts about logical consequence from Section 4.3.
7.2.12 Exercise
If \(X\) has a model, then \(X\) is proof-theoretically consistent: that is, \(X \not\vdash \bot\).

7.3 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 \(R_{ab}\). 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 \(R_{ab}\) or of the form \(a\equiv b\) is called an atomic formula.)
7.3. THE COMPLETENESS THEOREM

• **Stage 3.** Next, we’ll allow \( X \) to include formulas with the other logical connectives (\( \neg \), \( \land \), 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 satisfies \( A(x) \)—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!
7.3.1 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.

7.3.2 Exercise
Let $X$ be a set of relational formulas, and let $S$ and $g$ be the structure and assignment from Definition 7.3.1.

(a) Every $L$-term $a$ denotes itself: that is, $\llbracket a \rrbracket_Sg = a$.

(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 $\text{suc } 0 = \text{suc } 0 + 0$. Notice that the simple model from Stage 1 definitely won’t satisfy this sentence. On the “linguistic” interpretation, $\text{suc } 0$ denotes itself, the term $\text{suc } 0$, while $\text{suc } 0 + 0$ denotes the term $\text{suc } 0 + 0$, and these two terms are different. So on the “self-referential” Stage 1 interpretation, $\text{suc } 0 = \text{suc } 0 + 0$ 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 \( a \equiv b \).

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.

### 7.3.3 Definition

Let \( X \) be a set of \( L \)-formulas.

1. Terms \( a \) and \( b \) are equivalent given \( X \) iff \( X \models a \equiv b \).
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,

\[
E(a) = \{ b \in L\text{-terms} \mid X \models a \equiv b \}
\]

(\( E \) is a function that takes each \( L \)-term to a set of \( L \)-terms.)

### 7.3.4 Exercise

For any \( L \)-terms \( a \) and \( b \), \( E(a) = E(b) \) iff \( a \) and \( b \) are equivalent in \( X \).

### 7.3.5 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.)

### 7.3.6 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 $E(x)$.

### 7.3.7 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:

$$\llbracket a \rrbracket_{Sg} = E(a)$$

(b) For any two-place relation symbol $R$ and any $L$-terms $a$ and $b$,

$$(S, g) \text{ satisfies } Rab \iff X \vdash Rab$$

(c) For any $L$-terms $a$ and $b$,

$$(S, g) \text{ satisfies } (a=b) \iff X \vdash (a=b)$$

### 7.3.8 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 \( X = \{ \exists x \ (f(x) = c) \} \). 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 \( \{ f(t) \} \). This function doesn’t map anything to the singleton set \( \{ c \} \). So if we construct the canonical model for \( X \) in the same way as Stage 2, the existential claim \( \exists x \ (f(x) = c) \) 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.

7.3.9 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 4.3.)

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 \( \forall x \neg A(x) \in X \).

7.3.10 Exercise
If \( X \) is consistent and negation-complete, then \( X \vdash A \) iff \( A \in X \).

7.3.11 Exercise
Suppose that \( X \) is consistent, negation-complete, and witness-complete.

(a) \( \neg A \in X \) iff \( A \notin X \).

(b) \( A \land B \in X \) iff \( A \in X \) and \( B \in X \).

(c) \( \forall x A(x) \in X \) iff for every term \( t \), \( A(t) \in X \).

7.3.12 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 7.3.6). For any formula $A$,

$$(S, g) \text{ satisfies } A \iff A \in X$$

*Hint.* Use induction on the complexity of $A$. Exercise 7.3.7 and Exercise 7.3.11 will help.

### 7.3.13 Lemma

Suppose $X$ is a consistent, negation-complete, and witness-complete set of formulas. Then $X$ has a model.

*Proof*

By Exercise 7.3.12, the canonical model for the set of atomic formulas in $X$ is a model of $X$. \(\square\)

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.

#### 7.3.14 Exercise

If $X \cup \{A\}$ is inconsistent, and $X \cup \{\neg A\}$ is inconsistent, then $X$ is inconsistent.

#### 7.3.15 Exercise

If $X_0 \subseteq X_1 \subseteq X_2 \subseteq \ldots$ 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 $X_n$.

#### 7.3.16 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 7.3.14: 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 \( X_0 \subseteq X_1 \subseteq \ldots \). Thus, by Exercise 7.3.15, 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 \( X \).

7.3.17 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.

7.3.18 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.\(^3\)

\(^3\)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.
The proof is very similar to Lemma 7.3.16. 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 \( A_0(x), \ldots, A_n(x) \), and which is distinct from each of the earlier variables \( y_0, \ldots, y_n \). 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 7.3.17: 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 \( X \).

\[\square\]

7.3.19 Exercise (Henkin’s Lemma)
If \( X \) is a consistent set of sentences, then \( X \) has a model.

\textit{Hint.} Put the previous three lemmas together (in the right order).

7.3.20 Exercise (The Completeness Theorem)
If \( X \models A \), then \( X \vdash A \).

7.3.21 Exercise (The Compactness Theorem)
If \( X \models A \), then there is a finite subset \( X_0 \subseteq X \) such that \( X_0 \notmodels A \).

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.

**7.3.22 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:

**7.3.23 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:

**7.3.24 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.

### 7.4 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.)

**7.4.1 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:

\[ 0 \ \text{suc} \ 0 \ \text{suc suc} \ 0 \ \ldots \]

7.4.2 Exercise

Consider the signature of the language of arithmetic with one additional constant symbol \( c \). The theory

\[ \text{Th} \mathbb{N} \cup \{ c \neq 0, c \neq 1, c \neq 2, \ldots \} \]

has a model.

7.4.3 Exercise

There is a non-standard model of arithmetic: that is, there is a structure which is a model of \( \text{Th} \mathbb{N} \) and which is not isomorphic to the standard model \( \mathbb{N} \).

TODO. Discuss the gap between the induction schema and the Inductive Principle.

7.5 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 4.4) 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 $A$,

$$A \in T \iff X \vdash A$$

### 7.5.1 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.

### 7.5.2 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 7.2.1**.

(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 6.6**.

### 7.5.3 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.

### 7.5.4 Exercise

(a) Any effectively axiomatizable theory is effectively enumerable.

(b) The set of logical truths is effectively enumerable.

### 7.5.5 Exercise

If $X$ is a set of sentences which is effectively enumerable, consistent, and negation-complete, then $X$ is decidable.
7.5.6 Exercise (Gödel’s First Incompleteness Theorem)
No theory is sufficiently strong, axiomatizable, consistent, and complete.

7.5.7 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 Th S.
(b) The theory of arithmetic Th 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.

7.6 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. In several respects, we actually proved a bit more

4In 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
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
enumerate. 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 $T$:

$$
T \vdash \text{Prov}_T(A) \quad \text{if} \quad T \vdash A \\
T \vdash \neg \text{Prov}_T(A) \quad \text{otherwise}
$$

Then, by Gödel’s Fixed Point Theorem, we have a sentence $G$ which is equivalent
(in $T$) to $\neg \text{Prov}_T(G)$. 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 $\text{Prov}_T(x)$ that represents $T$ within $T$. This
is exactly what Tarski’s Theorem (Exercise 5.7.5) 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 $\text{Proof}_T(x, y)$.

\[
\begin{align*}
T \vdash \text{Proof}_T(P)(A) & \quad \text{if } P \text{ is a proof of } A \text{ from } T'\text{’s axioms} \\
T \vdash \neg \text{Proof}_T(P)(A) & \quad \text{otherwise}
\end{align*}
\]

Now consider the formula $\exists x \, \text{Proof}_T(x, y)$. It’s customary to call this formula $\text{Prov}_T(y)$—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 ??. It’s analogous to semi-decidability.)

### 7.6.1 Exercise

Suppose that $\text{Proof}_T(x, y)$ represents the $T$-proof relation in $T$. Let $\text{Prov}_T(y)$ be $\exists x \, \text{Proof}_T(x, y)$.

(a) For any sentence $A$, if $T \vdash A$, then $T \vdash \text{Prov}_T(A)$.

(b) Suppose furthermore that the theory $T$ is true in the standard string structure $\mathcal{S}$. In that case, if $T \nvdash A$, then $T \nvdash \text{Prov}_T(A)$.

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.

### 7.6.2 Definition

Let $T$ be a sufficiently strong, effectively axiomatizable theory. Let $\text{Proof}_T(x, y)$ be a formula that represents the $T$-proof relation in $T$. (There is such a formula, by the Representability Theorem (Principle 6.7.8).) Let $\text{Prov}_T(y)$ be the provability formula $\exists x \, \text{Proof}_T(x, y)$.

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

\[
G_T \equiv \mathcal{T} \neg \text{Prov}_T(G_T)
\]

### 7.6.3 Lemma

Any sufficiently strong, effectively axiomatizable theory $T$ has a Gödel sentence $G_T$. 
7.6. GÖDEL SENTENCES

Proof
This immediately follows from Gödel’s Fixed Point Theorem (Exercise 5.5.11). □

7.6.4 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 $S$, 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.

7.6.5 Definition
A theory $T$ is $\omega$-inconsistent (pronounced “omega inconsistent”) iff there is some formula $A(x)$ such that

(a) $T \vdash \exists x A(x)$
(b) For every string $s$, $T \vdash \neg A(s)$.

If there is no such formula, then $T$ is $\omega$-consistent.

7.6.6 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 $\omega$-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 $\omega$-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.)

### 7.7 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 $\omega$-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 $\omega$-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.

### 7.8 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 7.6. 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*
7.8. CONSISTENCY IS UNPROVABLE

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 7.6, there is a formula $\text{Proof}_T(x, y)$ such that

$$
T \vdash \text{Proof}_T(P)\langle A \rangle \quad \text{if } P \text{ is a proof of } A \text{ from } X
$$

$$
T \vdash \neg \text{Proof}_T(P)\langle A \rangle \quad \text{otherwise}
$$

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 $\text{Provet}(y)$ be $\exists x \text{ Proof}(x, y)$. 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 $T \vdash \text{Proof}_T(P)\langle A \rangle$, and thus by existential generalization, $T \vdash \text{Provet}(A)$. In short, for any sentence $A$,

If $T \vdash A$ then $T \vdash \text{Provet}(A)$

But, to reiterate, we don’t get the other half of the definition of representability: if $T$ 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 $T$.

7.8.1 Definition

The consistency sentence for $T$ is the sentence $\neg \text{Provet}(\bot)$. This is abbreviated $\text{Con}_T$. That is, to spell this out, $\text{Con}_T$ is the sentence

$$
\neg \exists x \text{ Proof}_T(x, \langle \bot \rangle)
$$

where $\text{Proof}_T(x, y)$ 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 $\text{Proof}_T(x, y)$ 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:

$$\text{If } T \vdash \text{Con}_T \text{ then } T \vdash \bot$$

For the first step, recall from Section 7.6 that any sufficiently strong, effectively axiomatizable theory $T$ has a Gödel sentence $G_T$, such that

$$T \vdash G_T \leftrightarrow \neg \text{Pr} \text{ov}_T(G)$$

Recall also from Exercise 7.6.4 that if $T$ is consistent, then $T$ does not prove its own Gödel sentence. Putting that the other way around:

$$\text{If } T \vdash G_T \text{ then } T \vdash \bot$$

(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:

$$\text{If } T \vdash A \rightarrow B \text{ and } T \vdash A \text{ then } T \vdash B$$

$$\text{If } T \vdash A \text{ then } T \vdash \text{Pr} \text{ov}_T(A)$$

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.

7.8.2 Definition

A theory $T$ satisfies the derivability conditions iff

$$T \vdash \text{Pr} \text{ov}(A \rightarrow B) \rightarrow \text{Pr} \text{ov}(A) \rightarrow \text{Pr} \text{ov}(B)$$

$$T \vdash \text{Pr} \text{ov}(A) \rightarrow \text{Pr} \text{ov} \text{(Pr} \text{ov}(A))$$

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
7.8. CONSISTENCY IS UNPROVABLE

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 \( \text{Proof}(x, y) \). I’m ignoring some complications here.) But here’s one important example: first-order Peano Arithmetic PA satisfies the derivability conditions.

7.8.3 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 \( \square A \) as an abbreviation for the sentence \( \text{Prov}(A) \). Using box notation, we can summarize the key facts about provability more concisely like this:

\[
T \vdash G \quad \rightarrow \\
\text{If } T \vdash G \text{ then } T \vdash \bot
\]

\[
T \vdash A \rightarrow B \quad \text{and} \quad T \vdash A \quad \text{then} \quad T \vdash B
\]

\[
T \vdash \square(A \rightarrow B) \rightarrow \square A \rightarrow \square B
\]

\[
T \vdash A \quad \text{then} \quad T \vdash \square A
\]

\[
T \vdash \square A \rightarrow \square \square A
\]

We can also rewrite the consistency sentence \( \text{Con}_T \) as \( \neg \square \bot \).

7.8.4 Exercise

Here is a pretty basic logical fact: for any sentence \( A \),

\[
\text{If } T \vdash A \quad \text{and} \quad T \vdash \neg A \quad \text{then} \quad T \text{ is inconsistent}
\]

Use the facts about provability to show that \( T \) “knows” this fact. That is:

\[
T \vdash \text{Prov}(A) \rightarrow \text{Prov}(\neg A) \rightarrow \text{Prov}(\bot)
\]

In box notation:

\[
T \vdash \square A \rightarrow \square \neg A \rightarrow \square \bot
\]

7.8.5 Exercise

We have already proved this fact (Exercise 7.6.4 (a)):

\[
\text{If } T \vdash G \quad \text{then} \quad T \vdash \bot
\]
In this exercise, we’ll show that the proof of this fact can be carried out within $T$.

(a) $T \vdash \square G \boxed{\rightarrow} \square \neg \neg \square G$

(b) $T \vdash \square G \boxed{\rightarrow} \square \bot$

7.8.6 Exercise (Gödel’s Second Incompleteness Theorem)

Use the previous exercise and Exercise 7.6.4 (a) to show that if $T$ proves the consistency sentence for $T$, then $T$ is inconsistent. That is:

If $T \vdash \text{Con}_T$ then $T \vdash \bot$

Or in other words:

If $T \vdash \neg \square \bot$ then $T \vdash \bot$
Chapter 8

Second-Order Logic*

1. The idea of second-order logic
2. Semantics for second-order logic
3. Second-order Peano Arithmetic (PA₂) does not have non-standard models
4. Thus PA₂ is negation-complete
5. Thus PA₂ is not effectively enumerable
6. Thus second-order logic has no sound and complete proof system
7. Second-order logic is not compact
8. Type theory
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. (Thus ZFC does not prove there are large cardinals.)
5. Some independence results (stated without proof): large cardinals, the Continuum Hypothesis.
7. ZFC$_2$ does not have countable models; categoricity
CHAPTER 9. SET THEORY*

