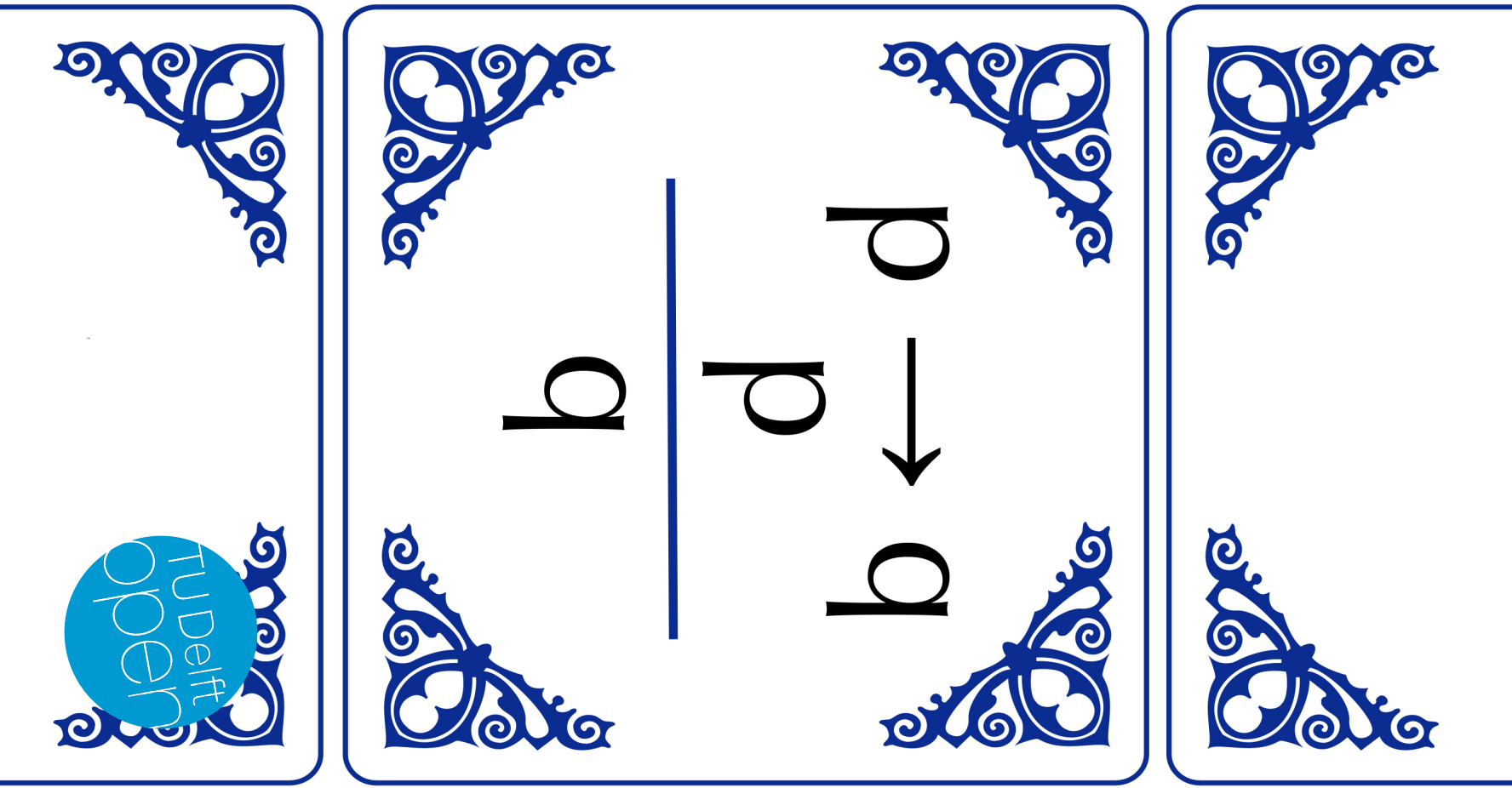
Delftse Foundations of Computation

Stefan Hugtenburg and Neil Yorke-Smith



TU Delft open

Delftse Foundations of Computation

First TU Delft Edition (TUD-1.1, August 2019)

Stefan Hugtenburg and **Neil Yorke-Smith**

Delftse Foundations of Computation

First TU Delft Edition (TUD-1.1, August 2019)

Stefan Hugtenburg and **Neil Yorke-Smith**
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology
Delft, The Netherlands
{s.hugtenburg,n.yorke-smith}@tudelft.nl

Delftse Foundations of Computation is a textbook for a one quarter introductory course in theoretical computer science. It includes topics from propositional and predicate logic, proof techniques, set theory and the theory of computation, along with practical applications to computer science. It has no prerequisites other than a general familiarity with computer programming.

This book is derived from *Foundations of Computation* by Carol Critchlow and David Eck, Version 2.3 (Summer 2011), which is licensed under CC BY-NC-SA 4.0. Critchlow and Eck are not associated with the TU Delft editions. This book also uses some material from Wikipedia (English) (`en.wikipedia.org`), which is licensed under CC BY-SA 3.0. The authors of the TU Delft editions are responsible for any errors, and welcome bug reports and suggestions by email or in person.

Thanks to M. de Jong, T. Klos, F. Mulder, H. Tonino and E. Walraven.

The latest edition of this book is available for online use and for free download from the TU Delft Open Textbook repository at `textbooks.open.tudelft.nl`. The original (non-TU Delft) version contains additional material, and is available at `math.hws.edu/FoundationsOfComputation/`.

Typeset in TEX Gyre Pagella

Published by Delft University of Technology, Delft, The Netherlands, August 2019

Version TUD-1.1

ISBN 978-94-6366-083-9 | DOI 10.5074/t.isbn.9789463660839

To the students of CSE1300

Contents

Chapter 1

Introduction

Logic originally meant 'the word' or 'what is spoken' in Ancient Greece, and today means 'thought' or 'reason'. As a subject, logic is concerned with the most general laws of truth. Why study this kind of reasoning in computer science?

Logic is important because digital computers work with precision, and because designing algorithms requires precision, and because comparing algorithms requires precision.

Even when when a computer is, seemingly, computing with vague or imprecise quantities, the underlying computation is precise.[1] For example, when a deep neural network is being trained to recognise cats, the algorithm being used to train the network is specified precisely. More than this, the criteria we use to assess whether the network has learned well enough are also specified precisely. And any theoretical properties about the algorithm have been proven precisely.

Reasoning and logic, and related mathematical concepts such as sets, are foundational for computer science. A third of your first year TUDelft CSE curriculum is mathematics: *Reasoning & Logic*, *Calculus*, *Linear Algebra* and *Probability Theory & Statistics*.

As a computer scientist, you have to be capable of solving complex problems. One important aspect is to be able to come to the right conclusions. On the basis of theorems and partial observations you can acquire more knowledge and evidence to help prove that a specific conclusion is mathematically and logically correct. You learn how to do this with the course *Reasoning & Logic*.

The foundational mathematical skills you learn in *Reasoning & Logic* are used in all the other mathematics courses you will take, and in *Computer Organisation*, *Algorithms & Data Structures*, *Information & Data Management*, *Machine Learning*, and many other courses. In fact, logic is studied and used not only in mathematics and computer science, but also in philosophy (since Ancient Greece) and today in fields such as linguistics and psychology.

[1] You can take a course on quantum computing to learn whether they are an exception.

1

This book is designed to help you achieve the learning goals of *Reasoning & Logic*:

1. Translate a logically-precise claim to and from natural language.
2. Describe the operation of logical connectors and quantifiers.
3. Describe the notion of logical validity.
4. Explain and apply basic set operations.
5. Define and perform computations with functions, relations and equivalence classes.
6. Construct and interpret recursive definitions.
7. Construct an appropriate function or relation given a description (in natural language or formal notation).
8. Construct a direct or indirect proof (by contradiction, division into cases, generalisation, or [structural] induction) or logical equivalence—or counterexample for (in)valid arguments—in propositional logic, predicate logic and set theory.
9. Identify what type of proof is appropriate for a given claim.
10. Solve simple Boolean Satisfiability (SAT) instances.
11. Develop specifications for verification tools like SAT or SMT solvers.
12. Interpret the response of verification tools like SAT or SMT solvers.

We do not cover every topic at the same level of detail. Some topics, such as SAT and SMT solvers, we do not cover at all. Further, the lectures will not cover everything in the book. Some topics in the lectures you will prepare for using other materials: these will be announced.

Starred sections in the contents of this book are not included in the syllabus for *Reasoning & Logic*.

We include solutions to some of the exercises, starting on page 163. Exercises that have a solution are marked with a dagger (†) symbol.

1

The theme of the book is about coming to the right conclusion: proving the logical validity of *arguments*. What is a valid argument? When is an argument logically valid and when is it not? How can we determine whether an argument is logically valid? How can we derive a logically valid conclusion from the premises? Or how can we prove that a conclusion is not a logical consequence of the premises?

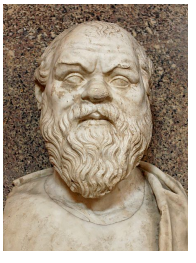
We will begin by talking further about logic.

Chapter 2

Logic

In a sense, we know a lot more than we realise, because everything that we know has consequences—*logical* consequences—that follow automatically. If you know that all humans are mortal, and you know that Socrates is human, then in a sense you know that Socrates is mortal, whether or not you have ever considered or wanted to consider that fact. This is an example of ***logical deduction***: from the ***premises*** that "All humans are mortal" and "Socrates is human", the ***conclusion*** that "Socrates is mortal" can be deduced by logic.

Socrates was a Greek philosopher who suffered a most unfortunate fate. In one of the most famous mathematical arguments—the one outlined above—he is the man that is destined to die. History has since taught us we were right as Socrates died after a long life (71 years), sentenced to death for corrupting the minds of the youths of Athens. His disciple Plato wrote many Socratic dialogues which give an insight into the philosophy of Socrates, often summarised as: "I know that I know nothing". The last words of this mortal man were (according to Plato): "Crito, we owe a cock to Asclepius: pay it and do not neglect it." The fate of the chicken is unknown...



Source: `en.wikipedia.org/wiki/Socrates`.

Logical deduction is a kind of computation. By applying rules of logic to a given set

of premises, conclusions that follow from those premises can be generated automatically. This computational process could for instance be carried out by a computer. Once you know the premises, or are willing to accept them for the sake of argument, you are forced *by logic* to accept the conclusions. Still, to say that you 'know' those conclusions would be misleading. The problem is that there are too many of them (infinitely many), and, in general, most of them are not particularly interesting. Until you have actually made the deduction, you don't *really* know the conclusion, and knowing which of the possible chains of deduction to follow is not easy. The *art* of logic is to find an interesting conclusion and a chain of logical deductions that leads from the premises to that conclusion. Checking that the deductions are valid is the mechanical, computational side of logic.

Later in *Reasoning & Logic*, you will see some automated computational techniques that can help us checking the deductions. I don't cover these in this edition of this book. In fact, there are *proof assistants* that can even help us finding interesting conclusions.

This chapter is mostly about the mechanics of logic. We will investigate logic as a branch of mathematics, with its own symbols, formulas and rules of computation. Your objective is to learn the rules of logic, to understand why they are valid, and to develop skill in applying them. As with any branch of mathematics, there is a certain beauty to the symbols and formulas themselves. But it is the applications that bring the subject to life for most people. We will, of course, cover some applications as we go along. In a sense, though, the real applications of logic include much of computer science and of mathematics itself.

Among the fundamental elements of thought, and therefore of logic, are propositions. A ***proposition*** is a statement that has a truth value: It is either true or false. "Delft is a city" and "$2 + 2 = 42$" are propositions. In the first part of this chapter, we will study ***propositional logic***, which takes propositions and considers how they can be combined and manipulated. This branch of logic has surprising application to the design of the electronic circuits that make up computers. This ties closely to the digital and boolean logic you will study in your course *Computer Organisation*.

Logic gets more interesting when we consider the internal structure of propositions. In English, a proposition is expressed as a sentence, and, as you know from studying grammar, sentences have parts. A simple sentence like "Delft is a city" has a ***subject*** and a ***predicate***. The sentence says something about its subject. The subject of "Delft is a city" is Delft. The sentence says something about Delft. The *something* that the sentence says about its subject is the predicate. In the example, the predicate is the phrase 'is a city'. Once we start working with predicates, we can create propositions using ***quantifiers*** like 'all', 'some', and 'no'. For example, working with the predicate 'has a university' we can move from simple propositions like "Delft has a university" to "All cities have a

university" or to "No city has a university" or to the rather more realistic "Some cities have a university".

Logical deduction usually deals with quantified statements, as shown by the basic example of human mortality with which we began this chapter. Logical deduction will be a major topic of this chapter; and under the name of ***proof***, it will be the topic of the next chapter and a major tool for the rest of this book.

2.1 Propositional Logic

We humans use a ***natural language*** when we speak, such as Dutch, English or Flemish. Natural languages are ambiguous and often vague. To start modelling them we first consider ***propositional logic***. This form of logic is arguably the easiest to work with, but also has limited expressive power. However even with this form we can already encapsulate many arguments and power a number of applications, for instance digital logic in chip design.

2.1.1 Propositions

A proposition is a statement which is either true or false. In propositional logic, we reason only about propositions and see what we can do with them. Since this is mathematics, we need to be able to talk about propositions without saying which particular propositions we are talking about, so we use symbolic names to represent them. We will always use lowercase letters such as p, q, and r to represent propositions. A letter used in this way is called a ***propositional variable***. Remember that when I say something like "Let p be a proposition", I mean "For the rest of this discussion, let the symbol p stand for some particular statement, which is either true or false (although I am not at the moment making any assumption about which it is)." The discussion has ***mathematical generality*** in that p can represent any statement, and the discussion will be valid no matter which statement it represents.

Propositional variables are a little bit like variables in a programming language such as Java. A basic Java variable such as `int x` can take any integer value. I say there is "a little bit" of similarity between the two notions of variables—don't take the analogy too far at this point in your learning!

2.1.2 Logical operators

What we do with propositions is combine them with ***logical operators***, also referred to as ***logical connectives***. A logical operator can be applied to one or more propositions

to produce a new proposition. The truth value of the new proposition is completely determined by the operator and by the truth values of the propositions to which it is applied.[1] In English, logical operators are represented by words such as 'and', 'or', and 'not'. For example, the proposition "I wanted to leave and I left" is formed from two simpler propositions joined by the word 'and'. Adding the word 'not' to the proposition "I left" gives "I did not leave" (after a bit of necessary grammatical adjustment).

2

But English is a little too rich for mathematical logic. When you read the sentence "I wanted to leave and I left", you probably see a connotation of causality: I left *because* I wanted to leave. This implication does not follow from the logical combination of the truth values of the two propositions "I wanted to leave" and "I left". Or consider the proposition "I wanted to leave but I did not leave". Here, the word 'but' has the same *logical* meaning as the word 'and', but the connotation is very different. So, in mathematical logic, we use *symbols* to represent logical operators. These symbols do not carry any connotation beyond their defined logical meaning. The logical operators corresponding to the English words 'and', 'or', and 'not' are $\wedge$, $\vee$, and $\neg$.[2]

Definition 2.1. Let p and q be propositions. Then $p \vee q$, $p \wedge q$, and $\neg p$ are propositions, whose truth values are given by the rules:

- $p \wedge q$ is true when both p is true and q is true, and in no other case
- $p \vee q$ is true when either p is true, or q is true, or both p and q are true, and in no other case
- $\neg p$ is true when p is false, and in no other case

The operators $\wedge$, $\vee$, and $\neg$ are referred to as ***conjunction***, ***disjunction***, and ***negation***, respectively. (Note that $p \wedge q$ is read as 'p and q', $p \vee q$ is read as 'p or q', and $\neg p$ is read as 'not p'.)

Consider the statement "I am a CSE student or I am not a TPM student." Taking p to mean "I am a CSE student" and q to mean "I am a TPM student", you can write this as $p \vee \neg q$.

[1] It is not always true that the truth value of a sentence can be determined from the truth values of its component parts. For example, if p is a proposition, then 'Johan Cruyff believes p' is also a proposition, so 'Cruyff believes' is some kind of operator. However, it does not count as a *logical* operator because just from knowing whether or not p is true, we get no information at all about whether 'Johan Cruyff believes p' is true.

[2] Other textbooks might use different notations to represent a negation. For instance a bar over the variable $\bar{x}$ or a $\sim$ symbol. In Boolean algebra (and thus in your *Computer Organisation* course) you will also often find the $+$ symbol to represent an 'or' and a $\cdot$ (dot) symbol to represent an 'and'.

2.1.3 Precedence rules

These operators can be used in more complicated expressions, such as $p \wedge \neg q$ or $(p \vee q) \wedge (q \vee r)$. A proposition made up of simpler propositions and logical operators is called a ***compound proposition***. Just like in mathematics, parentheses can be used in compound expressions to indicate the order in which the operators are to be evaluated. In the absence of parentheses, the order of evaluation is determined by ***precedence rules***. For the logical operators defined above, the rules are that $\neg$ has higher precedence than $\wedge$, and $\wedge$ has precedence over $\vee$. This means that in the absence of parentheses, any $\neg$ operators are evaluated first, followed by any $\wedge$ operators, followed by any $\vee$ operators.

For example, the expression $\neg p \vee q \wedge r$ is equivalent to the expression $(\neg p) \vee (q \wedge r)$, while $p \vee q \wedge q \vee r$ is equivalent to $p \vee (q \wedge q) \vee r$.

This still leaves open the question of which of the $\wedge$ operators in the expression $p \wedge q \wedge r$ is evaluated first. This is settled by the following rule: When several operators of equal precedence occur in the absence of parentheses, they are evaluated from left to right. Thus, the expression $p \wedge q \wedge r$ is equivalent to $(p \wedge q) \wedge r$ rather than to $p \wedge (q \wedge r)$. In this particular case, as a matter of fact, it doesn't really matter which $\wedge$ operator is evaluated first, since the two compound propositions $(p \wedge q) \wedge r$ and $p \wedge (q \wedge r)$ always have the same value, no matter what logical values the component propositions p, q, and r have. We say that $\wedge$ is an ***associative*** operation. We'll see more about associativity and other properties of operations in the next section.

In practice however you should **always** add parentheses in places where ambiguity may arise. In fact some textbooks even add them to single operators as well, e.g., writing $(p \wedge q)$ instead of $p \wedge q$. Although for this course we do not require them around single operators, we should never need the precedence rules outlined above. Your parentheses should make clear the order of operations!

Every compound proposition has a ***main connective***. The main connective is the connective that is evaluated last, according to the precedence rules and parentheses. There should be no ambiguity over which is the main connective in a compound proposition.

2.1.4 Logical equivalence

Suppose we want to verify that, in fact, $(p \wedge q) \wedge r$ and $p \wedge (q \wedge r)$ do always have the same value. To do so, we have to consider all possible combinations of values of p, q, and r, and check that for all such combinations, the two compound expressions do indeed have the same value. It is convenient to organize this computation into a truth table. A ***truth table*** is a table that shows the value of one or more compound propositions for

p	q	r	$p \wedge q$	$q \wedge r$	$(p \wedge q) \wedge r$	$p \wedge (q \wedge r)$
0	0	0	0	0	0	0
0	0	1	0	0	0	0
0	1	0	0	0	0	0
0	1	1	0	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	0
1	1	0	1	0	0	0
1	1	1	1	1	1	1

Figure 2.1: *A truth table that demonstrates the logical equivalence of $(p \wedge q) \wedge r$ and $p \wedge (q \wedge r)$. The fact that the last two columns of this table are identical shows that these two expressions have the same value for all eight possible combinations of values of p, q, and r.*

each possible combination of values of the propositional variables that they contain. We call each such combination a ***situation***. Figure 2.1 is a truth table that compares the value of $(p \wedge q) \wedge r$ to the value of $p \wedge (q \wedge r)$ for all possible values of p, q, and r. There are eight rows in the table because there are exactly eight different ways in which truth values can be assigned to p, q, and r.[3] In this table, we see that the last two columns, representing the values of $(p \wedge q) \wedge r$ and $p \wedge (q \wedge r)$, are identical.

I discuss the creation of truth tables for statements written in propositional logic in more detail in one of the pencasts of this course: `youtu.be/oua_nvpFECQ`.

In another pencast of this course, we dicuss how you should formulate your answer when using truth tables to test for equivalence: `youtu.be/sWu0fUu7s5c`.

[3] In general, if there are n variables, then there are 2^n different ways to assign truth values to the variables, i.e., 2^n situations. This might become clear to you if you try to come up with a scheme for systematically listing all possible sets of values. As this should not satisfy you, you'll find a rigorous proof of the fact later in this chapter.

You can write the rows in a truth table in any order you like. I suggest you write them in a sorted order, as in Table 2.1. This helps you to be systematic in writing out the table. It also helps us to provide feedback on your answers!

More generally, we say that two compound propositions are ***logically equivalent*** if they always have the same value, no matter what truth values are assigned to the propositional variables that they contain. If the number of propositional variables is small, it is easy to use a truth table to check whether or not two propositions are logically equivalent.

When writing a piece of code you will often have your code make decisions. For instance in a bit of Java code—such as in your *Object-Oriented Programming* course—you might encounter an if-statement to check if the user has inputted the right type of data. Since the input you expect can be rather difficult, the if-statement is a complex combination of many simple checked chained together by `&&`'s and `||`'s. After taking a look at the code, you believe it can be simplified to a much smaller expression. Using a truth table you can prove that your simplified version is equivalent to the original.

2.1.5 More logical operaters

There are other logical operators besides $\wedge$, $\vee$, and $\neg$. We will consider the ***conditional operator***, $\rightarrow$, the ***biconditional operator***, $\leftrightarrow$, and the ***exclusive or operator***, $\oplus$.[4] These operators can be completely defined by a truth table that shows their values for the four possible combinations of truth values of p and q.

Definition 2.2. For any propositions p and q, we define the propositions $p \rightarrow q$, $p \leftrightarrow q$, and $p \oplus q$ according to the truth table:

p	q	$p \rightarrow q$	$p \leftrightarrow q$	$p \oplus q$
0	0	1	1	0
0	1	1	0	1
1	0	0	0	1
1	1	1	1	0

[4]Note that the symbols for these operations also differ from textbook to textbook. While $\rightarrow$ is fairly standard, $\leftrightarrow$ is sometimes represented by $\equiv$ or $\Leftrightarrow$. There is even less standardization of the exclusive or operator, but that operator is generally not so important as the others.

When these operators are used in expressions, in the absence of parentheses to indicate order of evaluation, we use the following precedence rules: The exclusive or operator, $\oplus$, has the same precedence as $\vee$. The conditional operator, $\rightarrow$, has lower precedence than $\wedge$, $\vee$, $\neg$, and $\oplus$, and is therefore evaluated after them. Finally, the biconditional operator, $\leftrightarrow$, has the lowest precedence and is therefore evaluated last. For example, the expression $p \rightarrow q \wedge r \leftrightarrow \neg p \oplus s$ is evaluated as if it were written $(p \rightarrow (q \wedge r)) \leftrightarrow ((\neg p) \oplus s)$. But again you should always include the parentheses!

In order to work effectively with the logical operators, you need to know more about their meaning and how they relate to ordinary English expressions. To that end we first consider the conditional operator in more detail in the next section.

2.1.6 Implications in English

The proposition $p \rightarrow q$ is called an ***implication*** or a ***conditional***. It is usually read as 'p implies q'. In such an implication p and q also get special names of their own. p is called the ***hypothesis*** or ***antecedent*** and q is called the ***conclusion*** or ***consequent***.

Furthermore we say that if the implication $p \rightarrow q$ holds, then p is ***sufficient*** for q. That is if p is true that is sufficient to also make q true. Conversely we say that q is ***necessary*** for p. Without q being true, it is impossible for p to be true. That is if q is false, then p also has to be false.

In English, $p \rightarrow q$ is often expressed as 'if p then q'. For example, if p represents the proposition "Karel Luyben is Rector Magnificus of TU Delft" and q represents "Prometheus is blessed by the gods", then $p \rightarrow q$ could be expressed in English as "If Karel Luyben is Rector Magnificus of TU Delft, then Prometheus is blessed by the gods." In this example, p is false and q is also false. Checking the definition of $p \rightarrow q$, we see that $p \rightarrow q$ is a true statement. Most people would agree with this, even though it is not immediately obvious.

The letter 'T' in the **TU**Delft logo bears a stylized flame on top, referring to the flame that Prometheus brought from Mount Olympus to the people, against the will of Zeus. Because of this, Prometheus is sometimes considered as the first engineer, and he is an important symbol for the university. His bronze statue stands in the Mekelpark at the centre of campus.

Source: `en.wikipedia.org/wiki/Delft_University_of_Technology`. Image: `weblog.library.tudelft.nl/2016/01/04/english-prometheus-is-back/`.

It is worth looking at a similar example in more detail. Suppose that I assert that "If Feyenoord is a great team, then I'm the King of the Netherlands". This statement has the form $m \rightarrow k$ where m is the proposition "Feyenoord is a great team" and k is the proposition "I'm the king of the Netherlands". Now, demonstrably I am not the king of the Netherlands, so k is false. Since k is false, the only way for $m \rightarrow k$ to be true is for m to be false as well. (Check the definition of $\rightarrow$ in the table, if you are not convinced!) So, by asserting $m \rightarrow k$, I am really asserting that the Feyenoord is *not* a great team.

Or consider the statement, "If the party is on Tuesday, then I'll be there." What am I trying to say if I assert this statement? I am asserting that $p \rightarrow q$ is true, where p represents "The party is on Tuesday" and q represents "I will be at the party". Suppose that p is true, that is, the party does in fact take place on Tuesday. Checking the definition of $\rightarrow$, we see that in the only case where p is true and $p \rightarrow q$ is true, q is also true. So from the truth of "If the party is on Tuesday, then I will be at the party" and "The party is in fact on Tuesday", you can deduce that "I will be at the party" is also true. But suppose, on the other hand, that the party is actually on Wednesday. Then p is false. When p is false and $p \rightarrow q$ is true, the definition of $p \rightarrow q$ allows q to be either true or false. So, in this case, you can't make any deduction about whether or not I will be at the party. The statement "If the party is on Tuesday, then I'll be there" doesn't assert anything about what will happen if the party is on some other day than Tuesday.

2.1.7 More forms of implication

The implication $\neg q \to \neg p$ is called the ***contrapositive*** of $p \to q$. An implication is logically equivalent to its contrapositive. The contrapositive of "If this is Tuesday, then we are in Belgium" is "If we aren't in Belgium, then this isn't Tuesday". These two sentences assert exactly the same thing.

Note that $p \to q$ is *not* logically equivalent to $q \to p$. The implication $q \to p$ is called the ***converse*** of $p \to q$. The converse of "If this is Tuesday, then we are in Belgium" is "If we are in Belgium, then this is Tuesday". Note that it is possible for either one of these statements to be true while the other is false. In English, I might express the fact that both statements are true by saying "If this is Tuesday, then we are in Belgium, *and conversely*". In logic, this would be expressed with a proposition of the form $(p \to q) \wedge (q \to p)$.

Similarly $p \to q$ is *not* logically equivalent to $\neg p \to \neg q$. The implication $\neg p \to \neg q$ is called the ***inverse*** of $p \to q$. Although this mistake is commonly made in English, for instance people often assume that when I say: "If it is morning, I drink some coffee", I also mean that when it is not morning I do not drink coffee. But my original statement does not tell you anything about what I do when it is not morning.

The ***biconditional operator*** is closely related to the conditional operator. In fact, $p \leftrightarrow q$ is logically equivalent to $(p \to q) \wedge (q \to p)$. The proposition $p \leftrightarrow q$ is usually read as 'p if and only if q'. (The 'p if q' part represents $q \to p$, while 'p only if q' is another way of asserting that $p \to q$.) It could also be expressed as 'if p then q, and conversely'. Occasionally in English, 'if... then' is used when what is really meant is 'if and only if'. For example, if a parent tells a child, "If you are good, Sinterklaas will bring you toys", the parent probably really means to say "Sinterklaas will bring you toys if and only if you are good". (The parent would probably not respond well to the child's perfectly logical plea "But you never said what would happen if I wasn't good!")

2.1.8 Exclusive or

Finally, we turn to the exclusive or operator. The English word 'or' is actually somewhat ambiguous. The two operators $\oplus$ and $\vee$ express the two possible meanings of this word. The proposition $p \vee q$ can be expressed unambiguously as "p or q, *or both*", while $p \oplus q$ stands for "p or q, *but not both*". If a menu says that you can choose soup or salad, it doesn't mean that you can have both. In this case, 'or' is an exclusive or. On the other hand, in "You are at risk of heart disease if you smoke or drink", the or is inclusive since you certainly don't get off the hook if you both smoke and drink. In theoretical computer science and mathematics, the word 'or' is always taken in the inclusive sense of $p \vee q$.

2.1.9 Universal operators

Now, any compound proposition that uses any of the operators $\to$, $\leftrightarrow$, and $\oplus$ can be rewritten as a logically equivalent proposition that uses only $\wedge$, $\vee$, and $\neg$. It is easy to

check that $p \to q$ is logically equivalent to $\neg p \vee q$. (Just make a truth table for $\neg p \vee q$.) Similarly, $p \leftrightarrow q$ can be expressed as $(\neg p \vee q) \wedge (\neg q \vee p)$, So, in a strict logical sense, $\to$, $\leftrightarrow$, and $\oplus$ are unnecessary. (Nevertheless, they are useful and important, and we won't give them up.)

Even more is true: In a strict logical sense, we could do without the conjunction operator $\wedge$. It is easy to check that $p \wedge q$ is logically equivalent to $\neg(\neg p \vee \neg q)$, so any expression that uses $\wedge$ can be rewritten as one that uses only $\neg$ and $\vee$. Alternatively, we could do without $\vee$ and write everything in terms of $\neg$ and $\wedge$. We shall study some of these rewrite rules in more detail in Section 2.2.

We call a set of operators that can express all operations: ***functionally complete***. More formally we would state the following:

Definition 2.3. A set of logical operators is ***functionally complete*** if and only if all formulas in propositional logic can be rewritten to an equivalent form that uses only operators from the set.

Consider for instance the set $\{\neg, \vee\}$. As shown above the $\wedge$, $\to$ and $\leftrightarrow$-operators can be expressed using only these operators. In fact all possible operations can be expressed using only $\{\neg, \vee\}$. To prove this you will show in one of the exercises that all possible formulas in propositional logic can be expressed using $\{\neg, \vee, \wedge, \to, \leftrightarrow\}$. So by showing that we do not need $\wedge$, $\to$, and $\leftrightarrow$ we can prove that $\{\neg, \vee\}$ is also functionally complete.

2.1.10 Classifying propositions

Certain types of proposition will play a special role in our further work with logic. In particular, we define tautologies, contradictions, and contingencies as follows:

Definition 2.4. A compound proposition is said to be a ***tautology*** if and only if it is *true* for all possible combinations of truth values of the propositional variables which it contains. A compound proposition is said to be a ***contradiction*** if and only if it is *false* for all possible combinations of truth values of the propositional variables which it contains. A compound proposition is said to be a ***contingency*** if and only if it is neither a tautology nor a contradiction.

For example, the proposition $((p \vee q) \wedge \neg q) \to p$ is a tautology. This can be checked with a truth table:

p	q	$p \vee q$	$\neg q$	$(p \vee q) \wedge \neg q$	$((p \vee q) \wedge \neg q) \to p$
0	0	0	1	0	1
0	1	1	0	0	1
1	0	1	1	1	1
1	1	1	0	0	1

The fact that all entries in the last column are true tells us that this expression is a tautology. Note that for any compound proposition P, P is a tautology if and only if $\neg P$ is a contradiction. (Here and moving forward, I use uppercase letters to represent compound propositions. P stands for any formula made up of simple propositions, propositional variables, and logical operators.)

Logical equivalence can be defined in terms of tautology:

Definition 2.5. Two compound propositions, P and Q, are said to be ***logically equivalent*** if and only if the proposition $P \leftrightarrow Q$ is a tautology.

The assertion that P is logically equivalent to Q will be expressed symbolically as '$P \equiv Q$'. For example, $(p \rightarrow q) \equiv (\neg p \vee q)$, and $p \oplus q \equiv (p \vee q) \wedge \neg(p \wedge q)$.

What if $P \rightarrow Q$ and P is false? From a false premise we can derive any conclusion (check the truth table of $\rightarrow$). So if k stands for "I'm the King of the Netherlands", then $k \rightarrow Q$ is true for any compound proposition Q. You can substitute anything for Q, and the implication $k \rightarrow Q$ will hold. For example, it a logically valid deduction that: If I'm the King of the Netherlands, then unicorns exist. Taking this further, from a contradiction we can derive any conclusion. This is called the ***Principle of Explosion***.

Exercises

Recall that solutions to some of the exercises start on page 163. Exercises that have a solution are marked with a dagger (†) symbol. I suggest you attempt the exercise first before looking at the solution!

1. Give the three truth tables that define the logical operators $\wedge$, $\vee$, and $\neg$.

†**2.** Some of the following compound propositions are tautologies, some are contradictions, and some are neither (i.e., so are contingencies). In each case, use a truth table to decide to which of these categories the proposition belongs:

a) $(p \wedge (p \rightarrow q)) \rightarrow q$ **b)** $((p \rightarrow q) \wedge (q \rightarrow r)) \rightarrow (p \rightarrow r)$
c) $p \wedge \neg p$ **d)** $(p \vee q) \rightarrow (p \wedge q)$
e) $p \vee \neg p$ **f)** $(p \wedge q) \rightarrow (p \vee q)$

3. Use truth tables to show that each of the following propositions is logically equivalent to $p \leftrightarrow q$.

a) $(p \rightarrow q) \wedge (q \rightarrow p)$ **b)** $\neg p \leftrightarrow \neg q$
c) $(p \rightarrow q) \wedge (\neg p \rightarrow \neg q)$ **d)** $\neg(p \oplus q)$

†**4.** Is $\rightarrow$ an associative operation? This is, is $(p \rightarrow q) \rightarrow r$ logically equivalent to $p \rightarrow (q \rightarrow r)$?

5. Let p represent the proposition "You leave" and let q represent the proposition "I leave". Express the following sentences as compound propositions using p and q, and show that they are logically equivalent:
   a) Either you leave or I do. (Or both!)
   b) If you don't leave, I will.

†**6.** Suppose that m represents the proposition "The Earth moves", c represents "The Earth is the centre of the universe", and g represents "Galileo was falsely accused". Translate each of the following compound propositions into English:
   a) $\neg g \wedge c$ **b**) $m \rightarrow \neg c$
   c) $m \leftrightarrow \neg c$ **d**) $(m \rightarrow g) \wedge (c \rightarrow \neg g)$

7. Give the converse and the contrapositive of each of the following English sentences:
   a) If you are good, Sinterklaas brings you toys.
   b) If the package weighs more than one kilo, then you need extra postage.
   c) If I have a choice, I don't eat courgette.

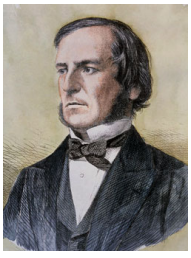
8. In an ordinary deck of fifty-two playing cards, for how many cards is it true
   a) that "This card is a ten and this card is a heart"?
   b) that "This card is a ten or this card is a heart"?
   c) that "If this card is a ten, then this card is a heart"?
   d) that "This card is a ten if and only if this card is a heart"?

9. Define a logical operator $\downarrow$ so that $p \downarrow q$ is logically equivalent to $\neg(p \vee q)$. (This operator is usually referred to as 'NOR', short for 'not or'.) Show that each of the propositions $\neg p$, $p \wedge q$, $p \vee q$, $p \rightarrow q$, $p \leftrightarrow q$, and $p \oplus q$ can be rewritten as a logically equivalent proposition that uses $\downarrow$ as its only operator.

10. For our proof that $\{\neg, \vee\}$ is functionally complete, we need to show that all formulas in propositional logic can be expressed in an equivalent form using only $\{\neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$.
   a) How many unique truth tables exist for formulas containing two atoms?
   b) Create a function for each of the possible truth tables that uses only the 5 operators listed above.
   c) Give an (informal) argument why this means all formulas in propositional logic can be expressed using only these five operators.

2.2 Boolean Algebra

So far we have discussed how to write and interpret propositions. This section deals with *manipulating* them. For this, we need algebra. Ordinary algebra, of the sort taught in high school, is about manipulating numbers, variables that represent numbers, and operators such as $+$ and $\times$ that apply to numbers. Now, we need an algebra that applies to logical values, propositional variables, and logical operators. The first person to think of logic in terms of algebra was the mathematician, George Boole, who introduced the idea in a book that he published in 1854. The algebra of logic is now called ***Boolean algebra*** in his honour.



George Boole (1815–1864) was a largely self-taught British mathematician, philosopher and logician, most of whose short career was spent as the first professor of mathematics at Queen's College, Cork in Ireland. He worked in the fields of differential equations and algebraic logic, and is best known as the author of The Laws of Thought (1854). Among TU Delft students he is best known for the room named after him in the EEMCS building 36.

Boolean logic is credited with laying the foundations for the information age: essentially, computer science. Boole maintained that: "No general method for the solution of questions in the theory of probabilities can be established which does not explicitly recognise, not only the special numerical bases of the science, but also those universal laws of thought which are the basis of all reasoning, and which, whatever they may be as to their essence, are at least mathematical as to their form."

Source: `en.wikipedia.org/wiki/George_Boole`.

The algebra of numbers includes a large number of rules for manipulating expressions. The distributive law, for example, says that $x(y+z) = xy + xz$, where x, y, and z are variables that stand for any numbers or numerical expressions. This law means that whenever you see something of the form $xy + xz$ in a numerical expression, you can substitute $x(y+z)$ without changing the value of the expression, and *vice versa*. Note that the equals sign in $x(y+z) = xy + xz$ means "has the same value as, no matter what numerical values x, y, and z have".

In Boolean algebra, we work with logical values instead of numerical values. There are only two logical values, true and false. We will write these values as $\mathbb{T}$ and $\mathbb{F}$ or 1 and 0. The symbols $\mathbb{T}$ and $\mathbb{F}$ play a similar role in Boolean algebra to the role that constant numbers such as 1 and 3.14159 play in ordinary algebra. Instead of the equals sign, Boolean algebra uses logical equivalence, $\equiv$, which has essentially the same meaning.[5] For example, for propositions p, q, and r, the $\equiv$ operator in $p \wedge (q \wedge r) \equiv (p \wedge q) \wedge r$ means "has the same value as, no matter what logical values p, q, and r have".

[5] In ordinary algebra, it is easy to be confused by the equals sign, because it has two very different roles. In an identity such as the distributive law, it means 'is always equal to'. On the other hand, an equation such as $x^2 + 3x = 4$ is a statement that might or might not be true, depending on the value of x. Boolean algebra has two operators, $\equiv$ and $\leftrightarrow$, that play roles similar to the two roles of the equals sign. $\equiv$ is used for identity, whereas $\leftrightarrow$ is used in equations that may or may not be true.

Double negation	$\neg(\neg p) \equiv p$
Excluded middle	$p \vee \neg p \equiv \mathbb{T}$
Contradiction	$p \wedge \neg p \equiv \mathbb{F}$
Identity laws	$\mathbb{T} \wedge p \equiv p$
	$\mathbb{F} \vee p \equiv p$
Idempotent laws	$p \wedge p \equiv p$
	$p \vee p \equiv p$
Commutative laws	$p \wedge q \equiv q \wedge p$
	$p \vee q \equiv q \vee p$
Associative laws	$(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$
	$(p \vee q) \vee r \equiv p \vee (q \vee r)$
Distributive laws	$p \wedge (q \vee r) \equiv (p \wedge q) \vee (p \wedge r)$
	$p \vee (q \wedge r) \equiv (p \vee q) \wedge (p \vee r)$
DeMorgan's laws	$\neg(p \wedge q) \equiv (\neg p) \vee (\neg q)$
	$\neg(p \vee q) \equiv (\neg p) \wedge (\neg q)$

Figure 2.2: *Laws of Boolean Algebra. These laws hold for any propositions p, q, and r.*

2.2.1 Basics of Boolean Algebra

Many of the rules of Boolean algebra are fairly obvious, if you think a bit about what they mean. Even those that are not obvious can be verified easily by using a truth table. Figure 2.2 lists the most important of these laws. You will notice that all these laws, except the first, come in pairs: each law in the pair can be obtained from the other by interchanging $\wedge$ with $\vee$ and $\mathbb{T}$ with $\mathbb{F}$. This cuts down on the number of facts you have to remember.[6]

Just as an example, let's verify the first rule in the table, the Law of Double Negation. This law is just the old, basic grammar rule that two negatives make a positive. Although the way this rule applies to English is questionable, if you look at how it is used—no matter what the grammarian says, "I can't get no satisfaction" doesn't really mean "I can get satisfaction"—the validity of the rule in logic can be verified just by computing the two possible cases: when p is true and when p is false. When p is true, then by the

[6]It is also an example of a more general fact known as ***duality***, which asserts that given any tautology that uses only the operators $\wedge$, $\vee$, and $\neg$, another tautology can be obtained from it by interchanging $\wedge$ with $\vee$ and $\mathbb{T}$ with $\mathbb{F}$. We won't attempt to prove this here, but I encourage you to try it!

definition of the $\neg$ operator, $\neg p$ is false. But then, again by the definition of $\neg$, the value of $\neg(\neg p)$ is true, which is the same as the value of p. Similarly, in the case where p is false, $\neg(\neg p)$ is also false. Organized into a truth table, this argument takes the rather simple form

p	$\neg p$	$\neg(\neg p)$
0	1	0
1	0	1

The fact that the first and last columns are identical shows the logical equivalence of p and $\neg(\neg p)$. The point here is not just that $\neg(\neg p) \equiv p$, but also that this logical equivalence is valid because it can be verified computationally based just on the relevant definitions. Its validity does *not* follow from the fact that "it's obvious" or "it's a well-known rule of grammar".

Students often ask "Why do I have to prove something when it's obvious?" The point is that logic—and mathematics more generally—is its own little world with its own set of rules. Although this world is related somehow to the real world, when you say that something is obvious (in the real world), you aren't playing by the rules of the world of logic. The real *magic* of mathematics is that by playing by its rules, you can come up with things that are decidedly not obvious, but that still say something about the real world or the computational world—often, something interesting and important.

Each of the rules in Figure 2.2 can be verified in the same way, by making a truth table to check all the possible cases. In one of the pencasts of this course we further discuss how to check the equivalence of two propositions using truth tables.

2.2.2 Substitution laws

It's important to understand that the propositional variables in the laws of Boolean algebra can stand for any propositions, including compound propositions. It is not just true, as the Double Negation Law states, that $\neg(\neg p) \equiv p$. It is also true that $\neg(\neg q) \equiv q$, that $\neg(\neg(p \wedge q)) \equiv (p \wedge q)$, that $\neg(\neg(p \rightarrow (q \wedge \neg p))) \equiv (p \rightarrow (q \wedge \neg p))$, and an infinite number of other statements of the same form. Here, a 'statement of the same form' is one that can be obtained by substituting something for p in both places where it occurs in $\neg(\neg p) \equiv p$. How can I be sure that all these infinitely many statements are valid when all that I've checked is one little two-line truth table? The answer is that any given proposition, Q, no matter how complicated, has a particular truth value, either true or false. So, the question of the validity of $\neg(\neg Q) \equiv Q$ always reduces to one of the two cases I

already checked in the truth table. (Note that for this argument to be valid, the same Q must be substituted for p in every position where it occurs.) While this argument may be 'obvious', it is not exactly a proof, but for now we will just accept the validity of the following theorem:

Theorem 2.1 (First Substitution Law)**.** *Suppose that Q is any proposition, and that p is a propositional variable. Consider any tautology. If (Q) is substituted for p in all places where p occurs in the tautology, then the result is also a tautology.*

Since logical equivalence is defined in terms of tautology, it is also true that when (Q) is substituted for p in a logical equivalence, the result is again a logical equivalence.[7]

The First Substitution Law lets you do algebra! For example, you can substitute $p \rightarrow q$ for p in the law of double negation, $\neg(\neg p) \equiv p$. This allows you to 'simplify' the expression $\neg(\neg(r \rightarrow q))$ to $r \rightarrow q$ with confidence that the resulting expression has the same logical value as the expression you started with. (That's what it means for $\neg(\neg(r \rightarrow q))$ and $r \rightarrow q$ to be logically equivalent.) You can play similar tricks with all the laws in Figure 2.2. Even more important is the Second Substitution Law, which says that you can substitute an expression for a logically equivalent expression, wherever it occurs. Once again, we will accept this as a theorem without trying to prove it here. It is surprisingly hard to put this law into words:

Theorem 2.2 (Second Substitution Law)**.** *Suppose that P and Q are any propositions such that $P \equiv Q$. Suppose that R is any compound proposition in which (P) occurs as a sub-proposition. Let R' be the proposition that is obtained by substituting (Q) for that occurrence of (P) in R. Then $R \equiv R'$.*

Note that in this case, the theorem does not require (Q) to be substituted for *every* occurrence of (P) in R. You are free to substitute for one, two, or as many occurrences of (P) as you like, and the result is still logically equivalent to R.

The Second Substitution Law allows us to use the logical equivalence $\neg(\neg p) \equiv p$ to 'simplify' the expression $q \rightarrow (\neg(\neg p))$ by substituting $\neg(\neg p)$ for p. The resulting expression, $q \rightarrow p$, is logically equivalent to the original $q \rightarrow (\neg(\neg p))$. Once again, we have to be careful about parentheses: The fact that $p \vee p \equiv p$ does *not* allow us to rewrite $q \wedge p \vee p \wedge r$ as $q \wedge p \wedge r$. The problem is that $q \wedge p \vee p \wedge r$ means $(q \wedge p) \vee (p \wedge r)$, so that $(p \vee p)$ is not a sub-expression. This again underlines the importance of always writing parentheses in your propositional formulas.

[7] I've added parentheses around Q here for technical reasons. Sometimes, the parentheses are necessary to make sure that Q is evaluated as a whole, so that its final value is used in place of p. As an example of what can go wrong, consider $q \wedge r$. If this is substituted literally for p in $\neg(\neg p)$, without parentheses, the result is $\neg(\neg q \wedge r)$. But this expression means $\neg((\neg q) \wedge r)$, which is *not* equivalent to $q \wedge r$. Did I say to always write parentheses if you're in doubt? See page 9.

2.2.3 Simplifications

The final piece of algebra in Boolean algebra is the observation that we can chain logical equivalences together. That is, from $P \equiv Q$ and $Q \equiv R$, it follows that $P \equiv R$. This is really just a consequence of the Second Substitution Law. The equivalence $Q \equiv R$ allows us to substitute R for Q in the statement $P \equiv Q$, giving $P \equiv R$. (Remember that, by Definition 2.5, logical equivalence is defined in terms of a proposition.) This means that we can show that two compound propositions are logically equivalent by finding a chain of logical equivalences that lead from one to the other.

Here is an example of such a chain of logical equivalences:

$$\begin{aligned} p \wedge (p \rightarrow q) &\equiv p \wedge (\neg p \vee q) && \text{definition of } p \rightarrow q\text{, Theorem 2.2} \\ &\equiv (p \wedge \neg p) \vee (p \wedge q) && \text{Distributive Law} \\ &\equiv \mathbb{F} \vee (p \wedge q) && \text{Law of Contradiction, Theorem 2.2} \\ &\equiv (p \wedge q) && \text{Identity Law} \end{aligned}$$

Each step in the chain has its own justification. In several cases, a substitution law is used without stating as much. In the first line, for example, the definition of $p \rightarrow q$ is that $p \rightarrow q \equiv \neg p \vee q$. The Second Substitution Law allows us to substitute $(\neg p \vee q)$ for $(p \rightarrow q)$. In the last line, we implicitly applied the First Substitution Law to the Identity Law, $\mathbb{F} \vee p \equiv p$, to obtain $\mathbb{F} \vee (p \wedge q) \equiv (p \wedge q)$.

The chain of equivalences in the above example allows us to conclude that $p \wedge (p \rightarrow q)$ is logically equivalent to $p \wedge q$. This means that if you were to make a truth table for these two expressions, the truth values in the column for $p \wedge (p \rightarrow q)$ would be identical to those in the column for $p \wedge q$. We know this without actually making the table. Don't believe it? Go ahead and make the truth table. In this case, the table is only be four lines long and easy enough to make. But Boolean algebra can be applied in cases where the number of propositional variables is too large for a truth table to be practical.

Let's do another example. Recall that a compound proposition is a tautology if it is true for all possible combinations of truth values of the propositional variables that it contains. But another way of saying the same thing is that P is a tautology if $P \equiv \mathbb{T}$. So, we can prove that a compound proposition, P, is a tautology by finding a chain of logical equivalences leading from P to $\mathbb{T}$. For example:

$$
\begin{array}{lll}
((p \vee q) \wedge \neg p) \rightarrow q & & \\
\quad \equiv (\neg((p \vee q) \wedge \neg p)) \vee q & & \text{definition of } \rightarrow \\
\quad \equiv (\neg(p \vee q) \vee \neg(\neg p)) \vee q & & \text{DeMorgan's Law, Theorem 2.2} \\
\quad \equiv (\neg(p \vee q) \vee p) \vee q & & \text{Double Negation, Theorem 2.2} \\
\quad \equiv (\neg(p \vee q)) \vee (p \vee q) & & \text{Associative Law for } \vee \\
\quad \equiv \mathbb{T} & & \text{Law of Excluded Middle}
\end{array}
$$

From this chain of equivalences, we can conclude that $((p \vee q) \wedge \neg p) \rightarrow q$ is a tautology.

Now, it takes some practice to look at an expression and see which rules can be applied to it; to see $(\neg(p \vee q)) \vee (p \vee q)$ as an application of the law of the excluded middle for example, you need to mentally substitute $(p \vee q)$ for p in the law as it is stated in Figure 2.2. Often, there are several rules that apply, and there are no definite guidelines about which one you should try. This is what makes algebra something of an art.

2.2.4 More rules of Boolean algebra

It is certainly not true that all possible rules of Boolean algebra are given in Figure 2.2. For one thing, there are many rules that are easy consequences of the rules that are listed there. For example, although the table asserts only that $\mathbb{F} \vee p \equiv p$, it is also true that $p \vee \mathbb{F} \equiv p$. This can be checked directly or by a simple calculation:

$$
\begin{array}{lll}
p \vee \mathbb{F} \equiv \mathbb{F} \vee p & & \text{Commutative Law} \\
\quad \equiv p & & \text{Identity Law as given in the table}
\end{array}
$$

Additional rules can be obtained by applying the Commutative Law to other rules in the table, and we will use such rules freely in the future.

Another sort of easy extension can be applied to the Associative Law, $(p \vee q) \vee r \equiv p \vee (q \vee r)$. The law is stated for the $\vee$ operator applied to three terms, but it generalizes

to four or more terms. For example

$$
\begin{aligned}
&((p \vee q) \vee r) \vee s \\
&\qquad \equiv (p \vee q) \vee (r \vee s) && \text{by the Associative Law for three terms} \\
&\qquad \equiv p \vee (q \vee (r \vee s)) && \text{by the Associative Law for three terms}
\end{aligned}
$$

We will, of course, often write this expression as $p \vee q \vee r \vee s$, with no parentheses at all, knowing that wherever we put the parentheses the value is the same.

One other thing that you should keep in mind is that rules can be applied in either direction. The Distributive Law, for example, allows you to distribute the p in $p \vee (q \wedge \neg p)$ to get $(p \vee q) \wedge (p \vee \neg p)$. But it can also be used in reverse to 'factor out' a term, as when you start with $(q \vee (p \rightarrow q)) \wedge (q \vee (q \rightarrow p))$ and factor out the q to get $q \vee ((p \rightarrow q) \wedge (q \rightarrow p))$.

So far in this section, we have been working with the laws of Boolean algebra without saying much about what they mean or why they are reasonable. Of course, you can apply the laws in calculations without understanding them. But if you want to figure out *which* calculations to do, you need some understanding. Most of the laws are clear enough with a little thought. For example, if we already know that q is false, then $p \vee q$ will be true when p is true and false when p is false. That is, $p \vee \mathbb{F}$ has the same logical value as p. But that's just what the Identity Law for $\vee$ says. A few of the laws need more discussion.

The Law of the Excluded Middle, $p \vee \neg p \equiv \mathbb{T}$, says that given any proposition p, at least one of p or $\neg p$ must be true. Since $\neg p$ is true exactly when p is false, this is the same as saying that p must be either true or false. There is no middle ground. The Law of Contradiction, $p \wedge \neg p \equiv \mathbb{F}$, says that it is not possible for *both* p and $\neg p$ to be true. Both of these rules are obvious.

There are some who set out to question the law of there being no middle ground. Already in the 1920's people like Tarski (who we will meet later) talked about other forms of logic where another value representing 'unknown' or 'not proven' also exists. You can also see this in some programming languages where they are referred to as 'tri-state booleans'.

These so-called non-standard logics have been developed and have also lead to things like 'fuzzy logic', which some consider quite controversial. Lotfi Zadeh is credited as the first person to refer to this type of logic as fuzzy logic in his work on 'fuzzy sets' in 1965. Zadeh was later quoted as saying: "Not being afraid to get embroiled in controversy. ... That's part of my character, too. I can be very stubborn. That's probably been beneficial for the development of Fuzzy Logic."

Source: `en.wikipedia.org/wiki/Lotfi_A._Zadeh`

The Distributive Laws cannot be called obvious, but a few examples can show that they are reasonable. Consider the statement, "This card is the ace of spades or clubs." Clearly, this is equivalent to "This card is the ace of spaces or this card is the ace of clubs." But this is just an example of the first distributive law! For, let a represent the proposition "This card is an ace", let s represent "This card is a spade" and let c represent "This card is a club". Then "This card is the ace of spades or clubs" can be translated into logic as $a \wedge (s \vee c)$, while "This card is the ace of spades or this card is the ace of clubs" becomes $(a \wedge s) \vee (a \wedge c)$. And the distributive law assures us that $a \wedge (s \vee c) \equiv (a \wedge s) \vee (a \wedge c)$. The second distributive law tells us, for example, that "This card is either a joker or is the ten of diamonds" is logically equivalent to "This card is either a joker or a ten, and it is either a joker or a diamond". That is, $j \vee (t \wedge d) \equiv (j \vee t) \wedge (j \vee d)$. The distributive laws are powerful tools and you should keep them in mind whenever you are faced with a mixture of $\wedge$ and $\vee$ operators.

DeMorgan's Laws must also be less than obvious, since people often get them wrong. Fortunately you get to practice them both in *Reasoning & Logic*, as well as in *Computer Organisation*, so you will soon get them right. More importantly perhaps they do also make sense. When considering $\neg(p \wedge q)$, you should ask yourself, how can 'p and q' *fail* to be true. It will fail to be true if either p is false *or* if q is false (or both). That is, $\neg(p \wedge q)$ is equivalent to $(\neg p) \vee (\neg q)$. Consider the sentence "A raven is large and black." If a bird is *not* large and black, then it is not a raven. But what exactly does it mean to be '*not* (*large and black*)'? How can you tell whether the assertion 'not (large and black)' is true of something? This will be true if it is either not large *or* not black. (It doesn't have to be

both—it could be large and white, it could be small and black.) Similarly, for 'p or q' to fail to be true, *both* p and q must be false. That is, $\neg(p \vee q)$ is equivalent to $(\neg p) \wedge (\neg q)$. This is DeMorgan's second law.

Recalling that $p \to q$ is equivalent to $(\neg p) \vee q$, we can apply DeMorgan's law to obtain a formula for the negation an implication:

$$\begin{aligned} \neg(p \to q) &\equiv \neg((\neg p) \vee q) \\ &\equiv (\neg(\neg p)) \wedge (\neg q) \\ &\equiv p \wedge \neg q \end{aligned}$$

That is, $p \to q$ is false exactly when both p is true and q is false. For example, the negation of "If you have an ace, you win" is "You have an ace, and you don't win". Think of it this way: if you had an ace and you didn't win, then the statement "If you have an ace, you win" was not true.

Exercises

1. Construct truth tables to demonstrate the validity of each of the distributive laws.

2. Construct the following truth tables:

a) Construct truth tables to demonstrate that $\neg(p \wedge q)$ is **not** logically equivalent to $(\neg p) \wedge (\neg q)$.

b) Construct truth tables to demonstrate that $\neg(p \vee q)$ is **not** logically equivalent to $(\neg p) \vee (\neg q)$.

c) Construct truth tables to demonstrate the validity of both DeMorgan's Laws.

3. Construct truth tables to demonstrate that $\neg(p \to q)$ is **not** logically equivalent to any of the following.

a) $(\neg p) \to (\neg q)$

b) $(\neg p) \to q$

c) $p \to (\neg q)$

Refer back to this section for a formula that **is** logically equivalent to $\neg(p \to q)$.

†**4.** Is $\neg(p \leftrightarrow q)$ logically equivalent to $(\neg p) \leftrightarrow (\neg q)$?

5. In the algebra of numbers, there is a distributive law of multiplication over addition: $x(y+z) = xy + xz$. What would a distributive law of addition over multiplication look like? Is it a valid law in the algebra of numbers?

6. The distributive laws given in Figure 2.2 are sometimes called the *left* distributive laws. The ***right distributive laws*** say that $(p \vee q) \wedge r \equiv (p \wedge r) \vee (q \wedge r)$ and that $(p \wedge q) \vee r \equiv (p \vee r) \wedge (q \vee r)$. Show that the right distributive laws are also valid laws of Boolean algebra. (Note: In practice, both the left and the right distributive laws are referred to simply as the distributive laws, and both can be used freely in proofs.)

7. Show that $p \wedge (q \vee r \vee s) \equiv (p \wedge q) \vee (p \wedge r) \vee (p \wedge s)$ for any propositions p, q, r, and s. In words, we can say that conjunction distributes over a disjunction of three terms. (Recall that

the $\wedge$ operator is called conjunction and $\vee$ is called disjunction.) Translate into logic and verify the fact that conjunction distributes over a disjunction of four terms. Argue that, in fact, conjunction distributes over a disjunction of any number of terms.

8. There are two additional basic laws of logic, involving the two expression $p \wedge \mathbb{F}$ and $p \vee \mathbb{T}$. What are the missing laws? Show that your answers are, in fact, laws.

9. For each of the following pairs of propositions, show that the two propositions are logically equivalent by finding a chain of equivalences from one to the other. State which definition or law of logic justifies each equivalence in the chain.

a) $p \wedge (q \wedge p)$, $p \wedge q$ **b)** $(\neg p) \rightarrow q$, $p \vee q$
c) $(p \vee q) \wedge \neg q$, $p \wedge \neg q$ **d)** $p \rightarrow (q \rightarrow r)$, $(p \wedge q) \rightarrow r$
e) $(p \rightarrow r) \wedge (q \rightarrow r)$, $(p \vee q) \rightarrow r$ **f)** $p \rightarrow (p \wedge q)$, $p \rightarrow q$

†**10.** For each of the following compound propositions, find a simpler proposition that is logically equivalent. Try to find a proposition that is as simple as possible.

a) $(p \wedge q) \vee \neg q$ **b)** $\neg(p \vee q) \wedge p$ **c)** $p \rightarrow \neg p$
d) $\neg p \wedge (p \vee q)$ **e)** $(q \wedge p) \rightarrow q$ **f)** $(p \rightarrow q) \wedge (\neg p \rightarrow q)$

†**11.** Express the negation of each of the following sentences in natural English:

a) It is sunny and cold.
b) I will have stroopwafel or I will have appeltaart.
c) If today is Tuesday, this is Belgium.
d) If you pass the final exam, you pass the course.

12. Apply one of the laws of logic to each of the following sentences, and rewrite it as an equivalent sentence. State which law you are applying.

a) I will have coffee and stroopwafel or appeltaart.
b) He has neither talent nor ambition.
c) You can have oliebollen, or you can have oliebollen.

13. Suppose it is simultaneously true that "All lemons are yellow" and "Not all lemons are yellow". Derive the conclusion "Unicorns exist". (If you get stuck, check out `en.wikipedia.org/wiki/Principle_of_explosion`.)

2.3 Application: Logic Circuits

As we saw in Chapter 1, computers have a reputation—not always deserved—for being 'logical'. But fundamentally, deep down, they are made of logic in a very real sense. The building blocks of computers are ***logic gates***, which are electronic components that compute the values of simple propositions such as $p \wedge q$ and $\neg p$. (Each gate is in turn built of even smaller electronic components called transistors, but this needn't concern us here: see the course *Computer Organisation*.)

Don't worry, logic circuits will be examined in *Computer Organisation*, not in *Reasoning & Logic*. They are a good example and application of propositional logic, and that's why we're talking about them in this section. Normal forms (Section 2.3.4) are definitely on the syllabus, however, so pay attention!

2.3.1 Logic gates

A wire in a computer can be in one of two states, which we can think of as being *on* and *off*. These two states can be naturally associated with the Boolean values $\mathbb{T}$ and $\mathbb{F}$. When a computer computes, the multitude of wires inside it are turned on and off in patterns that are determined by certain rules. The rules involved can be most naturally expressed in terms of logic. A simple rule might be: "turn wire C on whenever wire A is on and wire B is on". This rule can be implemented in hardware as an ***AND gate***. An AND gate is an electronic component with two input wires and one output wire, whose job is to turn its output on when both of its inputs are on and to turn its output off in any other case. If we associate 'on' with $\mathbb{T}$ and 'off' with $\mathbb{F}$, and if we give the names A and B to the inputs of the gate, then the gate computes the value of the logical expression $A \wedge B$. In effect, A is a proposition with the meaning "the first input is on", and B is a proposition with the meaning "the second input is on". The AND gate functions to ensure that the output is described by the proposition $A \wedge B$. That is, the output is on if and only if the first input is on and the second input is on.

As you hopefully know from *Computer Organisation*, an ***OR gate*** is an electronic component with two inputs and one output which turns its output on if either (or both) of its inputs is on. If the inputs are given names A and B, then the OR gate computes the logical value of $A \vee B$. A ***NOT gate*** has one input and one output, and it turns its output off when the input is on and on when the input is off. If the input is named A, then the NOT gate computes the value of $\neg A$.

As I mentioned earlier, other textbooks might use different notations to represent a negation. For instance a bar over the variable $\bar{x}$ or a $\sim$ symbol. In digital logic (and thus in your *Computer Organisation* course) you will also often find the $+$ symbol to represent an 'or' and a $\cdot$ (dot) symbol to represent an 'and'.

Other types of logic gates are, of course, possible. Gates could be made to compute $A \rightarrow B$ or $A \oplus B$, for example. However, any computation that can be performed by logic gates can be done using only AND, OR, and NOT gates, as we will see below. (In practice, however, NAND gates and NOR gates, which compute the values of $\neg(A \wedge B)$ and $\neg(A \vee B)$ respectively, are often used because they are easier to build from transistors than AND and OR gates.)

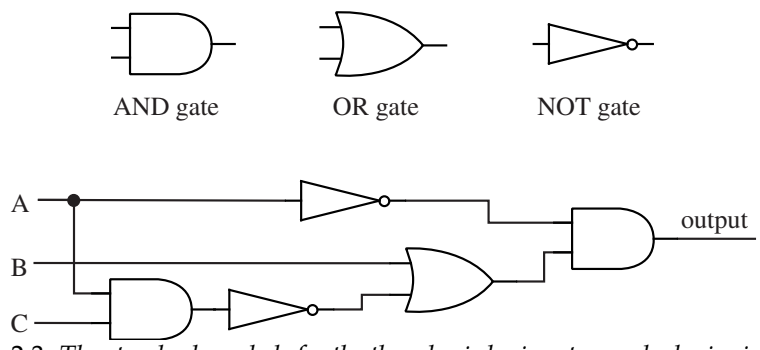


Figure 2.3: *The standard symbols for the three basic logic gates, and a logic circuit that computes the value of the logical expression* $(\neg A) \wedge (B \vee \neg(A \wedge C))$. *The input wires to each logic gate are on the left, with the output wire on the right. Note that when wires cross each other in a diagram such as this, the wires don't actually intersect unless there is a black circle at the point where they cross.*

2.3.2 Combining gates to create circuits

The three types of logic gates are represented by standard symbols, as shown in Figure 2.3. Since the inputs and outputs of logic gates are just wires carrying on/off signals, logic gates can be wired together by connecting outputs from some gates to inputs of other gates. The result is a ***logic circuit***. An example is also shown in Figure 2.3.

The logic circuit in the figure has three inputs, labeled A, B, and C. The circuit computes the value of the compound proposition $(\neg A) \wedge (B \vee \neg(A \wedge C))$. That is, when A represents the proposition "the input wire labeled A is on," and similarly for B and C, then the output of the circuit is on if and only if the value of the compound proposition $(\neg A) \wedge (B \vee \neg(A \wedge C))$ is true.

Given any compound proposition made from the operators $\wedge$, $\vee$, and $\neg$, it is possible to build a logic circuit that computes the value of that proposition. The proposition itself is a blueprint for the circuit. As noted in Section 2.1, every logical operator that we have encountered can be expressed in terms of $\wedge$, $\vee$, and $\neg$, so in fact every compound proposition that we know how to write can be computed by a logic circuit.

Given a proposition constructed from $\wedge$, $\vee$, and $\neg$ operators, it is easy to build a circuit to compute it. First, identify the main operator in the proposition—the one whose value will be computed *last*. Consider $(A \vee B) \wedge \neg(A \wedge B)$. This circuit has two input values, A and B, which are represented by wires coming into the circuit. The circuit has an output wire that represents the computed value of the proposition. The main operator in $(A \vee B) \wedge \neg(A \wedge B)$, is the first $\wedge$, which computes the value of the expression as a whole by combining the values of the subexpressions $A \vee B$ and $\neg(A \wedge B)$. This $\wedge$ operator corresponds to an AND gate in the circuit that computes the final output of the

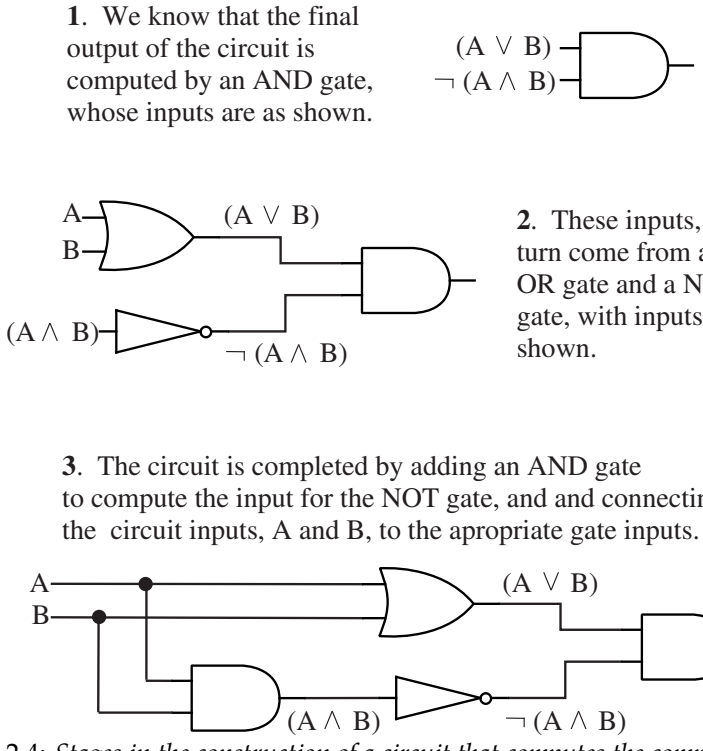


Figure 2.4: *Stages in the construction of a circuit that computes the compound proposition* $(A \vee B) \wedge \neg(A \wedge B)$.

circuit.

Once the main operator has been identified and represented as a logic gate, you just have to build circuits to compute the input or inputs to that operator. In the example, the inputs to the main AND gate come from two subcircuits. One subcircuit computes the value of $A \vee B$ and the other computes the value of $\neg(A \wedge B)$. Building each subcircuit is a separate problem, but smaller than the problem you started with. Eventually, you'll come to a gate whose input comes directly from one of the input wires—A or B in this case—instead of from a subcircuit.

2.3.3 From circuits to propositions

So, every compound proposition is computed by a logic circuit with one output wire. Is the reverse true? That is, given a logic circuit with one output, is there a proposition

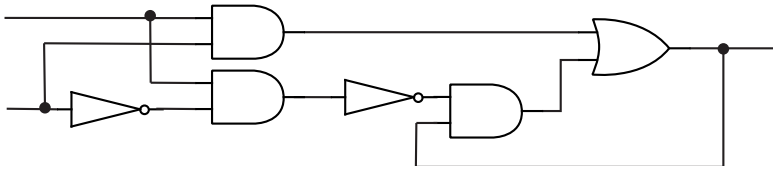


Figure 2.5: *This circuit contains a feedback loop, so it is not a combinatorial logic circuit. The feedback loop includes the* AND *gate and the* OR *gate on the right. This circuit does not compute the value of a compound proposition. This circuit does, however, play an important role in computer memories, since it can be used to store a logical value.*

that expresses the value of the output in terms of the values of the inputs? Not quite. When you wire together some logic gates to make a circuit, there is nothing to stop you from introducing feedback loops. A feedback loop occurs when the output from a gate is connected—possibly through one or more intermediate gates—back to an input of the same gate. Figure 2.5 shows an example of a circuit with a feedback loop. Feedback loops cannot be described by compound propositions, basically because there is no place to start, no input to associate with a propositional variable. But feedback loops are the only thing that can go wrong. A logic circuit that does not contain any feedback loops is called a ***combinatorial logic circuit***. Every combinatorial logic circuit with just one output computes the value of some compound proposition. The propositional variables in the compound proposition are just names associated with the input wires of the circuit. (Of course, if the circuit has more than one output, you can simply use a different proposition for each output.)

The key to understanding why this is true is to note that each wire in the circuit—not just the final output wire—represents the value of some proposition. Furthermore, once you know which proposition is represented by each input wire to a gate, it's obvious what proposition is represented by the output: You just combine the input propositions with the appropriate $\wedge$, $\vee$, or $\neg$ operator, depending on what type of gate it is. To find the proposition associated with the final output, you just have to start from the inputs and move through the circuit, labeling the output wire of each gate with the proposition that it represents. Figure 2.6 illustrates this process.

2.3.4 Disjunctive Normal Form

Compound propositions, then, correspond naturally with combinatorial logic circuits. But we have still not quite settled the question of just how powerful these circuits and propositions are. We've looked at a number of logical operators and noted that they can all be expressed in terms of $\wedge$, $\vee$, and $\neg$. But might there be other operators that cannot be so expressed? Equivalently, might there be other types of logic gates—possibly with

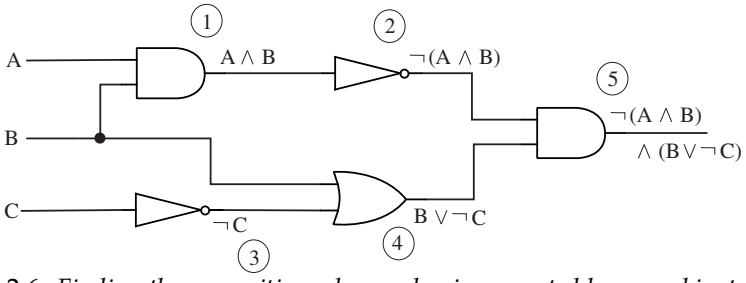


Figure 2.6: *Finding the proposition whose value is computed by a combinatorial logic circuit. Each wire in the circuit is labeled with the proposition that it represents. The numbering of the labels shows one of the orders in which they can be associated with the wires. The circuit as a whole computes the value of* $\neg(A \wedge B) \wedge (B \vee \neg C)$.

some large number of inputs—whose computations cannot be duplicated with AND, OR, and NOT gates? Any logical operator or logic gate computes a value for each possible combination of logical values of its inputs. We could always make a truth table showing the output for each possible combination of inputs. As it turns out, given *any* such truth table, it is possible to find a proposition, containing only the $\wedge$, $\vee$, and $\neg$ operators, whose value for each combination of inputs is given precisely by that table.

To see why this is true, it is useful to introduce a particular type of compound proposition. Define a ***simple term*** to be either a propositional variable or the negation of a propositional variable. A conjunction of simple terms would then consist of one or more simple terms put together with $\wedge$ operators. (A 'conjunction of one simple term' is just a single simple term by itself. This might not make grammatical sense, but it's the way mathematicians think.) Some examples of conjunctions of simple terms would be $p \wedge q$, p, $\neg q$, and $p \wedge \neg r \wedge \neg w \wedge s \wedge t$. Finally, we can take one or more such conjunctions and join them into a 'disjunction of conjunctions of simple terms'. This is the type of compound proposition we need. We can avoid some redundancy by assuming that no propositional variable occurs more than once in a single conjunction (since $p \wedge p$ can be replaced by p, and if p and $\neg p$ both occur in a conjunction, then the value of the conjuction is false, and it can be eliminated.) We can also assume that the same conjunction does not occur twice in the disjunction.

Normal forms are part of the syllabus for *Reasoning & Logic*. These normal forms, such as Disjunctive Normal Form (this subsection) and Conjunctive Normal Form (see the exercises), are important in propositional logic. There are normal forms for other logics, too, such as for *predicate logic* which we'll look at in the next Section 2.4.

Definition 2.6. A compound proposition is said to be in ***disjunctive normal form***, or DNF, if it is a disjunction of conjunctions of simple terms, and if, furthermore, each propositional variable occurs at most once in each conjunction and each conjunction occurs at most once in the disjunction.

Using p, q, r, s, A, and B as propositional variables, here are a few examples of propositions that are in disjunctive normal form:

$$(p \wedge q \wedge r) \vee (p \wedge \neg q \wedge r \wedge s) \vee (\neg p \wedge \neg q)$$
$$(p \wedge \neg q)$$
$$(A \wedge \neg B) \vee (\neg A \wedge B)$$
$$p \vee (\neg p \wedge q) \vee (\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge \neg q \wedge \neg r \wedge w)$$

Propositions in DNF are just what we need to deal with input/output tables of the type that we have been discussing. Any such table can be computed by a proposition in disjunctive normal form. It follows that it is possible to build a circuit to compute that table using only AND, OR, and NOT gates.

Theorem 2.3. *Consider a table that lists a logical output value for every combination of values of several propositional variables. Assume that at least one of the output values is true. Then there is a proposition containing those variables such that the value of the proposition for each possible combination of the values of the variables is precisely the value specified in the table. It is possible to choose the proposition to be in disjunctive normal form.*

Proof. Consider any row in the table for which the output value is $\mathbb{T}$. Form a conjunction of simple terms as follows: For each variable, p, whose value is $\mathbb{T}$ in that row, include p itself in the conjunction; for each variable, q, whose value is $\mathbb{F}$ in the row, include $\neg q$ in the conjunction. The value of this conjunction is $\mathbb{T}$ for the combination of variable values given in that row of the table, since each of the terms in the conjuction is true for that combination of variables. Furthermore, for any *other* possible combination of variable values, the value of the conjunction will be $\mathbb{F}$, since at least one of the simple terms in the conjunction will be false.

Take the disjunction of all such conjunctions constructed in this way, for each row in the table where the output value is true. This disjunction has the value $\mathbb{T}$ if and only if one of the conjunctions that make it up has the value $\mathbb{T}$—and that is precisely when the output value specified by the table is $\mathbb{T}$. So, this disjunction of conjunctions satisfies the requirements of the theorem. □

This is the first proof of a non-trivial claim that we've seen. You will learn about theorems and proofs, and proof techniques, at the end of this chapter and in Chapter 3.

p	q	r	output	
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	
$\mathbb{F}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$(\neg p \wedge \neg q \wedge r)$
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$	
$\mathbb{F}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$(\neg p \wedge q \wedge r)$
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$	$\mathbb{F}$	
$\mathbb{T}$	$\mathbb{F}$	$\mathbb{T}$	$\mathbb{F}$	
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{F}$	$\mathbb{F}$	
$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$\mathbb{T}$	$p \wedge q \wedge r$

Figure 2.7: *An input/output table specifying a desired output for each combination of values of the propositional variables p, q, and r. Each row where the output is $\mathbb{T}$ corresponds to a conjunction, shown next to that row in the table. The disjunction of these conjunctions is a proposition whose output values are precisely those specified by the table.*

As an example, consider the table in Figure 2.7. This table specifies a desired output value for each possible combination of values for the propositional variables p, q, and r. Look at the second row of the table, where the output value is true. According to the proof of the theorem, this row corresponds to the conjunction $(\neg p \wedge \neg q \wedge r)$. This conjunction is true when p is false, q is false, and r is true; in all other cases it is false, since in any other case at least one of the terms $\neg p$, $\neg q$, or r is false. The other two rows where the output is true give two more conjunctions. The three conjunctions are combined to produce the DNF proposition $(\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (p \wedge q \wedge r)$. This proposition computes all the output values specified in the table. Using this proposition as a blueprint, we get a logic circuit whose outputs match those given in the table.

Now, given any combinatorial logic circuit, there are many other circuits that have the same input/output behaviour. When two circuits have the same input/output table, the compound propositions associated with the two circuits are logically equivalent. To put this another way, propositions that are logically equivalent produce circuits that have the same input/output behaviour. As a practical matter, we will usually prefer the circuit that is simpler. The correspondence between circuits and propositions allows us to apply Boolean algebra to the simplification of circuits.

Our preference for simpler applies to compound propositions, whether or not they correspond to circuits. We usually prefer the equivalent form of the proposition that is simpler. Any proposition has an equivalent proposition in DNF. So when proving a theorem about compound propositions, it is sufficient to consider only DNF propositions. This can make the proof easier to write.

For example, consider the DNF proposition corresponding to the table in Figure 2.7. In $(\neg p \wedge \neg q \wedge r) \vee (\neg p \wedge q \wedge r) \vee (p \wedge q \wedge r)$, we can factor $(q \wedge r)$ from the last two terms, giving $(\neg p \wedge \neg q \wedge r) \vee ((\neg p \vee p) \wedge (q \wedge r))$. Since $\neg p \vee p \equiv \mathbb{T}$, and $\mathbb{T} \wedge Q \equiv Q$ for any proposition Q, this can be simplified to $(\neg p \wedge \neg q \wedge r) \vee (q \wedge r)$. Again, we can apply the distributive law to this to factor out an r, giving $((\neg p \wedge \neg q) \vee q) \wedge r)$. This compound proposition is logically equivalent to the one we started with, but implementing it in a circuit requires only five logic gates, instead of the ten required by the original proposition.[8]

If you start with a circuit instead of a proposition, it is often possible to find the associated proposition, simplify it using Boolean algebra, and use the simplified proposition to build an equivalent circuit that is simpler than the original. And simplifying a proposition to DNF is often a sensible approach.

One way to simplify propositions is using a ***Karnaugh-map*** (or *K-map* for short) as you will learn in *Computer Organisation*. Using a K-map you can find what they will call a 'minimal sum of products'. Notice that a sum of products is just a proposition written in DNF. For the course of *Reasoning & Logic* we may ask you to translate propositions to a DNF form. You can then choose to either do so using rewrite rules, but you are also free to use a K-map if you prefer. In one of the pencasts of this course I show how both methods lead to a result in DNF: `youtu.be/GwVngCU9eYY`.

2.3.5 Binary addition

All this explains nicely the relationship between logic and circuits, but it doesn't explain why logic circuits should be used in computers in the first place. Part of the explanation is found in the fact that computers use binary numbers. A binary number is a string of zeros and ones. Binary numbers are easy to represent in an electronic device like a

[8] No, I didn't count wrong. There are eleven logical operators in the original expression, but you can get by with ten gates in the circuit: Use a single NOT gate to compute $\neg p$, and connect the output of that gate to two different AND gates. Reusing the output of a logic gate is an obvious way to simplify circuits that does not correspond to any operation on propositions.

A	B	C	output
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	1

A	B	C	output
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Figure 2.8: *Input/output tables for the addition of three binary digits, A, B, and C.*

computer: Each position in the number corresponds to a wire. When the wire is on, it represents one; when the wire is off, it represents zero. When we are thinking in terms of logic, the same states of the wire represent true and false, but either representation is just an interpretation of the reality, which is a wire that is on or off. The question is whether the interpretation is fruitful.

Once wires are thought of as representing zeros and ones, we can build circuits to do computations with binary numbers. Which computations? Any that we want! If we know what the answer should be for each combination of inputs, then by Theorem 2.3 we can build a circuit to compute that answer. Of course, the procedure described in that theorem is only practical for small circuits, but small circuits can be used as building blocks to make all the calculating circuits in a computer.

For example, let's look at binary addition. To add two ordinary, decimal numbers, you line them up one on top of the other, and add the digits in each column. In each column, there might also be a carry from the previous column. To add up a column, you only need to remember a small number of rules, such as $7+6+1=14$ and $3+5+0=8$. For binary addition, it's even easier, since the only digits are 0 and 1. There are only eight rules:

$$0+0+0=00 \qquad 1+0+0=01$$
$$0+0+1=01 \qquad 1+0+1=10$$
$$0+1+0=01 \qquad 1+1+0=10$$
$$0+1+1=10 \qquad 1+1+1=11$$

Here, I've written each sum using two digits. In a multi-column addition, one of these digits is carried over to the next column. Here, we have a calculation that has three inputs and two outputs. We can make an input/output table for each of the two outputs.

The tables are shown in Figure 2.8. We know that these tables can be implemented as combinatorial circuits, so we know that circuits can add binary numbers. To add multi-digit binary numbers, we just need one copy of the basic addition circuit for each column in the sum.

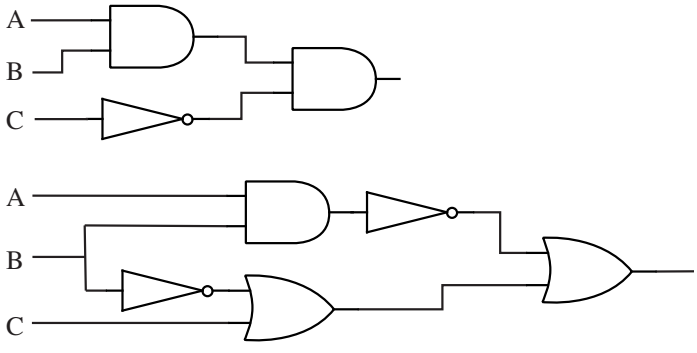
Exercises

1. Using only AND, OR, and NOT gates, draw circuits that compute the value of each of the propositions $A \rightarrow B$, $A \leftrightarrow B$, and $A \oplus B$.

2. For each of the following propositions, find a combinatorial logic circuit that computes that proposition:

a) $A \wedge (B \vee \neg C)$ **b)** $(p \wedge q) \wedge \neg(p \wedge \neg q)$
c) $(p \vee q \vee r) \wedge (\neg p \vee \neg q \vee \neg r)$ **d)** $\neg(A \wedge (B \vee C)) \vee (B \wedge \neg A)$

3. Find the compound proposition computed by each of the following circuits:

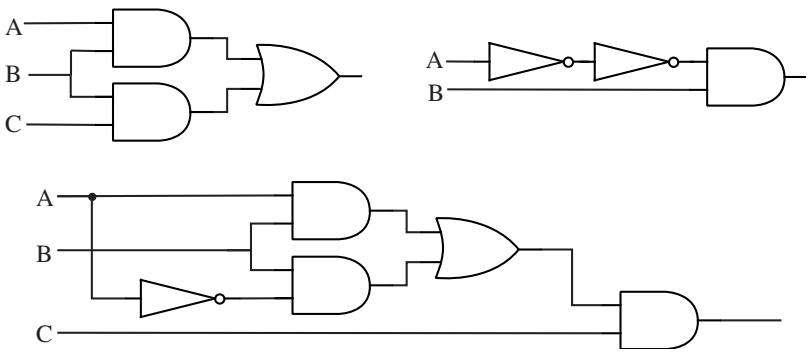


4. This section describes a method for finding the compound proposition computed by any combinatorial logic circuit. This method fails if you try to apply it to a circuit that contains a feedback loop. What goes wrong? Give an example.

5. Show that every compound proposition which is not a contradiction is equivalent to a proposition in disjunctive normal form. (Note: We can eliminate the restriction that the compound proposition is not a contradiction by agreeing that "$\mathbb{F}$" counts as a proposition in disjunctive normal form. $\mathbb{F}$ is logically equivalent to any contradiction.)

6. A proposition in ***conjunctive normal form*** (CNF) is a conjunction of disjunctions of simple terms (with the proviso, as in the definition of DNF that a single item counts as a conjunction or disjunction). Show that every compound proposition which is not a tautology is logically equivalent to a compound proposition in conjunctive normal form. (Hint: What happens if you take the negation of a DNF proposition and apply DeMorgan's Laws?)

7. Use the laws of Boolean algebra to simplify each of the following circuits:



8. Design circuits to implement the input/output tables for addition, as given in Figure 2.8. Try to make your circuits as simple as possible. (The circuits that are used in real computers for this purpose are more simplified than the ones you will probably come up with, but the general approach of using logic to design computer circuits is valid. If you are interested to learn more about this, the second year variant course *Digital Systems* describes circuit design in more detail.)

2.4 Predicate Logic

In propositional logic, we can let p stand for "Roses are red" and q stand for "Violets are blue". Then $p \wedge q$ will stand for "Roses are red and violets are blue". But we lose a lot in the translation into logic. Since propositional logic only deals with truth values, there's nothing we can do with p and q in propositional logic that has anything to do with roses, violets, or colour. To apply logic to such things, we need ***predicates***. The type of logic that uses predicates is called ***predicate logic*** or, when the emphasis is on manipulating and reasoning with predicates, ***predicate calculus***.

2.4.1 Predicates

A predicate is a kind of incomplete proposition, which becomes a proposition when it is applied to some entity (or, as we'll see later, to several entities). In the proposition "the rose is red", the predicate is *is red*. By itself, 'is red' is not a proposition. Think of it as having an empty slot, that needs to be filled in to make a proposition: "— is red". In the proposition "the rose is red", the slot is filled by the entity "the rose", but it could just as well be filled by other entities: "the barn is red"; "the wine is red"; "the banana is red". Each of these propositions uses the same predicate, but they are different propositions and they can have different truth values.

If P is a predicate and a is an entity, then $P(a)$ stands for the proposition that is formed when P is applied to a. If P represents 'is red' and a stands for 'the rose', then $P(a)$ is 'the rose is red'. If M is the predicate 'is mortal' and s is 'Socrates', then $M(s)$ is the proposition "Socrates is mortal".

Now, you might be asking, just what is an *entity* anyway? I am using the term here to mean some specific, identifiable thing to which a predicate can be applied. Generally, it doesn't make sense to apply a given predicate to every possible entity, but only to entities in a certain category. For example, it probably doesn't make sense to apply the predicate 'is mortal' to your living room sofa. This predicate only applies to entities in the category of living things, since there is no way something can be mortal unless it is alive. This category is called the domain of discourse for the predicate.[9]

We are now ready for a formal definition of one-place predicates. A one-place predicate, like all the examples we have seen so far, has a single slot which can be filled in with one entity:

Definition 2.7. A ***one-place predicate*** associates a proposition with each entity in some collection of entities. This collection is called the ***domain of discourse*** for the predicate. If P is a predicate and a is an entity in the domain of discourse for P, then $P(a)$ denotes the proposition that is associated with a by P. We say that $P(a)$ is the result of ***applying*** P to a.

We can obviously extend this to predicates that can be applied to two or more entities. In the proposition "John loves Mary", *loves* is a two-place predicate. Besides John and Mary, it could be applied to other pairs of entities: "John loves Jane", "Bill loves Mary", "John loves Bill", "John loves John". If Q is a two-place predicate, then $Q(a, b)$ denotes the proposition that is obtained when Q is applied to the entities a and b. Note that each of the 'slots' in a two-place predicate can have its own domain of discourse. For example, if Q represents the predicate 'owns', then $Q(a, b)$ will only make sense when a is a person and b is an inanimate object. An example of a three-place predicate is "a gave b to c", and a four-place predicate would be "a bought b from c for d euros". But keep in mind that not every predicate has to correspond to an English sentence.

When predicates are applied to entities, the results are propositions, and all the operators of propositional logic can be applied to these propositions just as they can to any propositions. Let R be the predicate 'is red', and let L be the two-place predicate 'loves'. If a, b, j, m, and b are entities belonging to the appropriate categories, then we can form compound propositions such as:

$R(a) \wedge R(b)$	a is red and b is red
$\neg R(a)$	a is not red
$L(j, m) \wedge \neg L(m, j)$	j loves m, and m does not love j
$L(j, m) \rightarrow L(b, m)$	if j loves m then b loves m
$R(a) \leftrightarrow L(j, j)$	a is red if and only if j loves j

[9] In the language of set theory, which will be introduced in Chapter 4, we would say that a domain of discourse is a set, U, and a predicate is a function from U to the set of truth values. The definition should be clear enough without the formal language of set theory, and in fact you should think of this definition—and many others—as motivation for that language.

Predicate logic is founded on the ideas developed by Charles Sanders Peirce (1839–1914), an American philosopher, logician, mathematician, and scientist.[10] Many of his contributions to logic were appreciated only years after he died. He has been called "the most original and versatile of American philosophers and America's greatest logician." and "one of the greatest philosophers ever". As early as 1886 he saw that logical operations could be carried out by electrical switching circuits; the same idea was used decades later to produce digital computers, as we saw in Section 2.3. You can read about his colourful life at the link below.

Source: `en.wikipedia.org/wiki/Charles_Sanders_Peirce`.

2.4.2 Quantifiers

Let's go back to the proposition with which we started this section: "Roses are red". This sentence is more difficult to handle than it might appear. We still can't express it properly in logic. The problem is that this proposition is not saying something about some particular entity. It really says that *all* roses are red (which happens to be a false statement, but that's what it means). Predicates can only be applied to individual entities.

Many other sentences raise similar difficulties: "All persons are mortal." "Some roses are red, but no roses are black." "All maths courses are interesting." "Every prime number greater than two is odd." Words like *all*, *no*, *some*, and *every* are called ***quantifiers***. We need to be able to express similar concepts in logic.

Suppose that P is a predicate, and we want to express the proposition that P is true when applied to any entity in the domain of discourse. That is, we want to say "for any entity x in the domain of discourse, $P(x)$ is true". In predicate logic, we write this in symbols as $\forall x(P(x))$. The $\forall$ symbol, which looks like an upside-down 'A', is usually read 'for all', so that $\forall x(P(x))$ is read as 'for all x, $P(x)$'. (It is understood that this means for all x in the domain of discourse for P.) For example, if R is the predicate 'is red' and the domain of discourse consists of all roses, then $\forall x(R(x))$ expresses the proposition "All roses are red". Note that the same proposition could be expressed in English as "Every rose is red" or "Any rose is red".

Now, suppose we want to say that a predicate, P, is true for *some* entity in its domain of discourse. This is expressed in predicate logic as $\exists x(P(x))$. The $\exists$ symbol, which

[10] And also on the ideas developed independently around the same time by German philosopher and logician Gottlob Frege, who we'll see in Section 3.1.

looks like a backwards 'E', is usually read 'there exists', but a more exact reading would be 'there is at least one'. Thus, $\exists x(P(x))$ is read as 'There exists an x such that $P(x)$' , and it means "there is at least one x in the domain of discourse for P for which $P(x)$ is true". If, once again, R stands for 'is red' and the domain of discourse is 'roses', then $\exists x(R(x))$ could be expressed in English as "There is a red rose" or "At least one rose is red" or "Some rose is red". It might also be expressed as "Some roses are red", but the plural is a bit misleading since $\exists x(R(x))$ is true even if there is only one red rose. We can now give the formal definitions:

Definition 2.8. Suppose that P is a one-place predicate. Then $\forall x(P(x))$ is a proposition, which is true if and only if $P(a)$ is true for every entity a in the domain of discourse for P. And $\exists x(P(x))$ is a proposition which is true if and only if there is at least one entity, a, in the domain of discourse for P for which $P(a)$ is true. The $\forall$ symbol is called the ***universal quantifier***, and $\exists$ is called the ***existential quantifier***.

The x in $\forall x(P(x))$ and $\exists x(P(x))$ is a variable. (More precisely, it is an *entity* variable, since its value can only be an entity.) Note that a plain $P(x)$—without the $\forall x$ or $\exists x$—is not a proposition. $P(x)$ is neither true nor false because x is not some particular entity, but just a placeholder in a slot that can be filled in with an entity. $P(x)$ would stand for something like the statement 'x is red', which is not really a statement in English at all. But it becomes a statement when the x is replaced by some particular entity, such as 'the rose'. Similarly, $P(x)$ becomes a proposition if some entity a is substituted for the x, giving $P(a)$.[11]

An ***open statement*** is an expression that contains one or more entity variables, which becomes a proposition when entities are substituted for the variables. (An open statement has open 'slots' that need to be filled in.) $P(x)$ and "x is red" are examples of open statements that contain one variable. If L is a two-place predicate and x and y are variables, then $L(x, y)$ is an open statement containing two variables. An example in English would be "x loves y". The variables in an open statement are called ***free variables***. An open statement that contains x as a free variable can be quantified with $\forall x$ or $\exists x$. The variable x is then said to be ***bound***. For example, x is free in $P(x)$ and is bound in $\forall x(P(x))$ and $\exists x(P(x))$. The free variable y in $L(x, y)$ becomes bound in $\forall y(L(x, y))$ and in $\exists y(L(x, y))$.

Note that $\forall y(L(x, y))$ is still an open statement, since it contains x as a free variable. Therefore, it is possible to apply the quantifier $\forall x$ or $\exists x$ to $\forall y(L(x, y))$, giving $\forall x\big(\forall y(L(x, y))\big)$ and $\exists x\big(\forall y(L(x, y))\big)$. Since all the variables are bound in these expressions, they are propositions. If $L(x, y)$ represents 'x loves y', then $\forall y(L(x, y))$ is something like "x loves everyone", and $\exists x\big(\forall y(L(x, y))\big)$ is the proposition, "There is someone who loves everyone". Of course, we could also have started with $\exists x(L(x, y))$: "There is someone who loves y". Applying $\forall y$ to this gives $\forall y\big(\exists x(L(x, y))\big)$, which means

[11] There is certainly room for confusion about names here. In this discussion, x is a variable and a is an entity. But that's only because I said so. Any letter could be used in either role, and you have to pay attention to the context to figure out what is going on. Usually, x, y, and z will be variables.

"For every person, there is someone who loves that person". Note in particular that $\exists x\big(\forall y(L(x,y))\big)$ and $\forall y\big(\exists x(L(x,y))\big)$ do *not* mean the same thing. Altogether, there are eight different propositions that can be obtained from $L(x,y)$ by applying quantifiers, with six distinct meanings among them.

From now on, I will leave out parentheses when there is no ambiguity. For example, I will write $\forall x\, P(x)$ instead of $\forall x(P(x))$ and $\exists x\, \exists y\, L(x,y)$ instead of $\exists x\big(\exists y(L(x,y))\big)$. Make sure though that when you leave out the parentheses you do so only when no ambiguity exists. In one of the problems of this chapter, you will see an example of two very similar statements where the parentheses do change the meaning significantly!

Further, I will sometimes give predicates and entities names that are complete words instead of just letters, as in $Red(x)$ and $Loves(john, mary)$. This might help to make examples more readable.

2.4.3 Operators

In predicate logic, the operators and laws of Boolean algebra still apply. For example, if P and Q are one-place predicates and a is an entity in the domain of discourse, then $P(a) \rightarrow Q(a)$ is a proposition, and it is logically equivalent to $\neg P(a) \vee Q(a)$. Furthermore, if x is a variable, then $P(x) \rightarrow Q(x)$ is an open statement, and $\forall x(P(x) \rightarrow Q(x))$ is a proposition. So are $P(a) \wedge (\exists x\, Q(x))$ and $(\forall x\, P(x)) \rightarrow (\exists x P(x))$. Obviously, predicate logic can be very expressive. Unfortunately, the translation between predicate logic and English sentences is not always obvious.

One of the commonly-made mistakes in predicate logic is the difference in translation between statements like: "All humans are mortal" and "There is a human that is mortal". I discuss the difference in translation of these statements in one of our pencasts: `youtu.be/BJeGHIX_ldY`.

Let's look one more time at the proposition "Roses are red". If the domain of discourse consists of roses, this translates into predicate logic as $\forall x\, Red(x)$. However, the sentence makes more sense if the domain of discourse is larger—for example if it consists of all flowers. Then "Roses are red" has to be read as "All flowers which are roses are red", or "For any flower, if that flower is a rose, then it is red". The last form translates directly into logic as $\forall x\big(Rose(x) \rightarrow Red(x)\big)$. Suppose we want to say that all red roses are pretty. The phrase 'red rose' is saying both that the flower is a rose and that it is red, and it must be translated as a conjunction, $Rose(x) \wedge Red(x)$. So, "All red roses are pretty" can be

rendered as $\forall x\big((Rose(x) \wedge Red(x)) \rightarrow Pretty(x)\big)$.

Here are a few more examples of translations from predicate logic to English. Let $H(x)$ represent 'x is happy', let $C(y)$ represent 'y is a computer', and let $O(x, y)$ represent 'x owns y'. Then we have the following translations:

- Jack owns a computer: $\exists x\big(O(jack, x) \wedge C(x)\big)$. (That is, there is at least one thing such that Jack owns that thing and that thing is a computer.)
- Everything Jack owns is a computer: $\forall x\big(O(jack, x) \rightarrow C(x)\big)$.
- If Jack owns a computer, then he's happy:
  $\big(\exists y(O(jack, y) \wedge C(y))\big) \rightarrow H(jack)$.
- Everyone who owns a computer is happy:
  $\forall x\big(\ \big(\exists y(O(x, y) \wedge C(y))\big) \rightarrow H(x)\big)\ \big)$.
- Everyone owns a computer: $\forall x\, \exists y\big(C(y) \wedge O(x, y)\big)$. (Note that this allows each person to own a different computer. The proposition $\exists y\, \forall x\big(C(y) \wedge O(x, y)\big)$ would mean that there is a single computer which is owned by everyone.)
- Everyone is happy: $\forall x H(x)$.
- Everyone is unhappy: $\forall x(\neg H(x))$.
- Someone is unhappy: $\exists x(\neg H(x))$.
- At least two people are happy: $\exists x \exists y\big(H(x) \wedge H(y) \wedge (x \neq y)\big)$. (The stipulation that $x \neq y$ is necessary because two different variables can refer to the same entity. The proposition $\exists x \exists y(H(x) \wedge H(y))$ is true even if there is only one happy person.)
- There is exactly one happy person:
  $\big(\exists x H(x)\big)\big) \wedge \big(\forall y \forall z((H(y) \wedge H(z)) \rightarrow (y = z))\big)$. (The first part of this conjunction says that there is at least one happy person. The second part says that if y and z are both happy people, then they are actually the same person. That is, it's not possible to find two *different* people who are happy. The statement can be simplified a little however, to get: $\exists x(H(x) \wedge \forall y(H(y) \rightarrow (x = y)))$ Do you see why this works as well?)

2.4.4 Tarski's world and formal structures

To help you reason about sets of predicate logic statements, or even arguments expressed in predicate logic, we often use a 'mathematical structure'. For some of these structures a visualisation in the form of ***Tarski's world*** can sometimes be useful.

What is truth? In 1933, Polish mathematician Alfred Tarski (1901–1983) published a very long paper in Polish (titled *Pojęcie prawdy w językach nauk dedukcyjnych*), setting out a mathematical definition of truth for formal languages. "Along with his contemporary, Kurt Gödel [who we'll see in Chapter 4], he changed the face of logic in the twentieth century, especially through his work on the concept of truth and the theory of models."

Source: `en.wikipedia.org/wiki/Alfred_Tarski`.

In *Tarski's world*, it is possible to describe situations using formulas whose truth can be evaluated, which are expressed in a first-order language that uses predicates such as Rightof(x, y), which means that x is situated—somewhere, not necessarily directly—to the right of y, or Blue(x), which means that x is blue. In the world in Figure 2.9, for instance, the formula $\forall x(\text{Triangle}(x) \rightarrow \text{Blue}(x))$ holds, since all triangles are blue, but the converse of this formula, $\forall x(\text{Blue}(x) \rightarrow \text{Triangle}(x))$, does not hold, since object c is blue but not a triangle.

Such an instance of Tarski world can be more formally described as a 'mathematical structure' (which we refer to as a ***formal structure*** occasionally). These structures allow us to evaluate statements in predicate logic as being true or false. To formalise a structure,

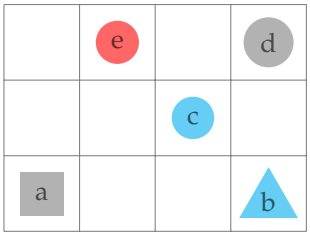


Figure 2.9: An instance of a Tarski World.

we need to describe two things: the domain of discourse D of the structure and for all of the predicates, for which objects of the domain they are true. We do so using *set-notation* which we discuss in more depth in Chapter 4. The formal description of the structure S depicted in Figure 2.9 is:

- $D = \{a, b, c, d, e\}$
- $Blue^S = \{b, c\}$
- $Gray^S = \{a, d\}$
- $Red^S = \{e\}$
- $Square^S = \{a\}$
- $Triangle^S = \{b\}$
- $Circle^S = \{c, d, e\}$
- $RightOf^S = \{(b,a), (c,a), (d,a), (e,a), (b,c), (d,c), (b,e), (c,e), (d,e)\}$
- $LeftOf^S = \{(a,b), (c,b), (e,b), (a,c), (e,c), (a,d), (c,d), (e,d), (a,e)\}$
- $BelowOf^S = \{(a,c), (a,d), (a,e), (b,c), (b,d), (b,e), (c,d), (c,e)\}$
- $AboveOf^S = \{(c,a), (c,b), (d,a), (d,b), (d,c), (e,a), (e,b), (e,c)\}$

Notice that for the one-place predicates we have a set of objects for which this predicate is true (e.g., only b and c are blue) and such a set is denoted using '{' and '}' symbols, called 'curly braces' or just 'braces'.[12] For the two-place predicates we have a set of ***tuples*** that are denoted using '(' and ')' symbols, called 'parentheses' or 'round brackets'. In this case, for instance, the fact that (a, b) is in the set $LeftOf^S$ means that $LeftOf(a, b)$ is true for this structure, i.e., a is left of b.

Such formal structures can also be defined to disprove arguments written in predicate logic, as we will see in Section 2.5.3.

2.4.5 Logical equivalence

To calculate in predicate logic, we need a notion of logical equivalence. Clearly, there are pairs of propositions in predicate logic that mean the same thing. Consider the propositions $\neg(\forall x H(x))$ and $\exists x(\neg H(x))$, where $H(x)$ represents 'x is happy'. The first of these propositions means "Not everyone is happy", and the second means "Someone is not happy". These statements have the same truth value: If not everyone is happy, then someone is unhappy and vice versa. But logical equivalence is much stronger than just having the same truth value. In propositional logic, logical equivalence is defined in terms of propositional variables: two compound propositions are logically equivalent if they have the same truth values for all possible truth values of the propositional variables they contain. In predicate logic, two formulas are logically equivalent if they have the same truth value for all possible predicates.

Consider $\neg(\forall x P(x))$ and $\exists x(\neg P(x))$. These formulas make sense for any predicate P, and for any predicate P they have the same truth value. Unfortunately, we can't—as we did in propositional logic—just check this fact with a truth table: there are no

[12] See Chapter 4.

$\neg(\forall x P(x)) \equiv \exists x(\neg P(x))$
$\neg(\exists x P(x)) \equiv \forall x(\neg P(x))$
$\forall x \forall y Q(x,y) \equiv \forall y \forall x Q(x,y)$
$\exists x \exists y Q(x,y) \equiv \exists y \exists x Q(x,y)$

Figure 2.10: *Four important rules of predicate logic. P can be any one-place predicate, and Q can be any two-place predicate. The first two rules are called DeMorgan's Laws for predicate logic.*

subpropositions, connected by $\wedge$, $\vee$, etc, out of which to build a table. So, let's reason it out: To say $\neg(\forall x P(x))$ is true is just to say that it is not the case that $P(x)$ is true for all possible entities x. So, there must be some entity a for which $P(a)$ is false. Since $P(a)$ is false, $\neg P(a)$ is true. But saying that there is an a for which $\neg P(a)$ is true is just saying that $\exists x(\neg P(x))$ is true. So, the truth of $\neg(\forall x P(x))$ implies the truth of $\exists x(\neg P(x))$. On the other hand, if $\neg(\forall x P(x))$ is false, then $\forall x P(x)$ is true. Since $P(x)$ is true for every x, $\neg P(x)$ is false for every x; that is, there is no entity a for which the statement $\neg P(a)$ is true. But this just means that the statement $\exists x(\neg P(x))$ is false. In any case, then, the truth values of $\neg(\forall x P(x))$ and $\exists x(\neg P(x))$ are the same. Since this is true for any predicate P, we will say that these two formulas are logically equivalent and write $\neg(\forall x P(x)) \equiv \exists x(\neg P(x))$.

A similar argument would show that $\neg(\exists x P(x)) \equiv \forall x(\neg P(x))$. These two equivalences, which explicate the relation between negation and quantification, are known as DeMorgan's Laws for predicate logic. (They are closely related to DeMorgan's Laws for propositional logic; see the exercises.) These laws can be used to help simplify expressions. For example,

$$\begin{aligned} \neg\forall y(R(y) \vee Q(y)) &\equiv \exists y(\neg(R(y) \vee Q(y))) \\ &\equiv \exists y((\neg R(y)) \wedge (\neg Q(y)) \end{aligned}$$

It might not be clear exactly why this qualifies as a 'simplification', but it's generally considered simpler to have the negation operator applied to basic propositions such as $R(y)$, rather than to quantified expressions such as $\forall y(R(y) \vee Q(y))$. For a more complicated

example:

$$\begin{aligned}
&\neg\,\exists x\big(P(x)\wedge(\forall y(Q(y)\to Q(x)))\big)\\
&\qquad\equiv \forall x\big(\neg\big(P(x)\wedge(\forall y(Q(y)\to Q(x)))\big)\\
&\qquad\equiv \forall x\big((\neg P(x))\vee(\neg\forall y(Q(y)\to Q(x)))\big)\\
&\qquad\equiv \forall x\big((\neg P(x))\vee(\exists y(\neg(Q(y)\to Q(x))))\big)\\
&\qquad\equiv \forall x\big((\neg P(x))\vee(\exists y(\neg(\neg Q(y)\vee Q(x))))\big)\\
&\qquad\equiv \forall x\big((\neg P(x))\vee(\exists y(\neg\neg Q(y)\wedge\neg Q(x)))\big)\\
&\qquad\equiv \forall x\big((\neg P(x))\vee(\exists y(Q(y)\wedge\neg Q(x)))\big)
\end{aligned}$$

DeMorgan's Laws are listed in Figure 2.10 along with two other laws of predicate logic. The other laws allow you to interchange the order of the variables when two quantifiers of the same type (both $\exists$ or $\forall$) occur together.

Notice however that we may not change the order of quantifiers that are not the same! For instance: $\forall x\exists y(R(x,y)) \not\equiv \exists y\forall x(R(x,y))$. If you are not convinced about this, try to draw up a Tarski's world that shows this unequivalence.

To define logical equivalence in predicate logic more formally, we need to talk about formulas that contain predicate variables, that is, variables that act as place-holders for arbitrary predicates in the same way that propositional variables are place-holders for propositions and entity variables are place-holders for entities. With this in mind, we can define logical equivalence and the closely related concept of tautology for predicate logic. We'll see that these are crucial pieces of writing proofs.

Definition 2.9. Let $\mathcal{P}$ be a formula of predicate logic which contains one or more predicate variables. $\mathcal{P}$ is said to be a ***tautology*** if it is true whenever all the predicate variables that it contains are replaced by actual predicates. Two formulas $\mathcal{P}$ and $\mathcal{Q}$ are said to be ***logically equivalent*** if $\mathcal{P}\leftrightarrow\mathcal{Q}$ is a tautology, that is if $\mathcal{P}$ and $\mathcal{Q}$ always have the same truth value when the predicate variables they contain are replaced by actual predicates. The notation $\mathcal{P}\equiv\mathcal{Q}$ asserts that $\mathcal{P}$ is logically equivalent to $\mathcal{Q}$.

Exercises

†**1.** Simplify each of the following propositions. In your answer, the $\neg$ operator should be applied only to individual predicates.

a) $\neg\,\forall x(\neg P(x))$ **b)** $\neg\,\exists x(P(x)\wedge Q(x))$

c) $\neg\forall z(P(z) \to Q(z))$ **d)** $\neg\big((\forall x P(x)) \wedge (\forall y Q(y))\big)$
e) $\neg\forall x\exists y P(x,y)$ **f)** $\neg\exists x(R(x) \wedge \forall y S(x,y))$
g) $\neg\exists y(P(y) \leftrightarrow Q(y))$ **h)** $\neg\big(\forall x(P(x) \to (\exists y Q(x,y)))\big)$

2. Give a careful argument to show that the second of DeMorgan's laws for predicate calculus, $\neg(\forall x P(x)) \equiv \exists x(\neg P(x))$, is valid.

3. Find the negation of each of the following propositions. Simplify the result; in your answer, the $\neg$ operator should be applied only to individual predicates.
a) $\exists n(\forall s C(s,n))$
b) $\exists n(\forall s(L(s,n) \to P(s)))$
c) $\exists n(\forall s(L(s,n) \to (\exists x\exists y\exists z Q(x,y,z))))$.
d) $\exists n(\forall s(L(s,n) \to (\exists x\exists y\exists z(s = xyz \wedge R(x,y) \wedge T(y) \wedge U(x,y,z)))))$.

4. Suppose that the domain of discourse for a predicate P contains only two entities. Show that $\forall x P(x)$ is equivalent to a conjunction of two simple propositions, and $\exists x P(x)$ is equivalent to a disjunction. Show that in this case, DeMorgan's Laws for propositional logic and DeMorgan's Laws for predicate logic actually say exactly the same thing. Extend the results to a domain of discourse that contains exactly three entities.

5. Let $H(x)$ stand for "x is happy", where the domain of discourse consists of people. Express the proposition "There are exactly three happy people" in predicate logic.

6. What is the difference between the following two statements?
$\exists x Red(x) \wedge \exists x Square(x)$ and $\exists x(Red(x) \wedge Square(x))$

7. Draw a Tarski world for the last exercise.

†**8.** Express Johan Cruyff's statement "There is only one ball, so you need to have it" in predicate logic.

9. Let $T(x,y)$ stand for 'x has taken y', where the domain of discourse for x consists of students and the domain of discourse for y consists of CS courses (at **TU**Delft). Translate each of the following propositions into an unambiguous English sentence:
a) $\forall x\,\forall y\,T(x,y)$ **b)** $\forall x\,\exists y\,T(x,y)$ **c)** $\forall y\,\exists x\,T(x,y)$
d) $\exists x\,\exists y\,T(x,y)$ **e)** $\exists x\,\forall y\,T(x,y)$ **f)** $\exists y\,\forall x\,T(x,y)$

10. Let $F(x,t)$ stand for "You can fool person x at time t." Translate the following sentence into predicate logic: "You can fool some of the people all of the time, and you can fool all of the people some of the time, but you can't fool all of the people all of the time."

11. Translate each of the following sentences into a proposition using predicate logic. Make up any predicates you need. State clearly what each predicate means.
a) All crows are black.
b) Any white bird is not a crow.
c) Not all politicians are honest.
d) All purple elephants have green feet.
e) There is no one who does not like pizza.
f) Anyone who passes the final exam will pass the course.[13]
g) If x is any positive number, then there is a number y such that $y^2 = x$.

[13] This is not true for *Reasoning & Logic*: see the syllabus.

12. Consider the following description of a Tarski World. Does an instance of a Tarski World exist with these properties? If so, give one with a domain of at most 5 elements. If no such instance exists, explain why not.

- $\forall x(Circle(x) \rightarrow \neg Blue(x))$
- $\exists x(Circle(x)) \wedge \exists x(Blue(x))$
- $RightOf(a, b)$
- $LeftOf(a, b) \vee Square(c)$

†**13.** The sentence "Someone has the answer to every question" is ambiguous. Give two translations of this sentence into predicate logic, and explain the difference in meaning.

14. The sentence "Jane is looking for a dog" is ambiguous. One meaning is that there is some particular dog—maybe the one she lost—that Jane is looking for. The other meaning is that Jane is looking for any old dog—maybe because she wants to buy one. Express the first meaning in predicate logic. Explain why the second meaning is *not* expressed by $\forall x(Dog(x) \rightarrow LooksFor(jane, x))$. In fact, the second meaning cannot be expressed in predicate logic. Philosophers of language spend a lot of time thinking about things like this. They are especially fond of the sentence "Jane is looking for a unicorn", which is not ambiguous when applied to the real world. Why is that?

2.5 Deduction

Logic can be applied to draw conclusions from a set of ***premises***. A premise is just a proposition that is known to be true or that has been accepted to be true for the sake of argument, and a conclusion is a proposition that can be deduced logically from the premises. The idea is that if you believe that the premises are true, then logic forces you to accept that the conclusion is true. An ***argument*** is a claim that a certain conclusion follows from a given set of premises. Here is an argument laid out in a traditional format:

If today is Tuesday, then this is Belgium
Today is Tuesday

$\therefore$ This is Belgium

The premises of the argument are shown above the line, and the conclusion below. The symbol $\therefore$ is read 'therefore'. The claim is that the conclusion, "This is Belgium", can be deduced logically from the two premises, "If today is Tuesday, then this is Belgium" and "Today is Tuesday". In fact, this claim is true. Logic forces you to accept this argument. Why is that?

2.5.1 Arguments

Let p stand for the proposition "Today is Tuesday", and let q stand for the proposition "This is Belgium". Then the above argument has the form

$$\begin{array}{l} p \to q \\ \underline{p \qquad\quad} \\ \therefore\ q \end{array}$$

Now, for *any* propositions p and q—not just the ones in this particular argument—if $p \to q$ is true and p is true, then q must also be true. This is easy to check in a truth table:

p	q	$p \to q$
0	0	1
0	1	1
1	0	0
1	1	1

The only case where both $p \to q$ and p are true is on the last line of the table, and in this case, q is also true. If you believe $p \to q$ and p, you have no logical choice but to believe q. This applies no matter what p and q represent. For example, if you believe "If Jill is breathing, then Jill pays taxes", and you believe that "Jill is breathing", logic forces you to believe that "Jill pays taxes". Note that we can't say for sure that the conclusion is true, only that *if* the premises are true, *then* the conclusion must be true.

This fact can be rephrased by saying that $\big((p \to q) \wedge p\big) \to q$ is a tautology. More generally, for any compound propositions $\mathcal{P}$ and $\mathcal{Q}$, saying "$\mathcal{P} \to \mathcal{Q}$ is a tautology" is the same as saying that "in all cases where $\mathcal{P}$ is true, $\mathcal{Q}$ is also true".[14] We will use the notation $\mathcal{P} \Longrightarrow \mathcal{Q}$ to mean that $\mathcal{P} \to \mathcal{Q}$ is a tautology. Think of $\mathcal{P}$ as being the premise of an argument or the conjunction of several premises. To say $\mathcal{P} \Longrightarrow \mathcal{Q}$ is to say that $\mathcal{Q}$ follows logically from $\mathcal{P}$. We will use the same notation in both propositional logic and predicate logic. (Note that the relation of $\Longrightarrow$ to $\to$ is the same as the relation of $\equiv$ to $\leftrightarrow$.)

Definition 2.10. Let $\mathcal{P}$ and $\mathcal{Q}$ be any formulas in either propositional logic or predicate logic. The notation $\mathcal{P} \Longrightarrow \mathcal{Q}$ is used to mean that $\mathcal{P} \to \mathcal{Q}$ is a tautology. That is, in all cases where $\mathcal{P}$ is true, $\mathcal{Q}$ is also true. We then say that $\mathcal{Q}$ can be ***logically deduced*** from $\mathcal{P}$ or that $\mathcal{P}$ ***logically implies*** $\mathcal{Q}$.

An argument in which the conclusion follows logically from the premises is said to be a ***valid argument***. To test whether an argument is valid, you have to replace the particular propositions or predicates that it contains with variables, and then test whether the conjunction of the premises logically implies the conclusion. We have seen that any argument of the form

[14] Here, "in all cases" means for all combinations of truth values of the propositional variables in $\mathcal{P}$ and $\mathcal{Q}$, i.e., in every situation. Saying $\mathcal{P} \to \mathcal{Q}$ is a tautology means it is true in all cases. But by definition of $\to$, it is automatically true in cases where $\mathcal{P}$ is false. In cases where $\mathcal{P}$ is true, $\mathcal{P} \to \mathcal{Q}$ will be true if and only if $\mathcal{Q}$ is true.

$$\begin{array}{l} p \to q \\ \underline{p \qquad\qquad} \\ \therefore\ q \end{array}$$

is valid, since $\big((p \to q) \wedge p\big) \to q$ is a tautology. This rule of deduction is called ***modus ponens***. It plays a central role in logic. Another, closely related rule is ***modus tollens***, which applies to arguments of the form

$$\begin{array}{l} p \to q \\ \underline{\neg q \qquad\qquad} \\ \therefore\ \neg p \end{array}$$

To verify that this is a valid argument, just check that $\big((p \to q) \wedge \neg q\big) \implies \neg p$, that is, that $\big((p \to q) \wedge \neg q\big) \to \neg p$ is a tautology. As an example, the following argument has the form of *modus tollens* and is therefore a valid argument:

> If Feyenoord is a great team, then I'm the king of the Netherlands
> I am not the king of the Netherlands
> ∴ Feyenoord is not a great team

You might remember this argument from page 13. You should note carefully that the validity of this argument has nothing to do with whether or not Feyenoord can play football well. The argument forces you to accept the conclusion *only if* you accept the premises. You can logically believe that the conclusion is false, as long as you believe that at least one of the premises is false.[15]

Another named rule of deduction is the ***Law of Syllogism***, which has the form

$$\begin{array}{l} p \to q \\ \underline{q \to r \qquad\qquad} \\ \therefore\ p \to r \end{array}$$

For example:

> If you study hard, you do well in school
> If you do well in school, you get a good job
> ∴ If you study hard, you get a good job

There are many other rules. Here are a few that might prove useful. Some of them might look trivial, but don't underestimate the power of a simple rule when it is combined with other rules.

$$\begin{array}{l} p \vee q \\ \underline{\neg p \qquad\qquad} \\ \therefore\ q \end{array} \qquad \begin{array}{l} p \\ \underline{q \qquad\qquad} \\ \therefore\ p \wedge q \end{array} \qquad \begin{array}{l} \underline{p \wedge q} \\ \therefore\ p \end{array} \qquad \begin{array}{l} \underline{p} \\ \therefore\ p \vee q \end{array}$$

[15] Unless the conclusion is a tautology. If that's the case, then even when a premise is false the conclusion will still be true. You do always know that if the conclusion is false then at least one of the premises is false.

Logical deduction is related to logical equivalence. We defined $\mathcal{P}$ and $\mathcal{Q}$ to be logically equivalent if $\mathcal{P} \leftrightarrow \mathcal{Q}$ is a tautology. Since $\mathcal{P} \leftrightarrow \mathcal{Q}$ is equivalent to $(\mathcal{P} \rightarrow \mathcal{Q}) \wedge (\mathcal{Q} \rightarrow \mathcal{P})$, we see that $\mathcal{P} \equiv \mathcal{Q}$ if and only if both $\mathcal{Q} \implies \mathcal{P}$ and $\mathcal{P} \implies \mathcal{Q}$. Thus, we can show that two statements are logically equivalent if we can show that each of them can be logically deduced from the other. Also, we get a lot of rules about logical deduction for free—two rules of deduction for each logical equivalence we know. For example, since $\neg(p \wedge q) \equiv (\neg p \vee \neg q)$, we get that $\neg(p \wedge q) \implies (\neg p \vee \neg q)$. For example, if we know "It is not both sunny and warm", then we can logically deduce "Either it's not sunny or it's not warm." (And vice versa.)

2.5.2 Valid arguments and proofs

In general, arguments are more complicated that those we've considered so far. Here, for example, is an argument that has five premises:

$$\begin{array}{l} (p \wedge r) \rightarrow s \\ q \rightarrow p \\ t \rightarrow r \\ q \\ t \\ \hline \therefore\ s \end{array}$$

Is this argument valid? Of course, you could use a truth table to check whether the conjunction of the premises logically implies the conclusion. But with five propositional variables, the table would have 32 lines, and the size of the table grows quickly when more propositional variables are used. So, in general, truth tables are not practical when we have a large number of variables.

For a relatively small number of variables (say three or fewer) a truth table can be a rather efficient method to test validity of an argument. In one of the pencasts of this course I show how you can use truth tables to test for validity as well as how you can use them to find counterexamples for invalid arguments: `youtu.be/lSZS3qbA88o`

Fortunately, there is another way to proceed, based on the fact that it is possible to chain several logical deductions together. That is, if $\mathcal{P} \implies \mathcal{Q}$ and $\mathcal{Q} \implies \mathcal{R}$, it follows that $\mathcal{P} \implies \mathcal{R}$. This means we can demonstrate the validity of an argument by deducing the conclusion from the premises in a sequence of steps. These steps can be presented in the form of a proof:

Definition 2.11. A ***formal proof*** that an argument is valid consists of a sequence of propositions such that the last proposition in the sequence is the conclusion of the argument,

and every proposition in the sequence is either a premise of the argument or follows by logical deduction from propositions that precede it in the list.

The existence of such a proof shows that the conclusion follows logically from the premises, and therefore that the argument is valid. Here is a formal proof that the argument given above is valid. The propositions in the proof are numbered, and each proposition has a justification.

Proof.

1.	$q \to p$	premise
2.	q	premise
3.	p	from 1 and 2 (*modus ponens*)
4.	$t \to r$	premise
5.	t	premise
6.	r	from 4 and 5 (*modus ponens*)
7.	$p \wedge r$	from 3 and 6
8.	$(p \wedge r) \to s$	premise
9.	s	from 7 and 8 (*modus ponens*)

□

Once a formal proof has been constructed, it is convincing. Unfortunately, it's not necessarily easy to come up with the proof. Usually, the best method is a combination of working forward ("Here's what I know, what can I deduce from that?") and working backwards ("Here's what I need to prove, what other things would imply that?"). For this proof, I might have thought: I want to prove s. I know that $p \wedge r$ implies s, so if I can prove $p \wedge r$, I'm okay. But to prove $p \wedge r$, it'll be enough to prove p and r separately....

Of course, not every argument is valid, so the question also arises, how can we show that an argument is invalid? Let's assume that the argument has been put into general form, with all the specific propositions replaced by propositional variables. The argument is valid if in all cases where all the premises are true, the conclusion is also true. The argument is invalid if there is even one case where all the premises are true and the conclusion is false. We can prove that an argument is invalid by finding an assignment of truth values to the propositional variables which makes all the premises true but makes the conclusion false. We call such an assignment a *counterexample* . To disprove the validity of an argument you should always provide a counterexample. This holds in propositional logic, predicate logic, and any other type of argument you may be asked to disprove.

For example, consider an argument of the form:

$$\begin{array}{l} p \to q \\ q \to (p \wedge r) \\ \underline{r} \\ \therefore\ p \end{array}$$

In the case where p is false, q is false, and r is true, the three premises of this argument are all true, but the conclusion is false. This counterexample shows that the argument is invalid.

To apply all this to arguments stated in English, we have to introduce propositional variables to represent all the propositions in the argument. For example, consider:

> John will be at the party if Mary is there and Bill is not there. Mary will be at the party if it's on Friday or Saturday. If Bill is at the party, Tom will be there. Tom won't be at the party if it's on Friday. The party is on Friday. Therefore, John will be at the party.

Let j stand for "John will be at the party", m for "Mary will be there", b for "Bill will be there", t for "Tom will be there", f for "The party is on Friday", and s for "The party is on Saturday". Then this argument has the form

$$\begin{array}{l} (m \wedge \neg b) \to j \\ (f \vee s) \to m \\ b \to t \\ f \to \neg t \\ \underline{f} \\ \therefore\ j \end{array}$$

This is a valid argument, as the following proof shows:

Proof.

1.	$f \to \neg t$	premise
2.	f	premise
3.	$\neg t$	from 1 and 2 (*modus ponens*)
4.	$b \to t$	premise
5.	$\neg b$	from 4 and 3 (*modus tollens*)
6.	$f \vee s$	from 2
7.	$(f \vee s) \to m$	premise
8.	m	from 6 and 7 (*modus ponens*)
9.	$m \wedge \neg b$	from 8 and 5
10.	$(m \wedge \neg b) \to j$	premise
11.	j	from 10 and 9 (*modus ponens*)

□

You may have noticed that we start our proofs with the word 'proof' and end it with a little square. This is done to illustrate clearly where our proof starts and ends. Historically different symbols and expressions have been used to indicate that a proof is done. You may have heard of the abbreviation Q.E.D. for instance for 'Quod Erat Demonstrandum', which translates to: 'what was to be shown'. Even in ancient Greece a Greek version of Q.E.D. was used by Greek mathematicians like Euclid. You are free to choose between Q.E.D. and the open square, so long as you remember that no proof is complete if it does not have either one of them.

2.5.3 Proofs in predicate logic

So far in this section, we have been working mostly with propositional logic. But the definitions of valid argument and logical deduction apply to predicate logic as well.

One of the most basic rules of deduction in predicate logic says that $(\forall x P(x)) \implies P(a)$ for any entity a in the domain of discourse of the predicate P. That is, if a predicate is true of all entities, then it is true of any given particular entity. This rule can be combined with rules of deduction for propositional logic to give the following valid arguments:

$$\begin{array}{l} \forall x(P(x) \to Q(x)) \\ P(a) \\ \hline \therefore\ Q(a) \end{array} \qquad \begin{array}{l} \forall x(P(x) \to Q(x)) \\ \neg Q(a) \\ \hline \therefore\ \neg P(a) \end{array}$$

These valid arguments go by the names of *modus ponens* and *modus tollens* for predicate logic. Note that from the premise $\forall x(P(x) \to Q(x))$ we can deduce $P(a) \to Q(a)$. From this and from the premise that $P(a)$, we can deduce $Q(a)$ by *modus ponens*. So the first argument above is valid. The second argument is similar, using *modus tollens*.

The most famous logical deduction of them all is an application of *modus ponens* for predicate logic:

$$\begin{array}{l} \text{All humans are mortal} \\ \text{Socrates is human} \\ \hline \therefore\ \text{Socrates is mortal} \end{array}$$

This has the form of *modus ponens* with $P(x)$ standing for "x is human", $Q(x)$ standing for "x is mortal", and a standing for the noted entity, Socrates.

To disprove validity of arguments in predicate logic, you again need to provide a counterexample. These are most easily given in the form of a mathematical structure. Consider for instance the following argument:

$$\begin{array}{l}\exists x P(x) \\ \forall x(P(x) \to Q(x)) \\ \hline \therefore\ \forall x Q(x)\end{array}$$

This argument is not valid and we can prove that using the following structure $\mathcal{A}$.

- $D = \{a, b\}$
- $P^{\mathcal{A}} = \{a\}$
- $Q^{\mathcal{A}} = \{a\}$

As you can see, the first premise is true. There is an x such that $P(x)$ holds, namely $x = a$. The second premis is also true, as for all x for which $P(x)$ holds (so only $x = a$), $Q(x)$ also holds (and indeed $Q(a)$) holds. However the conclusion is false, as $Q(b)$ does not hold, so the $Q(x)$ does not hold for all x.

There is a lot more to say about logical deduction and proof in predicate logic, and we'll spend the whole of the next chapter on the subject.

Exercises

1. Verify the validity of *modus tollens* and the Law of Syllogism.

2. Each of the following is a valid rule of deduction. For each one, give an example of a valid argument in English that uses that rule.

$$\begin{array}{l}p \vee q \\ \neg p \\ \hline \therefore\ q\end{array} \qquad \begin{array}{l}p \\ q \\ \hline \therefore\ p \wedge q\end{array} \qquad \begin{array}{l}p \wedge q \\ \hline \therefore\ p\end{array} \qquad \begin{array}{l}p \\ \hline \therefore\ p \vee q\end{array}$$

3. There are two notorious invalid arguments that look deceptively like *modus ponens* and *modus tollens*:

$$\begin{array}{l}p \to q \\ q \\ \hline \therefore\ p\end{array} \qquad \begin{array}{l}p \to q \\ \neg p \\ \hline \therefore\ \neg q\end{array}$$

Show that each of these arguments is invalid. Give an English example that uses each of these arguments.

4. Decide whether each of the following arguments is valid. If it is valid, give a formal proof. If it is invalid, show that it is invalid by finding an appropriate assignment of truth values to propositional variables.

a) $\begin{array}{l}p \to q \\ q \to s \\ s \\ \hline \therefore\ p\end{array}$ **b)** $\begin{array}{l}p \wedge q \\ q \to (r \vee s) \\ \neg r \\ \hline \therefore\ s\end{array}$ **c)** $\begin{array}{l}p \vee q \\ q \to (r \wedge s) \\ \neg p \\ \hline \therefore\ s\end{array}$

d) $(\neg p) \to t$
$q \to s$
$r \to q$
$\neg(q \vee t)$
$\therefore\ p$

e) p
$s \to r$
$q \vee r$
$q \to \neg p$
$\therefore\ \neg s$

f) $q \to t$
$p \to (t \to s)$
p
$\therefore\ q \to s$

5. For each of the following English arguments, express the argument in terms of propositional logic and determine whether the argument is valid or invalid.

a) If it is Sunday, it rains or snows. Today, it is Sunday and it's not raining. Therefore, it must be snowing.

b) If there is herring on the pizza, Jack won't eat it. If Jack doesn't eat pizza, he gets angry. Jack is angry. Therefore, there was herring on the pizza.

c) At 8:00, Jane studies in the library or works at home. It's 8:00 and Jane is not studying in the library. So she must be working at home.

Chapter 3

Proof

Mathematics is unique in that it claims a certainty that is beyond all possible doubt or argument. A mathematical proof shows how some result follows by logic alone from a given set of assumptions, and once the result has been proven, it is as solid as the foundations of logic themselves. Of course, mathematics achieves this certainty by restricting itself to an artificial, mathematical world, and its application to the real world does not carry the same degree of certainty.

Within the world of mathematics, consequences follow from assumptions with the force of logic, and a proof is just a way of pointing out logical consequences. Of course, the fact that mathematical results follow logically does not mean that they are obvious in any normal sense. Proofs are convincing once they are discovered, but finding them is often very difficult. They are written in a language and style that can seem obscure to the uninitiated. Often, a proof builds on a long series of definitions and previous results, and while each step along the way might be 'obvious' the end result can be surprising and powerful. This is what makes the search for proofs worthwhile.

In this chapter, we'll look at some approaches and techniques that can be used for proving mathematical results, including two important proof techniques known as proof by contradiction and mathematical induction. Along the way, we'll encounter a few new definitions and notations. Hopefully, you will be left with a higher level of confidence for exploring the mathematical world on your own.

3.1 A Little Historical Background

The mathematical world and the real world weren't always quite so separate. Until around the middle of the nineteenth century, the statements of mathematics were regarded as statements about the world. A proof was simply a convincing argument, rather than a chain forged of absolute logic. It was something closer to the original mean-

ing of the word 'proof', as a test or trial: you might have heard of the proverb, "The proof of the pudding is in the eating." So, to prove something was to test its truth by putting it to the trial of logical argument.

A commonly made mistake centres around the difference between 'proof' (a noun) and 'to prove' (a verb). You can prove a certain claim using a proof. But grammatically speaking you cannot proof a certain claim using a prove. Historically, in the course *Reasoning & Logic* this is one of the most common spelling/grammar mistakes on exams. By including it in this book, we hope that you will now know better.

The first rumble of trouble came in the form of ***non-Euclidean geometry***. For two thousand years, the geometry of the Greek mathematician Euclid had been accepted, simply, as the geometry of the world. In the middle of the nineteenth century, it was discovered that there are other systems of geometry, which are at least as valid and self-consistent as Euclid's system. Mathematicians can work in any of these systems, but they cannot all claim to be working in the real world.

Near the end of the nineteenth century came another shock, in the form of cracks in the very foundation of mathematics. At that time, mathematician Gottlob Frege was finishing a book on set theory that represented his life's work. In Frege's set theory, a set could be defined by any property. You could have, for example, the set consisting of all sets that contain three objects. As he was finishing his book, Frege received a letter from a young mathematician named Bertrand Russell which described what became known as ***Russell's Paradox***. Russell pointed out that the set of all sets—that is, the set that contains every entity that satisfies the property of being a set—cannot logically exist. We'll see Russell's reasoning in the following chapter. Frege could only include a postscript in his book stating that the basis of the work had been swept away.

A contemporary of Peirce (see page 40), Friedrich Ludwig Gottlob Frege (1848–1925) was a German philosopher, logician, and mathematician. Peirce and Frege were apparently mostly unaware of each other's work. Frege is understood by many to be the father of analytic philosophy, concentrating on the philosophy of language and mathematics. Like Peirce, his work was largely ignored during his lifetime, and came to be recognised by later mathematicans—including Russell.

Source: `en.wikipedia.org/wiki/Gottlob_Frege`.

Mathematicians responded to these problems by banishing appeals to facts about the real world from mathematical proof. Mathematics was to be its own world, built on its own secure foundation. The foundation would be a basic set of assumptions or 'axioms' from which everything else would follow by logic. It would only be necessary to show that the axioms themselves were logically consistent and complete, and the world of mathematics would be secure. Unfortunately, even this was not to be. In the 1930s, Kurt Gödel showed that there is no consistent, finite set of axioms that completely describes even the corner of the mathematical world known as arithmetic. Gödel showed that given any finite, consistent set of axioms, there are true statements about arithmetic that do not follow logically from those axioms. I will return to Gödel and his contemporaries in Chapter 5.

We are left with a mathematical world in which iron chains of logic still bind conclusions to assumptions. But the assumptions are no longer rooted in the real world. Nor is there any finite core of axioms to which the rest of the mathematical world can be chained. In this world, axioms are set up as signposts in a void, and then structures of logic are built around them. For example, in the next chapter, instead of talking about *the* set theory that describes the real world, we have *a* set theory, based on a given set of axioms. That set theory is necessarily incomplete, and it might differ from other set theories which are based on other sets of axioms.

3.2 Mathematical Proof

Understandably, mathematicians are very picky about getting their proofs right. It's how they construct their world. Students sometimes object that mathematicians are too picky about proving things that are 'obvious'. But the fact that something is obvious in the real world counts for very little in the constructed world of mathematics. Too many obvious

things have turned out to be dead wrong. (For that matter, even things in the real world that seem 'obviously' true are not necessarily true at all.)

For example, consider the quantity $f(n) = n^2 + n + 41$. When $n = 0$, $f(n) = 41$ which is prime; when $n = 1$, $f(n) = 43$ which is prime; when $n = 2$, $f(n) = 47$, which is prime. By the time you had calculated $f(3), f(4), \ldots, f(10)$ and found that they were all prime, you might conclude that it is 'obvious' that $f(n)$ is prime for all $n \geq 0$. But this is not in fact the case! (See exercises.)

As we saw in Section 2.5, a formal proof consists of a sequence of statements where each statement is either an assumption or follows by a rule of logic from previous statements. The examples in that section all worked with unspecified generic propositions (p, q, etc). Let us now look at how one might use the same techniques to prove a specific proposition about the mathematical world. We will prove that for all integers n, if n is even then n^2 is even. (Definition: an integer n is *even* iff $n = 2k$ for some integer k. For example, 2 is even since $2 = 2 \cdot 1$; 66 is even since $66 = 2 \cdot 33$; 0 is even since $0 = 2 \cdot 0$.)

Proof. This is a proposition of the form $\forall n(P(n) \to Q(n))$ where $P(n)$ is "n is even" and $Q(n)$ is "n^2 is even." We need to show that $P(n) \to Q(n)$ is true for all values of n. Or alternatively we can phrase it as: $\forall n(E(n) \to E(n^2))$ where $E(x)$ is 'x is even'.

In the language of Section 2.5, we need to show that for any n, $E(n)$ logically implies $E(n^2)$; or, equivalently, that $E(n^2)$ can be logically deduced from $E(n)$; or, equivalently, that

$$\frac{n \text{ is even}}{\therefore\ n^2 \text{ is even}}$$

is a valid argument. Here is a formal proof that

$$\frac{n \text{ is even}}{\therefore\ n^2 \text{ is even}}$$

is in fact a valid argument for any value of n:

Let n be an arbitrary integer.

1.	n is even	premise
2.	if n is even, then $n = 2k$ for some integer k	definition of even
3.	$n = 2k$ for some integer k	from 1, 2 (*modus ponens*)
4.	if $n = 2k$ for some integer k, then $n^2 = 4k^2$ for that integer k	basic algebra
5.	$n^2 = 4k^2$ for some integer k	from 3, 4 (*modus ponens*)
6.	if $n^2 = 4k^2$ for some integer k, then $n^2 = 2(2k^2)$ for that k	basic algebra
7.	$n^2 = 2(2k^2)$ for some integer k	from 5, 6 (*modus ponens*)
8.	if $n^2 = 2(2k^2)$ for some integer k, then $n^2 = 2k'$ for some integer k'	basic fact about integers
9.	$n^2 = 2k'$ for some integer k'	from 7, 8 (*modus ponens*)
10.	if $n^2 = 2k'$ for some integer k', then n^2 is even	definition of even
11.	n^2 is even	from 9, 10 (*modus ponens*)

(The "basic fact about integers" referred to above is that the product of integers is again an integer.) Since n could be replaced by any integer throughout this argument, we have proved the statement "if n is even then n^2 is even" is true for all integers n. (You might worry that the argument is only valid for even n; see the disclaimer about Feyenoord's football ability on page 51, or remind yourself that $P(n) \to Q(n)$ is automatically true if $P(n)$ is false.) □

Mathematical proofs are rarely presented with this degree of detail and formality. A slightly less formal proof of our proposition might leave out the explicit implications and instances of *modus ponens* and appear as follows:

Proof. Let n be an arbitrary integer.

1.	n is even	premise
2.	$n = 2k$ for some integer k	definition of even
3.	$n^2 = 4k^2$ for that integer k	basic algebra
4.	$n^2 = 2(2k^2)$ for that k	basic algebra
5.	$n^2 = 2k'$ for some integer k'	substituting $k' = 2k^2$
6.	n^2 is even	definition of even

Since n was an arbitrary integer, the statement is true for all integers. □

A more typical proof would take the argument above and present it in prose rather than list form:

Proof. Let n be an arbitrary integer and assume n is even. Then $n = 2k$ for some integer k by the definition of even, and $n^2 = 4k^2 = 2(2k^2)$. Since the product of integers is an integer, we have $n^2 = 2k'$ for some integer k'. Therefore n^2 is even. Since n was an arbitrary integer, the statement is true for all integers. □

Typically, in a 'formal' proof, it is this kind of (relatively) informal discussion that is given, with enough details to convince the reader that a complete, formal proof could be constructed. Of course, how many details the reader can be expected to fill in depends on the reader, and reading proofs is a skill that must be developed and practiced.

In the course *Reasoning & Logic* you are learning to write proper formal proofs, and as a part of that we also need to evaluate your performance. To this end we ask you to write proofs similar to the second example given above (the list form rather than the prose form) as this shows more clearly that you are aware of the formalisms required in a proof.

Writing a proof is even more difficult. Every proof involves a creative act of discovery, in which a chain of logic that leads from assumptions to conclusion is discovered. It also involves a creative act of expression, in which that logic is presented in a clear and convincing way. There is no algorithm for producing correct, coherent proofs. There are, however, some general guidelines for discovering and writing proofs. Let's look at some of these next.

3.2.1 How to write a proof

One of the most important pieces of advice to keep in mind is: "Use the definition". In the world of mathematics, terms mean exactly what they are defined to mean and nothing more. Definitions allow very complex ideas to be summarized as single terms. When you are trying to prove things about those terms, you generally need to 'unwind' the definitions. In our example above, we used the definition of even to write $n = 2k$, and then we worked with that equation. When you are trying to prove something about equivalence relations in Chapter 4, you can be pretty sure that you will need to use the fact that equivalence relations, by definition, are symmetric, reflexive, and transitive. (And, of course, you'll need to know how the term 'relation' is defined in the first place! We mean something quite different than the idea that 'relations' are something like your aunt and uncle.)

More advice along the same line is to check whether you are using the assumptions of the theorem. An assumption that is made in a theorem is called an ***hypothesis***. The hypotheses of the theorem state conditions whose truth will guarantee the conclusion of the theorem. To prove the theorem means to assume that the hypotheses are true, and to show, under that assumption, that the conclusion must be true. It's likely (though

not guaranteed) that you will need to use the hypotheses explicitly at some point in the proof, as we did in our example above.[1] Also, you should keep in mind that any result that has already been proved is available to be used in your proof.

A proof is a logical argument, based on the rules of logic. Since there are really not all that many basic rules of logic, the same patterns keep showing up over and over. Let's look at some of the patterns.

The most common pattern arises in the attempt to prove that something is true 'for all' or 'for every' or 'for any' entity in a given category. In terms of logic, the statement you are trying to prove is of the form $\forall x\, P(x)$. In this case, the most likely way to begin the proof is by saying something like, "Let x be an arbitrary entity in the domain of discourse. We want to show that $P(x)$." We call this a **proof by generalisation** In the rest of the proof, x refers to some unspecified but definite entity in the domain of discourse. Since x is arbitrary, proving $P(x)$ amounts to proving $\forall x\, P(x)$. You only have to be careful that you don't use any facts about x beyond what you have assumed. For example, in our proof above, we cannot make any assumptions about the integer n except that it is even; if we for instance also assume $x = 6$ or that x is divisible by 3, then the proof would have been incorrect, or at least incomplete.

Sometimes, you have to prove that an entity exists that satisfies certain stated properties. Such a proof is called an ***existence proof***. In this case, you are attempting to prove a statement of the form $\exists x\, P(x)$. The way to do this is to find an example, that is, to find a specific entity a for which $P(a)$ is true. One way to prove the statement "There is an even prime number" is to find a specific number that satisfies this description. The same statement could also be expressed "Not every prime number is odd." This statement has the form $\neg(\forall x\, P(x))$, which is equivalent to the statement $\exists x\, (\neg P(x))$. An example that proves the statement $\exists x\, (\neg P(x))$ also proves $\neg(\forall x\, P(x))$. Such an example is called a ***counterexample*** to the statement $\forall x\, P(x)$: A counterexample proves that the statement $\forall x\, P(x)$ is false. The number 2 is a counterexample to the statement "All prime numbers are odd." In fact, 2 is the only counterexample to this statement; many statements have multiple counterexamples.

Note that we have now discussed how to prove and disprove universally quantified statements, and how to prove existentially quantified statements. How do you *disprove* $\exists x\, P(x)$? Recall that $\neg\exists x\, P(x)$ is logically equivalent to $\forall x\, (\neg P(x))$, so to disprove $\exists x\, P(x)$ you need to prove $\forall x\, (\neg P(x))$.

Many statements, like that in our example above, have the logical form of an implication, $p \to q$. (More accurately, they are of the form "$\forall x\, (P(x) \to Q(x))$", but as discussed above the strategy for proving such a statement is to prove $P(x) \to Q(x)$ for an arbitrary element x of the domain of discourse.) The statement might be "For all natural numbers n, if n is even then n^2 is even," or "For all strings x, if x is in the language

[1] Of course, if you set out to discover new theorems on your own, you aren't given the hypotheses and conclusion in advance, which makes things quite a bit harder—and more interesting.

L then x is generated by the grammar G,"[2] or "For all elements s, if $s \in A$ then $s \in B$." Sometimes the implication is implicit rather than explicit: for example, "The sum of two rationals is rational" is really short for "For any numbers x and y, if x and y are rational then $x + y$ is rational." A proof of such a statement often begins something like this: "Assume that p. We want to show that q." In the rest of the proof, p is treated as an assumption that is known to be true. As discussed above, the logical reasoning behind this is that you are essentially proving that

$$\frac{p}{\therefore\ q}$$

is a valid argument. Another way of thinking about it is to remember that $p \to q$ is automatically true in the case where p is false, so there is no need to handle that case explicitly. In the remaining case, when p is true, we can show that $p \to q$ is true by showing that the truth of q follows from the truth of p. So remember than proving an implication you should assume the antecedent and prove the consequent (you can refresh your memory of what those words mean on page 12).

A statement of the form $p \wedge q$ can be proven by proving p and q separately. A statement of the form $p \vee q$ can be proved by proving the logically equivalent statement $(\neg p) \to q$: to prove $p \vee q$, you can assume that p is false and prove, under that assumption, that q is true. For example, the statement "Every integer is even or odd" is equivalent to the statement "Every integer that is not even is odd".

Since $p \leftrightarrow q$ is equivalent to $(p \to q) \wedge (q \to p)$, a statement of the form $p \leftrightarrow q$ is often proved by giving two proofs, one of $p \to q$ and one of $q \to p$. In English, $p \leftrightarrow q$ can be stated in several forms such as "p if and only if q", "if p then q and conversely," and "p is necessary and sufficient for q". The phrase 'if and only if' is so common in mathematics that it is often abbreviated ***iff***.

You should also keep in mind that you can prove $p \to q$ by displaying a chain of valid implications $p \to r \to s \to \cdots \to q$. Similarly, $p \leftrightarrow q$ can be proved with a chain of valid biconditionals $p \leftrightarrow r \leftrightarrow s \leftrightarrow \cdots \leftrightarrow q$.

3.2.2 Some terminology

Before we look at some sample proofs, here is some terminology that we will use throughout our sample proofs and the rest of the course of *Reasoning & Logic*.

- The ***natural numbers*** (denoted $\mathbb{N}$) are the numbers $0, 1, 2, \ldots$. Note that the sum and product of natural numbers are natural numbers.
- The ***integers*** (denoted $\mathbb{Z}$) are the numbers $0, -1, 1, -2, 2, -3, 3, \ldots$. Note that the sum, product, and difference of integers are integers.

[2] You will learn about this in the course *Automata, Computability, and Complexity* in your second year.

- The ***rational numbers*** (denoted $\mathbb{Q}$) are all numbers that can be written in the form $\frac{m}{n}$ where m and n are integers and $n \neq 0$. So $\frac{1}{3}$ and $\frac{-65}{7}$ are rationals; so, less obviously, are 6 and $\frac{\sqrt{27}}{\sqrt{12}}$ since $6 = \frac{6}{1}$ (or, for that matter, $6 = \frac{-12}{-2}$), and $\frac{\sqrt{27}}{\sqrt{12}} = \sqrt{\frac{27}{12}} = \sqrt{\frac{9}{4}} = \frac{3}{2}$. Note the restriction that the number in the denominator cannot be 0: $\frac{3}{0}$ is not a number at all, rational or otherwise; it is an undefined quantity. Note also that the sum, product, difference, and quotient of rational numbers are rational numbers (provided you don't attempt to divide by 0).

- The ***real numbers*** (denoted $\mathbb{R}$) are numbers that can be written in decimal form, possibly with an infinite number of digits after the decimal point. Note that the sum, product, difference, and quotient of real numbers are real numbers (provided you don't attempt to divide by 0).

- The ***irrational numbers*** are real numbers that are not rational, i.e., that cannot be written as a ratio of integers. Such numbers include $\sqrt{3}$ (which we will prove is not rational) and π (if anyone ever told you that $\pi = \frac{22}{7}$, remember that $\frac{22}{7}$ is only an *approximation* of the value of π). Later you will learn that we can describe this set of irrational numbers as $\mathbb{R} - \mathbb{Q}$, that is: it is all the numbers that are in $\mathbb{R}$ but are not in $\mathbb{Q}$.

- An integer n is ***divisible by*** m iff $n = mk$ for some integer k. This can also be expressed by saying that m ***e***venly divides n, which has the mathematical notation $m \mid n$. So for example, $2 \mid 8$, but $8 \nmid 2$. $2 \mid n$ iff $n = 2k$ for some integer k; n is divisible by 3 iff $n = 3k$ for some integer k, and so on. Note that if $2 \nmid n$ (i.e., n is *not* divisible by 2), then n must be 1 more than a multiple of 2 so $n = 2k + 1$ for some integer k. Similarly, if n is not divisible by 3 then n must be 1 or 2 more than a multiple of 3, so $n = 3k + 1$ or $n = 3k + 2$ for some integer k.

- An integer is ***even*** iff it is divisible by 2 and ***odd*** iff it is not.

- An integer $n > 1$ is ***prime*** if it is divisible by exactly two positive integers, namely 1 and itself. Note that a number must be greater than 1 to even have a chance of being termed 'prime'. In particular, neither 0 nor 1 is prime.

3.2.3 Examples

Let's look now at another example of a proof. We set out to prove that the sum of any two rational numbers is rational.

Proof. We start by assuming that x and y are arbitrary rational numbers. Here's a formal proof that the inference rule

$$\frac{\begin{array}{l}x \text{ is rational}\\ y \text{ is rational}\end{array}}{\therefore\ x+y \text{ is rational}}$$

is a valid rule of inference:

1. x is rational	premise
2. if x is rational, then $x = \frac{a}{b}$ for some integers a and $b \neq 0$	definition of rationals
3. $x = \frac{a}{b}$ for some integers a and $b \neq 0$	from 1,2 (*modus ponens*)
4. y is rational	premise
5. if y is rational, then $y = \frac{c}{d}$ for some integers c and $d \neq 0$	definition of rational
6. $y = \frac{c}{d}$ for some c and $d \neq 0$	from 4,5 (*modus ponens*)
7. $x = \frac{a}{b}$ for some a and $b \neq 0$ and $y = \frac{c}{d}$ for some c and $d \neq 0$	from 3,6
8. if $x = \frac{a}{b}$ for some a and $b \neq 0$ and $y = \frac{c}{d}$ for c and $d \neq 0$ then $x + y = \frac{ad+bc}{bd}$ where a, b, c, d are integers and $b, d \neq 0$	basic algebra
9. $x + y = \frac{ad+bc}{bd}$ for some a, b, c, d where $b, d \neq 0$	from 7,8 (*modus ponens*)
10. if $x + y = \frac{ad+bc}{bd}$ for some a, b, c, d where $b, d \neq 0$ then $x + y = \frac{m}{n}$ where m, n are integers and $n \neq 0$	properties of integers
11. $x + y = \frac{m}{n}$ where m and n are integers and $n \neq 0$	from 9,10 (*modus ponens*)
12. if $x + y = \frac{m}{n}$ where m and n are integers and $n \neq 0$ then $x + y$ is rational	definition of rational
13. $x + y$ is rational	from 11,12 (*modus ponens*)

So the rule of inference given above is valid. Since x and y are arbitrary rationals, we have proved that the rule is valid for all rationals, and hence the sum of any two rationals is rational. □

Again, a more informal presentation that we expect from you during the course would look like:

Proof. Proof by generalisation:

- Let x and y be arbitrary rational numbers.
- By the definition of rational, there are integers $a, b \neq 0, c, d \neq 0$ such that $x = \frac{a}{b}$ and $y = \frac{c}{d}$.
- Then $x + y = \frac{ad+bc}{bd}$;
- We know $ad + bc$ and bd are integers since the sum and product of integers are integers, and we also know $bd \neq 0$ since neither b nor d is 0.
- So we have written $x + y$ as the ratio of two integers, the denominator being non-zero.
- Therefore, by the definition of rational numbers, $x + y$ is rational.
- Since x and y were arbitrary rationals, the sum of any two rationals is rational.

□

And one more example: we will prove that any 4-digit number $d_1d_2d_3d_4$ is divisible by 3 iff the sum of the four digits is divisible by 3.

Proof. This statement is of the form $p \leftrightarrow q$; recall that $p \leftrightarrow q$ is logically equivalent to $(p \rightarrow q) \wedge (q \rightarrow p)$. So we need to prove for any 4-digit number $d_1d_2d_3d_4$ that (1) if $d_1d_2d_3d_4$ is divisible by 3 then $d_1 + d_2 + d_3 + d_4$ is divisible by 3, and (2) if $d_1 + d_2 + d_3 + d_4$ is divisible by 3 then $d_1d_2d_3d_4$ is divisible by 3. So let $d_1d_2d_3d_4$ be an arbitrary 4-digit number.

(1) Assume $d_1d_2d_3d_4$ is divisible by 3, i.e., $d_1d_2d_3d_4 = 3k$ for some integer k. The number $d_1d_2d_3d_4$ is actually $d_1 \times 1000 + d_2 \times 100 + d_3 \times 10 + d_4$, so we have the equation

$$d_1 \times 1000 + d_2 \times 100 + d_3 \times 10 + d_4 = 3k.$$

Since $1000 = 999 + 1$, $100 = 99 + 1$, and $10 = 9 + 1$, this equation can be rewritten

$$999d_1 + d_1 + 99d_2 + d_2 + 9d_3 + d_3 + d_4 = 3k.$$

Rearranging gives

$$\begin{aligned} d_1 + d_2 + d_3 + d_4 &= 3k - 999d_1 - 99d_2 - 9d_3 \\ &= 3k - 3(333d_1) - 3(33d_2) - 3(3d_3). \end{aligned}$$

We can now factor a 3 from the right side to get

$$d_1 + d_2 + d_3 + d_4 = 3(k - 333d_1 - 33d_2 - d_3).$$

Since $(k - 333d_1 - 33d_2 - d_3)$ is an integer, we have shown that $d_1 + d_2 + d_3 + d_4$ is divisible by 3.

(2) Assume $d_1 + d_2 + d_3 + d_4$ is divisible by 3. Consider the number $d_1d_2d_3d_4$. As remarked above,

$$d_1d_2d_3d_4 = d_1 \times 1000 + d_2 \times 100 + d_3 \times 10 + d_4$$

so

$$\begin{aligned} d_1d_2d_3d_4 &= 999d_1 + d_1 + 99d_2 + d_2 + 9d_3 + d_3 + d_4 \\ &= 999d_1 + 99d_2 + 9d_3 + (d_1 + d_2 + d_3 + d_4). \end{aligned}$$

We assumed that $d_1 + d_2 + d_3 + d_4 = 3k$ for some integer k, so we can substitute into the last equation to get

$$d_1d_2d_3d_4 = 999d_1 + 99d_2 + 9d_3 + 3k = 3(333d_1 + 33d_2 + 3d_3 + k).$$

Since the quantity in parentheses is an integer, we have proved that $d_1d_2d_3d_4$ is divisible by 3.

In (1) and (2) above, the number $d_1d_2d_3d_4$ was an arbitrary 4-digit integer, so we have proved that for all 4-digit integers, $d_1d_2d_3d_4$ is divisible by 3 iff the sum of the four digits is divisible by 3. □

Now suppose we wanted to prove the statement "For all integers n, n^2 is even if and only if n is even." We have already proved half of this statement ("For all integers n, if n is even then n^2 is even"), so all we need to do is prove the statement "For all integers n, if n^2 is even then n is even" and we'll be done. Unfortunately, this is not as straightforward as it seems: suppose we started in our standard manner and let n be an arbitrary integer and assumed that $n^2 = 2k$ for some integer k. Then we'd be stuck! Taking the square root of both sides would give us n on the left but would leave a $\sqrt{2k}$ on the right. This quantity is not of the form $2k'$ for any integer k'; multiplying it by $\frac{\sqrt{2}}{\sqrt{2}}$ would give $2\frac{\sqrt{k}}{\sqrt{2}}$ but there is no way for us to prove that $\frac{\sqrt{k}}{\sqrt{2}}$ is an integer. So we've hit a dead end. What do we do now?

The answer is that we need a different proof technique. The proofs we have written so far are what are called ***direct proofs***: to prove $p \to q$ you assume p is true and prove that the truth of q follows. Sometimes, when a direct proof of $p \to q$ fails, an ***indirect proof*** will work. Recall that the *contrapositive* of the implication $p \to q$ is the implication $\neg q \to \neg p$, and that this proposition is logically equivalent to $p \to q$. An indirect proof of $p \to q$, then, is a direct proof of the contrapositive $\neg q \to \neg p$. In our current example, instead of proving "if n^2 is even then n is even" directly, we can prove its contrapositive "if n is not even (i.e., n is odd) then n^2 is not even (i.e., n^2 is odd.)" We call this method a **proof by contrapositive**. The proof of this contrapositive is a routine direct argument which we leave to the exercises.

Alternatively we sometimes need a **proof by division into cases**. Consider for instance that we want to prove that $3 \mid (n^3 + 3n^2 + 2n)$ for all integers n. What we can do is split our proof into three different case based on the divisibility by 3. Recall from the definition of divisibility that every number can be written as either $n = 3k$, $n = 3k + 1$, or $n = 3k + 2$. In a proof by division into cases, we prove that the claim holds for all of these cases and thereby prove the claim holds for all numbers.

I have also created a pencast about several of the different proof techniques outlined in this chapter. This includes one in which we prove that $3 \mid (n^3 + 3n^2 + 2n)$ using a proof by division into cases, found here: `youtu.be/4OHyyGY_WpI`

Exercises

1. Find a natural number n for which $n^2 + n + 41$ is not prime.

2. Show that the propositions $p \vee q$ and $(\neg p) \to q$ are logically equivalent.

3. Show that the proposition $(p \vee q) \to r$ is equivalent to $(p \to r) \wedge (q \to r)$.

4. Determine whether each of the following statements is true. If it true, prove it. If it is false, give a counterexample.

- **a)** Every prime number is odd.
- **b)** Every prime number greater than 2 is odd.
- **c)** If x and y are integers with $x < y$, then there is an integer z such that $x < z < y$.
- **d)** If x and y are real numbers with $x < y$, then there is a real number z such that $x < z < y$.

†**5.** Suppose that r, s, and t are integers, such that r evenly divides s and s evenly divides t. Prove that r evenly divides t.

6. Prove that for all integers n, if n is odd then n^2 is odd.

7. Prove that an integer n is divisible by 3 iff n^2 is divisible by 3. (Hint: give an indirect proof of "if n^2 is divisible by 3 then n is divisible by 3.")

8. Prove or disprove each of the following statements. Remember that to disprove a statement we always expect a counterexample!

- **a)** The product of two even integers is even.
- **b)** The product of two integers is even only if both integers are even.
- **c)** The product of two rational numbers is rational.
- **d)** The product of two irrational numbers is irrational.
- **e)** For all integers n, if n is divisible by 4 then n^2 is divisible by 4.
- **f)** For all integers n, if n^2 is divisible by 4 then n is divisible by 4.

3.3 Proof by Contradiction

Suppose that we start with some set of assumptions and apply rules of logic to derive a sequence of statements that can be proved from those assumptions, and suppose that we derive a statement that we know to be false. When the laws of logic are applied to true statements, the statements that are derived will also be true. If we derive a false statement by applying rules of logic to a set of assumptions, then at least one of the assumptions must be false. This observation leads to a powerful proof technique, which is known as **proof by contradiction**.

Suppose that you want to prove some proposition, p. To apply proof by contradiction, assume that $\neg p$ is true, and apply the rules of logic to derive conclusions based on this assumption. If it is possible to derive a statement that is known to be false, it follows that the assumption, $\neg p$, must be false. (Of course, if the derivation is based on several assumptions, then you only know that at least *one* of the assumptions must be false.) The fact that $\neg p$ is false proves that p is true. Essentially, you are arguing that p must be true, because if it weren't, then some statement that is known to be false could be proved to be true. Generally, the false statement that is derived in a proof by contradiction is of the form $q \wedge \neg q$. This statement is a contradiction in the sense that it is false no matter what the value of q. Note that deriving the contradiction $q \wedge \neg q$ is the same as showing that the two statements, q and $\neg q$, both follow from the assumption that $\neg p$.

As a first example of proof by contradiction, consider the following theorem:

Theorem 3.1. *The number $\sqrt{3}$ is irrational.*

Proof. Proof by contradiction:

- Assume for the sake of contradiction that $\sqrt{3}$ is rational.
- Then $\sqrt{3}$ can be written as the ratio of two integers, $\sqrt{3} = \frac{m'}{n'}$ for some integers m' and n'.
- Furthermore, the fraction $\frac{m'}{n'}$ can be reduced to lowest terms by canceling all common factors of m' and n'. So $\sqrt{3} = \frac{m}{n}$ for some integers m and n which have no common factors.
- Squaring both sides of this equation gives $3 = \frac{m^2}{n^2}$ and re-arranging gives $3n^2 = m^2$.
- From this equation we see that m^2 is divisible by 3; you proved in the previous section (Exercise 6) that m^2 is divisible by 3 iff m is divisible by 3. Therefore m is divisible by 3 and we can write $m = 3k$ for some integer k.
- Substituting $m = 3k$ into the last equation above gives $3n^2 = (3k)^2$ or $3n^2 = 9k^2$, which in turn becomes $n^2 = 3k^2$. From this we see that n^2 is divisible by 3, and again we know that this implies that n is divisible by 3.

- But now we have (i) m and n have no common factors, and (ii) m and n have a common factor, namely 3. It is impossible for both these things to be true, yet our argument has been logically correct.
- Therefore our original assumption, namely that $\sqrt{3}$ is rational, must be incorrect.
- Therefore $\sqrt{3}$ must be irrational.

□

One of the oldest mathematical proofs, which goes all the way back to Euclid, is a proof by contradiction. Recall that a prime number is an integer n, greater than 1, such that the only positive integers that evenly divide n are 1 and n. We will show that there are infinitely many primes. Before we get to the theorem, we need a lemma. (A ***lemma*** is a theorem that is introduced only because it is needed in the proof of another theorem. Lemmas help to organize the proof of a major theorem into manageable chunks.)

Lemma 3.2. *If N is an integer and $N > 1$, then there is a prime number which evenly divides N.*

Proof. Let D be the smallest integer which is greater than 1 and which evenly divides N. (D exists since there is at least one number, namely N itself, which is greater than 1 and which evenly divides N. We use the fact that any non-empty subset of $\mathbb{N}$ has a smallest member.) I claim that D is prime, so that D is a prime number that evenly divides N.

Suppose that D is not prime. We show that this assumption leads to a contradiction. Since D is not prime, then, by definition, there is a number k between 2 and $D - 1$, inclusive, such that k evenly divides D. But since D evenly divides N, we also have that k evenly divides N (by exercise 5 in the previous section). That is, k is an integer greater than one which evenly divides N. But since k is less than D, this contradicts the fact that D is the *smallest* such number. This contradiction proves that D is a prime number. □

Theorem 3.3. *There are infinitely many prime numbers.*

Proof. Suppose that there are only finitely many prime numbers. We will show that this assumption leads to a contradiction.

Let p_1, p_2, ..., p_n be a complete list of all prime numbers (which exists under the assumption that there are only finitely many prime numbers). Consider the number N obtained by multiplying all the prime numbers together and adding one. That is,

$$N = (p_1 \cdot p_2 \cdot p_3 \cdots p_n) + 1.$$

Now, since N is larger than any of the prime numbers p_i, and since p_1, p_2, ..., p_n is a *complete* list of prime numbers, we know that N cannot be prime. By the lemma, there is a prime number p which evenly divides N. Now, p must be one of the numbers p_1, p_2, ..., p_n. But in fact, none of these numbers evenly divides N, since dividing N by any p_i leaves a remainder of 1. This contradiction proves that the assumption that there are only finitely many primes is false. □

This proof demonstrates the power of proof by contradiction. The fact that is proved here is not at all obvious, and yet it can be proved in just a few paragraphs.

It is easy to get a proof by contradiction wrong however. In one of the pencasts of this course I treat a commonly-made mistake when using proofs by contradiction: `youtu.be/OqKvBWxanok`.

Exercises

1. Suppose that $a_1, a_2, ..., a_{10}$ are real numbers, and suppose that $a_1 + a_2 + \cdots + a_{10} > 100$. Use a proof by contradiction to conclude that at least one of the numbers a_i must be greater than 10.

2. Prove that each of the following statements is true. In each case, use a proof by contradiction. Remember that the negation of $p \rightarrow q$ is $p \wedge \neg q$.

a) Let n be an integer. If n^2 is an even integer, then n is an even integer.

b) $\sqrt{2}$ is irrational.

c) If r is a rational number and x is an irrational number, then $r + x$ is an irrational number. (That is, the sum of a rational number and an irrational number is irrational.)

d) If r is a non-zero rational number and x is an irrational number, then rx is an irrational number.

e) If r and $r + x$ are both rational, then x is rational.

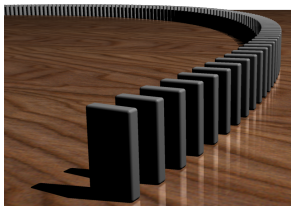
†**3.** The ***pigeonhole principle*** is the following obvious observation: If you have n pigeons in k pigeonholes and if $n > k$, then there is at least one pigeonhole that contains more than one pigeon. Even though this observation seems obvious, it's a good idea to prove it. Prove the pigeonhole principle using a proof by contradiction.

3.4 Mathematical Induction

The structure of the natural numbers—0, 1, 2, 3, and on to infinity—makes possible a powerful proof technique known as ***induction*** or ***mathematical induction***. Although the idea behind induction is simple, students often struggle with it. Take your time and study this material thoroughly! You will have opportunity to practice in the labs after we discuss induction in the lectures.

Let P be a one-place predicate whose domain of discourse includes the natural numbers. Suppose that we can prove that $P(0)$ is true. Suppose that we can also prove the statements $P(0) \rightarrow P(1)$, $P(1) \rightarrow P(2)$, $P(2) \rightarrow P(3)$, and so on. The principle of mathematical induction is the observation that we can then conclude that $P(n)$ is true for *all* natural numbers n. This should be clear. It's like a line of dominos, lined up and ready

to fall, one after the next.[3] Since $P(0)$ and $P(0) \to P(1)$ are true, we can apply the rule of *modus ponens* to conclude that $P(1)$ is true. Then, since $P(1)$ and $P(1) \to P(2)$ are true, we can conclude by *modus ponens* that $P(2)$ is true. From $P(2)$ and $P(2) \to P(3)$, we conclude that $P(3)$ is true. For any given n in the set $\mathbb{N}$, we can continue this chain of deduction for n steps to prove that $P(n)$ is true.



When applying induction, we don't actually prove each of the implications $P(0) \to P(1)$, $P(1) \to P(2)$, and so on, individually. That would require an infinite amount of work. The whole point of induction is to avoid any infinitely long process. Instead, we prove $\forall k\,(P(k) \to P(k+1))$ (where the domain of discourse for the predicate P is $\mathbb{N}$. The statement $\forall k\,(P(k) \to P(k+1))$ summarizes all the infinitely many implications in a single statement. Stated formally, the principle of mathematical induction says that if we can prove the statement $P(0) \wedge \big(\forall k\,(P(k) \to P(k+1))\big)$, then we can deduce that $\forall n\,P(n)$ (again, with $\mathbb{N}$ as the domain of discourse).

Khan Academy offers a video with a clear explanation of mathematical induction: `www.khanacademy.org/math/algebra-home/alg-series-and-induction/alg-induction/v/proof-by-induction`. I highly recommend that you watch it to also hear this information from another perspective.

3.4.1 How to write a proof by induction

It should be intuitively clear that the principle of induction is valid. It follows from the fact that the list 0, 1, 2, 3, ..., if extended long enough, will eventually include any given natural number. If we start from $P(0)$ and take enough steps of the form $P(k) \to P(k+1)$, we can get $P(n)$ for any given natural number n. However, whenever we deal with infinity, we are courting the possibility of paradox. We will prove the principle of induction rigorously in the next chapter (see Theorem 4.3), but for now we just state it as a theorem:

Theorem 3.4. *Let P be a one-place predicate whose domain of discourse includes the natural numbers. Suppose that $P(0) \wedge \big(\forall k \in \mathbb{N}\,(P(k) \to P(k+1))\big)$.*[4] *Then $P(n)$ is true for all*

[3] Image: `commons.wikimedia.org/wiki/File:Dominoeffect.png`.

[4] We will encounter this notation of $k \in \mathbb{N}$ again in Chapter 4. For now you should just remember that $k \in \mathbb{N}$ means: k is an integer from the set $\mathbb{N}$.

natural numbers n. (That is, the statement $\forall n\, P(n)$ is true, where the domain of discourse for P is the set of natural numbers.)

Mathematical induction can be applied in many situations: you can prove things about strings of characters by doing induction on the length of the string, things about graphs by doing induction on the number of nodes in the graph, things about grammars by doing induction on the number of productions in the grammar, and so on. We'll be looking at applications of induction for the rest of this chapter, and treat a form called structural induction in the next chapter.

Although proofs by induction can be very different from one another, they all follow just a few basic structures. A proof based on the preceding theorem always has two parts. First, $P(0)$ is proved. This is called the ***base case*** of the induction. Then the statement $\forall k\,(P(k) \to P(k+1))$ is proved. This statement can be proved by letting k be an arbitrary element of $\mathbb{N}$ and proving $P(k) \to P(k+1)$. This in turn can be proved by assuming that $P(k)$ is true and proving that the truth of $P(k+1)$ follows from that assumption. This case is called the ***inductive case***, and $P(k)$ is called the ***inductive hypothesis*** or the ***induction hypothesis***. Note that the base case is just as important as the inductive case. By itself, the truth of the statement $\forall k\,(P(k) \to P(k+1))$ says nothing at all about the truth of any of the individual statements $P(n)$. The chain of implications $P(0) \to P(1)$, $P(1) \to P(2)$, ..., $P(n-1) \to P(n)$ says nothing about $P(n)$ unless the chain is anchored at the other end by the truth of $P(0)$.

3.4.2 Examples

Let's look at a few examples.

Theorem 3.5. *The number $2^{2n} - 1$ is divisible by 3 for all natural numbers n.*

Proof. Here, $P(n)$ is the statement that $2^{2n} - 1$ is divisible by 3.

Base case: When $n = 0$, $2^{2n} - 1 = 2^0 - 1 = 1 - 1 = 0$ and 0 is divisible by 3 (since $0 = 3 \cdot 0$.) Therefore the statement holds when $n = 0$.

Inductive case: We want to show that if the statement is true for $n = k$ (where k is an arbitrary natural number), then it is true for $n = k + 1$ also. That is, we must prove the implication $P(k) \to P(k+1)$. So we assume $P(k)$, that is, we assume that 2^{2k} is divisible by 3. This means that $2^{2k} - 1 = 3m$ for some integer m. We want to prove $P(k+1)$, that

is, that $2^{2(k+1)} - 1$ is also divisible by 3:

$$\begin{aligned}
2^{2(k+1)} - 1 &= 2^{2k+2} - 1 \\
&= 2^{2k} \cdot 2^2 - 1 && \text{properties of exponents} \\
&= 4 \cdot 2^{2k} - 1 \\
&= 4 \cdot 2^{2k} - 4 + 4 - 1 \\
&= 4(2^{2k} - 1) + 3 && \text{algebra} \\
&= 4(3m) + 3 && \text{the inductive hypothesis} \\
&= 3(4m + 1) && \text{algebra}
\end{aligned}$$

and from the last line we see that 2^{2k+1} is in fact divisible by 3. (The third step—subtracting and adding 4—was done to enable us to use our inductive hypothesis.)

Altogether, we have proved that $P(0)$ holds and that, for all k, $P(k) \rightarrow P(k+1)$ is true. Therefore, by the principle of induction, $P(n)$ is true for all n in $\mathbb{N}$, i.e. $2^{2n} - 1$ is divisible by 3 for all n in $\mathbb{N}$. □

The principle of mathematical induction gives a method for proving $P(n)$ for all n in the set $\mathbb{N}$. It should be clear that if M is any natural number, a similar method can be used to show that $P(n)$ is true for all natural numbers n that satisfy $n \geq M$. Just start the induction with a base case of $n = M$ instead of with a base case of $n = 0$. I leave the proof of this extension of the principle of induction as an exercise. We can use the extended principle of induction to prove a result that was first mentioned in Section 2.1.

Theorem 3.6. *Suppose that a compound proposition contains exactly n propositional variables, where $n \geq 1$. Then there are exactly 2^n different ways of assigning truth values to the n variables.*

Proof. Let $P(n)$ be the statement "There are exactly 2^n different ways of assigning truth values to n propositional variables." We will use induction to prove the $P(n)$ is true for all $n \geq 1$.

Base case: First, we prove the statement $P(1)$. If there is exactly one variable, then there are exactly two ways of assigning a truth value to that variable. Namely, the variable can be either *true* or *false*. Since $2 = 2^1$, $P(1)$ is true.

Inductive case: Suppose that $P(k)$ is already known to be true. We want to prove that, under this assumption, $P(k+1)$ is also true. Suppose that p_1, p_2, ..., p_{k+1} are $k+1$ propositional variables. Since we are assuming that $P(k)$ is true, we know that there are 2^k ways of assigning truth values to p_1, p_2, ..., p_k. But each assignment of truth values to p_1, p_2, ..., p_k can be extended to the complete list p_1, p_2, ..., p_k, p_{k+1} in two ways. Namely, p_{k+1} can be assigned the value *true* or the value *false*. It follows that there are $2 \cdot 2^k$ ways of assigning truth values to p_1, p_2, ..., p_{k+1}. Since $2 \cdot 2^k = 2^{k+1}$, this finishes the proof. □

3.4.3 More examples

The sum of an arbitrary number of terms is written using the symbol $\sum$. (This symbol is the Greek letter sigma, which is equivalent to the Latin letter S and stands for 'sum'.) Thus, we have

$$\sum_{i=1}^{5} i^2 = 1^2 + 2^2 + 3^2 + 4^2 + 5^2$$
$$\sum_{k=3}^{7} a_k = a_3 + a_4 + a_5 + a_6 + a_7$$
$$\sum_{n=0}^{N} \frac{1}{n+1} = \frac{1}{0+1} + \frac{1}{1+1} + \frac{1}{2+1} + \cdots + \frac{1}{N+1}$$

This notation for a sum, using the $\sum$ operator, is called ***summation notation***. A similar notation for products uses the symbol $\prod$. (This is the Greek letter pi, which is equivalent to the Latin letter P and stands for 'product'.) For example,

$$\prod_{k=2}^{5} (3k+2) = (3 \cdot 2 + 2)(3 \cdot 3 + 2)(3 \cdot 4 + 2)(3 \cdot 5 + 2)$$
$$\prod_{i=1}^{n} \frac{1}{i} = \frac{1}{1} \cdot \frac{1}{2} \cdots \frac{1}{n}.$$

Induction can be used to prove many formulas that use these notations. Here are two examples:

Theorem 3.7. $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$ *for any integer n greater than zero.*

We'll prove this theorem in class. Can you do it yourself?

The summation of Theorem 3.7 is often attributed to mathematician and physicist Carl Friedrich Gauss (1777–1855). Gauss has been exceptionally influential in a variety of fields; he is called "the greatest mathematician since antiquity". For example, you might have heard of the Gaussian probability distribution, or perhaps of the Gauss unit for magnetic flux (although we now commonly use the Tesla for that, with 1 Tesla equalling 10^5 Gauss). His brilliance was already apparent in primary school when he allegedly used the 'Gauss sum' from Theorem 3.7 to solve the maths homework the teacher had set the class—to great astonishment of the teacher. You will see this summation again in the course *Algorithms & Data Structures* when analysing the runtime of algorithms containing loops.

Source: `en.wikipedia.org/wiki/Carl_Friedrich_Gauss`.

Theorem 3.8. $\sum_{i=1}^{n} i2^{i-1} = (n-1)\cdot 2^n + 1$ *for any natural number* $n > 0$.

Proof. Let $P(n)$ be the statement $\sum_{i=1}^{n} i2^{i-1} = (n-1)\cdot 2^n + 1$. We use induction to show that $P(n)$ is true for all $n > 0$

Base case: Consider the case $n = 1$. $P(1)$ is the statement that $\sum_{i=1}^{1} i2^{i-1} = (1-1)\cdot 2^1 + 1$. Since each side of this equation is equal to one, this is true.

Inductive case: Let $k > 1$ be arbitrary, and assume that $P(k)$ is true. We want to show that $P(k+1)$ is true. $P(k+1)$ is the statement $\sum_{i=1}^{k+1} i2^{i-1} = ((k+1)-1)\cdot 2^{k+1} + 1$. But, we can compute that

$$\begin{aligned}
\sum_{i=1}^{k+1} i2^{i-1} &= \left(\sum_{i=1}^{k} i2^{i-1}\right) + (k+1)2^{(k+1)-1} \\
&= \Big((k-1)\cdot 2^k + 1\Big) + (k+1)2^k && \text{(inductive hypothesis)} \\
&= \big((k-1) + (k+1)\big)2^k + 1 \\
&= (k\cdot 2)\cdot 2^k + 1 \\
&= k2^{k+1} + 1
\end{aligned}$$

which is what we wanted to show. This completes the induction. □

For example, these theorems show that $\sum_{i=1}^{100} i = 1+2+3+4+\cdots+100 = \frac{100(100+1)}{2} = 5050$ and that $1 \cdot 2^0 + 2 \cdot 2^1 + 3 \cdot 2^2 + 4 \cdot 2^3 + 5 \cdot 2^4 = (5-1)2^5 + 1 = 129$, as well as infinitely many other such sums.

3.5 Strong Mathematical Induction

There is a second form of the principle of mathematical induction which is useful in some cases. To apply the first form of induction, we assume $P(k)$ for an arbitrary natural number k and show that $P(k+1)$ follows from that assumption. In the second form of induction, the assumption is that $P(x)$ holds for all x between 0 and k inclusive, and we show that $P(k+1)$ follows from this. This gives us a lot more to work with when deducing $P(k+1)$. We will need this second, stronger form of induction in the next two sections. A proof will be given in the next chapter.

Theorem 3.9. *Let P be a one-place predicate whose domain of discourse includes the natural numbers. Suppose that $P(0)$ is true and that*

$$(P(0) \wedge P(1) \wedge \cdots \wedge P(k)) \rightarrow P(k+1)$$

is true for each natural number $k \geq 0$. Then $P(n)$ is true for every natural number n.

For example, we can use this theorem to prove that every integer greater than one can be written as a product of prime numbers (where a number that is itself prime is considered to be a product of one prime number). The proof illustrates an important point about applications of this theorem: When proving $P(k+1)$, you don't necessarily have to *use* the assumptions that $P(0)$, $P(1)$, ..., and $P(k)$ are true. If $P(k+1)$ is proved by *any* means—possibly including the assumptions—then the statement $(P(0) \wedge P(1) \wedge \cdots \wedge P(k)) \rightarrow P(k+1)$ has been shown to be true. It follows from this observation that several numbers, not just zero, can be 'base cases' in the sense that $P(x+1)$ can be proved independently of $P(0)$ through $P(x)$. In this sense, 0, 1, and every prime number are base cases in the following theorem.

Theorem 3.10. *Every natural number greater than one can be written as a product of prime numbers.*

Proof. Let $P(n)$ be the statement "if $n > 1$, then n can be written as a product of prime numbers". We will prove that $P(n)$ is true for all n by applying the second form of the principle of induction.

Note that $P(0)$ and $P(1)$ are both automatically true, since $n = 0$ and $n = 1$ do not satisfy the condition that $n > 1$, and $P(2)$ is true since 2 is the product of the single prime

number 2. Suppose that k is an arbitrary natural number with $k > 1$, and suppose that $P(0)$, $P(1)$, ..., $P(k)$ are already known to be true; we want to show that $P(k+1)$ is true. In the case where $k+1$ is a prime number, then $k+1$ is a product of one prime number, so $P(k+1)$ is true.

Consider the case where $k+1$ is not prime. Then, according to the definition of prime number, it is possible to write $k+1 = ab$ where a and b are numbers in the range from 2 to k inclusive. Since $P(0)$ through $P(k)$ are known to be true, a and b can each be written as a product of prime numbers. Since $k+1 = ab$, $k+1$ can also be written as a product of prime numbers. We have shown that $P(k+1)$ follows from $P(0) \wedge P(1) \wedge \cdots \wedge P(k)$, and this completes the induction. □

In two of the pencasts of this course, we treat two flawed induction proofs and examine what mistakes have been made. You can find them here: `youtu.be/m0EIgQyukdQ` and `youtu.be/2c-zw-ENNss`.

Exercises

1. Use induction to prove that $n^3 + 3n^2 + 2n$ is divisible by 3 for all natural numbers n.

2. Use induction to prove that

$$\sum_{i=0}^{n} r^i = \frac{1 - r^{n+1}}{1 - r}$$

for any natural number n and for any real number r such that $r \neq 1$.

3. Use induction to prove that for any natural number n,

$$\sum_{i=0}^{n} \frac{1}{2^i} = 2 - \frac{1}{2^n}$$

In addition to proving this by induction, show that it follows as a corollary of Exercise 2.

4. Use induction to prove that for any natural number n,

$$\sum_{i=0}^{n} 2^i = 2^{n+1} - 1$$

In addition to proving this by induction, show that it follows as a corollary of Exercise 2.

5. Use induction to prove that for any positive integer n,

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$$

6. Use induction to prove that for any positive integer n,

$$\sum_{i=1}^{n}(2i-1) = n^2$$

7. Evaluate the following sums, using results proved in this section and in the previous exercises:

a) $1+3+5+7+9+11+13+15+17+19$

b) $1+\frac{1}{3}+\frac{1}{3^2}+\frac{1}{3^3}+\frac{1}{3^4}+\frac{1}{3^5}+\frac{1}{3^6}$

c) $50+51+52+53+\cdots+99+100$

d) $1+4+9+16+25+36+49+81+100$

e) $\frac{1}{2^2}+\frac{1}{2^3}+\cdots+\frac{1}{2^{99}}$

8. Write each of the sums in the preceding problem using summation notation.

9. Rewrite the proof of Theorem 3.8 without using summation notation.

10. Use induction to prove the following generalized distributive laws for propositional logic: For any natural number $n > 1$ and any propositions q, p_1, p_2, ..., p_n,

a) $q \wedge (p_1 \vee p_2 \vee \cdots \vee p_n) = (q \wedge p_1) \vee (q \wedge p_2) \vee \cdots \vee (q \wedge p_n)$

b) $q \vee (p_1 \wedge p_2 \wedge \cdots \wedge p_n) = (q \vee p_1) \wedge (q \vee p_2) \wedge \cdots \wedge (q \vee p_n)$

3.6 Application: Recursion and Induction

In computer programming, there is a technique called ***recursion*** that is closely related to induction. In a computer program, a ***subroutine*** is a named sequence of instructions for performing a certain task. When that task needs to be performed in a program, the subroutine can be ***called*** by name. A typical way to organize a program is to break down a large task into smaller, simpler subtasks by calling subroutines to perform each of the subtasks. A subroutine can perform its task by calling other subroutines to perform subtasks of the overall task. A subroutine can also call itself. That is, in the process of performing some large task, a subroutine can call itself to perform a subtask. This is known as recursion, and a subroutine that does this is said to be a ***recursive subroutine***. Recursion is appropriate when a large task can be broken into subtasks where some or all of the subtasks are smaller, simpler versions of the main task.

Prolog is an example of a programming language that uses recursion to powerful effect. Classical Prolog has no loop construct: loops are defined using recursive subroutines.

Like induction, recursion is often considered to be a 'hard' topic by students. Experienced computer scientists, on the other hand, often say that they can't see what all the

fuss is about, since induction and recursion are elegant methods which 'obviously' work. In fairness, students have a point, since induction and recursion both manage to pull infinite rabbits out of very finite hats. But the magic is indeed elegant, and learning the trick is very worthwhile.

In your course *Computer Organisation* you will be tasked to write a small recursive program to compute the factorial in Assembly. You can have a sneak-preview below of of what the pseudocode for such a program might be. In future courses like *Algorithms & Data Structures* and *Algorithm Design* you will also be tasked to write and analyse recursive algorithms.

3.6.1 Recursive factorials

A simple example of a recursive subroutine is a function that computes $n!$ for a non-negative integer n. $n!$, which is read "n factorial", is defined as follows:

$$0! = 1$$
$$n! = \prod_{i=1}^{n} i \qquad \text{for } n > 0$$

For example, $5! = 1 \cdot 2 \cdot 3 \cdot 4 \cdot 5 = 120$. Note that for $n > 1$,

$$n! = \prod_{i=1}^{n} i = \left(\prod_{i=1}^{n-1} i \right) \cdot n = \big((n-1)!\big) \cdot n$$

It is also true that $n! = \big((n-1)!\big) \cdot n$ when $n = 1$. This observation makes it possible to write a recursive function to compute $n!$.

All the programming examples in this section are written in the Java programming language. I won't put these blue boxes around them.

To compute *factorial*(n) for $n > 0$, we can write a function (in Java). This function computes *factorial*$(n-1)$ first by calling itself recursively. The answer from that computation is then multiplied by n to give the value of $n!$. The recursion has a base case, namely the case when $n = 0$. For the base case, the answer is computed directly rather than by using recursion. The base case prevents the recursion from continuing forever, in an infinite chain of recursive calls.

Now, as it happens, recursion is not the best way to compute $n!$. It can be computed more efficiently using a loop. Furthermore, except for small values of n, the value of $n!$ is

outside the range of numbers that can be represented as 32-bit *ints*. However, ignoring these problems, the *factorial* function provides a first example of the interplay between recursion and induction. We can use induction to prove that *factorial*(n) does indeed compute $n!$ for $n \geq 0$.[5]

Theorem 3.11. *Assume that the data type int can represent arbitrarily large integers. Under this assumption, the factorial function defined above correctly computes $n!$ for any natural number n.*

Proof. Let $P(n)$ be the statement "*factorial*(n) correctly computes $n!$". We use induction to prove that $P(n)$ is true for all natural numbers n.

Base case: In the case $n = 0$, the *if* statement in the function assigns the value 1 to the answer. Since 1 is the correct value of $0!$, *factorial*(0) correctly computes $0!$.

Inductive case: Let k be an arbitrary natural number, and assume that $P(k)$ is true. From this assumption, we must show that $P(k+1)$ is true. The assumption is that *factorial*(k) correctly computes $k!$, and we want to show that *factorial*$(k+1)$ correctly computes $(k+1)!$.

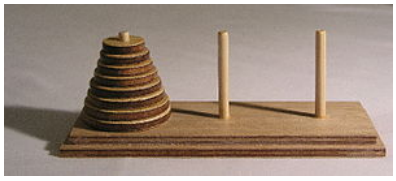
When the function computes *factorial*$(k+1)$, the value of the parameter n is $k+1$. Since $k+1 > 0$, the *if* statement in the function computes the value of *factorial*$(k+1)$ by applying the computation *factorial*$(k) * (k+1)$. We know, by the induction hypothesis, that the value computed by *factorial*(k) is $k!$. It follows that the value computed by *factorial*$(k+1)$ is $(k!) \cdot (k+1)$. As we observed above, for any $k+1 > 0$, $(k!) \cdot (k+1) = (k+1)!$. We see that *factorial*$(k+1)$ correctly computes $(k+1)!$. This completes the induction. □

In this proof, we see that the base case of the induction corresponds to the base case of the recursion, while the inductive case corresponds to a recursive subroutine call. A recursive subroutine call, like the inductive case of an induction, reduces a problem to a 'simpler' or 'smaller' problem, which is closer to the base case.

3.6.2 Towers of Hanoi

Another standard example of recursion is the Towers of Hanoi problem. Let n be a positive integer. Imagine a set of n discs of decreasing size, piled up in order of size, with the largest disc on the bottom and the smallest disc on top. The problem is to move this tower of discs to a second pile, following certain rules: Only one disc can be moved at a time, and a disc can only be placed on top of another disc if the disc on top is smaller. While the discs are being moved from the first pile to the second pile, discs can be kept in a third, spare pile. All the discs must at all times be in one of the three piles.

[5] In the proof, we pretend that the data type *int* is not limited to 32 bits. In reality, the function only gives the correct answer when the answer can be represented as a 32-bit binary number. This is the kind of implementation issue that matters in practice, especially in lower-level languages like Assembly.



The Towers of Hanoi puzzle was first published by Édouard Lucas in 1883. The puzzle is based on a legend of temple wherein there initially was one pile of discs neatly sorted from largest to smallest. In Lucas's story, monks have since been continuously moving discs from this pile of 64 discs according to the rules of the puzzle to again created a sorted stack at the other end of the temple. It is said that when the last disc is placed, the world will end. But on the positive side, even if the monks move one disc every second, it will take approximately 42 times the age of the universe until they are done. And that is assuming they are using the optimal strategy...

Source: `en.wikipedia.org/wiki/Tower_of_Hanoi`

For example, if there are two discs, the problem can be solved by the following sequence of moves:

```
Move disc 1 from pile 1 to pile 3
Move disc 2 from pile 1 to pile 2
Move disc 1 from pile 3 to pile 2
```

A simple recursive subroutine can be used to write out the list of moves to solve the problem for any value of n. The recursion is based on the observation that for $n > 1$, the problem can be solved as follows: Move $n - 1$ discs from pile number 1 to pile number 3 (using pile number 2 as a spare). Then move the largest disc, disc number n, from pile number 1 to pile number 2. Finally, move the $n - 1$ discs from pile number 3 to pile number 2, putting them on top of the n^{th} disc (using pile number 1 as a spare). In both cases, the problem of moving $n - 1$ discs is a smaller version of the original problem and so can be done by recursion. Here is the subroutine, written in Java:

```
void Hanoi(int n, int A, int B, int C) {
  // List the moves for moving n discs from
  // pile number A to pile number B, using
  // pile number C as a spare.  Assume n > 0.
  if (n == 1) {
    System.out.println("Move disc 1 from pile " + A + " to pile " + B);
  }
  else {
    Hanoi(n-1, A, C, B);
    System.out.println("Move disc " + n + " from pile " +
                       A + " to pile " + B);
    Hanoi(n-1, C, B, A);
  }
```

```
}
```

This problem and its fame have led to implementations in a variety of languages, including a language called Brainf*ck.[6] In the *Computer Organisation* course, you can implement an interpreter for this language and test it on the implementation of the Hanoi algorithm.

We can use induction to prove that this subroutine does in fact solve the Towers of Hanoi problem.

Theorem 3.12. *The sequence of moves printed by the Hanoi subroutine as given above correctly solves the Towers of Hanoi problem for any integer $n \geq 1$.*

Proof. We prove by induction that whenever n is a positive integer and A, B, and C are the numbers 1, 2, and 3 in some order, the subroutine call $Hanoi(n, A, B, C)$ prints a sequence of moves that will move n discs from pile A to pile B, following all the rules of the Towers of Hanoi problem.

In the base case, $n = 1$, the subroutine call $Hanoi(1, A, B, C)$ prints out the single step "Move disc 1 from pile A to pile B", and this move does solve the problem for 1 disc.

Let k be an arbitrary positive integer, and suppose that $Hanoi(k, A, B, C)$ correctly solves the problem of moving the k discs from pile A to pile B using pile C as the spare, whenever A, B, and C are the numbers 1, 2, and 3 in some order. We need to show that $Hanoi(k + 1, A, B, C)$ correctly solves the problem for $k + 1$ discs. Since $k + 1 > 1$, $Hanoi(k + 1, A, B, C)$ begins by calling $Hanoi(k, A, C, B)$. By the induction hypothesis, this correctly moves k discs from pile A to pile C. disc number $k + 1$ is not moved during this process. At that point, pile C contains the k smallest discs and pile A still contains the $(k+1)^{st}$ disc, which has not yet been moved. So the next move printed by the subroutine, "Move disc $(k + 1)$ from pile A to pile B", is legal because pile B is empty. Finally, the subroutine calls $Hanoi(k, C, B, A)$, which, by the induction hypothesis, correctly moves the k smallest discs from pile C to pile B, putting them on top of the $(k + 1)^{\text{st}}$ disc, which does not move during this process. At that point, all $(k + 1)$ discs are on pile B, so the problem for $k + 1$ discs has been correctly solved. □

3.6.3 Binary trees

Recursion is often used with linked data structures, which are data structures that are constructed by linking several objects of the same type together with pointers. For an example, we'll look at the data structure known as a ***binary tree***.

[6] See `en.wikipedia.org/wiki/Brainfuck`

If you don't already know about objects and pointers, you will not be able to follow the rest of this section. Time to read about it on the internet?

A binary tree consists of nodes linked together in a tree-like structure. The nodes can contain any type of data, but we will consider binary trees in which each node contains an integer. A binary tree can be empty, or it can consist of a node (called the ***root*** of the tree) and two smaller binary trees (called the ***left subtree*** and the ***right subtree*** of the tree). You can already see the recursive structure: a tree can contain smaller trees. In Java, the nodes of a tree can be represented by objects belonging to the class


```
class BinaryTreeNode {
  int item;   // An integer value stored in the node.
  BinaryTreeNode left;   // Pointer to left subtree.
  BinaryTreeNode right;  // Pointer to right subtree.
}
```

An empty tree is represented by a pointer that has the special value ***null***. If *root* is a pointer to the root node of a tree, then *root.left* is a pointer to the left subtree and *root.right* is a pointer to the right subtree. Of course, both *root.left* and *root.right* can be *null* if the corresponding subtree is empty. Similarly, *root.item* is a name for the integer in the root node.

Binary trees are not part of the material for *Reasoning & Logic* and only serve as examples of induction on algorithms here. You will study this material in more detail in the course *Algorithms & Data Structures*.

Let's say that we want a function that will find the sum of all the integers in all the nodes of a binary tree. We can do this with a simple recursive function. The base case of the recursion is an empty tree. Since there are no integers in an empty tree, the sum of the integers in an empty tree is zero. For a non-empty tree, we can use recursion to find the sums of the integers in the left and right subtrees, and then add those sums to the integer in the root node of the tree. In Java, this can be expressed as follows:

```
int TreeSum( BinaryTreeNode root ) {
 // Find the sum of all the integers in the
 // tree that has the given root.
 int answer;
 if (root == null) {  // The tree is empty.
   answer = 0;
 } else {
  answer = TreeSum(root.left);
  answer += TreeSum(root.right);
  answer += root.item;
 }
 return answer;
```

```
13 }
```

We can use the second form of the principle of mathematical induction to prove that this function is correct.

Theorem 3.13. *The function TreeSum, defined above, correctly computes the sum of all the integers in a binary tree.*

Proof. We use induction on the number of nodes in the tree. Let $P(n)$ be the statement "*TreeSum* correctly computes the sum of the nodes in any binary tree that contains exactly n nodes". We show that $P(n)$ is true for every natural number n.

Consider the case $n = 0$. A tree with zero nodes is empty, and an empty tree is represented by a *null* pointer. In this case, the *if* statement in the definition of *TreeSum* assigns the value 0 to the answer, and this is the correct sum for an empty tree. So, $P(0)$ is true.

Let k be an arbitrary natural number, with $k > 0$. Suppose we already know $P(x)$ for each natural number x with $0 \leq x < k$. That is, *TreeSum* correctly computes the sum of all the integers in any tree that has fewer than k nodes. We must show that it follows that $P(k)$ is true, that is, that *TreeSum* works for a tree with k nodes. Suppose that *root* is a pointer to the root node of a tree that has a total of k nodes. Since the root node counts as a node, that leaves a total of $k - 1$ nodes for the left and right subtrees, so each subtree must contain fewer than k nodes. By the induction hypothesis, we know that *TreeSum*(*root.left*) correctly computes the sum of all the integers in the left subtree, and *TreeSum*(*root.right*) correctly computes the sum of all the integers in the right subtree. The sum of all the integers in the tree is *root.item* plus the sums of the integers in the subtrees, and this is the value computed by *TreeSum*. So, *TreeSum* does work for a tree with k nodes. This completes the induction. □

Note how closely the structure of the inductive proof follows the structure of the recursive function. In particular, the second principle of mathematical induction is very natural here, since the size of subtree could be anything up to one less than the size of the complete tree. It would be very difficult to use the first principle of induction in a proof about binary trees.

Exercises

1. The *Hanoi* subroutine given in this section does not just solve the Towers of Hanoi problem. It solves the problem using the minimum possible number of moves. Use induction to prove this fact.

2. Use induction to prove that the *Hanoi* subroutine uses $2^n - 1$ moves to solve the Towers of Hanoi problem for n discs.

3. Consider the following recursive function:

```
int power(int x, int n) {
  // Compute x raised to the power n.
  // Assume that n >= 0.
  int answer;
  if (n == 0) {
    answer = 1;
  } else if (n % 2 == 0) {
    answer = power(x * x, n / 2);
  } else {
    answer = x * power(x * x, (n-1) / 2);
  }
  return answer;
}
```

Show that for any integer x and any non-negative integer n, the function *power*(x,n) correctly computes the value of x^n. (Assume that the *int* data type can represent arbitrarily large integers.) Note that the test "`if (n % 2 == 0)`" tests whether n is evenly divisible by 2. That is, the test is true if n is an even number. (This function is actually a very efficient way to compute x^n.)

4. A ***leaf node*** in a binary tree is a node in which both the left and the right subtrees are empty. Prove that the following recursive function correctly counts the number of leaves in a binary tree:

```
int LeafCount(BinaryTreeNode root) {
  // Counts the number of leaf nodes in
  // the tree with the specified root.
  int count;
  if (root == null) {
    count = 0;
  } else if (root.left == null && root.right == null) {
    count = 1;
  } else {
    count = LeafCount(root.left);
    count += LeafCount(root.right);
  }
  return count;
}
```

5. A ***binary sort tree*** satisfies the following property: If *node* is a pointer to any node in the tree, then all the integers in the left subtree of *node* are less than *node.item* and all the integers in the right subtree of *node* are greater than or equal to *node.item*. Prove that the following recursive subroutine prints all the integers in a binary sort tree in non-decreasing order:

```
void SortPrint(BinaryTreeNode root) {
  // Assume that root is a pointer to the
  // root node of a binary sort tree.  This // subroutine prints the integers in
     the
  // tree in non-decreasing order.
  if (root == null) {
    // There is nothing to print.
```

```
    }
    else {
      SortPrint(root.left);
      System.out.println(root.item);
      SortPrint(root.right);
    }
  }
```

3.7 Recursive Definitions

Recursion occurs in programming when a subroutine is defined—or at least partially defined—in terms of itself. But recursion also occurs outside of programming. A ***recursive definition*** is a definition that includes a reference to the term that is being defined. A recursive definition defines something at least partially in terms of itself. As in the case of recursive subroutines, mathematical induction can often be used to prove facts about things that are defined recursively.

As I already noted, there is a recursive definition for $n!$, for n in $\mathbb{N}$, and we can use this definition to prove facts about the factorials. We can define $0! = 1$ and $n! = n \cdot (n-1)!$ for $n > 0$. Do you see how the base case and the inductive case in an inductive proof can correspond to the two parts of the recursive definition?

Other sequences of numbers can also be defined recursively. For example, the famous ***Fibonacci sequence*** is the sequence of numbers f_0, f_1, f_2, ..., defined recursively by

$$\begin{aligned} f_0 &= 0 \\ f_1 &= 1 \\ f_n &= f_{n-1} + f_{n-2} \qquad \text{for } n > 1 \end{aligned}$$

Using this definition, we compute that

$$\begin{aligned} f_2 &= f_1 + f_0 = 0 + 1 = 1 \\ f_3 &= f_2 + f_1 = 1 + 1 = 2 \\ f_4 &= f_3 + f_2 = 2 + 1 = 3 \\ f_5 &= f_4 + f_3 = 3 + 2 = 5 \\ f_6 &= f_5 + f_4 = 5 + 3 = 8 \\ f_7 &= f_6 + f_5 = 8 + 5 = 13 \end{aligned}$$

and so on. Based on this definition, we can use induction to prove facts about the Fibonacci sequence. We can prove, for example, that f_n grows exponentially with n, even without finding an exact formula for f_n:

Theorem 3.14. *The Fibonacci sequence, f_0, f_1, f_2, ..., satisfies $f_n > \left(\frac{3}{2}\right)^{n-1}$, for $n \geq 6$.*

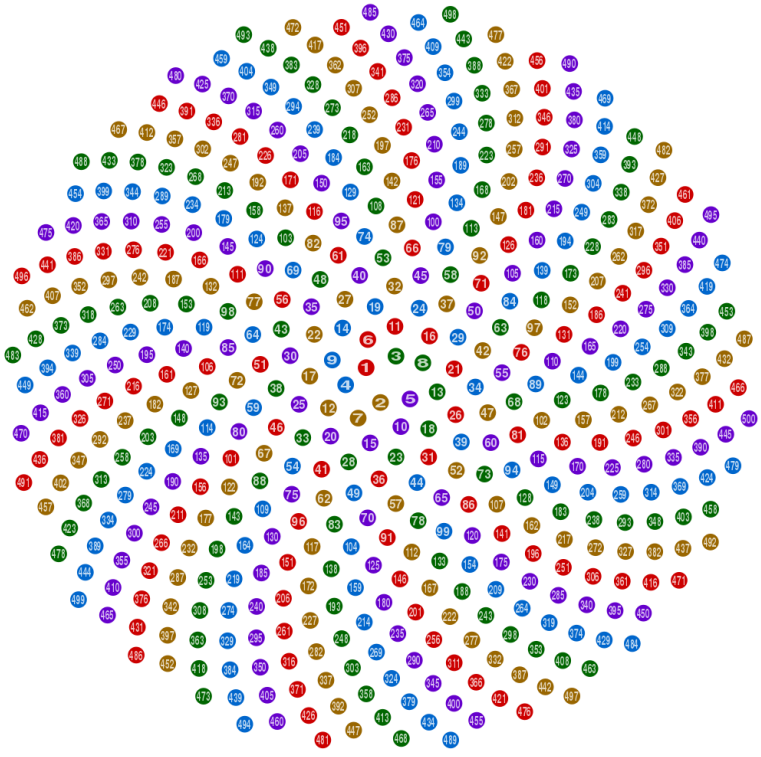


Figure 3.1: *Fibonacci numbers occur in nature, as in this model of the florets in the head of a sunflower. Source:* `commons.wikimedia.org/wiki/File:SunflowerModel.svg`

Proof. We prove this by induction on n. For $n = 6$, we have that $f_n = 8$ while $1.5^{n-1} = 1.5^5$, which is about 7.6. So $f_n > 1.5^{n-1}$ for $n = 6$. Similarly, for $n = 7$, we have $f_n = 13$ and $1.5^{n-1} = 1.5^6$, which is about 11.4. So $f_n > 1.5^{n-1}$ for $n = 7$.

Now suppose that k is an arbitrary integer with $k > 7$. Suppose that we already know that $f_n > 1.5^{n-1}$ for $n = k - 1$ and for $n = k - 2$. We want to show that the inequality

then holds for $n = k$ as well. But

$$\begin{aligned} f_k &= f_{k-1} + f_{k-2} \\ &> 1.5^{(k-1)-1} + 1.5^{(k-2)-1} && \text{(by the induction hypothesis)} \\ &= 1.5^{k-2} + 1.5^{k-3} \\ &= (1.5) \cdot (1.5^{k-3}) + (1.5^{k-3}) \\ &= (2.5) \cdot (1.5^{k-3}) \\ &> (1.5^2) \cdot (1.5^{k-3}) && \text{(since } 1.5^2 = 2.25\text{)} \\ &= 1.5^{k-1} \end{aligned}$$

This string of equalities and inequalities shows that $f_k > 1.5^{k-1}$. This completes the induction and proves the theorem. □

Exercises

1. Prove that the Fibonacci sequence, f_0, f_1, f_2, ..., satisfies $f_n < 2^n$ for all natural numbers $n \geq 1$.

2. Suppose that a_1, a_2, a_3, ..., is a sequence of numbers which is defined recursively by $a_1 = 1$ and $a_n = 2a_{n-1} + 2^{n-1}$ for $n > 1$. Prove that $a_n = n2^{n-1}$ for every positive integer n.

3.8 Invariants

Recursion is closely linked to iteration. In fact, a `while` loop can be written as a recursive subroutine (and this how the language Prolog achieves 'iteration': see page 82). In computer science we would like to prove correctness and other properties about algorithms. Proofs about algorithms can be more difficult than the proofs about simple properties of the integers that I often use as examples this book.

A tool that helps us to prove properties about algorithms is an ***invariant***. For example, a *loop invariant* is a property P of a loop such that:

1. P is true *before* the loop is executed, and

2. P remains true *after* each execution of the body of the loop (but not necessarily *in-between* the steps within the body).

So to prove that an algorithm A has the property Q (the *post-condition*), we can find an invariant P of A such that Q follows from P, together with the fact that A has terminated. This last fact that A has terminated means that the loop condition (the *guard* of the loop) has become false.

In more detail, we need to find an invariant and show four things about it:

1. **Initialization or basis property.** The invariant holds before the first iteration of the loop.

2. **Maintenance or inductive property.** If the invariant holds before an iteration, then it also holds before the next iteration.

3. **Termination and falsity of guard.** After a finite number of iterations the guard becomes false and the loop terminates.

4. **Correctness of the post-condition.** The invariant together with the negation of the guard imply that the post-condition holds, in which case the program is correct.

In one of the pencasts of this course, I prove the correctness of an algorithm using an invariant. You can find that pencast here: `youtu.be/GSvqF48TVM4`.

As an example, consider the simple loop:

```
1 while (x < 10)
      x = x+1;
```

What does this loop achieve? What is an invariant that helps us to prove the loop correctly achieves this? The invariant is $x \leq 10$—check that it does satisfy the above four properties!

While invariants can be useful, a suitable invariant can be difficult to find.

For a more complex example, consider the following. Note that you should call this bit of code with an integer $n \geq 0$ and $a > 0$.

```
  r = 0
2 b = n
  while (b >= a)
4   b -= a
    r += 1
```

Try to convince yourself that this code computes: $\lfloor n/a \rfloor$. Don't believe me? I will prove it to you:

Proof. Invariant: $r \cdot a + b = n$

1. **Initialization or basis property.** Before the loop runs, $b = n$ and $r = 0$. Thus $r \cdot a + b = b = n$.

2. **Maintenance or inductive property.** Assume the invariant holds before iteration k, thus: $r_{\text{old}} \cdot a + b_{\text{old}} = n$.
   Now we prove that it holds after iteration k, that is: $r_{\text{new}} \cdot a + b_{\text{new}} = n$.
   From line 4 we derive that: $b_{\text{new}} = b_{\text{old}} - a$ and $r_{\text{new}} = r_{\text{old}} + 1$. Thus $r_{\text{new}} \cdot a + b_{\text{new}} = (r_{\text{old}} + 1) \cdot a + b_{\text{old}} - a = r_{\text{old}} \cdot a + a + b_{\text{old}} - a = r_{\text{old}} \cdot a + b_{\text{old}} \overset{\text{IH}}{=} n$.

3. **Termination and falsity of guard.** Every iteration b decreases by a. Since $a > 0$, this means that eventually $b < a$ will hold.

4. **Correctness of the post-condition.** Since $0 \leq b < a$ and $r \cdot a + b = n$, we know that: $r \cdot a \leq n$ and $n = r \cdot a + b < r \cdot a + a < (r+1) \cdot a$. So we get: $r \leq n/a$ and $n/a < r + 1$, thus $n/a - 1 < r \leq n/a$. Since r is integer, this means: $f = \lfloor n/a \rfloor$.

□

There is a form of logic, *Floyd–Hoare logic*, in which we can express invariants and can formally prove the (partial) correctness of a program. Read about it on wikipedia: `en.wikipedia.org/wiki/Loop_invariant`.

Chapter 4

Sets, Functions, and Relations

We deal with the complexity of the world by putting things into categories. There are not just hordes of individual creatures. There are dogs, cats, elephants, and mice. There are mammals, insects, and fish. Animals, vegetables, and minerals. Solids, liquids, and gases. Things that are red. Big cities. Pleasant memories.... Categories build on categories. They are the subject and the substance of thought.

In mathematics, which operates in its own abstract and rigorous world, categories are modelled by ***sets***. A set is just a collection of elements. Along with logic, sets form the 'foundation' of mathematics, just as categories are part of the foundation of day-to-day thought. In this chapter, we study sets and relationships among sets. And, yes, that means we'll prove theorems about sets!

4.1 Basic Concepts

A ***set*** is a collection of ***elements***. A set is defined entirely by the elements that it contains. An element can be anything, including another set. You will notice that this is not a precise mathematical definition. Instead, it is an intuitive description of what the word 'set' is supposed to mean: any time you have a bunch of entities and you consider them as a unit, you have a set. Mathematically, sets are really defined by the operations that can be performed on them. These operations model things that can be done with collections of objects in the real world. These operations are the subject of the branch of mathematics known as ***set theory***.

The most basic operation in set theory is forming a set from a given list of specific entities. The set that is formed in this way is denoted by enclosing the list of entities between a left brace, '{', and a right brace, '}'. The entities in the list are separated by commas. For example, the set denoted by

$$\{\ 17,\ \pi,\ \texttt{New York City},\ \texttt{King Willem-Alexander},\ \texttt{Euromast}\ \}$$

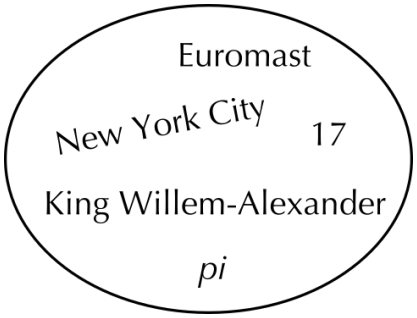


Figure 4.1: *Venn diagram of an example set.*

is the set that contains the entities 17, π, New York City, King Willem-Alexander, and Euromast. These entities are the elements of the set. Since we assume that a set is completely defined by the elements that it contains, the set is well-defined. Of course, we still haven't said what it means to be an 'entity'. Something as definite as 'New York City' should qualify, except that it doesn't seem like New York City really belongs in the world of mathematics. The problem is that mathematics is supposed to be its own self-contained world, but it is supposed to model the real world. When we use mathematics to model the real world, we admit entities such as New York City and even Euromast. But when we are doing mathematics *per se*, we'll generally stick to obviously mathematical entities such as the integer 17 or the real number π. We will also use letters such as a and b to refer to entities. For example, when I say something like "Let A be the set $\{a, b, c\}$", I mean a, b, and c to be particular, but unspecified, entities.

It's important to understand that a set is defined by the elements that it contains, and not by the order in which those elements might be listed. For example, the notations $\{a, b, c, d\}$ and $\{b, c, a, d\}$ define the same set. Furthermore, a set can only contain one copy of a given element, even if the notation that specifies the set lists the element twice. This means that $\{a, b, a, a, b, c, a\}$ and $\{a, b, c\}$ specify exactly the same set. Note in particular that it's incorrect to say that the set $\{a, b, a, a, b, c, a\}$ contains seven elements, since some of the elements in the list are identical. The notation $\{a, b, c\}$ can lead to some confusion, since it might not be clear whether the letters a, b, and c are assumed to refer to three *different* entities. A mathematician would generally *not* make this assumption without stating it explicitly, so that the set denoted by $\{a, b, c\}$ could actually contain either one, two, or three elements. When it is important that different letters refer to different entities, I will say so explicitly, as in "Consider the set $\{a, b, c\}$, where a, b, and c are distinct."

4.1.1 Elements of sets

The symbol $\in$ is used to express the relation 'is an element of'. That is, if a is an entity and A is a set, then $a \in A$ is a statement that is true if and only if a is one of the elements of A. In that case, we also say that a is a ***member*** of the set A. The assertion that a is not an element of A is expressed by the notation $a \notin A$. Note that both $a \in A$ and $a \notin A$ are statements in the sense of propositional logic. That is, they are assertions which can be either true or false. The statement $a \notin A$ is equivalent to $\neg(a \in A)$.

As you may have noticed by now, it is convention for sets to be denoted using capital letters (e.g. 'A') and elements within sets to be denoted using lower-case letters (e.g. 'a'). You should adhere to the same convention to prevent misunderstandings!

It is possible for a set to be empty, that is, to contain no elements whatsoever. Since a set is completely determined by the elements that it contains, there is only one set that contains no elements. This set is called the ***empty set***, and it is denoted by the symbol $\emptyset$. Note that for any element a, the statement $a \in \emptyset$ is false. The empty set, $\emptyset$, can also be denoted by an empty pair of braces, i.e., $\{\ \}$.

If A and B are sets, then, by definition, A is equal to B if and only if they contain exactly the same elements. In this case, we write $A = B$. Using the notation of predicate logic, we can say that $A = B$ if and only if $\forall x(x \in A \leftrightarrow x \in B)$.

Later, when proving theorems in set theory, we will find it can often help to use this predicate logic notation to simplify our proofs. To avoid having to look them up later, make sure that you understand why the predicate logic notation is equivalent to the set notation.

Suppose now that A and B are sets such that every element of A is an element of B. In that case, we say that A is a ***subset*** of B, i.e. A is a subset of B if and only if $\forall x(x \in A \rightarrow x \in B)$. The fact that A is a subset of B is denoted by $A \subseteq B$. Note that $\emptyset$ is a subset of every set B: $x \in \emptyset$ is false for any x, and so given any B, $(x \in \emptyset \rightarrow x \in B)$ is true for all x.

If $A = B$, then it is automatically true that $A \subseteq B$ and that $B \subseteq A$. The converse is also true: If $A \subseteq B$ and $B \subseteq A$, then $A = B$. This follows from the fact that for any x, the statement $(x \in A \leftrightarrow x \in B)$ is logically equivalent to the statement $(x \in A \rightarrow x \in B) \wedge (x \in B \rightarrow x \in A)$. This fact is important enough to state as a theorem.

Theorem 4.1. *Let A and B be sets. Then $A = B$ if and only if both $A \subseteq B$ and $B \subseteq A$.*

This theorem expresses the following advice: If you want to check that two sets, A

and B, are equal, you can do so in two steps. First check that every element of A is also an element of B, and then check that every element of B is also an element of A.

If $A \subseteq B$ but $A \neq B$, we say that A is a ***proper subset*** of B. We use the notation $A \subsetneq B$ to mean that A is a proper subset of B. That is, $A \subsetneq B$ if and only if $A \subseteq B \wedge A \neq B$. We will sometimes use $A \supseteq B$ as an equivalent notation for $B \subseteq A$, and $A \supsetneq B$ as an equivalent for $B \subsetneq A$. Other text books also sometimes use the $\subset$ symbol to represent proper subsets, e.g., $A \subset B \equiv A \subsetneq B$. Additionally, you may come across $A \not\subseteq B$ which means that A is not a subset of B. Notice that (especially in written text) the difference between $A \subsetneq B$ and $A \not\subseteq B$ can be small, so make sure to read properly and to write clearly!

4.1.2 Set-builder notation

A set can contain an infinite number of elements. In such a case, it is not possible to list all the elements in the set: we cannot give an ***extensional definition*** of the set. Sometimes the ellipsis '...' is used to indicate a list that continues on infinitely. For example, $\mathbb{N}$, the set of natural numbers, can be specified as

$$\mathbb{N} = \{0, 1, 2, 3, \dots\}$$

However, this is an informal notation, which is not really well-defined, and it should only be used in cases where it is clear what it means. It's not very useful to say that "the set of prime numbers is $\{2, 3, 5, 7, 11, 13, \dots\}$", and it is completely meaningless to talk about "the set $\{17, 42, 105, \dots\}$". Clearly, we need another way to specify sets besides listing their elements. The need is fulfilled by predicates.

If $P(x)$ is a predicate, then we can form the set that contains all entities a such that a is in the domain of discourse for P and $P(a)$ is true. The notation $\{x \mid P(x)\}$ is used to denote this set. This is the ***intensional definition*** of the set. The name of the variable, x, is arbitrary, so the same set could equally well be denoted as $\{z \mid P(z)\}$ or $\{r \mid P(r)\}$. The notation $\{x \mid P(x)\}$ can be read "the set of x such that $P(x)$". We call this the ***set-builder notation***, as you can think of the predicate as a building material for the elements of the set. For example, if $E(x)$ is the predicate 'x is an even number', and if the domain of discourse for E is the set $\mathbb{N}$, then the notation $\{x \mid E(x)\}$ specifies the set of even natural numbers. That is,

$$\{x \mid E(x)\} = \{0, 2, 4, 6, 8, \dots\}$$

It turns out, for deep and surprising reasons that we will discuss later, that we have to be a little careful about what counts as a predicate. In order for the notation $\{x \mid P(x)\}$ to be valid, we have to assume that the domain of discourse of P is in fact a set. (You might wonder how it could be anything else. That's the surprise!)

Often, it is useful to specify the domain of discourse explicitly in the notation that defines a set. In the above example, to make it clear that x must be a natural number, we could write the set as $\{x \in \mathbb{N} \mid E(x)\}$. This notation can be read as "the set of all x in $\mathbb{N}$ such that $E(x)$". More generally, if X is a set and P is a predicate whose domain of discourse includes all the elements of X, then the notation

$$\{x \in X \mid P(x)\}$$

is the set that consists of all entities a that are members of the set X and for which $P(a)$ is true. In this notation, we don't have to assume that the domain of discourse for P is a set, since we are effectively limiting the domain of discourse to the set X. The set denoted by $\{x \in X \mid P(x)\}$ could also be written as $\{x \mid x \in X \wedge P(x)\}$.

We can use this notation to define the set of prime numbers in a rigorous way. A prime number is a natural number n which is greater than 1 and which satisfies the property that for any factorization $n = xy$, where x and y are natural numbers, either x or y must be n. We can express this definition as a predicate and define the set of prime numbers as

$$\{n \in \mathbb{N} \mid (n > 1) \wedge \\ \forall x \forall y \big((x \in \mathbb{N} \wedge y \in \mathbb{N} \wedge n = xy) \rightarrow (x = n \vee y = n) \big) \}$$

Admittedly, this definition is hard to take in in one gulp. But this example shows that it is possible to define complex sets using predicates.

4.1.3 Operations on sets

Now that we have a way to express a wide variety of sets, we turn to operations that can be performed on sets. The most basic operations on sets are ***union*** and ***intersection***. If A and B are sets, then we define the union of A and B to be the set that contains all the elements of A together with all the elements of B. The union of A and B is denoted by $A \cup B$. The union can be defined formally as

$$A \cup B = \{x \mid x \in A \vee x \in B\}.$$

The intersection of A and B is defined to be the set that contains every entity that is both a member of A and a member of B. The intersection of A and B is denoted by $A \cap B$. Formally,

$$A \cap B = \{x \mid x \in A \wedge x \in B\}.$$

An entity gets into $A \cup B$ if it is in *either* A or B. It gets into $A \cap B$ if it is in *both* A and B. Note that the symbol for the logical 'or' operator, $\vee$, is similar to the symbol for the union operator, $\cup$, while the logical 'and' operator, $\wedge$, is similar to the intersection operator, $\cap$.

The ***set difference*** of two sets, A and B, is defined to be the set of all entities that are members of A but are not members of B. The set difference of A and B is denoted $A \setminus B$ or alternatively as $A - B$. The idea is that $A \setminus B$ is formed by starting with A and then removing any element that is also found in B. Formally,

$$A \setminus B = \{x \mid x \in A \wedge x \notin B\}.$$

Union and intersection are clearly commutative operations. That is, $A \cup B = B \cup A$ and $A \cap B = B \cap A$ for any sets A and B. However, set difference is not commutative. In general, $A \setminus B \neq B \setminus A$.

Suppose that $A = \{a, b, c\}$, that $B = \{b, d\}$, and that $C = \{d, e, f\}$. Then we can apply the definitions of union, intersection, and set difference to compute, for example, that:

$$\begin{array}{lll} A \cup B = \{a, b, c, d\} & A \cap B = \{b\} & A \setminus B = \{a, c\} \\ A \cup C = \{a, b, c, d, e, f\} & A \cap C = \emptyset & A \setminus C = \{a, b, c\} \end{array}$$

In this example, the sets A and C have no elements in common, so that $A \cap C = \emptyset$. There is a term for this: Two sets are said to be ***disjoint*** if they have no elements in common. That is, for any sets A and B, A and B are said to be disjoint if and only if $A \cap B = \emptyset$.

Of course, the set operations can also be applied to sets that are defined by predicates. The next example illustrates this.

let $L(x)$ be the predicate 'x is lucky', and let $W(x)$ be the predicate 'x is wise', where the domain of discourse for each predicate is the set of people. Let $X = \{x \mid L(x)\}$, and let $Y = \{x \mid W(x)\}$. Then

$$\begin{aligned} X \cup Y &= \{x \mid L(x) \vee W(x)\} = \{\text{people who are lucky or wise}\} \\ X \cap Y &= \{x \mid L(x) \wedge W(x)\} = \{\text{people who are lucky and wise}\} \\ X \setminus Y &= \{x \mid L(x) \wedge \neg W(x)\} = \{\text{people who are lucky but not wise}\} \\ Y \setminus X &= \{x \mid W(x) \wedge \neg L(x)\} = \{\text{people who are wise but not lucky}\} \end{aligned}$$

Notation	Definition
$a \in A$	a is a member (or element) of A
$a \notin A$	$\neg(a \in A)$, a is not a member of A
$\emptyset$	the empty set, which contains no elements
$A \subseteq B$	A is a subset of B, $\forall x(x \in A \rightarrow x \in B)$
$A \subsetneq B$	A is a proper subset of B, $A \subseteq B \wedge A \neq B$
$A \supseteq B$	A is a superset of B, same as $B \subseteq A$
$A \supsetneq B$	A is a proper superset of B, same as $B \subsetneq A$
$A = B$	A and B have the same members, $A \subseteq B \wedge B \subseteq A$
$A \cup B$	union of A and B, $\{x \mid x \in A \vee x \in B\}$
$A \cap B$	intersection of A and B, $\{x \mid x \in A \wedge x \in B\}$
$A \setminus B$	set difference of A and B, $\{x \mid x \in A \wedge x \notin B\}$
$\mathcal{P}(A)$	power set of A, $\{X \mid X \subseteq A\}$

Figure 4.2: *Some of the notations that are defined in this section. A and B are sets, and a is an entity.*

You have to be a little careful with the English word 'and'. We might say that the set $X \cup Y$ contains people who are lucky *and* people who are wise. But what this means is that a person gets into the set $X \cup Y$ either by being lucky *or* by being wise, so $X \cup Y$ is defined using the logical 'or' operator, $\vee$.

4.1.4 Visualising sets[1]

A ***Venn diagram*** shows all possible logical relations between a finite collection of different sets. These diagrams depict elements as points in the plane, and sets as regions inside closed curves. So a Venn diagram consists of multiple overlapping closed curves, usually circles, each representing a set. The points inside a curve (circle) labelled S represent elements of the set S, while points outside the boundary represent elements not in the set S. Figure 4.1 shows our example set which opened the section.

Venn diagrams help us to visualise sets and set operations. For example, the set of

[1]This subsection is derived from `en.wikipedia.org/wiki/Venn_diagram`.

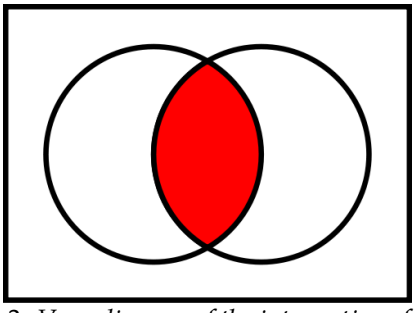


Figure 4.3: *Venn diagram of the intersection of two sets.*

all elements that are members of both sets S and T, $S \cap T$, is represented visually by the area of overlap of the regions S and T: see Figure 4.3. In Venn diagrams the curves are overlapped in every possible way, showing all possible relations between the sets. You can find it useful to draw a Venn diagram to gain intuition of what's happening. On their own, Venn diagrams do not offer a proof for theorems in set theory however.

4.1.5 Sets of sets

Sets can contain other sets as elements. For example, the notation $\{a, \{b\}\}$ defines a set that contains two elements, the entity a and the set $\{b\}$. Since the set $\{b\}$ is a member of the set $\{a, \{b\}\}$, we have that $\{b\} \in \{a, \{b\}\}$. On the other hand, provided that $a \neq b$, the statement $\{b\} \subseteq \{a, \{b\}\}$ is false, since saying $\{b\} \subseteq \{a, \{b\}\}$ is equivalent to saying that $b \in \{a, \{b\}\}$, and the entity b is not one of the two members of $\{a, \{b\}\}$. For the entity a, it is true that $\{a\} \subseteq \{a, \{b\}\}$ and for the set $\{b\}$, it is true that $\{\{b\}\} \subseteq \{a, \{b\}\}$. Study these examples carefully before you continue, as many students struggle with the notion and notation of putting sets in sets.

Given a set A, we can construct the set that contains all the subsets of A. This set is called the ***power set*** of A, and is denoted $\mathcal{P}(A)$. Formally, we define

$$\mathcal{P}(A) = \{X \mid X \subseteq A\}.$$

For example, if $A = \{a, b\}$, then the subsets of A are the empty set, $\{a\}$, $\{b\}$, and $\{a, b\}$, so the power set of A is set given by

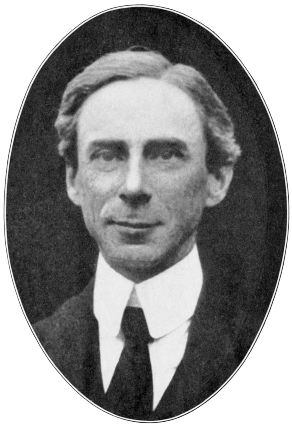
$$\mathcal{P}(A) = \{\, \emptyset,\ \{a\},\ \{b\},\ \{a, b\}\, \}.$$

So the power set of of A has four elements. Does this surprise you?

Note that since the empty set is a *subset* of any set, the empty set is an *element* of the power set of any set. That is, for any set A, $\emptyset \subseteq A$ and $\emptyset \in \mathcal{P}(A)$. Since the empty set is a subset of itself, and is its only subset, we have that $\mathcal{P}(\emptyset) = \{\emptyset\}$. The set $\{\emptyset\}$ is not empty. It contains one element, namely $\emptyset$.

The Nobel Prize was won by Bertrand Russell (1872–1970), a dominant figure in British thought during the twentieth century. Russell was a philosopher and mathematician, and also a historian, social critic and political activist. With A. N. Whitehead, Russell wrote Principia Mathematica, an epic attempt to create a logical basis for mathematics. His work has had a considerable influence on computer science, and not just for his contributions to logic and set theory: he proposed the beginnings of what are now type systems.

Source: `en.wikipedia.org/wiki/Bertrand_Russell`.



We remarked earlier in this section that the notation $\{x \mid P(x)\}$ is only valid if the domain of discourse of P is a set. This might seem a rather puzzling thing to say—after all, why and how would the domain of discourse be anything else? The answer is related to Russell's Paradox, which we mentioned briefly in Chapter 3 and which shows that it is logically impossible for the set of all sets to exist. This impossibility can be demonstrated using a proof by contradiction. In the proof, we use the existence of the set of all sets to define another set which cannot exist because its existence would lead to a logical contradiction.

Theorem 4.2. *There is no set of all sets.*

Proof. Suppose that the set of all sets exists. We will show that this assumption leads to a contradiction. Let V be the set of all sets. We can then define the set R to be the set which contains every set that does not contain itself. That is,

$$R = \{X \in V \mid X \notin X\}$$

Now, we must have either $R \in R$ or $R \notin R$. We will show that either case leads to a contradiction.

Consider the case where $R \in R$. Since $R \in R$, R must satisfy the condition for membership in R. A set X is in R iff $X \notin X$. To say that R satisfies this condition means that $R \notin R$. That is, from the fact that $R \in R$, we deduce the contradiction that $R \notin R$.

Now consider the remaining case, where $R \notin R$. Since $R \notin R$, R does not satisfy the condition for membership in R. Since the condition for membership is that $R \notin R$, and this condition is false, the statement $R \notin R$ must be false. But this means that the statement $R \in R$ is true. From the fact that $R \notin R$, we deduce the contradiction that $R \in R$.

Since both possible cases, $R \in R$ and $R \notin R$, lead to contradictions, we see that it is not possible for R to exist. Since the existence of R follows from the existence of V, we see that V also cannot exist. □

This (in)famous contradiction has been adapted to natural language to make it easier to convey the problem to laymen. Unfortunately many of these translations are flawed. Can you think of a solution for the following for instance? "The barber of Seville shaves all men who do not shave themselves. Who shaves the barber?"

To avoid Russell's paradox, we must put limitations on the construction of new sets. We can't force the set of all sets into existence simply by thinking of it. We can't form the set $\{x \mid P(x)\}$ unless the domain of discourse of P is a set. Any predicate Q can be used to form a set $\{x \in X \mid Q(x)\}$, but this notation requires a pre-existing set X. Predicates can be used to form subsets of existing sets, but they can't be used to form new sets completely from scratch.

The notation $\{x \in A \mid P(x)\}$ is a convenient way to effectively limit the domain of discourse of a predicate, P, to members of a set, A, that we are actually interested in. We will use a similar notation with the quantifiers $\forall$ and $\exists$. The proposition $(\forall x \in A)(P(x))$ is true if and only if $P(a)$ is true for every element a of the set A. And the proposition $(\exists x \in A)(P(x))$ is true if and only if there is some element a of the set A for which $P(a)$ is true. These notations are valid only when A is contained in the domain of discourse for P. As usual, we can leave out parentheses when doing so introduces no ambiguity. So, for example, we might write $\forall x \in A\ P(x)$.

4.1.6 Mathematical induction revisited

We end this section by returning to the topic of mathematical induction. First, I will give proofs of the two forms of the principle of mathematical induction. These proofs were omitted from the previous chapter, but only for the lack of a bit of set notation. In fact,

the principle of mathematical induction is valid only because it follows from one of the basic axioms that define the natural numbers, namely the fact that any non-empty set of natural numbers has a smallest element. Given this axiom, we can use it to prove the following two theorems:

Theorem 4.3. *Let P be a one-place predicate whose domain of discourse includes the natural numbers. Suppose that $P(0) \wedge \big(\forall k \in \mathbb{N}\ (P(k) \rightarrow P(k+1))\big)$. Then $\forall n \in \mathbb{N}$, $P(n)$.*

Proof. Suppose that both $P(0)$ and $\big(\forall k \in \mathbb{N}\,(P(k) \rightarrow P(k+1))\big)$ are true, but that $\big(\forall n \in \mathbb{N},\ P(n)\big)$ is false. We show that this assumption leads to a contradiction.

Since the statement $\forall n \in \mathbb{N}$, $P(n)$ is false, its negation, $\neg(\forall n \in \mathbb{N},\ P(n))$, is true. The negation is equivalent to $\exists n \in \mathbb{N}$, $\neg P(n)$. Let $X = \{n \in \mathbb{N} \mid \neg P(n)\}$. Since $\exists n \in \mathbb{N}$, $\neg P(n)$ is true, we know that X is not empty. Since X is a non-empty set of natural numbers, it has a smallest element. Let x be the smallest element of X. That is, x is the smallest natural number such that $P(x)$ is false. Since we know that $P(0)$ is true, x cannot be 0. Let $y = x - 1$. Since $x \neq 0$, y is a natural number. Since $y < x$, we know, by the definition of x, that $P(y)$ is true. We also know that $\forall k \in \mathbb{N}\,(P(k) \rightarrow P(k+1))$ is true. In particular, taking $k = y$, we know that $P(y) \rightarrow P(y+1)$. Since $P(y)$ and $P(y) \rightarrow P(y+1)$, we deduce by *modus ponens* that $P(y+1)$ is true. But $y + 1 = x$, so we have deduced that $P(x)$ is true. This contradicts the fact that $P(x)$ is false. This contradiction proves the theorem. □

Theorem 4.4. *Let P be a one-place predicate whose domain of discourse includes the natural numbers. Suppose that $P(0)$ is true and that*

$$(P(0) \wedge P(1) \wedge \cdots \wedge P(k)) \rightarrow P(k+1)$$

is true for each natural number $k \geq 0$. Then it is true that $\forall n \in \mathbb{N}$, $P(n)$.

Proof. Suppose that P is a predicate that satisfies the hypotheses of the theorem, and suppose that the statement $\forall n \in \mathbb{N}$, $P(n)$ is false. We show that this assumption leads to a contradiction.

Let $X = \{n \in \mathbb{N} \mid \neg P(n)\}$. Because of the assumption that $\forall n \in \mathbb{N}$, $P(n)$ is false, X is non-empty. It follows that X has a smallest element. Let x be the smallest element of X. The assumption that $P(0)$ is true means that $0 \notin X$, so we must have $x > 0$. Since x is the smallest natural number for which $P(x)$ is false, we know that $P(0)$, $P(1)$, ..., and $P(x-1)$ are all true. From this and the fact that $(P(0) \wedge P(1) \wedge \cdots \wedge P(x-1)) \rightarrow P(x)$, we deduce that $P(x)$ is true. But this contradicts the fact that $P(x)$ is false. This contradiction proves the theorem. □

4.1.7 Structural Induction

Next, while we are on the topic of induction, let's generalise the idea of induction to also apply it to sets. This more general form of induction is often called ***structural induction***.

Structural induction is used to prove that some proposition $P(x)$ holds for all x of some sort of recursively defined structure, such as formulae, lists, or trees—or recursively-defined sets. In a proof by structural induction we show that the proposition holds for all the 'minimal' structures, and that if it holds for the immediate substructures of a certain structure S, then it must hold for S also. Structural induction is useful for proving properties about algorithms; sometimes it is used together with invariants for this purpose.

To get an idea of what a 'recursively defined set' might look like, consider the following definition of the set of natural numbers $\mathbb{N}$.

Basis: $0 \in \mathbb{N}$.

Succession: $x \in \mathbb{N} \rightarrow x + 1 \in \mathbb{N}$.

Exclusivity: No other elements other than those outlined by the rules above are in $\mathbb{N}$.

This definition is similar to one we have seen before, first stating that $0 \in \mathbb{N}$ and then saying that we can add 1 to an element in $\mathbb{N}$ to get another element of $\mathbb{N}$. The final clause is needed to ensure that other items are not part of $\mathbb{N}$. Without it, you and me, as well as π, 'New York City', and 'King Willem-Alexander' might have been in the set. After all there was no reason for those elements not to be in there.

Now compare that recursive definition, with the method for mathematical induction we have seen before:

Base case: Prove that $P(0)$ holds.

Inductive case: Prove that $\forall k \in \mathbb{N}(P(k) \rightarrow P(k+1))$ holds.

Conclusion: $\forall n \in \mathbb{N}P(n))$ holds.

As we can see mathematical induction and this recursive definition show large similarities. The base case of the induction proves the property for the basis of our recursive definition and the inductive step proves the property for the succession rule. In fact, this similarity is no coincidence and we can generalise this method to get to structural induction.

Consider for instance the set *PROP*, which represents all valid formula in propositional logic:

Atoms: $p_i \in PROP$ for all $i \in \mathbb{N}$.

Negation: $x \in PROP \rightarrow \neg x \in PROP$.

Binary connective: $x, y \in PROP \rightarrow (x * y) \in PROP$, s.t. $* \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$.

Exclusivity: Nothing else is in *PROP*.

Using this definition of the set *PROP* we can use structural induction to prove certain claims about *PROP*. For instance we can prove that every formula in *PROP* has equally many left parentheses '(' and right parentheses ')'.

Proof. Let $l(\phi)$ denote the number of left parentheses in a formula ϕ. Similarly let $r(\phi)$ denote the number of right parentheses. Let $P(\phi)$ be the statement that $l(\phi) = r(\phi)$. We need to prove that $\forall \phi \in PROP(P(\phi))$.

Base case: Consider the Atoms rule of the definition of *PROP*: $l(p_i) = 0 = r(p_i)$. Therefore $P(p_i)$ holds.

Inductive case: We want to show that if the statement is true for $x, y \in PROP$ (where x and y are arbitrary formula), then it is true for $\neg x$ and $(x * y)$ for all $* \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$. That is, we must prove the implication $(P(x) \wedge P(y)) \rightarrow (P(\neg x) \wedge P((x * y)))$. So we assume $P(x) \wedge P(y)$, that is, we assume that for both formula x and y: $l(x) = r(x)$ and $l(y) = r(y)$. We want to prove $P(\neg x)$, that is, that for $\neg x$ $l(\neg x) = r(\neg x)$

$$\begin{aligned} l(\neg x) &= l(x) && \text{by the Negation rule of } PROP \\ &= r(x) && \text{by the inductive hypothesis} \\ &= r(\neg x) && \text{by the Negation rule of } PROP \end{aligned}$$

Secondly we prove that $P((x * y))$ holds for all $* \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$:

$$\begin{aligned} l((x * y)) &= 1 + l(x) + l(y) && \text{by the Binary connective rule of } PROP \\ &= 1 + r(x) + r(y) && \text{by the inductive hypothesis} \\ &= r((x * y)) && \text{by the Binary connective rule of } PROP \end{aligned}$$

Altogether, we have shown that $P(p_i)$ holds and that, for all $x, y \in PROP$ and $* \in \{\vee, \wedge, \rightarrow, \leftrightarrow\}$, $(P(x) \wedge P(y)) \rightarrow (P(\neg x) \wedge P((x * y))$ is true. Therefore, by the principle of structural induction, $P(\phi)$ is true for all $\phi \in PROP$, so for all propositional formula the number of left parentheses equals the number of right parentheses. This completes the proof by structural induction. □

Such structural induction proofs can be applied on any recursively defined set of numbers, formulae or even strings (pieces of text) or lists or trees, making this a very powerful generalised proof method.

Exercises

1. If we don't make the assumption that a, b, and c are distinct, then the set denoted by $\{a, b, c\}$ might actually contain either 1, 2, or 3 elements. How many different elements might the set $\{ a, b, \{a\}, \{a, c\}, \{a, b, c\} \}$ contain? Explain your answer.

†**2.** Compute $A \cup B$, $A \cap B$, and $A \setminus B$ for each of the following pairs of sets

a) $A = \{a, b, c\}$, $B = \emptyset$
b) $A = \{1, 2, 3, 4, 5\}$, $B = \{2, 4, 6, 8, 10\}$
c) $A = \{a, b\}$, $B = \{a, b, c, d\}$
d) $A = \{a, b, \{a, b\}\}$, $B = \{\{a\}, \{a, b\}\}$

3. Draw a Venn diagram for each of the four sets of the last exercise.

4. Recall that $\mathbb{N}$ represents the set of natural numbers. That is, $\mathbb{N} = \{0, 1, 2, 3, \dots\}$. Let $X = \{n \in \mathbb{N} \mid n \geq 5\}$, let $Y = \{n \in \mathbb{N} \mid n \leq 10\}$, and let $Z = \{n \in \mathbb{N} \mid n \text{ is an even number}\}$. Find each of the following sets:

a) $X \cap Y$ **b)** $X \cup Y$ **c)** $X \setminus Y$ **d)** $\mathbb{N} \setminus Z$
e) $X \cap Z$ **f)** $Y \cap Z$ **g)** $Y \cup Z$ **h)** $Z \setminus \mathbb{N}$

5. Find $\mathcal{P}(\{1, 2, 3\})$. (Hint: It has eight elements.)

6. Assume that a and b are entities and that $a \neq b$. Let A and B be the sets defined by $A = \{\, a, \{b\}, \{a, b\} \,\}$ and $B = \{\, a, b, \{a, \{b\}\} \,\}$. Determine whether each of the following statements is true or false. Explain your answers.

a) $b \in A$ **b)** $\{a, b\} \subseteq A$ **c)** $\{a, b\} \subseteq B$
d) $\{a, b\} \in B$ **e)** $\{a, \{b\}\} \in A$ **f)** $\{a, \{b\}\} \in B$

7. Since $\mathcal{P}(A)$ is a set, it is possible to form the set $\mathcal{P}(\mathcal{P}(A))$. What is $\mathcal{P}(\mathcal{P}(\emptyset))$? What is $\mathcal{P}(\mathcal{P}(\{a, b\}))$? (Hint: It has sixteen elements.)

8. In the English sentence, "She likes dogs that are small, cuddly, and cute", does she like an intersection or a union of sets of dogs? How about in the sentence, "She likes dogs that are small, dogs that are cuddly, and dogs that are cute"?

†**9.** If A is any set, what can you say about $A \cup A$? About $A \cap A$? About $A \setminus A$? Why?

10. Suppose that A and B are sets such that $A \subseteq B$. What can you say about $A \cup B$? About $A \cap B$? About $A \setminus B$? Why?

11. Suppose that A, B, and C are sets. Show that $C \subseteq A \cap B$ if and only if $(C \subseteq A) \wedge (C \subseteq B)$.

12. Suppose that A, B, and C are sets, and that $A \subseteq B$ and $B \subseteq C$. Show that $A \subseteq C$.

13. Suppose that A and B are sets such that $A \subseteq B$. Is it necessarily true that $\mathcal{P}(A) \subseteq \mathcal{P}(B)$? Why or why not?

14. Let M be any natural number, and let $P(n)$ be a predicate whose domain of discourse includes all natural numbers greater than or equal to M. Suppose that $P(M)$ is true, and suppose that $P(k) \rightarrow P(k + 1)$ for all $k \geq M$. Show that $P(n)$ is true for all $n \geq M$.

15. Prove that the number of propositional variables is always at most one more than the number of connectives for every formula $\phi \in PROP$.

4.2 The Boolean Algebra of Sets

It is clear that set theory is closely related to logic. The intersection and union of sets can be defined in terms of the logical 'and' and logical 'or' operators. The notation $\{x \mid P(x)\}$ makes it possible to use predicates to specify sets. And if A is any set, then the formula $x \in A$ defines a one place predicate that is true for an entity x if and only if x is a member of A. So it should not be a surprise that many of the rules of logic have analogues in set theory.

For example, we have already noted that $\cup$ and $\cap$ are commutative operations. This fact can be verified using the rules of logic. Let A and B be sets. According to the definition of equality of sets, we can show that $A \cup B = B \cup A$ by showing that $\forall x \big((x \in$

$A \cup B) \leftrightarrow (x \in B \cup A)\big)$. But for any x,

$$\begin{aligned} x \in A \cup B &\leftrightarrow x \in A \vee x \in B && \text{(definition of } \cup) \\ &\leftrightarrow x \in B \vee x \in A && \text{(commutativity of } \vee) \\ &\leftrightarrow x \in B \cup A && \text{(definition of } \cup) \end{aligned}$$

The commutativity of $\cap$ follows in the same way from the definition of $\cap$ in terms of $\wedge$ and the commutativity of $\wedge$, and a similar argument shows that union and intersection are associative operations.

The distributive laws for propositional logic give rise to two similar rules in set theory. Let A, B, and C be any sets. Then

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

and

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

These rules are called the *distributive laws* for set theory. To verify the first of these laws, we just have to note that for any x,

$$\begin{aligned} & x \in A \cup (B \cap C) \\ & \quad \leftrightarrow (x \in A) \vee ((x \in B) \wedge (x \in C)) && \text{(definition of } \cup, \cap) \\ & \quad \leftrightarrow ((x \in A) \vee (x \in B)) \wedge ((x \in A) \vee (x \in C)) && \text{(distributivity of } \vee) \\ & \quad \leftrightarrow (x \in A \cup B) \wedge (x \in A \cup C) && \text{(definition of } \cup) \\ & \quad \leftrightarrow x \in ((A \cup B) \cap (A \cup C)) && \text{(definition of } \cap) \end{aligned}$$

The second distributive law for sets follows in exactly the same way.

4.2.1 Set complement

While $\cup$ is analogous to $\vee$ and $\cap$ is analogous to $\wedge$, we have not yet seen any operation in set theory that is analogous to the logical 'not' operator, $\neg$. Given a set A, it is tempting to try to define $\{x \mid \neg(x \in A)\}$, the set that contains everything that does not belong to A. Unfortunately, the rules of set theory do not allow us to define such a set. The notation $\{x \mid P(x)\}$ can only be used when the domain of discourse of P is a set, so there must be an underlying set from which the elements that are/are not in A are chosen, i.e., some underlying set of which A is a subset. We can get around this problem by restricting the discussion to subsets of some fixed set. This set will be known as the ***universal set***. Keep in mind that the universal set is only universal for some particular discussion. It is simply some set that is large enough to contain all the sets under discussion as subsets. Given a universal set U and any subset A of U, we can define the set $\{x \in U \mid \neg(x \in A)\}$.

Definition 4.1. Let U be a given universal set, and let A be any subset of U. We define the ***complement*** of A in U to be the set $\overline{A}$ that is defined by $\overline{A} = \{x \in U \mid x \notin A\}$.

Usually, we will refer to the complement of A in U simply as the complement of A, but you should remember that whenever complements of sets are used, there must be some universal set in the background. Other textbooks may use A^c to denote the complement of A instead.

Given the complement operation on sets, we can look for analogues to the rules of logic that involve negation. For example, we know that $p \wedge \neg p = \mathbb{F}$ for any proposition p. It follows that for any subset A of U,

$$\begin{aligned} A \cap \overline{A} &= \{x \in U \mid (x \in A) \wedge (x \in \overline{A})\} && \text{(definition of } \cap) \\ &= \{x \in U \mid (x \in A) \wedge (x \notin A)\} && \text{(definition of complement)} \\ &= \{x \in U \mid (x \in A) \wedge \neg(x \in A)\} && \text{(definition of } \notin) \\ &= \emptyset \end{aligned}$$

the last equality following because the proposition $(x \in A) \wedge \neg(x \in A)$ is false for any x. Similarly, we can show that $A \cup \overline{A} = U$ and that $\overline{\overline{A}} = A$ (where $\overline{\overline{A}}$ is the complement of the complement of A, that is, the set obtained by taking the complement of $\overline{A}$.)

The most important laws for working with complements of sets are DeMorgan's Laws for sets. These laws, which follow directly from DeMorgan's Laws for logic, state that for any subsets A and B of a universal set U,

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$

and

$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

For example, we can verify the first of these laws with the calculation

$$\begin{aligned} \overline{A \cup B} &= \{x \in U \mid x \notin (A \cup B)\} && \text{(definition of complement)} \\ &= \{x \in U \mid \neg(x \in A \cup B)\} && \text{(definition of } \notin) \\ &= \{x \in U \mid \neg(x \in A \vee x \in B)\} && \text{(definition of } \cup) \\ &= \{x \in U \mid (\neg(x \in A)) \wedge (\neg(x \in B))\} && \text{(DeMorgan's Law for logic)} \\ &= \{x \in U \mid (x \notin A) \wedge (x \notin B)\} && \text{(definition of } \notin) \\ &= \{x \in U \mid (x \in \overline{A}) \wedge (x \in \overline{B})\} && \text{(definition of complement)} \\ &= \overline{A} \cap \overline{B} && \text{(definition of } \cap) \end{aligned}$$

Double complement	$\overline{\overline{A}} = A$
Miscellaneous laws	$A \cup \overline{A} = U$ $A \cap \overline{A} = \emptyset$ $\emptyset \cup A = A$ $\emptyset \cap A = \emptyset$
Idempotent laws	$A \cap A = A$ $A \cup A = A$
Commutative laws	$A \cap B = B \cap A$ $A \cup B = B \cup A$
Associative laws	$A \cap (B \cap C) = (A \cap B) \cap C$ $A \cup (B \cup C) = (A \cup B) \cup C$
Distributive laws	$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$ $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$
DeMorgan's laws	$\overline{A \cap B} = \overline{A} \cup \overline{B}$ $\overline{A \cup B} = \overline{A} \cap \overline{B}$

Figure 4.4: *Some Laws of Boolean Algebra for sets. A, B, and C are sets. For the laws that involve the complement operator, they are assumed to be subsets of some universal set, U. For the most part, these laws correspond directly to laws of Boolean Algebra for propositional logic as given in Figure 2.2.*

An easy inductive proof can be used to verify generalized versions of DeMorgan's Laws for set theory. (In this context, all sets are assumed to be subsets of some unnamed universal set.) A simple calculation verifies DeMorgan's Law for three sets:

$$\begin{aligned}
\overline{A \cup B \cup C} &= \overline{(A \cup B) \cup C} \\
&= \overline{(A \cup B)} \cap \overline{C} && \text{(by DeMorgan's Law for two sets)} \\
&= (\overline{A} \cap \overline{B}) \cap \overline{C} && \text{(by DeMorgan's Law for two sets)} \\
&= \overline{A} \cap \overline{B} \cap \overline{C}
\end{aligned}$$

From there, we can derive similar laws for four sets, five sets, and so on. However, just saying 'and so on' is not a rigorous proof of this fact. Whereas we may have excused ourselves about that in Chapter 2, we can now prove this fact. Here is a rigorous inductive proof of a generalized DeMorgan's Law:

Theorem 4.5. *For any natural number $n \geq 2$ and for any sets X_1, X_2, ..., X_n,*

$$\overline{X_1 \cup X_2 \cup \cdots \cup X_n} = \overline{X_1} \cap \overline{X_2} \cap \cdots \cap \overline{X_2}$$

Proof. We give a proof by induction. In the base case, $n = 2$, the statement is that $\overline{X_1 \cup X_2} = \overline{X_1} \cap \overline{X_n}$. This is true since it is just an application of DeMorgan's law for two sets.

For the inductive case, suppose that the statement is true for $n = k$. We want to show that it is true for $n = k + 1$. Let X_1, X_2, ..., X_{k+1} be any $k + 1$ sets. Then we have:

$$\begin{aligned}
\overline{X_1 \cup X_2 \cup \cdots \cup X_{k+1}} &= \overline{(X_1 \cup X_2 \cup \cdots \cup X_k) \cup X_{k+1}} \\
&= \overline{(X_1 \cup X_2 \cup \cdots \cup X_k)} \cap \overline{X_{k+1}} \\
&= (\overline{X_1} \cap \overline{X_2} \cap \cdots \cap \overline{X_k}) \cap \overline{X_{k+1}} \\
&= \overline{X_1} \cap \overline{X_2} \cap \cdots \cap \overline{X_{k+1}}
\end{aligned}$$

In this computation, the second step follows by DeMorgan's Law for two sets, while the third step follows from the induction hypothesis. Therefore by the principle of induction we have proven the theorem. □

4.2.2 Link between logic and set theory

Just as the laws of logic allow us to do algebra with logical formulas, the laws of set theory allow us to do algebra with sets. Because of the close relationship between logic and set theory, their algebras are very similar. The algebra of sets, like the algebra of logic, is Boolean algebra. When George Boole wrote his 1854 book about logic, it was really as much about set theory as logic. In fact, Boole did not make a clear distinction between a predicate and the set of objects for which that predicate is true. His algebraic laws and formulas apply equally to both cases. More exactly, if we consider only subsets of some given universal set U, then there is a direct correspondence between the basic symbols and operations of propositional logic and certain symbols and operations in set theory, as shown in this table:

Logic	Set Theory
$\mathbb{T}$	U
$\mathbb{F}$	$\emptyset$
$p \wedge q$	$A \cap B$
$p \vee q$	$A \cup B$
$\neg p$	$\overline{A}$

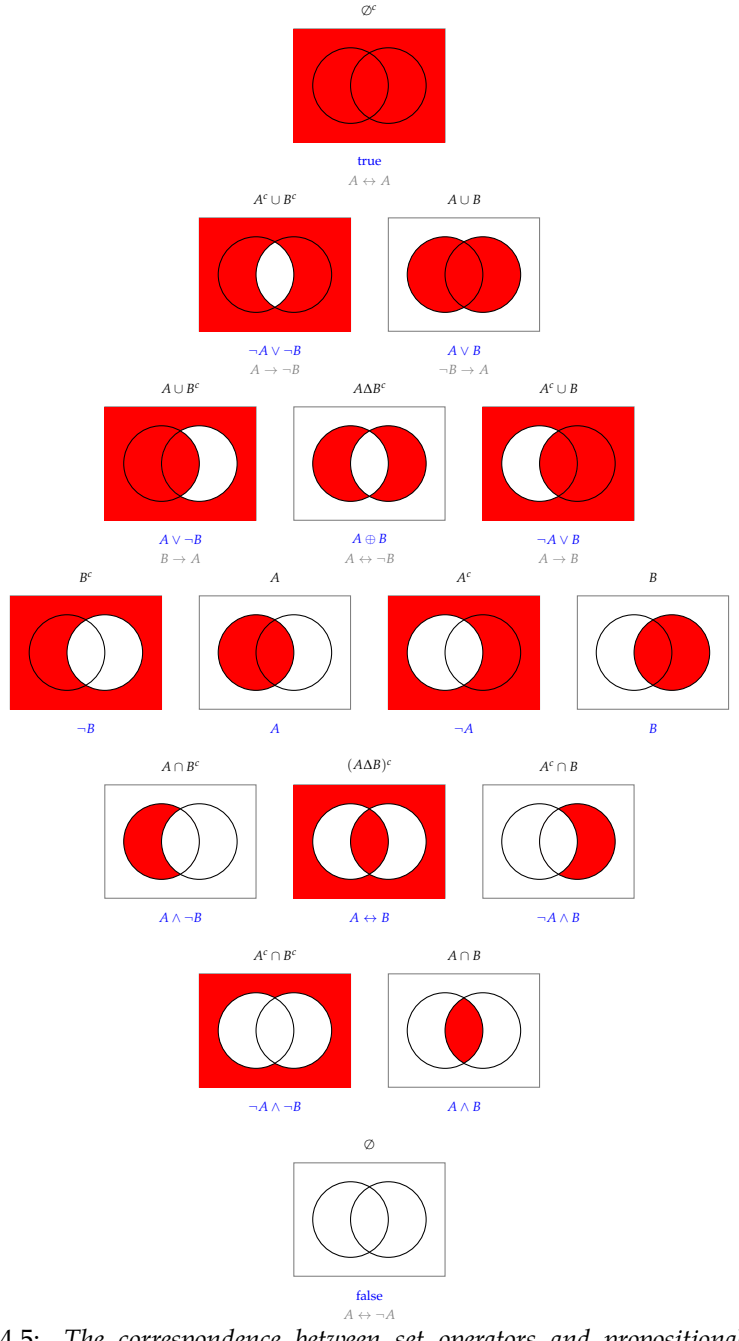


Figure 4.5: *The correspondence between set operators and propositional operators. Recreation of:* `commons.wikimedia.org/wiki/Template:Operations_and_relations_in_set_theory_and_logic`

Any valid logical formula or computation involving propositional variables and the symbols $\mathbb{T}$, $\mathbb{F}$, $\wedge$, $\vee$, and $\neg$ can be transformed into a valid formula or computation in set theory by replacing the propositions in the formula with subsets of U and replacing the logical symbols with U, $\emptyset$, $\cap$, $\cup$, and the complement operator. Figure 4.5 illustrates.

Just as in logic, the operations of set theory can be combined to form complex expressions such as $\overline{(A \cup C) \cap (B \cup \overline{C} \cup D)}$. Parentheses can always be used in such expressions to specify the order in which the operations are to be performed. In the absence of parentheses, we need precedence rules to determine the order of operation. The precedence rules for the Boolean algebra of sets are carried over directly from the Boolean algebra of propositions. When union and intersection are used together without parentheses, intersection has precedence over union. Furthermore, when several operators of the same type are used without parentheses, then they are evaluated in order from left to right. (Of course, since $\cup$ and $\cap$ are both associative operations, it really doesn't matter whether the order of evaluation is left-to-right or right-to-left.) For example, $A \cup B \cap C \cup D$ is evaluated as $(A \cup (B \cap C)) \cup D$. The complement operation is a special case. Since it is denoted by drawing a line over its operand, there is never any ambiguity about which part of a formula it applies to.

Unfortunately in hand-written work this is not always true. Make sure to write neatly and unambiguously when working with complements. Also note that, similarly to parentheses in propositional logic, although parentheses may not be necessary I strongly encourage you to add them them to improve readability of the order of operations.

The laws of set theory can be used to simplify complex expressions involving sets. (As usual, of course, the meaning of 'simplification' is partly in the eye of the beholder.) For example, for any sets X and Y,

$$\begin{aligned} (X \cup Y) \cap (Y \cup X) &= (X \cup Y) \cap (X \cup Y) && \text{(Commutative Law)} \\ &= (X \cup Y) && \text{(Idempotent Law)} \end{aligned}$$

where in the second step, the Idempotent Law, which says that $A \cap A = A$, is applied with $A = X \cup Y$. For expressions that use the complement operation, it is usually considered to be simpler to apply the operation to an individual set, as in $\overline{A}$, rather than to a formula, as in $\overline{A \cap B}$. DeMorgan's Laws can always be used to simplify an expression

in which the complement operation is applied to a formula. For example,

$$\begin{aligned} A \cap \overline{B \cup \overline{A}} &= A \cap (\overline{B} \cap \overline{\overline{A}}) && \text{(DeMorgan's Law)} \\ &= A \cap (\overline{B} \cap A) && \text{(Double Complement)} \\ &= A \cap (A \cap \overline{B}) && \text{(Commutative Law)} \\ &= (A \cap A) \cap \overline{B}) && \text{(Associative Law)} \\ &= A \cap \overline{B} && \text{(Idempotent Law)} \end{aligned}$$

As a final example of the relationship between set theory and logic, consider the set-theoretical expression $A \cap (A \cup B)$ and the corresponding compound proposition $p \wedge (p \vee q)$. (These correspond since for any x, $x \in A \cap (A \cup B) \equiv (x \in A) \wedge ((x \in A) \vee (x \in B))$.) You might find it intuitively clear that $A \cap (A \cup B) = A$. Formally, this follows from the fact that $p \wedge (p \vee q) \equiv p$, which might be less intuitively clear and is surprising difficult to prove algebraically from the laws of logic. However, there is another way to check that a logical equivalence is valid: Make a truth table. Consider a truth table for $p \wedge (p \vee q)$:

p	q	$p \vee q$	$p \wedge (p \vee q)$
0	0	0	0
0	1	1	0
1	0	1	1
1	1	1	1

The fact that the first column and the last column of this table are identical shows that $p \wedge (p \vee q) \equiv p$. Taking p to be the proposition $x \in A$ and q to be the proposition $x \in B$, it follows that the sets A and $A \cap (A \cup B)$ have the same members and therefore are equal.

In one of the pencasts of the course, I describe how Venn diagrams can also be used as alternatives to truth tables based on an old exam question. You can find that pencast here: `youtu.be/5xle5qfrh0k`.

Exercises

1. Use the laws of logic to verify the associative laws for union and intersection. That is, show that if A, B, and C are sets, then $A \cup (B \cup C) = (A \cup B) \cup C$ and $A \cap (B \cap C) = (A \cap B) \cap C$.

2. Show that for any sets A and B, $A \subseteq A \cup B$ and $A \cap B \subseteq A$.

3. Recall that the symbol $\oplus$ denotes the logical exclusive or operation. If A and B sets, define the set $A \triangle B$ by $A \triangle B = \{x \mid (x \in A) \oplus (x \in B)\}$. Show that $A \triangle B = (A \setminus B) \cup (B \setminus A)$. ($A \triangle B$ is known as the ***symmetric difference*** of A and B.)

4. Choose three non-empty sets A, B, C. Draw a Venn diagram of $A \triangle B \triangle C$.

5. Let A be a subset of some given universal set U. Verify that $\overline{\overline{A}} = A$ and that $A \cup \overline{A} = U$.

6. Verify the second of DeMorgan's Laws for sets, $\overline{A \cap B} = \overline{A} \cup \overline{B}$. For each step in your verification, state why that step is valid.

7. The subset operator, $\subseteq$, is defined in terms of the logical implication operator, $\rightarrow$. However, $\subseteq$ differs from the $\cap$ and $\cup$ operators in that $A \cap B$ and $A \cup B$ are *sets*, while $A \subseteq B$ is a *statement*. So the relationship between $\subseteq$ and $\rightarrow$ isn't quite the same as the relationship between $\cup$ and $\vee$ or between $\cap$ and $\wedge$. Nevertheless, $\subseteq$ and $\rightarrow$ do share some similar properties. This problem shows one example.

a) Show that the following three compound propositions are logically equivalent: $p \rightarrow q$, $(p \wedge q) \leftrightarrow p$, and $(p \vee q) \leftrightarrow q$.

b) Show that for any sets A and B, the following three statements are equivalent: $A \subseteq B$, $A \cap B = A$, and $A \cup B = B$.

8. DeMorgan's Laws apply to subsets of some given universal set U. Show that for a subset X of U, $\overline{X} = U \setminus X$. It follows that DeMorgan's Laws can be written as $U \setminus (A \cup B) = (U \setminus A) \cap (U \setminus B)$ and $U \setminus (A \cap B) = (U \setminus A) \cup (U \setminus B)$. Show that these laws hold whether or not A and B are subsets of U. That is, show that for any sets A, B, and C, $C \setminus (A \cup B) = (C \setminus A) \cap (C \setminus B)$ and $C \setminus (A \cap B) = (C \setminus A) \cup (C \setminus B)$.

9. Show that $A \cup (A \cap B) = A$ for any sets A and B.

10. Let X and Y be sets. Simplify each of the following expressions. Justify each step in the simplification with one of the rules of set theory.

a) $X \cup (Y \cup X)$ **b)** $(X \cap Y) \cap \overline{X}$

c) $(X \cup Y) \cap \overline{Y}$ **d)** $(X \cup Y) \cup (X \cap Y)$

11. Let A, B, and C be sets. Simplify each of the following expressions. In your answer, the complement operator should only be applied to the individual sets A, B, and C.

a) $\overline{A \cup B \cup C}$ **b)** $\overline{A \cup B \cap C}$ **c)** $\overline{\overline{A \cup B}}$

d) $\overline{B \cap \overline{C}}$ **e)** $\overline{A \cap \overline{B \cap \overline{C}}}$ **f)** $A \cap \overline{A \cup B}$

12. Use induction to prove the following generalized DeMorgan's Law for set theory: For any natural number $n \geq 2$ and for any sets X_1, X_2, ..., X_n,

$$\overline{X_1 \cap X_2 \cap \cdots \cap X_n} = \overline{X_1} \cup \overline{X_2} \cup \cdots \cup \overline{X_n}$$

13. State and prove generalized distributive laws for set theory.

4.3 Application: Programming with Sets

On a computer, all pieces of data are represented, ultimately, as strings of zeros and ones. At times, computers need to work with sets. How can sets be represented as strings of

zeros and ones? And once we have represented sets on a computer, how do we program with them?

In this section we go into detail about representing and computing with sets. You won't be examined on this in *Reasoning & Logic*. You will find that there is again quite some overlap with your study materials from *Computer Organisation* however, as we discuss different numeric systems.

4.3.1 Representing sets

A set is determined by its elements. Given a set A and an entity x, the fundamental question is, does x belong to A or not? If we know the answer to this question for each possible x, then we know the set. For a given x, the answer to the question, "Is x a member of A", is either *yes* or *no*. The answer can be encoded by letting 1 stand for yes and 0 stand for no. The answer, then, is a single ***bit***, that is, a value that can be either zero or one. To represent the set A as a string of zeros and ones, we could use one bit for each possible member of A. If a possible member x is in the set, then the corresponding bit has the value one. If x is not in the set, then the corresponding bit has the value zero.

Now, in cases where the number of possible elements of the set is very large or infinite, it is not practical to represent the set in this way. It would require too many bits, perhaps an infinite number. In such cases, some other representation for the set can be used. However, suppose we are only interested in subsets of some specified small set. Since this set plays the role of a universal set, let's call it U. To represent a subset of U, we just need one bit for each member of U. If the number of members of U is n, then a subset of U is represented by a string of n zeros and ones. Furthermore, every string of n zeros and ones determines a subset of U, namely that subset that contains exactly the elements of U that correspond to ones in the string. You know by now from *Computer Organisation* that a string of n zeros and ones is called an n-bit ***binary number***. So, we see that if U is a set with n elements, then the subsets of U correspond to n-bit binary numbers.

To make things more definite, let U be the set $\{0, 1, 2, \ldots, 31\}$. This set consists of the 32 integers between 0 and 31, inclusive. Then each subset of U can be represented by a 32-bit binary number. We use 32 bits because most computer languages can work directly with 32-bit numbers. For example, the programming languages Java has a data type named *int*. A value of type *int* is a 32-bit binary number.[2] Before we get a definite correspondence between subsets of U and 32-bit numbers, we have to decide which bit in the number will correspond to each member of U. Following tradition, we assume that the bits are numbered from right to left. That is, the rightmost bit corresponds to

[2]If in a future version of Java, a value of type *int* is, for instance, a 64-bit number—which can be used to represent a subset of the set $\{0, 1, 2, \ldots, 63\}$—the principle remains the same.

Hex.	Binary	Hex.	Binary
0	0000_2	8	1000_2
1	0001_2	9	1001_2
2	0010_2	A	1010_2
3	0011_2	B	1011_2
4	0100_2	C	1100_2
5	0101_2	D	1101_2
6	0110_2	E	1110_2
7	0111_2	F	1111_2

Figure 4.6: *The 16 hexadecimal digits and the corresponding binary numbers. Each hexadecimal digit corresponds to a 4-bit binary number. Longer binary numbers can be written using two or more hexadecimal digits. For example,* $101000011111_2 = 0xA1F$.

the element 0 in U, the second bit from the right corresponds to 1, the third bit from the right to 2, and so on. For example, the 32-bit number

$$10000000000000000000001001110110$$

corresponds to the subset $\{1, 2, 4, 5, 6, 9, 31\}$. Since the leftmost bit of the number is 1, the number 31 is in the set; since the next bit is 0, the number 30 is not in the set; and so on.

From now on, I will write binary numbers with a subscript of 2 to avoid confusion with ordinary numbers. Furthermore, I will often leave out leading zeros. For example, 1101_2 is the binary number that would be written out in full as

$$00000000000000000000000000001101$$

and which corresponds to the set $\{0, 2, 3\}$. On the other hand 1101 represents the ordinary number one thousand one hundred and one.

Even with this notation, it can be very annoying to write out long binary numbers—and almost impossible to read them. So binary numbers are never written out as sequences of zeros and ones in computer programs. An alternative is to use ***hexadecimal numbers***. Hexadecimal numbers are written using the sixteen symbols 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F. These symbols are knows as the hexadecimal digits. Each hexadecimal digit corresponds to a 4-bit binary number, as shown in Figure 4.6. To represent a longer binary number, several hexadecimal digits can be strung together. For example, the hexadecimal number C7 represents the binary number 11000111_2. In Java and many

related languages, a hexadecimal number is written with the prefix `0x`. Thus, the hexadecimal number C7 would appear in the program as 0xC7. I will follow the same convention here. Any 32-bit binary number can be written using eight hexadecimal digits (or fewer if leading zeros are omitted). Thus, subsets of $\{0, 1, 2, \ldots, 31\}$ correspond to 8-digit hexadecimal numbers. For example, the subset $\{1, 2, 4, 5, 6, 9, 31\}$ corresponds to 0x80000276, which represents the binary number $10000000000000000000001001110110_2$. Similarly, 0xFF corresponds to $\{0, 1, 2, 3, 4, 5, 6, 7\}$ and 0x1101 corresponds to the binary number 0001000100000001_2 and to the set $\{0, 8, 12\}$.

Now, if you have worked with binary numbers or with hexadecimal numbers, you know that they have another, more common interpretation. They represent ordinary integers. Just as 342 represents the integer $3 \cdot 10^2 + 4 \cdot 10^1 + 2 \cdot 10^0$, the binary number 1101_2 represents the integer $1 \cdot 2^3 + 1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0$, or 13. When used in this way, binary numbers are known as ***base-2 numbers***, just as ordinary numbers are base-10 numbers. Hexadecimal numbers can be interpreted as base-16 numbers. For example, 0x3C7 represents the integer $3 \cdot 16^2 + 12 \cdot 16^1 + 7 \cdot 16^0$, or 874. So, does 1101_2 really represent the integer 13, or does it represent the set $\{0, 2, 3\}$? The answer is that to a person, 1101_2 can represent either. Both are valid interpretations, and the only real question is which interpretation is useful in a given circumstance. On the other hand, to the computer, 1101_2 doesn't represent *anything*. It's just a string of bits, and the computer manipulates the bits according to its program, without regard to their interpretation.

Of course, we still have to answer the question of whether it is ever useful to interpret strings of bits in a computer as representing sets.

4.3.2 Computing with sets

If all we could do with sets were to 'represent' them, it wouldn't be very useful. We need to be able to compute with sets. That is, we need to be able to perform set operations such as union and complement.

Many programming languages provide operators that perform set operations. In Java and related languages, the operators that perform union, intersection, and complement are written as `|`, `&`, and `~`. For example, if x and y are 32-bit integers representing two subsets, X and Y, of $\{0, 1, 2, \ldots, 31\}$, then $x \,|\, y$ is a 32-bit integer that represents the set $X \cup Y$. Similarly, $x \,\&\, y$ represents the set $X \cap Y$, and $\sim x$ represents the complement, $\overline{X}$.

The operators `|`, `&`, and `~` are called ***bitwise logical operators*** because of the way they operate on the individual bits of the numbers to which they are applied. If 0 and 1 are interpreted as the logical values *false* and *true*, then the bitwise logical operators perform the logical operations $\vee$, $\wedge$, and $\neg$ on individual bits. To see why this is true, let's look at the computations that these operators have to perform.

Let k be one of the members of $\{0, 1, 2, \ldots, 31\}$. In the binary numbers x, y, $x \,|\, y$, $x \,\&\, y$, and $\sim x$, the number k corresponds to the bit in position k. That is, k is in the set represented by a binary number if and only if the bit in position k in that binary number

is 1. Considered as sets, $x \,\&\, y$ is the intersection of x and y, so k is a member of the set represented by $x \,\&\, y$ if and only if k is a member of both of the sets represented by x and y. That is, bit k is 1 in the binary number $x \,\&\, y$ if and only if bit k is 1 in x and bit k is 1 in y. When we interpret 1 as *true* and 0 as *false*, we see that bit k of $x \,\&\, y$ is computed by applying the logical 'and' operator, $\wedge$, to bit k of x and bit k of y. Similarly, bit k of $x \mid y$ is computed by applying the logical 'or' operator, $\vee$, to bit k of x and bit k of y. And bit k of $\sim x$ is computed by applying the logical 'not' operator, $\neg$, to bit k of x. In each case, the logical operator is applied to each bit position separately. (Of course, this discussion is just a translation to the language of bits of the definitions of the set operations in terms of logical operators: $A \cap B = \{x \mid x \in A \wedge x \in B\}$, $A \cup B = \{x \mid x \in A \vee x \in B\}$, and $\overline{A} = \{x \in U \mid \neg(x \in A)\}$.)

For example, consider the binary numbers 1011010_2 and 10111_2, which represent the sets $\{1,3,4,6\}$ and $\{0,1,2,4\}$. Then $1011010_2 \,\&\, 10111_2$ is 10010_2. This binary number represents the set $\{1,4\}$, which is the intersection $\{1,3,4,6\} \cap \{0,1,2,4\}$. It's easier to see what's going on if we write out the computation in columns, the way you probably first learned to do addition:

$$\begin{array}{rr} & 1\,0\,1\,1\,0\,1\,0 \\ \& & 0\,0\,1\,0\,1\,1\,1 \\ \hline & 0\,0\,1\,0\,0\,1\,0 \end{array} \qquad \begin{array}{c} \{\ 6,\ 4,\ 3,\ 1\ \} \\ \{\ 4,\ 2,\ 1,\ 0\ \} \\ \hline \{\ 4,\ 1\ \} \end{array}$$

Note that in each column in the binary numbers, the bit in the bottom row is computed as the logical 'and' of the two bits that lie above it in the column. I've written out the sets that correspond to the binary numbers to show how the bits in the numbers correspond to the presence or absence of elements in the sets. Similarly, we can see how the union of two sets is computed as a bitwise 'or' of the corresponding binary numbers.

$$\begin{array}{rr} & 1\,0\,1\,1\,0\,1\,0 \\ \mid & 0\,0\,1\,0\,1\,1\,1 \\ \hline & 1\,0\,1\,1\,1\,1\,1 \end{array} \qquad \begin{array}{c} \{\ 6,\ 4,\ 3,\ 1\ \} \\ \{\ 4,\ 2,\ 1,\ 0\ \} \\ \hline \{\ 6,\ 4,\ 3,\ 2,\ 1,\ 0\ \} \end{array}$$

The complement of a set is computed using a bitwise 'not' operation. Since we are working with 32-bit binary numbers, the complement is taken with respect to the universal set $\{0,1,2,\ldots,31\}$. So, for example,

$$\sim 1011010_2 = 11111111111111111111111110100101_2$$

Of course, we can apply the operators &, $\mid$, and $\sim$ to numbers written in hexadecimal form, or even in ordinary, base-10 form. When doing such calculations by hand, it is probably best to translate the numbers into binary form. For example,

$$\begin{aligned} 0\text{x}AB7 \,\&\, 0\text{x}168E &= 1010\,1011\,0111_2 \,\&\, 1\,0110\,1000\,1110_2 \\ &= 0\,0010\,1000\,0110_2 \\ &= 0\text{x}286 \end{aligned}$$

When computing with sets, it is sometimes necessary to work with individual elements. Typical operations include adding an element to a set, removing an element from a set, and testing whether an element is in a set. However, instead of working with an element itself, it's convenient to work with the set that contains that element as its only member. For example, testing whether $5 \in A$ is the same as testing whether $\{5\} \cap A \neq \emptyset$. The set $\{5\}$ is represented by the binary number 100000_2 or by the hexadecimal number 0x20. Suppose that the set A is represented by the number x. Then, testing whether $5 \in A$ is equivalent to testing whether 0x20 & $x \neq 0$. Similarly, the set $A \cup \{5\}$, which is obtained by adding 5 to A, can be computed as x | 0x20. The set $A \setminus \{5\}$, which is the set obtained by removing 5 from A if it occurs in A, is represented by x & ~0x20.

The sets $\{0\}, \{1\}, \{2\}, \{3\}, \{4\}, \{5\}, \{6\}, ..., \{31\}$ are represented by the hexadecimal numbers 0x1, 0x2, 0x4, 0x8, 0x10, 0x20, ..., 0x80000000. In typical computer applications, some of these numbers are given names, and these names are thought of as names for the possible elements of a set (although, properly speaking, they are names for sets containing those elements). Suppose, for example, that a, b, c, and d are names for four of the numbers from the above list. Then $a \mid c$ is the set that contains the two elements corresponding to the numbers a and c. If x is a set, then x & ~d is the set obtained by removing d from x. And we can test whether b is in x by testing if x & $b \neq 0$.

Here is an actual example, which is used in the Apple Mac operating systems (macOS). Characters can be printed or displayed on the screen in various sizes and styles. A ***font*** is a collection of pictures of characters in a particular size and style. On the Mac, a basic font can be modified by specifying any of the following style attributes: *bold*, *italic*, *underline*, *outline*, *shadow*, *condense*, and *extend*. The style of a font is a subset of this set of attributes. A style set can be specified by or-ing together individual attributes. For example, an underlined, bold, italic font has style set *underline* | *bold* | *italic*. For a plain font, with none of the style attributes set, the style set is the empty set, which is represented by the number zero.

The Java programming language uses a similar scheme to specify style attributes for fonts, but currently there are only two basic attributes, `Font.BOLD` and `Font.ITALIC`. A more interesting example in Java is provided by event types. An event in Java represents some kind of user action, such as pressing a key on the keyboard. Events are associated with 'components' such as windows, push buttons, and scroll bars. Components can be set to ignore a given type of event. We then say that that event type is disabled for that component. If a component is set to process events of a given type, then that event type is said to be enabled. Each component keeps track of the set of event types that are currently enabled. It will ignore any event whose type is not in that set. Each event type has an associated constant with a name such as `AWTEvent.MOUSE_EVENT_MASK`. These constants represent the possible elements of a set of event types. A set of event types can be specified by or-ing together a number of such constants. If c is a component and x is a number representing a set of event types, then the command '*c.enableEvents*(x)' enables the events in the set x for the component c. If y represents the set of event types

that were already enabled for c, then the effect of this command is to replace y with the union, $y \mid x$. Another command, "$c.disableEvents(x)$", will disable the event types in x for the component c. It does this by replacing the current set, y, with $y \,\&\, \sim x$.

Exercises

1. Suppose that the numbers x and y represent the sets A and B. Show that the set $A \setminus B$ is represented by $x \,\&\, (\sim y)$.

2. Write each of the following binary numbers in hexadecimal:
a) 10110110_2 **b)** 10_2 **c)** 111100001111_2 **d)** 101001_2

3. Write each of the following hexadecimal numbers in binary:
a) 0x123 **b)** 0xFADE **c)** 0x137F **d)** 0xFF11

4. Give the value of each of the following expressions as a hexadecimal number:
a) 0x73 | 0x56A **b)** ~0x3FF0A2FF
c) (0x44 | 0x95) & 0xE7 **d)** 0x5C35A7 & 0xFF00
e) 0x5C35A7 & ~0xFF00 **f)** ~(0x1234 & 0x4321)

5. Find a calculator (or a calculator program on a computer) that can work with hexadecimal numbers. Write a short report explaining how to work with hexadecimal numbers on that calculator. You should explain, in particular, how the calculator can be used to do the previous problem.

6. This question assumes that you know how to add binary numbers. Suppose x and y are binary numbers. Under what circumstances will the binary numbers $x + y$ and $x \mid y$ be the same?

7. In addition to hexadecimal numbers, the programming languages Java, C, and C++ support ***octal numbers***. Look up and report on octal numbers in Java, C, or C++. Explain what octal numbers are, how they are written, and how they are used.

8. In the Unix (and Linux, macOS, ...) operating system, every file has an associated set of permissions, which determine who can use the file and how it can be used. The set of permissions for a given file is represented by a nine-bit binary number. This number is sometimes written as an octal number. Research and report on the Unix systems of permissions. What set of permissions is represented by the octal number 752? by the octal number 622? Explain what is done by the Unix commands `chmod g+rw filename` and `chmod o-w filename` in terms of sets.

9. Most modern programming languages have a boolean data type that has the values *true* and *false*. The usual logical and, or, and not operators on boolean values are represented in Java, C and C++ by the operators &&, ||, and !. C and C++ (but not Java) allow integer values to be used in places where boolean values are expected. In this case, the integer zero represents the boolean value *false* while any non-zero integer represents the boolean value *true*. This means that if x and y are integers, then both $x \,\&\, y$ and $x \,\&\&\, y$ are valid expressions, and both can be considered to represent boolean values. Do the expressions $x \,\&\, y$ and $x \,\&\&\, y$ always represent the same boolean value, for any integers x and y? Do the expressions $x \mid y$ and $x \mid\mid y$ always represent the same boolean values? Explain your answers.

10. Suppose that you, as a programmer, want to write a subroutine that will open a window on the computer's screen. The window can have any of the following options: a close box, a zoom

box, a resize box, a minimize box, a vertical scroll bar, a horizontal scroll bar. Design a scheme whereby the options for the window can be specified by a single parameter to the subroutine. The parameter should represent a set of options. How would you use your subroutine to open a window that has a close box and both scroll bars and no other options? Inside your subroutine, how would you determine which options have been specified for the window?

4.4 Functions

Both the real world and the world of mathematics are full of what are called, in mathematics, 'functional relationships'. A functional relationship is a relationship between two sets, which associates exactly one element from the second set to each element of the first set.

For example, each item for sale in a store has a price. The first set in this relationship is the set of items in the store. For each item in the store, there is an associated price, so the second set in the relationship is the set of possible prices. The relationship is a functional relationship because each item has a price. That is, the question "What is the price of this item?" has a single, definite answer for each item in the store.

Similarly, the question "Who is the (biological) mother of this person?" has a single, definite answer for each person. So, the relationship 'mother of' defines a functional relationship. In this case, the two sets in the relationship are the same set, namely the set of people. On the other hand, the relationship 'child of' is not a functional relationship. The question "Who is the child of this person?" does not have a single, definite answer for each person. A given person might not have any child at all. And a given person might have more than one child. Either of these cases—a person with no child or a person with more than one child—is enough to show that the relationship 'child of' is not a functional relationship.

Or consider an ordinary map, such as a map of Zuid-Holland or a street map of Rome. The whole point of the map, if it is accurate, is that there is a functional relationship between points on the map and points on the surface of the Earth. Perhaps because of this example, a functional relationship is sometimes called a ***mapping***.

There are also many natural examples of functional relationships in mathematics. For example, every rectangle has an associated area. This fact expresses a functional relationship between the set of rectangles and the set of numbers. Every natural number n has a square, n^2. The relationship 'square of' is a functional relationship from the set of natural numbers to itself.

4.4.1 Formalising the notion of functions

In mathematics, of course, we need to work with functional relationships in the abstract. To do this, we introduce the idea of ***function***. You should think of a function as a mathematical object that expresses a functional relationship between two sets. The notation

$f\colon A \to B$ expresses the fact that f is a function from the set A to the set B. That is, f is a name for a mathematical object that expresses a functional relationship from the set A to the set B. The notation $f\colon A \to B$ is read as "f is a function from A to B" or more simply as "f maps A to B".

Mathematical functions are different to functions in a programming language in Java. We'll come back to this in the next section.

If $f\colon A \to B$ and if $a \in A$, the fact that f is a functional relationship from A to B means that f associates some element of B to a. That element is denoted $f(a)$. That is, for each $a \in A$, $f(a) \in B$ and $f(a)$ is the single, definite answer to the question "What element of B is associated to a by the function f?" The fact that f is a function from A to B means that this question has a single, well-defined answer. Given $a \in A$, $f(a)$ is called the ***value*** of the function f at a.

For example, if I is the set of items for sale in a given store and M is the set of possible prices, then there is function $c\colon I \to M$ which is defined by the fact that for each $x \in I$, $c(x)$ is the price of the item x. Similarly, if P is the set of people, then there is a function $m\colon P \to P$ such that for each person p, $m(p)$ is the mother of p. And if $\mathbb{N}$ is the set of natural numbers, then the formula $s(n) = n^2$ specifies a function $s\colon \mathbb{N} \to \mathbb{N}$.

It is in the form of formulas such as $s(n) = n^2$ or $f(x) = x^3 - 3x + 7$ that most people first encounter functions. But you should note that a formula by itself is not a function, although it might well specify a function between two given sets of numbers. Functions are much more general than formulas, and they apply to all kinds of sets, not just to sets of numbers.

4.4.2 Operations on functions

Suppose that $f\colon A \to B$ and $g\colon B \to C$ are functions. Given $a \in A$, there is an associated element $f(a) \in B$. Since g is a function from B to C, and since $f(a) \in B$, g associates some element of C to $f(a)$. That element is $g(f(a))$. Starting with an element a of A, we have produced an associated element $g(f(a))$ of C. This means that we have defined a new function from the set A to the set C. This function is called the ***composition*** of g with f, and it is denoted by $g \circ f$. That is, if $f\colon A \to B$ and $g\colon B \to C$ are functions, then $g \circ f\colon A \to C$ is the function which is defined by

$$(g \circ f)(a) = g(f(a))$$

for each $a \in A$. For example, suppose that p is the function that associates to each item in a store the price of the item, and suppose that t is a function that associates the amount of tax on a price to each possible price. The composition, $t \circ p$, is the function that associates to each item the amount of tax on that item. Or suppose that $s\colon \mathbb{N} \to \mathbb{N}$ and $r\colon \mathbb{N} \to \mathbb{N}$ are the functions defined by the formulas $s(n) = n^2$ and $r(n) = 3n + 1$, for each $n \in \mathbb{N}$. Then $r \circ s$ is a function from $\mathbb{N}$ to $\mathbb{N}$, and for $n \in \mathbb{N}$, $(r \circ s)(n) = r(s(n)) = r(n^2) = 3n^2 + 1$. In this case, we also have the function $s \circ r$, which satisfies $(s \circ r)(n) = s(r(n)) = s(3n + 1) = (3n + 1)^2 = 9n^2 + 6n + 1$. Note in particular that $r \circ s$ and $s \circ r$ are not the same function. The operation $\circ$ is not commutative.

If A is a set and $f\colon A \to A$, then $f \circ f$, the composition of f with itself, is defined. For example, using the function s from the preceding example, $s \circ s$ is the function from $\mathbb{N}$ to $\mathbb{N}$ given by the formula $(s \circ s)(n) = s(s(n)) = s(n^2) = (n^2)^2 = n^4$. If m is the function from the set of people to itself which associates to each person that person's mother, then $m \circ m$ is the function that associates to each person that person's maternal grandmother.

If a and b are entities, then (a, b) denotes the ***ordered pair*** containing a and b. The ordered pair (a, b) differs from the set $\{a, b\}$ because a set is not ordered. That is, $\{a, b\}$ and $\{b, a\}$ denote the same set, but if $a \neq b$, then (a, b) and (b, a) are different ordered pairs. More generally, two ordered pairs (a, b) and (c, d) are equal if and only if both $a = c$ and $b = d$. If (a, b) is an ordered pair, then a and b are referred to as the ***coordinates*** of the ordered pair. In particular, a is the first coordinate and b is the second coordinate.

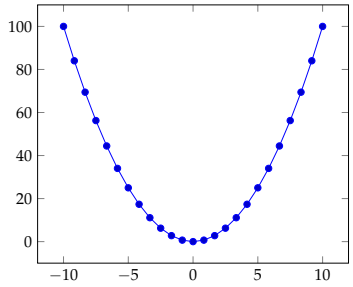
In high school you would also have to write (x,y)-coordinates using this ordered pair notation. For instance you would say that the line $y = ax + b$ intersects the y-axis at $(0, b)$ and the x-axis at $(-\frac{b}{a}, 0)$.

If A and B are sets, then we can form the set $A \times B$ which is defined by

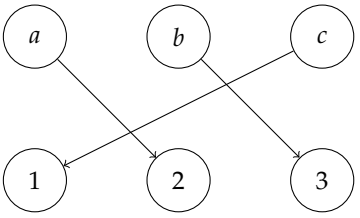
$$A \times B = \{(a, b) \mid a \in A \text{ and } b \in B\}.$$

This set is called the ***cross product*** or ***Cartesian product*** of the sets A and B. The set $A \times B$ contains every ordered pair whose first coordinate is an element of A and whose second coordinate is an element of B. For example, if $X = \{c, d\}$ and $Y = \{1, 2, 3\}$, then $X \times Y = \{(c, 1), (c, 2), (c, 3), (d, 1), (d, 2), (d, 3)\}$. It is possible to extend this idea to the cross product of more than two sets. The cross product of the three sets A, B, and C is denoted $A \times B \times C$. It consists of all ***ordered triples*** (a, b, c) where $a \in A$, $b \in B$, and $c \in C$. The definition for four or more sets is similar. The general term for a member of a cross product is ***tuple*** (recall page 45) or, more specifically, ***ordered n-tuple***. For example, (a, b, c, d, e) is an ordered 5-tuple.

Given a function $f\colon A \to B$, consider the set $\{(a, b) \in A \times B \mid a \in A \text{ and } b = f(a)\}$. This set of ordered pairs consists of all pairs (a, b) such that $a \in A$ and b is the element of



(a) Graph for the formula $f(n) = n^2$ for $-10 \leq n \leq 10$



(b) Graph for the function $g : A \to B$ with $A = \{a, b, c\}$ and $B = \{1, 2, 3\}$, such that $g = \{(a, 2), (b, 3), (c, 1)\}$.

Figure 4.7: Two different graphs representing functions.

B that is associated to a by the function f. The set $\{(a, b) \in A \times B \mid a \in A \text{ and } b = f(a)\}$ is called the ***graph*** of the function f. Since f is a function, each element $a \in A$ occurs once and only once as a first coordinate among the ordered pairs in the graph of f. Given $a \in A$, we can determine $f(a)$ by finding that ordered pair and looking at the second coordinate. In fact, it is convenient to consider the function and its graph to be the same thing, and to use this as our official mathematical definition.[3]

You are already familiar with the process of graphing numerical functions from high school (for instance you can graph the formula $f(n) = n^2$ as depicted in Figure 4.7a), but we can also graph non-numerical functions as indicated in Figure 4.7b. To do so, we draw all elements from the set A and connect them to the elements of B they map to.

Definition 4.2. Let A and B be sets. A ***function*** from A to B is a subset of $A \times B$ which has the property that for each $a \in A$, the set contains one and only one ordered pair whose first coordinate is a. If (a, b) is that ordered pair, then b is called the value of the function at a and is denoted $f(a)$. If $b = f(a)$, then we also say that the function f ***maps*** a to b. The fact that f is a function from A to B is indicated by the notation $f : A \to B$.

For example, if $X = \{a, b\}$ and $Y = \{1, 2, 3\}$, then the set $\{(a, 2), (b, 1)\}$ is a function from X to Y, and $\{(1, a), (2, a), (3, b)\}$ is a function from Y to X. On the other hand, $\{(1, a), (2, b)\}$ is not a function from Y to X, since it does not specify any value for 3. And $\{(a, 1), (a, 2), (b, 3)\}$ is not a function from X to Y because it specifies two different values, 1 and 2, associated with the same element, a, of X.

[3] This is a convenient definition for the mathematical world, but as is often the case in mathematics, it neglects much of the real world. Functional relationships in the real world are *meaningful*, but we model them in mathematics with meaningless sets of ordered pairs. We do this for the usual reason: to have something precise and rigorous enough that we can make logical deductions and prove things about it.

Even though the technical definition of a function is a set of ordered pairs, it's usually better to think of a function from A to B as something that associates some element of B to every element of A. The set of ordered pairs is one way of expressing this association. If the association is expressed in some other way, it's easy to write down the set of ordered pairs. For example, the function $s\colon \mathbb{N} \to \mathbb{N}$ which is specified by the formula $s(n) = n^2$ can be written as the set of ordered pairs $\{(n, n^2) \mid n \in \mathbb{N}\}$.

4.4.3 Properties of functions

Suppose that $f\colon A \to B$ is a function from the set A to the set B. We say that A is the ***domain*** of the function f and that B is the ***range*** of the function. We define the ***image*** of the function f to be the set $\{b \in B \mid \exists a \in A\,(b = f(a))\}$. Put more simply, the image of f is the set $\{f(a) \mid a \in A\}$. That is, the image is the set of all values, $f(a)$, of the function, for all $a \in A$. For example, for the function $s\colon \mathbb{N} \to \mathbb{N}$ that is specified by $s(n) = n^2$, both the domain and the range are $\mathbb{N}$, and the image is the set $\{n^2 \mid n \in \mathbb{N}\}$, or $\{0, 1, 4, 9, 16, \dots\}$.

In some cases—particularly in courses like *Calculus*—the term 'range' is used to refer to what I am calling the image.

Note that the image of a function is a subset of its range. It can be a proper subset, as in the above example, but it is also possible for the image of a function to be equal to the range. In that case, the function is said to be ***onto***. Sometimes, the fancier term ***surjective*** is used instead. Formally, a function $f\colon A \to B$ is said to be onto (or surjective) if every element of B is equal to $f(a)$ for some element of A. In terms of logic, f is onto if and only if

$$\forall b \in B \left(\exists a \in A\,(b = f(a))\right).$$

For example, let $X = \{a, b\}$ and $Y = \{1, 2, 3\}$, and consider the function from Y to X specified by the set of ordered pairs $\{(1, a), (2, a), (3, b)\}$. This function is onto because its image, $\{a, b\}$, is equal to the range, X. However, the function from X to Y given by $\{(a, 1), (b, 3)\}$ is not onto, because its image, $\{1, 3\}$, is a proper subset of its range, Y.

As a further example, consider the function f from $\mathbb{Z}$ to $\mathbb{Z}$ given by $f(n) = n - 52$. To show that f is onto, we need to pick an arbitrary b in the range $\mathbb{Z}$ and show that there is some number a in the domain $\mathbb{Z}$ such that $f(a) = b$. So let b be an arbitrary integer; we want to find an a such that $a - 52 = b$. Clearly this equation will be true when $a = b + 52$. So every element b is the image of the number $a = b + 52$, and f is therefore onto. Note that if f had been specified to have domain $\mathbb{N}$, then f would *not* be onto, as for some $b \in \mathbb{Z}$ the number $a = b + 52$ is not in the domain $\mathbb{N}$ (for example, the integer -73 is not in the image of f, since -21 is not in $\mathbb{N}$.)

If $f\colon A \to B$ and if $a \in A$, then a is associated to only one element of B. This is part of the definition of a function. However, no such restriction holds for elements of B. If $b \in B$, it is possible for b to be associated to zero, one, two, three, ..., or even to an infinite number of elements of A. In the case where each element of the range is associated to at most one element of the domain, the function is said to be ***one-to-one***. Sometimes, the term ***injective*** is used instead. The function f is one-to-one (or injective) if for any two distinct elements x and y in the domain of f, $f(x)$ and $f(y)$ are also distinct. In terms of logic, $f\colon A \to B$ is one-to-one if and only if

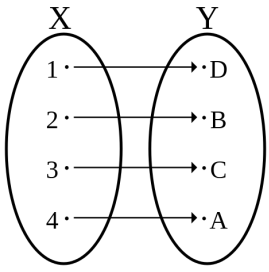
$$\forall x \in A\ \forall y \in A\ \big(x \neq y \to f(x) \neq f(y)\big).$$

Since a proposition is equivalent to its contrapositive, we can write this condition equivalently as

$$\forall x \in A\ \forall y \in A\ \big(f(x) = f(y) \to x = y\big).$$

Sometimes, it is easier to work with the definition of one-to-one when it is expressed in this form.

The function that associates every person to his or her mother is not one-to-one because it is possible for two different people to have the same mother. The function $s\colon \mathbb{N} \to \mathbb{N}$ specified by $s(n) = n^2$ is one-to-one. However, we can define a function $r\colon \mathbb{Z} \to \mathbb{Z}$ by the same formula: $r(n) = n^2$, for $n \in \mathbb{Z}$. The function r is *not* one-to-one since two different integers can have the same square. For example, $r(-2) = r(2)$.



A function that is both one-to-one and onto is said to be ***bijective***.[4] The function that associates each point in a map of Zuid-Holland to a point in the state itself is presumably bijective. For each point on the map, there is a corresponding point in the province, and *vice versa*. If we specify the function f from the set $\{1, 2, 3\}$ to the set $\{a, b, c\}$ as the set of ordered pairs $\{(1, b), (2, a), (3, c)\}$, then f is a bijective function. Or consider the function from $\mathbb{Z}$ to $\mathbb{Z}$ given by $f(n) = n - 52$. We have already shown that f is onto. We can show that it is also one-to-one.

Proof. Pick an arbitrary x and y in $\mathbb{Z}$ and assume that $f(x) = f(y)$. This means that $x - 52 = y - 52$, and adding 52 to both sides of the equation gives $x = y$. Since x and y were arbitrary, we have proved $\forall x \in \mathbb{Z}\ \forall y \in \mathbb{Z}\,(f(x) = f(y) \to x = y)$, that is, that f is one-to-one. □

Altogether, then, f is a bijection.

4.4.4 First-class objects

One difficulty that people sometimes have with mathematics is its generality. A set is a collection of entities, but an 'entity' can be anything at all, including other sets. Once we have defined ordered pairs, we can use ordered pairs as elements of sets. We could also make ordered pairs of sets. Now that we have defined functions, every function is itself an entity. This means that we can have sets that contain functions. We can even have a function whose domain and range are sets of functions. Similarly, the domain or range of a function might be a set of sets, or a set of ordered pairs. Computer scientists have a good name for this. They would say that sets, ordered pairs, and functions are ***first-class objects*** or **first-class citizens**. Once a set, ordered pair, or function has been defined, it can be used just like any other entity. If they were not first-class objects, there could be restrictions on the way they can be used. For example, it might not be possible to use functions as members of sets. (This would make them 'second class.')

One way that programming languages differ is by what they allow as first-class objects. For example, Java added a 'lambda syntax' to help writing 'closures' in version 8.

For example, suppose that A, B, and C are sets. Then since $A \times B$ is a set, we might have a function $f\colon A \times B \to C$. If $(a, b) \in A \times B$, then the value of f at (a, b) would be

[4] Image: `commons.wikimedia.org/wiki/File:Bijection.svg`.

denoted $f((a,b))$. In practice, though, one set of parentheses is usually dropped, and the value of f at (a,b) is denoted $f(a,b)$. As a particular example, we might define a function $p\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$ with the formula $p(n,m)=nm+1$. Similarly, we might define a function $q\colon \mathbb{N}\times\mathbb{N}\times\mathbb{N}\to\mathbb{N}\times\mathbb{N}$ by $q(n,m,k)=(nm-k,nk-n)$.

Suppose that A and B are sets. There are, in general, many functions that map A to B. We can gather all those functions into a set. This set, whose elements are all the functions from A to B, is denoted B^A. (We'll see later why this notation is reasonable.) Using this notation, saying $f\colon A\to B$ is exactly the same as saying $f\in B^A$. Both of these notations assert that f is a function from A to B. Of course, we can also form an unlimited number of other sets, such as the power set $\mathcal{P}(B^A)$, the cross product $B^A\times A$, or the set $A^{A\times A}$, which contains all the functions from the set $A\times A$ to the set A. And of course, any of these sets can be the domain or range of a function. An example of this is the function $\mathcal{E}\colon B^A\times A\to B$ defined by the formula $\mathcal{E}(f,a)=f(a)$. Let's see if we can make sense of this notation. Since the domain of $\mathcal{E}$ is $B^A\times A$, an element in the domain is an ordered pair in which the first coordinate is a function from A to B and the second coordinate is an element of A. Thus, $\mathcal{E}(f,a)$ is defined for a function $f\colon A\to B$ and an element $a\in A$. Given such an f and a, the notation $f(a)$ specifies an element of B, so the definition of $\mathcal{E}(f,a)$ as $f(a)$ makes sense. The function $\mathcal{E}$ is called the 'evaluation function' since it captures the idea of evaluating a function at an element of its domain.

4

Exercises

1. Let $A=\{1,2,3,4\}$ and let $B=\{a,b,c\}$. Find the sets $A\times B$ and $B\times A$.

2. Let A be the set $\{a,b,c,d\}$. Let f be the function from A to A given by the set of ordered pairs $\{(a,b),(b,b),(c,a),(d,c)\}$, and let g be the function given by the set of ordered pairs $\{(a,b),(b,c),(c,d),(d,d)\}$. Find the set of ordered pairs for the composition $g\circ f$.

3. Let $A=\{a,b,c\}$ and let $B=\{0,1\}$. Find all possible functions from A to B. Give each function as a set of ordered pairs. (Hint: Every such function corresponds to one of the subsets of A.)

4. Consider the functions from $\mathbb{Z}$ to $\mathbb{Z}$ which are defined by the following formulas. Decide whether each function is onto and whether it is one-to-one; prove your answers.

a) $f(n)=2n$ **b)** $g(n)=n+1$ **c)** $h(n)=n^2+n+1$

d) $s(n)=\begin{cases} \text{n/2}, & \text{if } n \text{ is even} \\ \text{(n+1)/2}, & \text{if } n \text{ is odd} \end{cases}$

5. Prove that composition of functions is an associative operation. That is, prove that for functions $f\colon A\to B$, $g\colon B\to C$, and $h\colon C\to D$, the compositions $(h\circ g)\circ f$ and $h\circ(g\circ f)$ are equal.

6. Suppose that $f\colon A\to B$ and $g\colon B\to C$ are functions and that $g\circ f$ is one-to-one.

a) Prove that f is one-to-one. (Hint: use a proof by contradiction.)

b) Find a specific example that shows that g is not necessarily one-to-one.

7. Suppose that $f\colon A\to B$ and $g\colon B\to C$, and suppose that the composition $g\circ f$ is an onto function.

a) Prove that g is an onto function.

b) Find a specific example that shows that f is not necessarily onto.

4.5 Application: Programming with Functions

Functions are fundamental in computer programming, although not everything in programming that goes by the name of 'function' is a function according to the mathematical definition.

In this section we go into detail about functions in computer programming. You won't be examined on this in *Reasoning & Logic*! You will find that there is again quite some overlap with your study materials from *Object-Oriented Programming* and later on in your curriculum with *Concepts of Programming Languages*.

In computer programming, a function is a routine that is given some data as input and that will calculate and return an answer based on that data. For example, Java, a function that calculates the square of an integer could be written

```
1 int square(int n) {
    return n*n;
3 }
```

In Java, *int* is a data type. From the mathematical point of view, a data type is a set. The data type *int* is the set of all integers that can be represented as 32-bit binary numbers. Mathematically, then, $int \subseteq \mathbb{Z}$. (You should get used to the fact that sets and functions can have names that consist of more than one character, since it's done all the time in computer programming.) The first line of the above function definition, `int square(int n)`, says that we are defining a function named square whose range is *int* and whose domain is *int*. In the usual notation for functions, we would express this as $square\colon int \to int$, or possibly as $square \in int^{int}$, where int^{int} is the set of all functions that map the set *int* to the set *int*.

The first line of the function, `int square(int n)`, is called the ***prototype*** of the function. The prototype specifies the name, the domain, and the range of the function and so carries exactly the same information as the notation $f\colon A \to B$. The '*n*' in the prototype `int square(int n)` is a name for an arbitrary element of the data type *int*. We call *n* a ***parameter*** of the function. The rest of the definition of *square* tells the computer to calculate the value of $square(n)$ for any $n \in int$ by multiplying *n* times *n*. The statement `return n*n` says that $n * n$ is the value that is computed, or 'returned', by the function. (The $*$ stands for multiplication.)

Java has many data types in addition to *int*. There is a boolean data type named *boolean*. The values of type *boolean* are *true* and *false*. Mathematically, *boolean* is a name for the set $\{true, false\}$. The type *double* consists of real numbers, which can include a decimal point. Of course, on a computer, it's not possible to represent the entire infinite set of real numbers, so *double* represents some subset of the mathematical set of real

numbers. There is also a data type whose values are strings of characters, such as "Hello world" or "xyz152QQZ". The name for this data type in Java is *string*. All these types, and many others, can be used in functions. For example, in Java, $m \,\%\, n$ is the remainder when the integer m is divided by the integer n. We can define a function to test whether an integer is even as follows:

```
boolean even(int k) {
   return k % 2 == 0;
}
```

You don't need to worry about all the details here, but you should understand that the prototype, `boolean even(int k)`, says that *even* is a function from the set *int* to the set *boolean*. That is, $even$: $int \to boolean$. Given an integer N, $even(N)$ has the value *true* if N is an even integer, and it has the value *false* if N is an odd integer.

A function can have more than one parameter. For example, we might define a function with prototype `int index(string str, string sub)`. If s and t are strings, then $index(s,t)$ would be the *int* that is the value of the function at the ordered pair (s,t). We see that the domain of index is the cross product $string \times string$, and we can write $index$: $string \times string \to int$ or, equivalently, $index \in int^{string \times string}$.

Not every Java function is actually a function in the mathematical sense. In mathematics, a function must associate a single value in its range to each value in its domain. There are two things that can go wrong: the value of the function might not be defined for every element of the domain, and the function might associate several different values to the same element of the domain. Both of these things can happen with Java functions.

In computer programming, it is very common for a 'function' to be undefined for some values of its parameter. In mathematics, a ***partial function*** from a set A to a set B is defined to be a function from a subset of A to B. A partial function from A to B can be undefined for some elements of A, but when it is defined for some $a \in A$, it associates just one element of B to a. Many functions in computer programs are actually partial functions. (When dealing with partial functions, an ordinary function, which is defined for every element of its domain, is sometimes referred to as a ***total function***. Note that—with the mind-bending logic that is typical of mathematicians—a total function is a type of partial function, because a set is a subset of itself.)

It's also very common for a 'function' in a computer program to produce a variety of values for the same value of its parameter. A common example is a function with prototype `int random(int N)`, which returns a random integer between 1 and N. The value of $random(5)$ could be 1, 2, 3, 4, or 5. This is not the behaviour of a mathematical function—but it's very useful when programming!

Even though many functions in computer programs are not really mathematical functions, I will continue to refer to them as functions in this section. Mathematicians will just have to stretch their definitions a bit to accommodate the realities of computer programming.

4.5.1 Functions as first-class objects

In most programming languages, functions are not first-class objects. That is, a function cannot be treated as a data value in the same way as a *string* or an *int*. However, newer versions of Java do take a step in this direction with 'lambda expressions'. In Java it is not yet possible for a function to be a parameter to another function. For example, suppose in Java we could write the function prototype

```
double sumten(Function<Integer,Double> f)
```

This is a prototype for a function named *sumten* whose parameter is a function. The parameter is specified by the prototype `Function<Integer,Double> f`. This means that the parameter must be a function from *int* to *double*. The parameter name, f, stands for an arbitrary such function. Mathematically, $f \in double^{int}$, and so $sumten\colon double^{int} \to double$.

My idea is that $sumten(f)$ would compute $f(1) + f(2) + \cdots + f(10)$. A more useful function would be able to compute $f(a) + f(a+1) + \cdots + f(b)$ for any integers a and b. This just means that a and b should be parameters to the function. The prototype for the improved function would look like

```
double sum(Function<Integer,Double> f, int a, int b)
```

The parameters to *sum* form an ordered triple in which the first coordinate is a function and the second and third coordinates are integers. So, we could write

$$sum\colon double^{int} \times int \times int \to double$$

It's interesting that computer programmers deal routinely with such complex objects.

Languages where functions are first-class objects are for example Python and Scala. These languages support what is called ***functional programming***.

One of the most accessible languages that supports functional programming is JavaScript, a language that is used on webpages. (Although the names are similar, JavaScript and Java are only distantly related. You probably knew that.) In JavaScript, the function that computes the square of its parameter could be defined as

```
function square(n) {
   return n*n;
}
```

This is similar to the Java definition of the same function, but you'll notice that no type is specified for the parameter n or for the value computed by the function. Given this definition of square, $square(x)$ would be legal for any x of any type. (Of course, the value of $square(x)$ would be undefined for most types, so *square* is a *very* partial function, like most functions in JavaScript.) In effect, all possible data values in JavaScript are bundled together into one set, which I will call *data*. We then have $square\colon data \to data$.[5]

[5]Not all functional programming languages lump data types together in this way. There is a functional programming language, Haskell, for example, that is as strict about types as C++. For information about Haskell, see `www.haskell.org`.

In JavaScript, a function really is a first-class object. We can begin to see this by looking at an alternative definition of the function *square*:

```
square = function(n) { return n*n; }
```

Here, the notation `function(n) { return n*n; }` creates a function that computes the square of its parameter, but it doesn't give any name to this function. This function object is then assigned to a variable named *square*. The value of *square* can be changed later, with another assignment statement, to a different function or even to a different type of value. This notation for creating function objects can be used in other places besides assignment statements. Suppose, for example, that a function with prototype `function sum(f,a,b)` has been defined in a JavaScript program to compute $f(a) + f(a+1) + \cdots + f(b)$. Then we could compute $1^2 + 2^2 + \cdots + 100^2$ by saying


```
sum(function(n) { return n*n; }, 1, 100)
```

Here, the first parameter is the function that computes squares. We have created and used this function without ever giving it a name.

It is even possible in JavaScript for a function to return another function as its value. For example,

```
function monomial(a, n) {
   return (function(x) { a*Math.pow(x,n); });
}
```

Here, `Math.pow(x,n)` computes x^n, so for any numbers a and n, the value of *monomial*(a,n) is a function that computes ax^n. Thus,

```
f = monomial(2,3);
```

would define f to be the function that satisfies $f(x) = 2x^3$, and if *sum* is the function described above, then

```
sum( monomial(8,4), 3, 6 )
```

would compute $8 * 3^4 + 8 * 4^4 + 8 * 5^4 + 8 * 6^4$. In fact, *monomial* can be used to create an unlimited number of new functions from scratch. It is even possible to write *monomial*(2,3)(5) to indicate the result of applying the function *monomial*(2,3) to the value 5. The value represented by *monomial*(2,3)(5) is $2 * 5^3$, or 250. This is real functional programming and might give you some idea of its power.

Exercises

1. For each of the following Java-like function prototypes, translate the prototype into a standard mathematical function specification, such as *func*: *float* $\to$ *int*.

a) `int strlen(string s)`

b) `double pythag(double x, double y)`

c) `int round(double x)`
d) `string sub(string s, int n, int m)`
e) `string unlikely(Function<String,Integer> f )`
f) `int h( Function<Integer,Integer> f, Function<Integer,Integer> g )`

2. Write a Java-like function prototype for a function that belongs to each of the following sets.

a) $string^{string}$

b) $boolean^{float \times float}$

c) $float^{int^{int}}$

3. It is possible to define new types in Java by using classes. For example, the definition

```
Class point {
  double x;
  double y;
}
```

defines a new type named *point*. A value of type *point* contains two values of type *double*. What mathematical operation corresponds to the construction of this data type? Why?

4. Let *square*, *sum* and *monomial* be the JavaScript functions described in this section. What is the value of each of the following?

a) $sum(square, 2, 4)$
b) $sum(monomial(5,2), 1, 3)$
c) $monomial(square(2), 7)$
d) $sum(\text{function}(n)\ \{\ \text{return}\ 2 * n;\ \}, 1, 5)$
e) $square(sum(monomial(2,3), 1, 2))$

5. Write a JavaScript function named *compose* that computes the composition of two functions. That is, $compose(f,g)$ is $f \circ g$, where f and g are functions of one parameter. Recall that $f \circ g$ is the function defined by $(f \circ g)(x) = f(g(x))$.

4.6 Counting Past Infinity

As children, we all learned to answer the question "How many?" by counting with numbers: 1, 2, 3, 4, ... But the question of "How many?" was asked and answered long before the abstract concept of number was invented. The answer can be given in terms of "as many as." How many cousins do you have? As many cousins as I have fingers on both hands. How many sheep do you own? As many sheep as there are notches on this stick. How many baskets of wheat must I pay in taxes? As many baskets as there are stones in this box. The question of how many things are in one collection of objects is answered by exhibiting another, more convenient, collection of objects that has just as many members.

In set theory, the idea of one set having just as many members as another set is expressed in terms of ***one-to-one correspondence***. A one-to-one correspondence between two sets A and B pairs each element of A with an element of B in such a way that every

element of B is paired with one and only one element of A. The process of counting, as it is learned by children, establishes a one-to-one correspondence between a set of n objects and the set of numbers from 1 to n. The rules of counting are the rules of one-to-one correspondence: Make sure you count every object, make sure you don't count the same object more than once. That is, make sure that each object corresponds to *one* and *only one* number. Earlier in this chapter, we used the fancy name 'bijective function' to refer to this idea, but we can now see it as as an old, intuitive way of answering the question "How many?"

4.6.1 Cardinality

In counting, as it is learned in childhood, the set $\{1, 2, 3, \ldots, n\}$ is used as a typical set that contains n elements. In mathematics and computer science, it has become more common to start counting with zero instead of with one, so we define the following sets to use as our basis for counting:

$$\begin{array}{ll} N_0 = \emptyset, & \text{a set with 0 elements} \\ N_1 = \{0\}, & \text{a set with 1 element} \\ N_2 = \{0, 1\}, & \text{a set with 2 elements} \\ N_3 = \{0, 1, 2\}, & \text{a set with 3 elements} \\ N_4 = \{0, 1, 2, 3\}, & \text{a set with 4 elements} \end{array}$$

and so on. In general, $N_n = \{0, 1, 2, \ldots, n-1\}$ for each $n \in \mathbb{N}$. For each natural number n, N_n is a set with n elements. Note that if $n \neq m$, then there is no one-to-one correspondence between N_n and N_m. This is obvious, but like many obvious things is not all that easy to prove rigorously, and we omit the argument here.

Theorem 4.6. *For each $n \in \mathbb{N}$, let N_n be the set $N_n = \{0, 1, \ldots, n-1\}$. If $n \neq m$, then there is no bijective function from N_m to N_n.*

We can now make the following definitions:

Definition 4.3. A set A is said to be ***finite*** if there is a one-to-one correspondence between A and N_n for some natural number n. We then say that n is the ***cardinality*** of A. The notation $|A|$ is used to indicate the cardinality of A. That is, if A is a finite set, then $|A|$ is the natural number n such that there is a one-to-one correspondence between A and N_n. In layman's terms: $|A|$ is the number items in A. A set that is not finite is said to be ***infinite***. That is, a set B is infinite if for every $n \in \mathbb{N}$, there is *no* one-to-one correspondence between B and N_n.

Fortunately, we don't always have to count every element in a set individually to determine its cardinality. Consider, for example, the set $A \times B$, where A and B are finite sets. If we already know $|A|$ and $|B|$, then we can determine $|A \times B|$ by computation, without explicit counting of elements. In fact, $|A \times B| = |A| \cdot |B|$. The cardinality of

the cross product $A \times B$ can be computed by multiplying the cardinality of A by the cardinality of B. To see why this is true, think of how you might count the elements of $A \times B$. You could put the elements into piles, where all the ordered pairs in a pile have the same first coordinate. There are as many piles as there are elements of A, and each pile contains as many ordered pairs as there are elements of B. That is, there are $|A|$ piles, with $|B|$ items in each. By the definition of multiplication, the total number of items in all the piles is $|A| \cdot |B|$. A similar result holds for the cross product of more that two finite sets. For example, $|A \times B \times C| = |A| \cdot |B| \cdot |C|$.

It's also easy to compute $|A \cup B|$ in the case where A and B are disjoint finite sets. (Recall that two sets A and B are said to be disjoint if they have no members in common, that is, if $A \cap B = \emptyset$.) Suppose $|A| = n$ and $|B| = m$. If we wanted to count the elements of $A \cup B$, we could use the n numbers from 0 to $n - 1$ to count the elements of A and then use the m numbers from n to $n + m - 1$ to count the elements of B. This amounts to a one-to-one correspondence between $A \cup B$ and the set N_{n+m}. We see that $|A \cup B| = n + m$. That is, for disjoint finite sets A and B, $|A \cup B| = |A| + |B|$.

What about $A \cup B$, where A and B are not disjoint? We have to be careful not to count the elements of $A \cap B$ twice. After counting the elements of A, there are only $|B| - |A \cap B|$ new elements in B that still need to be counted. So we see that for any two finite sets A and B, $|A \cup B| = |A| + |B| - |A \cap B|$.

What about the number of subsets of a finite set A? What is the relationship between $|A|$ and $|\mathcal{P}(A)|$? The answer is provided by the following theorem.

Theorem 4.7. *A finite set with cardinality n has 2^n subsets.*

Proof. Let $P(n)$ be the statement "Any set with cardinality n has 2^n subsets". We will use induction to show that $P(n)$ is true for all $n \in N$.

Base case: For $n = 0$, $P(n)$ is the statement that a set with cardinality 0 has 2^0 subsets. The only set with 0 elements is the empty set. The empty set has exactly 1 subset, namely itself. Since $2^0 = 1$, $P(0)$ is true.

Inductive case: Let k be an arbitrary element of $\mathbb{N}$, and assume that $P(k)$ is true. That is, assume that any set with cardinality k has 2^k elements. (This is the induction hypothesis.) We must show that $P(k + 1)$ follows from this assumption. That is, using the assumption that any set with cardinality k has 2^k subsets, we must show that any set with cardinality $k + 1$ has 2^{k+1} subsets.

Let A be an arbitrary set with cardinality $k + 1$. We must show that $|\mathcal{P}(A)| = 2^{k+1}$. Since $|A| > 0$, A contains at least one element. Let x be some element of A, and let $B = A \setminus \{x\}$. The cardinality of B is k, so we have by the induction hypothesis that $|\mathcal{P}(B)| = 2^k$. Now, we can divide the subsets of A into two classes: subsets of A that do not contain x and subsets of A that do contain x. Let Y be the collection of subsets of A that do not contain x, and let X be the collection of subsets of A that do contain x. X and Y are disjoint, since it is impossible for a given subset of A both to contain and to not contain x. It follows that $|\mathcal{P}(A)| = |X \cup Y| = |X| + |Y|$.

Now, a member of Y is a subset of A that does not contain x. But that is exactly the same as saying that a member of Y is a subset of B. So $Y = \mathcal{P}(B)$, which we know contains 2^k members. As for X, there is a one-to-one correspondence between $\mathcal{P}(B)$ and X. Namely, the function $f\colon \mathcal{P}(B) \to X$ defined by $f(C) = C \cup \{x\}$ is a bijective function. (The proof of this is left as an exercise.) From this, it follows that $|X| = |\mathcal{P}(B)| = 2^k$. Putting these facts together, we see that $|\mathcal{P}(A)| = |X| + |Y| = 2^k + 2^k = 2 \cdot 2^k = 2^{k+1}$. This completes the proof that $P(k) \to P(k+1)$. □

We have seen that the notation A^B represents the set of all functions from B to A. Suppose A and B are finite, and that $|A| = n$ and $|B| = m$. Then $\left|A^B\right| = n^m = |A|^{|B|}$. (This fact is one of the reasons why the notation A^B is reasonable.) One way to see this is to note that there is a one-to-one correspondence between A^B and a cross product $A \times A \times \cdots A$, where the number of terms in the cross product is m. (This will be shown in one of the exercises at the end of this section.) It follows that $\left|A^B\right| = |A| \cdot |A| \cdots |A| = n \cdot n \cdots n$, where the factor n occurs m times in the product. This product is, by definition, n^m.

This discussion about computing cardinalities is summarized in the following theorem:

Theorem 4.8. *Let A and B be finite sets. Then*

- $|A \times B| = |A| \cdot |B|$.
- $|A \cup B| = |A| + |B| - |A \cap B|$.
- *If A and B are disjoint then* $|A \cup B| = |A| + |B|$.
- $\left|A^B\right| = |A|^{|B|}$.
- $|\mathcal{P}(A)| = 2^{|A|}$.

When it comes to counting and computing cardinalities, this theorem is only the beginning of the story. There is an entire large and deep branch of mathematics known as ***combinatorics*** that is devoted mostly to the problem of counting. But the theorem is already enough to answer many questions about cardinalities.

For example, suppose that $|A| = n$ and $|B| = m$. We can form the set $\mathcal{P}(A \times B)$, which consists of all subsets of $A \times B$. Using the theorem, we can compute that $|\mathcal{P}(A \times B)| = 2^{|A \times B|} = 2^{|A| \cdot |B|} = 2^{nm}$. If we assume that A and B are disjoint, then we can compute that $\left|A^{A \cup B}\right| = |A|^{|A \cup B|} = n^{n+m}$.

To be more concrete, let $X = \{a, b, c, d, e\}$ and let $Y = \{c, d, e, f\}$ where a, b, c, d, e, and f are distinct. Then $|X \times Y| = 5 \cdot 4 = 20$ while $|X \cup Y| = 5 + 4 - |\{c, d, e\}| = 6$ and $\left|X^Y\right| = 5^4 = 625$.

We can also answer some simple practical questions. Suppose that in a restaurant you

can choose one starter and one main course. What is the number of possible meals? If A is the set of possible appetizers and C is the set of possible main courses, then your meal is an ordered pair belonging to the set $A \times C$. The number of possible meals is $|A \times C|$, which is the product of the number of appetizers and the number of main courses.

Or suppose that four different prizes are to be awarded, and that the set of people who are eligible for the prizes is A. Suppose that $|A| = n$. How many different ways are there to award the prizes? One way to answer this question is to view a way of awarding the prizes as a function from the set of prizes to the set of people. Then, if P is the set of prizes, the number of different ways of awarding the prizes is $|A^P|$. Since $|P| = 4$ and $|A| = n$, this is n^4. Another way to look at it is to note that the people who win the prizes form an ordered tuple (a, b, c, d), which is an element of $A \times A \times A \times A$. So the number of different ways of awarding the prizes is $|A \times A \times A \times A|$, which is $|A| \cdot |A| \cdot |A| \cdot |A|$. This is $|A|^4$, or n^4, the same answer we got before.[6]

4.6.2 Counting to infinity

So far, we have only discussed finite sets. $\mathbb{N}$, the set of natural numbers $\{0, 1, 2, 3, \dots\}$, is an example of an infinite set. There is no one-to-one correspondence between $\mathbb{N}$ and any of the finite sets N_n. Another example of an infinite set is the set of even natural numbers, $E = \{0, 2, 4, 6, 8, \dots\}$. There is a natural sense in which the sets $\mathbb{N}$ and E have the same number of elements. That is, there is a one-to-one correspondence between them. The function $f : \mathbb{N} \to E$ defined by $f(n) = 2n$ is bijective. We will say that $\mathbb{N}$ and E have the same cardinality, even though that cardinality is not a finite number. Note that E is a proper subset of $\mathbb{N}$. That is, $\mathbb{N}$ has a proper subset that has the same cardinality as $\mathbb{N}$.

We will see that not all infinite sets have the same cardinality. When it comes to infinite sets, intuition is not always a good guide. Most people seem to be torn between two conflicting ideas. On the one hand, they think, it seems that a proper subset of a set should have fewer elements than the set itself. On the other hand, it seems that any two infinite sets should have the same number of elements. Neither of these is true, at least if we define having the same number of elements in terms of one-to-one correspondence.

A set A is said to be ***countably infinite*** if there is a one-to-one correspondence between $\mathbb{N}$ and A. A set is said to be ***countable*** if it is either finite or countably infinite. An infinite set that is not countably infinite is said to be ***uncountable***. If X is an uncountable set, then there is no one-to-one correspondence between $\mathbb{N}$ and X.

The idea of 'countable infinity' is that even though a countably infinite set cannot be counted in a finite time, we can imagine counting all the elements of A, one-by-one,

[6] This discussion assumes that one person can receive any number of prizes. What if the prizes have to go to four different people? This question takes us a little farther into combinatorics than I would like to go, but the answer is not hard. The first award can be given to any of n people. The second prize goes to one of the remaining $n - 1$ people. There are $n - 2$ choices for the third prize and $n - 3$ for the fourth. The number of different ways of awarding the prizes to four different people is the product $n(n - 1)(n - 2)(n - 3)$. What about dividing arbitrary objects between arbitrary numbers of people? That's one topic of *social choice theory*.

in an infinite process. A bijective function $f\colon \mathbb{N} \to A$ provides such an infinite listing: $(f(0), f(1), f(2), f(3), \dots)$. Since f is onto, this infinite list includes all the elements of A. In fact, making such a list effectively shows that A is countably infinite, since the list amounts to a bijective function from $\mathbb{N}$ to A. For an uncountable set, it is impossible to make a list, even an infinite list, that contains all the elements of the set.

Before you start believing in uncountable sets, you should ask for an example. In Chapter 3, we worked with the infinite sets $\mathbb{Z}$ (the integers), $\mathbb{Q}$ (the rationals), $\mathbb{R}$ (the reals), and $\mathbb{R} \setminus \mathbb{Q}$ (the irrationals). Intuitively, these are all 'bigger' than $\mathbb{N}$, but as I have already mentioned, intuition is a poor guide when it comes to infinite sets. Are any of $\mathbb{Z}$, $\mathbb{Q}$, $\mathbb{R}$, and $\mathbb{R} \setminus \mathbb{Q}$ in fact uncountable?

It turns out that both $\mathbb{Z}$ and $\mathbb{Q}$ are only countably infinite. The proof that $\mathbb{Z}$ is countable is left as an exercise; we will show here that the set of non-negative rational numbers is countable. (The fact that $\mathbb{Q}$ itself is countable follows easily from this.) The reason is that it's possible to make an infinite list containing all the non-negative rational numbers. Start the list with all the non-negative rational numbers n/m such that $n + m = 1$. There is only one such number, namely $0/1$. Next come numbers with $n + m = 2$. They are $0/2$ and $1/1$, but we leave out $0/2$ since it's just another way of writing $0/1$, which is already in the list. Now, we add the numbers with $n + m = 3$, namely $0/3$, $1/2$, and $2/1$. Again, we leave out $0/3$, since it's equal to a number already in the list. Next come numbers with $n + m = 4$. Leaving out $0/4$ and $2/2$ since they are already in the list, we add $1/3$ and $3/1$ to the list. We continue in this way, adding numbers with $n + m = 5$, then numbers with $n + m = 6$, and so on. The list looks like:

$$\left(\frac{0}{1}, \frac{1}{1}, \frac{1}{2}, \frac{2}{1}, \frac{1}{3}, \frac{3}{1}, \frac{1}{4}, \frac{2}{3}, \frac{3}{2}, \frac{4}{1}, \frac{1}{5}, \frac{5}{1}, \frac{1}{6}, \frac{2}{5}, \dots\right)$$

This process can be continued indefinitely, and every non-negative rational number will eventually show up in the list. So we get a complete, infinite list of non-negative rational numbers. This shows that the set of non-negative rational numbers is in fact countable.

4.6.3 Uncountable sets

On the other hand, $\mathbb{R}$ is uncountable. It is not possible to make an infinite list that contains every real number. It is not even possible to make a list that contains every real number between zero and one. Another way of saying this is that every infinite list of real numbers between zero and one, no matter how it is constructed, leaves something out. To see why this is true, imagine such a list, displayed in an infinitely long column. Each row contains one number, which has an infinite number of digits after the decimal point. Since it is a number between zero and one, the only digit before the decimal point is zero. For example, the list might look like this:

$$
\begin{aligned}
&0.\mathbf{9}039893724987956129792765485794 5\ldots\\
&0.1\mathbf{2}34934209405987598023923083454 9\ldots\\
&0.22\mathbf{4}0004329843623470932327998957 9\ldots\\
&0.500\mathbf{0}0000000000000000000000000 0\ldots\\
&0.7774\mathbf{3}4492342348769901209094800 9\ldots\\
&0.77755\mathbf{5}5558888888949888898000011 1\ldots\\
&0.1234567\mathbf{8}888888888888888880000000 0\ldots\\
&0.34835440\mathbf{0}0098487127121239403205777\ldots\\
&0.93473244\mathbf{4}4790049834099999094890 0\ldots\\
&\quad\vdots
\end{aligned}
$$

This is only (a small part of) one possible list. How can we be certain that *every* such list leaves out some real number between zero and one? The trick is to look at the digits shown in bold face. We can use these digits to build a number that is not in the list. Since the first number in the list has a 9 in the first position after the decimal point, we know that this number cannot equal any number of, for example, the form 0.4.... Since the second number has a 2 in the second position after the decimal point, *neither* of the first two numbers in the list is equal to any number that begins with 0.44.... Since the third number has a 4 in the third position after the decimal point, *none* of the first three numbers in the list is equal to any number that begins 0.445.... We can continue to construct a number in this way, and we end up with a number that is different from every number in the list. The n^{th} digit of the number we are building must differ from the n^{th} digit of the n^{th} number in the list. These are the digits shown in bold face in the above list. To be definite, I use a 5 when the corresponding boldface number is 4, and otherwise I use a 4. For the list shown above, this gives a number that begins 0.44544445.... The number constructed in this way is not in the given list, so the list is incomplete. The same construction clearly works for any list of real numbers between zero and one. No such list can be a complete listing of the real numbers between zero and one, and so there can be no complete listing of all real numbers. We conclude that the set $\mathbb{R}$ is uncountable.

The technique used in this argument is called ***diagonalization***. It is named after the fact that the bold face digits in the above list lie along a diagonal line.

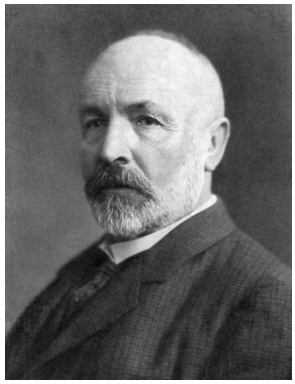
Proofs by diagonalisation are not a part of *Reasoning & Logic*. You should be able to prove a set is countably infinite (by finding a bijection from $\mathbb{N}$ to the set), but you will not be asked to prove a set is uncountable. We leave that for the course *Automata, Languages, and Computability* later in your curriculum.

This proof was discovered by a mathematician named Georg Cantor, who caused quite a fuss in the nineteenth century when he came up with the idea that there are different kinds of infinity. Since then, his notion of using one-to-one correspondence to define the cardinalities of infinite sets has been accepted. Mathematicians now consider

it almost intuitive that $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{Q}$ have the same cardinality while $\mathbb{R}$ has a strictly larger cardinality.

To say that George Cantor (1845–1918) caused a fuss in mathematics is the least you can say. Cantor's theory of transfinite numbers was originally regarded as so counter-intuitive—even shocking—that it encountered resistance from mathematical contemporaries. Like Frege, Cantor was a German mathematician contributing to logic and set theory. Like Peirce, Cantor's ideas did not find acceptance until later in his life. In 1904, the Royal Society awarded Cantor its Sylvester Medal, the highest honour it can confer for work in mathematics.

Source: `en.wikipedia.org/wiki/Georg_Cantor`.



Theorem 4.9. *Suppose that X is an uncountable set, and that K is a countable subset of X. Then the set $X \setminus K$ is uncountable.*

Proof. Let X be an uncountable set. Let $K \subseteq X$, and suppose that K is countable. Let $L = X \setminus K$. We want to show that L is uncountable. Suppose that L is countable. We will show that this assumption leads to a contradiction.

Note that $X = K \cup (X \setminus K) = K \cup L$. You will show in Exercise 11 of this section that the union of two countable sets is countable. Since X is the union of the countable sets K and L, it follows that X is countable. But this contradicts the fact that X is uncountable. This contradiction proves the theorem. □

In the proof, both q and $\neg q$ are shown to follow from the assumptions, where q is the statement 'X is countable'. The statement q is shown to follow from the assumption that $X \setminus K$ is countable. The statement $\neg q$ is true by assumption. Since q and $\neg q$ cannot both be true, at least one of the assumptions must be false. The only assumption that can be false is the assumption that $X \setminus K$ is countable.

This theorem, by the way, has the following easy corollary. (A ***corollary*** is a theorem that follows easily from another, previously proved theorem.)

Corollary 4.10. *The set of irrational real numbers is uncountable.*

Proof. Let I be the set of irrational real numbers. By definition, $I = \mathbb{R} \setminus \mathbb{Q}$. We have already shown that $\mathbb{R}$ is uncountable and that $\mathbb{Q}$ is countable, so the result follows immediately from the previous theorem. □

You might still think that $\mathbb{R}$ is as big as things get, that is, that any infinite set is in one-to-one correspondence with $\mathbb{R}$ or with some subset of $\mathbb{R}$. In fact, though, if X is any set then it's possible to find a set that has strictly larger cardinality than X. In fact, $\mathcal{P}(X)$ is such a set. A variation of the diagonalization technique can be used to show that there is no one-to-one correspondence between X and $\mathcal{P}(X)$. Note that this is obvious for finite sets, since for a finite set X, $|\mathcal{P}(X)| = 2^{|X|}$, which is larger than $|X|$. The point of the theorem is that it is true even for infinite sets.

Theorem 4.11. *Let X be any set. Then there is no one-to-one correspondence between X and $\mathcal{P}(X)$.*

Proof. Given an arbitrary function $f\colon X \to \mathcal{P}(X)$, we can show that f is not onto. Since a one-to-one correspondence is both one-to-one and onto, this shows that f is not a one-to-one correspondence.

Recall that $\mathcal{P}(X)$ is the set of subsets of X. So, for each $x \in X$, $f(x)$ is a subset of X. We have to show that no matter how f is defined, there is some subset of X that is not in the image of f.

Given f, we define A to be the set $A = \{x \in X \mid x \notin f(x)\}$. The test '$x \notin f(x)$' makes sense because $f(x)$ is a set. Since $A \subseteq X$, we have that $A \in \mathcal{P}(X)$. However, A is not in the image of f. That is, for every $y \in X$, $A \neq f(y)$.[7] To see why this is true, let y be any element of X. There are two cases to consider. Either $y \in f(y)$ or $y \notin f(y)$. We show that whichever case holds, $A \neq f(y)$. If it is true that $y \in f(y)$, then by the definition of A, $y \notin A$. Since $y \in f(y)$ but $y \notin A$, $f(y)$ and A do not have the same elements and therefore are not equal. On the other hand, suppose that $y \notin f(y)$. Again, by the definition of A, this implies that $y \in A$. Since $y \notin f(y)$ but $y \in A$, $f(y)$ and A do not have the same elements and therefore are not equal. In either case, $A \neq f(y)$. Since this is true for any $y \in X$, we conclude that A is not in the image of f and therefore f is not a one-to-one correspondence. □

From this theorem, it follows that there is no one-to-one correspondence between $\mathbb{R}$ and $\mathcal{P}(\mathbb{R})$. The cardinality of $\mathcal{P}(\mathbb{R})$ is strictly bigger than the cardinality of $\mathbb{R}$. But it doesn't stop there. $\mathcal{P}(\mathcal{P}(\mathbb{R}))$ has an even bigger cardinality, and the cardinality of $\mathcal{P}(\mathcal{P}(\mathcal{P}(\mathbb{R})))$ is bigger still. We could go on like this forever, and we still won't have exhausted all the possible cardinalities. If we let $\mathbb{X}$ be the infinite union $\mathbb{R} \cup \mathcal{P}(\mathbb{R}) \cup \mathcal{P}(\mathcal{P}(\mathbb{R})) \cup \cdots$, then $\mathbb{X}$ has larger cardinality than any of the sets in the union. And then

[7] In fact, we have constructed A so that the sets A and $f(y)$ differ in at least one element, namely y itself. This is where the 'diagonalization' comes in.

there's $\mathcal{P}(\mathbb{X})$, $\mathcal{P}(\mathcal{P}(\mathbb{X}))$, $\mathbb{X} \cup \mathcal{P}(\mathbb{X}) \cup \mathcal{P}(\mathcal{P}(\mathbb{X})) \cup \cdots$. There is no end to this. There is no upper limit on possible cardinalities, not even an infinite one! We have counted past infinity.

4.6.4 A final note on infinities

We have seen that $|\mathbb{R}|$ is strictly larger than $|\mathbb{N}|$. We end this section with what might look like a simple question: Is there a subset of $\mathbb{R}$ that is neither in one-to-one correspondence with $\mathbb{N}$ nor with $\mathbb{R}$? That is, is the cardinality of $\mathbb{R}$ the *next* largest cardinality after the cardinality of $\mathbb{N}$, or are there other cardinalities intermediate between them? This problem was unsolved for quite a while, and the solution, when it was found, proved to be completely unexpected. It was shown that both 'yes' and 'no' are consistent answers to this question! That is, the logical structure built on the system of axioms that had been accepted as the basis of set theory was not extensive enough to answer the question. The question is 'undecidable' in that system. We will come back to 'undecidability' in Chapter 5.

It is possible to extend the system of axioms underlying set theory in various ways. In some extensions, the answer is yes. In others, the answer is no. You might object, "Yes, but which answer is true for the *real* real numbers?" Unfortunately, it's not even clear whether this question makes sense, since in the world of mathematics, the real numbers are just part of a structure built from a system of axioms. And it's not at all clear whether the 'real numbers' exist in some sense in the real world. If all this sounds like it's a bit of a philosophical muddle, it is. That's the state of things today at the foundation of mathematics, and it has implications for the foundations of computer science, as we'll see in the next chapter.

Exercises

1. Suppose that A, B, and C are finite sets which are pairwise disjoint. (That is, $A \cap B = A \cap C = B \cap C = \emptyset$.) Express the cardinality of each of the following sets in terms of $|A|$, $|B|$, and $|C|$. Which of your answers depend on the fact that the sets are pairwise disjoint?

a) $\mathcal{P}(A \cup B)$	**b)** $A \times (B^C)$	**c)** $\mathcal{P}(A) \times \mathcal{P}(C)$
d) $A^{B \times C}$	**e)** $(A \times B)^C$	**f)** $\mathcal{P}(A^B)$
g) $(A \cup B)^C$	**h)** $(A \cup B) \times A$	**i)** $A \times A \times B \times B$

2. Suppose that A and B are finite sets which are not necessarily disjoint. What are all the possible values for $|A \cup B|$?

3. Let's say that an 'identifier' consists of one or two characters. The fist character is one of the twenty-six letters (A, B, ..., C). The second character, if there is one, is either a letter or one of the ten digits (0, 1, ..., 9). How many different identifiers are there? Explain your answer in terms of unions and cross products.

4. Suppose that there are five books that you might bring along to read on your vacation. In how many different ways can you decide which books to bring, assuming that you want to bring at least one? Why?

5. Show that the cardinality of a finite set is well-defined. That is, show that if f is a bijective function from a set A to N_n, and if g is a bijective function from A to N_m, then $n = m$.

6. Finish the proof of Theorem 4.7 by proving the following statement: Let A be a non-empty set, and let $x \in A$. Let $B = A \setminus \{x\}$. Let $X = \{C \subseteq A \mid x \in C\}$. Define $f\colon \mathcal{P}(B) \to X$ by the formula $f(C) = C \cup \{x\}$. Show that f is a bijective function.

7. Use induction on the cardinality of B to show that for any finite sets A and B, $\left|A^B\right| = |A|^{|B|}$. (Hint: For the case where $B \neq \emptyset$, choose $x \in B$, and divide A^B into classes according to the value of $f(x)$.)

8. Let A and B be finite sets with $|A| = n$ and $|B| = m$. Let us list the elements of B as $B = \{b_0, b_1, \ldots, b_{m-1}\}$. Define the function $\mathcal{F}\colon A^B \to A \times A \times \cdots \times A$, where A occurs m times in the cross product, by $\mathcal{F}(f) = \big(f(b_0), f(b_1), \ldots, f(b_{m-1})\big)$. Show that $\mathcal{F}$ is a one-to-one correspondence.

9. Show that $\mathbb{Z}$, the set of integers, is countable by finding a one-to-one correspondence between $\mathbb{N}$ and $\mathbb{Z}$.

10. Show that the set $\mathbb{N} \times \mathbb{N}$ is countable.

11. Complete the proof of Theorem 2.9 as follows:
- **a)** Suppose that A and B are countably infinite sets. Show that $A \cup B$ is countably infinite.
- **b)** Suppose that A and B are countable sets. Show that $A \cup B$ is countable.

12. Prove that each of the following statements is true. In each case, use a proof by contradiction.
- **a)** Let X be a countably infinite set, and let N be a finite subset of X. Then $X \setminus N$ is countably infinite.
- **b)** Let A be an infinite set, and let X be a subset of A. Then at least one of the sets X and $A \setminus X$ is infinite.
- **c)** Every subset of a finite set is finite.

13. Let A and B be sets and let $\perp$ be an entity that is *not* a member of B. Show that there is a one-to-one correspondence between the set of functions from A to $B \cup \{\perp\}$ and the set of partial functions from A to B. (Partial functions were defined in Section 4.5. The symbol '$\perp$' is sometimes used in theoretical computer science to represent the value 'undefined.')

4.7 Relations

In Section 4.4, we saw that 'mother of' is a functional relationship because every person has one and only one mother, but that 'child of' is not a functional relationship, because a person can have no children or more than one child. However, the relationship expressed by 'child of' is certainly one that we have a right to be interested in and one that we should be able to deal with mathematically.

There are many examples of relationships that are not functional relationships. The relationship that holds between two natural numbers n and m when $n \leq m$ is an example in mathematics. The relationship between a person and a book that that person has on loan from the library is another. Some relationships involve more than two entities, such as the relationship that associates a name, an address and a phone number in an

address book, or the relationship that holds among three real numbers x, y, and z if $x^2 + y^2 + z^2 = 1$. Each of these relationships can be represented mathematically by what is called a 'relation'.

A ***relation*** on two sets, A and B, is defined to be a subset of $A \times B$. Since a function from A to B is defined, formally, as a subset of $A \times B$ that satisfies certain properties, a function is a relation. However, relations are more general than functions, since *any* subset of $A \times B$ is a relation. We also define a relation among three or more sets to be a subset of the cross product of those sets. In particular, a relation on A, B, and C is a subset of $A \times B \times C$.

For example, if P is the set of people and B is the set of books owned by a library, then we can define a relation $\mathcal{R}$ on the sets P and B to be the set $\mathcal{R} = \{(p, b) \in P \times B \mid p \text{ has } b \text{ out on loan}\}$. The fact that a particular $(p, b) \in \mathcal{R}$ is a fact about the world that the library will certainly want to keep track of. When a collection of facts about the world is stored on a computer, it is called a database. We'll see in the next section that relations are the most common means of representing data in databases.

If A is a set and $\mathcal{R}$ is a relation on the sets A and A (that is, on two copies of A), then $\mathcal{R}$ is said to be a ***binary relation*** on A. That is, a binary relation on the set A is a subset of $A \times A$. The relation consisting of all ordered pairs (c, p) of people such that c is a child of p is a binary relation on the set of people. The set $\{(n, m) \in \mathbb{N} \times \mathbb{N} \mid n \leq m\}$ is a binary relation on $\mathbb{N}$. Similarly, we define a ***ternary relation*** on a set A to be a subset of $A \times A \times A$. The set $\{(x, y, z) \in \mathbb{R} \times \mathbb{R} \times \mathbb{R} \mid x^2 + y^2 + z^2 = 1\}$ is a ternary relation on $\mathbb{R}$. For complete generality, we can define an ***n-ary relation*** on A, for any positive integer n, to be a subset of $A \times A \times \cdots \times A$, where A occurs n times in the cross product.

For the rest of this section, we will be working exclusively with binary relations. Suppose that $\mathcal{R} \subseteq A \times A$. That is, suppose that $\mathcal{R}$ is a binary relation on a set A. If $(a, b) \in \mathcal{R}$, then we say that a is related to b by $\mathcal{R}$. Instead of writing '$(a, b) \in \mathcal{R}$', we will often write '$a \,\mathcal{R}\, b$'. This notation is used in analogy to the notation $n \leq m$ to express the relation that n is less than or equal to m. Remember that $a \,\mathcal{R}\, b$ is just an alternative way of writing $(a, b) \in \mathcal{R}$. In fact, we could consider the relation $\leq$ to be a set of ordered pairs and write $(n, m) \in \leq$ in place of the notation $n \leq m$.

4.7.1 Properties of relations

In many applications, attention is restricted to relations that satisfy some property or set of properties. (This is, of course, just what we do when we study functions.) We begin our discussion of binary relations by considering several important properties. In this discussion, let A be a set and let $\mathcal{R}$ be a binary relation on A, that is, a subset of $A \times A$.

$\mathcal{R}$ is said to be ***reflexive*** if $\forall a \in A\,(a\,\mathcal{R}\,a)$. That is, a binary relation on a set is reflexive if every element of the set is related to itself. This is true, for example, for the relation $\leq$ on the set $\mathbb{N}$, since $n \leq n$ for every $n \in \mathbb{N}$. On the other hand, it is not true for the relation $<$ on $\mathbb{N}$, since, for example, the statement $17 < 17$ is false.[8]

$\mathcal{R}$ is called ***transitive*** if $\forall a \in A$, $\forall b \in A$, $\forall c \in A\,\big((a\,\mathcal{R}\,b \wedge b\,\mathcal{R}\,c) \rightarrow (a\,\mathcal{R}\,c)\big)$. Transitivity allows us to 'chain together' two true statements $a\,\mathcal{R}\,b$ and $b\,\mathcal{R}\,c$, which are 'linked' by the b that occurs in each statement, to deduce that $a\,\mathcal{R}\,c$. For example, suppose P is the set of people, and define the relation $\mathcal{C}$ on P such that $x\,\mathcal{P}\,y$ if and only if x is a child of y. The relation $\mathcal{P}$ is not transitive because the child of a child of a person is not a child of that person. Suppose, on the other hand, that we define a relation $\mathcal{D}$ on P such that $x\,\mathcal{D}\,y$ if and only if x is a descendent of y. Then D is a transitive relation on the set of people, since a descendent of a descendent of a person is a descendent of that person. That is, from the facts that Elizabeth is a descendent of Victoria and Victoria is a descendent of James, we can deduce that Elizabeth is a descendent of James. In the mathematical world, the relations $\leq$ and $<$ on the set $\mathbb{N}$ are both transitive.

$\mathcal{R}$ is said to be ***symmetric*** if $\forall a \in A$, $\forall b \in B\,(a\,\mathcal{R}\,b \rightarrow b\,\mathcal{R}\,a)$. That is, whenever a is related to b, it follows that b is related to a. The relation 'is a first cousin of' on the set of people is symmetric, since whenever x is a first cousin of y, we have automatically that y is a first cousin of x. On the other hand, the 'child of' relation is certainly not symmetric. The relation $\leq$ on $\mathbb{N}$ is not symmetric. From the fact that $n \leq m$, we cannot conclude that $m \leq n$. It is true for *some* n and m in $\mathbb{N}$ that $n \leq m \rightarrow m \leq n$, but it is not true for *all* n and m in $\mathbb{N}$.

Finally, $\mathcal{R}$ is ***antisymmetric*** if $\forall a \in A$, $\forall b \in B\,\big((a\,\mathcal{R}\,b \wedge b\,\mathcal{R}\,a) \rightarrow a = b\big)$. The relation $\mathcal{R}$ is antisymmetric if for any two *distinct* elements x and y of A, we can't have both $x\,\mathcal{R}\,y$ and $y\,\mathcal{R}\,x$. The relation $\leq$ on $\mathbb{N}$ is antisymmetric because from the facts that $n \leq m$ and $m \leq n$, we can deduce that $n = m$. The relation 'child of' on the set of people is antisymmetric since it's impossible to have both that x is a child of y and y is a child of x.

During lectures, we'll think about how to draw relations graphically. See the figure in Section 4.4.3 for one kind of graphical depiction.

There are a few combinations of properties that define particularly useful types of binary relations. The relation $\leq$ on the set $\mathbb{N}$ is reflexive, antisymmetric, and transitive. These properties define what is called a partial order: a ***partial order*** on a set A is a binary relation on A that is reflexive, antisymmetric, and transitive.

Another example of a partial order is the subset relation, $\subseteq$, on the power set of any set. If X is a set, then of course $\mathcal{P}(X)$ is a set in its own right, and $\subseteq$ can be considered to be a binary relation on this set. Two elements A and B of $\mathcal{P}(X)$ are related by $\subseteq$ if and

[8] Note that to show that the relation $\mathcal{R}$ is *not* reflexive, you only need to find one a such that $a\,\mathcal{R}\,a$ is false. This follows from the fact that $\neg\big(\forall a \in A\,(a\,\mathcal{R}\,a)\big) \equiv \exists a \in A\,\big(\neg(a\,\mathcal{R}\,a)\big)$. A similar remark holds for each of the properties of relations that are discussed here.

only if $A \subseteq B$. This relation is reflexive since every set is a subset of itself. The fact that it is antisymmetric follows from Theorem 4.1. The fact that it is transitive was Exercise 11 in Section 4.1.

The ordering imposed on $\mathbb{N}$ by $\leq$ has one important property that the ordering of subsets by $\subseteq$ does not share. If n and m are natural numbers, then at least one of the statements $n \leq m$ and $m \leq n$ must be true. However, if A and B are subsets of a set X, it is certainly possible that both $A \subseteq B$ and $B \subseteq A$ are false. A binary relation $\mathcal{R}$ on a set A is said to be a ***total order*** if it is a partial order and furthermore for any two elements a and b of A, either $a \mathcal{R} b$ or $b \mathcal{R} a$. The relation $\leq$ on the set $\mathbb{N}$ is a total order. The relation $\subseteq$ on $\mathcal{P}(X)$ is not. (Note once again the slightly odd mathematical language: A total order is a kind of partial order—not, as you might expect, the opposite of a partial order.)

For another example of ordering, let L be the set of strings that can be made from lowercase letters. L contains both English words and nonsense strings such as "sxjja". There is a commonly used total order on the set L, namely alphabetical order.

4.7.2 Equivalence relations

We'll approach another important kind of binary relation indirectly, through what might at first appear to be an unrelated idea. Let A be a set. A ***partition*** of A is defined to be a collection of non-empty subsets of A such that each pair of distinct subsets in the collection is disjoint and the union of all the subsets in the collection is A. A partition of A is just a division of all the elements of A into non-overlapping subsets. For example, the sets $\{1, 2, 6\}$, $\{3, 7\}$, $\{4, 5, 8, 10\}$, and $\{9\}$ form a partition of the set $\{1, 2, \ldots, 10\}$. Each element of $\{1, 2, \ldots, 10\}$ occurs in exactly one of the sets that make up the partition. As another example, we can partition the set of all people into two sets, the set of males and the set of females. Biologists try to partition the set of all organisms into different species. Librarians try to partition books into various categories such as fiction, biography, and poetry. In the real world, classifying things into categories is an essential activity, although the boundaries between categories are not always well-defined. The abstract mathematical notion of a partition of a set models the real-world notion of classification. In the mathematical world, though, the categories are sets and the boundary between two categories is sharp.

In the real world, items are classified in the same category because they are related in some way. This leads us from partitions back to relations. Suppose that we have a partition of a set A. We can define a relation $\mathcal{R}$ on A by declaring that for any a and b in A, $a \mathcal{R} b$ if and only if a and b are members of the same subset in the partition. That is, two elements of A are related if they are in the same category. It is clear that the relation defined in this way is reflexive, symmetric, and transitive.

An ***equivalence relation*** is defined to be a binary relation that is reflexive, symmetric, and transitive. Any relation defined, as above, from a partition is an equivalence relation. Conversely, we can show that any equivalence relation defines a partition. Suppose that $\mathcal{R}$ is an equivalence relation on a set A. Let $a \in A$. We define the ***equivalence class*** of a

under the equivalence relation $\mathcal{R}$ to be the subset $[a]_{\mathcal{R}}$ defined as $[a]_{\mathcal{R}} = \{b \in A \mid b \mathcal{R} a\}$. That is, the equivalence class of a is the set of all elements of A that are related to a. In most cases, we'll assume that the relation in question is understood, and we'll write $[a]$ instead of $[a]_{\mathcal{R}}$. Note that each equivalence class is a subset of A. The following theorem shows that the collection of equivalence classes form a partition of A.

Theorem 4.12. *Let A be a set and let $\mathcal{R}$ be an equivalence relation on A. Then the collection of all equivalence classes under $\mathcal{R}$ is a partition of A.*

Proof. To show that a collection of subsets of A is a partition, we must show that each subset is non-empty, that the intersection of two distinct subsets is empty, and that the union of all the subsets is A.

If $[a]$ is one of the equivalence classes, it is certainly non-empty, since $a \in [a]$. (This follows from the fact that $\mathcal{R}$ is reflexive, and hence $a \mathcal{R} a$.) To show that A is the union of all the equivalence classes, we just have to show that each element of A is a member of one of the equivalence classes. Again, the fact that $a \in [a]$ for each $a \in A$ shows that this is true.

Finally, we have to show that the intersection of two distinct equivalence classes is empty. Suppose that a and b are elements of A and consider the equivalence classes $[a]$ and $[b]$. We have to show that if $[a] \neq [b]$, then $[a] \cap [b] = \emptyset$. Equivalently, we can show the converse: If $[a] \cap [b] \neq \emptyset$ then $[a] = [b]$. So, assume that $[a] \cap [b] \neq \emptyset$. Saying that a set is not empty just means that the set contains some element, so there must be an $x \in A$ such that $x \in [a] \cap [b]$. Since $x \in [a]$, $x \mathcal{R} a$. Since $\mathcal{R}$ is symmetric, we also have $a \mathcal{R} x$. Since $x \in [b]$, $x \mathcal{R} b$. Since $\mathcal{R}$ is transitive and since $(a \mathcal{R} x) \wedge (x \mathcal{R} b)$, it follows that $a \mathcal{R} b$.

Our goal is to deduce that $[a] = [b]$. Since $[a]$ and $[b]$ are sets, they are equal if and only if $[a] \subseteq [b]$ and $[b] \subseteq [a]$. To show that $[a] \subseteq [b]$, let c be an arbitrary element of $[a]$. We must show that $c \in [b]$. Since $c \in [a]$, we have that $c \mathcal{R} a$. And we have already shown that $a \mathcal{R} b$. From these two facts and the transitivity of $\mathcal{R}$, it follows that $c \mathcal{R} b$. By definition, this means that $c \in [b]$. We have shown that any member of $[a]$ is a member of $[b]$ and therefore that $[a] \subseteq [b]$. The fact that $[b] \subseteq [a]$ can be shown in the same way. We deduce that $[a] = [b]$, which proves the theorem. □

The point of this theorem is that if we can find a binary relation that satisfies certain properties, namely the properties of an equivalence relation, then we can classify things into categories, where the categories are the equivalence classes.

For example, suppose that U is a possibly infinite set. Define a binary relation $\sim$ on $\mathcal{P}(U)$ as follows: For X and Y in $\mathcal{P}(U)$, $X \sim Y$ if and only if there is a bijective function from the set X to the set Y. In other words, $X \sim Y$ means that X and Y have the same cardinality. Then $\sim$ is an equivalence relation on $\mathcal{P}(U)$. (The symbol $\sim$ is often used to denote equivalence relations. It is usually read 'is equivalent to'.) If $X \in \mathcal{P}(U)$, then the equivalence class $[X]_{\sim}$ consists of all the subsets of U that have the same cardinality as X. We have classified all the subsets of U according to their cardinality—even though

we have never said what an infinite cardinality *is*. (We have only said what it means to have the same cardinality.)

You might know the popular puzzle called Rubik's Cube[9], a cube made of smaller cubes with coloured sides that could be manipulated by twisting layers of little cubes. The object was to manipulate the cube so that the colours of the little cubes formed a certain configuration. Define two configurations of the cube to be equivalent if it's possible to manipulate one configuration into the other by a sequence of twists. This is, in fact, an equivalence relation on the set of possible configurations. (Symmetry follows from the fact that each move is reversible.) It has been shown that this equivalence relation has exactly twelve equivalence classes. The interesting fact is that it has more than one equivalence class: If the configuration that the cube is in and the configuration that you want to achieve are not in the same equivalence class, then you are doomed to failure.

Fortunately if you buy the cube in the configuration you want it to be in (i.e., with 6 faces all having just one colour), you can still be successful. Since all configurations you can get to using legal moves are in the same class, you can only move to another class of configurations with an illegal move. So it is only by taking the cube apart and incorrectly putting it back together that you can really change this puzzle from difficult to impossible.

Since we're talking about solving the Rubik's Cube, could you write a computer program to solve it? Given a cube in some initial state, what is the shortest sequence of moves to get to the solved state—or to prove it cannot be done? See Agostinelli, McAleer, Shmakov & Baldi, *Solving the Rubik's cube with deep reinforcement learning and search*, Nature Machine Intelligence (2019).

Suppose that $\mathcal{R}$ is a binary relation on a set A. Even though $\mathcal{R}$ might not be transitive, it is always possible to construct a transitive relation from $\mathcal{R}$ in a natural way. If we think of $a \mathcal{R} b$ as meaning that a is related by $\mathcal{R}$ to b 'in one step', then we consider the relationship that holds between two elements x and y when x is related by $\mathcal{R}$ to y 'in one or more steps'. This relationship defines a binary relation on A that is called the ***transitive closure*** of $\mathcal{R}$. The transitive closure of $\mathcal{R}$ is denoted $\mathcal{R}^*$. Formally, $\mathcal{R}^*$ is defined as follows: For a and b in A, $a \mathcal{R}^* b$ if there is a sequence $x_0, x_1, \ldots x_n$ of elements of A, where $n > 0$ and $x_0 = a$ and $x_n = b$, such that $x_0 \mathcal{R} x_1$, $x_1 \mathcal{R} x_2$, ..., and $x_{n-1} \mathcal{R} x_n$.

[9]Image: `commons.wikimedia.org/wiki/File:Rubiks_cube_scrambled.jpg`.

You will revist the notion of transitive closures in the course *Automata, Languages, and Computability*.

For example, if $a \mathcal{R} c$, $c \mathcal{R} d$, and $d \mathcal{R} b$, then we would have that $a \mathcal{R}^* b$. Of course, we would also have that $a \mathcal{R}^* c$, and $a \mathcal{R}^* d$.

For a practical example, suppose that C is the set of all cities and let $\mathcal{A}$ be the binary relation on C such that for x and y in C, $x \mathcal{A} y$ if there is a regularly scheduled airline flight from x to y. Then the transitive closure $\mathcal{A}^*$ has a natural interpretation: $x \mathcal{A}^* y$ if it's possible to get from x to y by a sequence of one or more regularly-scheduled airline flights. You'll find a few more examples of transitive closures in the exercises.

Exercises

1. For a finite set, it is possible to define a binary relation on the set by listing the elements of the relation, considered as a set of ordered pairs. Let A be the set $\{a, b, c, d\}$, where a, b, c, and d are distinct. Consider each of the following binary relations on A. Is the relation reflexive? Symmetric? Antisymmetric? Transitive? Is it a partial order? An equivalence relation?

a) $\mathcal{R} = \{(a,b), (a,c), (a,d)\}$.
b) $\mathcal{S} = \{(a,a), (b,b), (c,c), (d,d), (a,b), (b,a)\}$.
c) $\mathcal{T} = \{(b,b), (c,c), (d,d)\}$.
d) $\mathcal{C} = \{(a,b), (b,c), (a,c), (d,d)\}$.
e) $\mathcal{D} = \{(a,b), (b,a), (c,d), (d,c)\}$.

2. Let A be the set $\{1,2,3,4,5,6\}$. Consider the partition of A into the subsets $\{1,4,5\}$, $\{3\}$, and $\{2,6\}$. Write out the associated equivalence relation on A as a set of ordered pairs.

3. Consider each of the following relations on the set of people. Is the relation reflexive? Symmetric? Transitive? Is it an equivalence relation?

a) x is related to y if x and y have the same biological parents.
b) x is related to y if x and y have at least one biological parent in common.
c) x is related to y if x and y were born in the same year.
d) x is related to y if x is taller than y.
e) x is related to y if x and y have both visited Indonesia.

4. It is possible for a relation to be both symmetric and antisymmetric. For example, the equality relation, $=$, is a relation on any set which is both symmetric and antisymmetric. Suppose that A is a set and $\mathcal{R}$ is a relation on A that is both symmetric and antisymmetric. Show that $\mathcal{R}$ is a subset of $=$ (when both relations are considered as sets of ordered pairs). That is, show that for any a and b in A, $(a \mathcal{R} b) \rightarrow (a = b)$.

5. Let $\sim$ be the relation on $\mathbb{R}$, the set of real numbers, such that for x and y in $\mathbb{R}$, $x \sim y$ if and only if $x - y \in \mathbb{Z}$. For example, $\sqrt{2} - 1 \sim \sqrt{2} + 17$ because the difference, $(\sqrt{2} - 1) -$

$(\sqrt{2}+17)$, is -18, which is an integer. Show that $\sim$ is an equivalence relation. Show that each equivalence class $[x]_\sim$ contains exactly one number a which satisfies $0 \leq a < 1$. (Thus, the set of equivalence classes under $\sim$ is in one-to-one correspondence with the half-open interval $[0,1)$.)

6. Let A and B be any sets, and suppose $f\colon A \to B$. Define a relation $\sim$ on B such that for any x and y in A, $x \sim y$ if and only if $f(x) = f(y)$. Show that $\sim$ is an equivalence relation on A.

7. Let $\mathbb{Z}^+$ be the set of positive integers $\{1, 2, 3, \dots\}$. Define a binary relation $\mathcal{D}$ on $\mathbb{Z}^+$ such that for n and m in $\mathbb{Z}^+$, $n\,\mathcal{D}\,m$ if n divides evenly into m, with no remainder. Equivalently, $n\,\mathcal{D}\,m$ if n is a factor of m, that is, if there is a k in $\mathbb{Z}^+$ such that $m = nk$. Show that $\mathcal{D}$ is a partial order.

8. Consider the set $\mathbb{N} \times \mathbb{N}$, which consists of all ordered pairs of natural numbers. Since $\mathbb{N} \times \mathbb{N}$ is a set, it is possible to have binary relations on $\mathbb{N} \times \mathbb{N}$. Such a relation would be a subset of $(\mathbb{N} \times \mathbb{N}) \times (\mathbb{N} \times \mathbb{N})$. Define a binary relation $\preceq$ on $\mathbb{N} \times \mathbb{N}$ such that for (m,n) and (k,ℓ) in $\mathbb{N} \times \mathbb{N}$, $(m,n) \preceq (k,\ell)$ if and only if either $m < k$ or $((m = k) \wedge (n \leq \ell))$. Which of the following are true?

a) $(2,7) \preceq (5,1)$ **b)** $(8,5) \preceq (8,0)$
c) $(0,1) \preceq (0,2)$ **d)** $(17,17) \preceq (17,17)$

Show that $\preceq$ is a total order on $\mathbb{N} \times \mathbb{N}$.

9. Let $\sim$ be the relation defined on $\mathbb{N} \times \mathbb{N}$ such that $(n,m) \sim (k,\ell)$ if and only if $n + \ell = m + k$. Show that $\sim$ is an equivalence relation.

10. Let P be the set of people and let $\mathcal{C}$ be the 'child of' relation. That is $x\,\mathcal{C}\,y$ means that x is a child of y. What is the meaning of the transitive closure $\mathcal{C}^*$? Explain your answer.

11. Let $\mathcal{R}$ be the binary relation on $\mathbb{N}$ such that $x\,\mathcal{R}\,y$ if and only if $y = x + 1$. Identify the transitive closure $\mathcal{R}^*$. (It is a well-known relation.) Explain your answer.

12. Suppose that $\mathcal{R}$ is a reflexive, symmetric binary relation on a set A. Show that the transitive closure $\mathcal{R}^*$ is an equivalence relation.

4.8 Application: Relational Databases

One of the major uses of computer systems is to store and manipulate collections of data. A ***database*** is a collection of data that has been organized so that it is possible to add and delete information, to update the data that it contains, and to retrieve specified parts of the data. A ***Database Management System***, or DBMS, is a computer program that makes it possible to create and manipulate databases. A DBMS must be able to accept and process commands that manipulate the data in the databases that it manages. These commands are called ***queries***, and the languages in which they are written are called ***query languages***. A query language is a kind of specialized programming language.

There are many different ways that the data in a database could be represented. Different DBMS's use various data representations and various query languages. However, data is most commonly stored in relations. A relation in a database is a relation in the mathematical sense. That is, it is a subset of a cross product of sets. A database that stores its data in relations is called a ***relational database***. The query language for most relational database management systems is some form of the language known as ***Structured***

Query Language, or SQL. In this section, we'll take a very brief look at SQL, relational databases, and how they use relations.

You'll learn more about databases in the courses *Web & Database Technology* and *Information & Data Management*.

A relation is just a subset of a cross product of sets. Since we are discussing computer representation of data, the sets are data types. As in Section 4.5, we'll use data type names such as *int* and *string* to refer to these sets. A relation that is a subset of the cross product $int \times int \times string$ would consist of ordered 3-tuples such as (17, 42, "hike"). In a relational database, the data is stored in the form of one or more such relations. The relations are called tables, and the tuples that they contain are called rows or records.

As an example, consider a lending library that wants to store data about its members, the books that it owns, and which books the members have out on loan. This data could be represented in three tables, as illustrated in Figure 4.8. The relations are shown as tables rather than as sets of ordered tuples, but each table is, in fact, a relation. The rows of the table are the tuples. The `Members` table, for example, is a subset of $int \times string \times string \times string$, and one of the tuples is (1782, "Smit, Johan", "107 Main St", "New York, NY"). A table does have one thing that ordinary relations in mathematics do not have. Each column in the table has a name. These names are used in the query language to manipulate the data in the tables.

The data in the `Members` table is the basic information that the library needs in order to keep track of its members, namely the name and address of each member. A member also has a `MemberID` number, which is presumably assigned by the library. Two different members can't have the same `MemberID`, even though they might have the same name or the same address. The `MemberID` acts as a ***primary key*** for the `Members` table. A given value of the primary key uniquely identifies one of the rows of the table. Similarly, the `BookID` in the `Books` table is a primary key for that table. In the `Loans` table, which holds information about which books are out on loan to which members, a `MemberID` unambiguously identifies the member who has a given book on loan, and the `BookID` says unambiguously which book that is. Every table has a primary key, but the key can consist of more than one column. The DBMS enforces the uniqueness of primary keys. That is, it won't let users make a modification to the table if it would result in two rows having the same primary key.

The fact that a relation is a set—a set of tuples—means that it can't contain the same tuple more than once. In terms of tables, this means that a table shouldn't contain two identical rows. But since no two rows can contain the same primary key, it's impossible for two rows to be identical. So tables are in fact relations in the mathematical sense.

The library must have a way to add and delete members and books and to make a record when a book is borrowed or returned. It should also have a way to change the

address of a member or the due date of a borrowed book. Operations such as these are performed using the DBMS's query language. SQL has commands named `INSERT`, `DELETE`, and `UPDATE` for performing these operations. The command for adding Barack Obama as a member of the library with `MemberID` 999 would be

```
INSERT INTO Members
VALUES (999, "Barack Obama",
             "1600 Pennsylvania Ave", "Washington, DC")
```

When it comes to deleting and modifying rows, things become more interesting because it's necessary to specify which row or rows will be affected. This is done by specifying a condition that the rows must fulfill. For example, this command will delete the member with ID 4277:


```
DELETE FROM Members
WHERE MemberID = 4277
```

It's possible for a command to affect multiple rows. For example,

```
DELETE FROM Members
WHERE Name = "Smit, Johan"
```

would delete every row in which the name is "Smit, Johan". The update command also specifies what changes are to be made to the row:

```
UPDATE Members
SET Address="19 South St", City="Hartford, CT"
WHERE MemberID = 4277
```

Of course, the library also needs a way of retrieving information from the database. SQL provides the `SELECT` command for this purpose. For example, the query

```
SELECT Name, Address
FROM Members
WHERE City = "New York, NY"
```

asks for the name and address of every member who lives in New York City. The last line of the query is a condition that picks out certain rows of the "Members" relation, namely all the rows in which the `City` is "New York, NY". The first line specifies which data from those rows should be retrieved. The data is actually returned in the form of a table. For example, given the data in Figure 4.8, the query would return this table:

Smit, Johan	107 Main St
Jones, Mary	1515 Center Ave
Lee, Joseph	90 Park Ave
De Jong, Sally	89 Main St

The table returned by a SELECT query can even be used to construct more complex queries. For example, if the table returned by SELECT has only one column, then it can be used with the IN operator to specify any value listed in that column. The following query will find the BookID of every book that is out on loan to a member who lives in New York City:

```
SELECT BookID
FROM Loans
WHERE MemberID IN (SELECT MemberID
                   FROM Members
                   WHERE City = "New York, NY")
```

More than one table can be listed in the FROM part of a query. The tables that are listed are joined into one large table, which is then used for the query. The large table is essentially the cross product of the joined tables, when the tables are understood as sets of tuples. For example, suppose that we want the titles of all the books that are out on loan to members who live in New York City. The titles are in the Books table, while information about loans is in the Loans table. To get the desired data, we can join the tables and extract the answer from the joined table:

```
SELECT Title
FROM Books, Loans
WHERE MemberID IN (SELECT MemberID
                   FROM Members
                   WHERE City = "New York, NY")
```

In fact, we can do the same query without using the nested SELECT. We need one more bit of notation: If two tables have columns that have the same name, the columns can be named unambiguously by combining the table name with the column name. For example, if the Members table and Loans table are both under discussion, then the MemberID columns in the two tables can be referred to as Members.MemberID and Loans.MemberID. So, we can say:

```
SELECT Title
FROM Books, Loans
WHERE City ="New York, NY"
      AND Members.MemberID = Loans.MemberID
```

This is just a sample of what can be done with SQL and relational databases. The conditions in WHERE clauses can get very complicated, and there are other operations besides the cross product for combining tables. The database operations that are needed to complete a given query can be complex and time-consuming. Before carrying out a query, the DBMS tries to optimize it. That is, it manipulates the query into a form that can be carried out most efficiently. The rules for manipulating and simplifying queries form an *algebra* of relations, and the theoretical study of relational databases is in large part the study of the algebra of relations.

Exercises

1. Using the library database from Figure 4.8, what is the result of each of the following SQL commands?

a)
```
SELECT Name, Address
FROM Members
WHERE Name = "Smit, Johan"
```

b)
```
DELETE FROM Books
WHERE Author = "Isaac Asimov"
```

c)
```
UPDATE Loans
SET DueDate = "20 November"
WHERE BookID = 221
```

d)
```
SELECT Title
FROM Books, Loans
WHERE Books.BookID = Loans.BookID
```

e)
```
DELETE FROM Loans
WHERE MemberID IN (SELECT MemberID
                   FROM Members
                   WHERE Name = "Lee, Joseph")
```

2. Using the library database from Figure 4.8, write an SQL command to do each of the following database manipulations:
- **a**) Find the `BookID` of every book that is due on 1 November 2010.
- **b**) Change the `DueDate` of the book with `BookID` 221 to 15 November 2010.
- **c**) Change the `DueDate` of the book with title "Summer Lightning" to 14 November 2010. Use a nested `SELECT`.
- **d**) Find the name of every member who has a book out on loan. Use joined tables in the `FROM` clause of a `SELECT` command.

3. Suppose that a university colleges wants to use a database to store information about its students, the courses that are offered in a given term, and which students are taking which courses. Design tables that could be used in a relational database for representing this data. Then write SQL commands to do each of the following database manipulations. (You should design your tables so that they can support all these commands.)
- **a**) Enroll the student with ID number 1928882900 in "English 260".
- **b**) Remove "Johan Smit" from "Biology 110".
- **c**) Remove the student with ID number 2099299001 from every course in which that student is enrolled.
- **d**) Find the names and addresses of the students who are taking "Computer Science 229".
- **e**) Cancel the course "History 101".

Members

MemberID	Name	Address	City
1782	Smit, Johan	107 Main St	New York, NY
2889	Jones, Mary	1515 Center Ave	New York, NY
378	Lee, Joseph	90 Park Ave	New York, NY
4277	Smit, Johan	2390 River St	Newark, NJ
5704	De Jong, Sally	89 Main St	New York, NY

Books

BookID	Title	Author
182	I, Robot	Isaac Asimov
221	The Sound and the Fury	William Faulkner
38	Summer Lightning	P.G. Wodehouse
437	Pride and Prejudice	Jane Austen
598	Left Hand of Darkness	Ursula LeGuin
629	Foundation Trilogy	Isaac Asimov
720	The Amber Spyglass	Philip Pullman

Loans

MemberID	BookID	DueDate
378	221	8 October 2010
2889	182	1 November 2010
4277	221	1 November 2010
1782	38	30 October 2010

Figure 4.8: *Tables that could be part of a relational database. Each table has a name, shown above the table. Each column in the table also has a name, shown in the top row of the table. The remaining rows hold the data.*

Chapter 5

Looking Beyond

Coming to the right conclusion has been the theme of this book. We learned how to represent statements formally in Chapter 2, and how to prove or disprove statements in Chapter 3. We looked at more important foundational parts of computer science in Chapter 4: sets, functions and relations.

Last chapter, I said that the foundations of mathematics are in "a bit of a philosophical muddle". That was at the end of our discussion about counting past infinity (Section 4.6). The questions from the work of Cantor, Russell and others became more profound in the 1930s thanks to Kurt Gödel, whom we mentioned briefly in Chapter 3. All this just before practical computers were invented—yes, the theoretical study of computing began before digital computers existed!

Since this book is about the foundations of computation, let's say a little more about Gõdel and his contemporary, Alan Turing.

Gödel published his two ***incompleteness theorems*** in 1931. The first incompleteness theorem states that for any self-consistent recursive axiomatic system powerful enough to describe the arithmetic of the natural numbers[1], there are true propositions about the naturals that cannot be proved from the axioms. In other words, in any formal system of logic, there will always be statements that you can never prove nor disprove: you don't know.

These two theorems ended a half-century of attempts, beginning with the work of Frege and culminating in the work of Russell and others, to find a set of axioms sufficient for all mathematics.

[1] For example, Peano arithmetic, which is a recursive definition of $\mathbb{N}$: see `en.wikipedia.org/wiki/Peano_axioms`.

John von Neumann, one of the pioneers of the first computers, wrote "Kurt Gödel's achievement in modern logic is singular and monumental—indeed it is more than a monument, it is a landmark which will remain visible far in space and time." John von Neumann was a brilliant Hungarian-American mathematician, physicist and computer scientist. Among many other things, he invented the von Neumann architecture (familiar from *Computer Organisation*?) and is called the 'midwife' of the modern computer.

Source: `en.wikipedia.org/wiki/Kurt_Gödel` and `en.wikipedia.org/wiki/John_von_Neumann`.

Around the same time, one of the early models of computation was developed by the British mathematician, Alan Turing. Turing was interested in studying the theoretical abilities and limitations of computation. (This before the first computers! Von Neumann knew Turing and emphasized that "the fundamental conception [of the modern computer] is owing to Turing" and not to himself.[2]) His model for computation is a very simple abstract computing machine which has come to be known as a ***Turing machine***. While Turing machines are not very practical, their use as a fundamental model for computation means that every computer scientist should be familiar with them, at least in a general way.[3] You'll learn about them in *Automata, Computability and Complexity*.

We can also use Turing machines to define 'computable languages'. The idea is that anything that it is possible to compute, you can compute using a Turing machine (just very slowly). Turing, with American mathematician Alonzo Church, made a hypothesis about the nature of computable functions.[4] It states that a function on the natural numbers is computable by a human being following an algorithm (ignoring resource limitations), if and only if it is computable by a Turing machine.

So Gödel established that there are some things we can never tell whether they are true or false, and Turing established a computational model for that the things we can

[2] A quote from von Neumann's colleague Frankel. See `en.wikipedia.org/wiki/Von_Neumann_architecture`.

[3] In fact, the Brainf*ck programming language we mentioned earlier is not much more than a Turing machine.

[4] It's not a theorem because it is not proved, but all theoretical computer scientists believe it to be true. Between them, Church and Turing did prove that a function is λ-computable if and only if it is Turing computable if and only if it is general recursive. Another thesis that is widely believed but not proved is: for the things we can compute, there is a difference between those that need only polynomial time (in the size of the input) for the computation and those that need more than polynomial time. More about that in *Automata, Computability and Complexity*.

compute.

Is there a link between these results? The ***halting problem*** is to determine, given a program and an input to the program, whether the program will eventually halt when run with that input. Turing proved in 1936 that a general algorithm to solve the halting problem for all possible program-input pairs cannot exist.

We end the book with the idea of the proof:

Proof. Consider a model of computation, such a Turing machine. For any program f that might determine if programs halt, construct a 'pathological' program g as follows. When called with an input, g passes its own source and its input to f, and when f returns an output (halt/not), g then specifically does the opposite of what f predicts g will do. No f can exist that handles this case. □

Selected Solutions

Attempt the exercise first before looking at the solution!

Solutions 2.1

2.

a)

p	q	A $\overbrace{p \to q}$	B $\overbrace{p \wedge (A)}$	$(B) \to q$
0	0	1	0	1
0	1	1	0	1
1	0	0	0	1
1	1	1	1	1

Since the proposition is always true, the proposition is a tautology.

b)

p	q	r	A $\overbrace{p \to q}$	B $\overbrace{q \to r}$	C $\overbrace{(A) \wedge (B)}$	D $\overbrace{p \to r}$	$(C) \to (D)$
0	0	0	1	1	1	1	1
0	0	1	1	1	1	1	1
0	1	0	1	0	0	1	1
0	1	1	1	1	1	1	1
1	0	0	0	1	0	0	1
1	0	1	0	1	0	1	1
1	1	0	1	0	0	0	1
1	1	1	1	1	1	1	1

Since the proposition is always true, the proposition is a tautology.

c)

p	$p \wedge \neg p$
0	0
1	0

Since the proposition is always false, the proposition is a contradiction.

d)

p	q	A $p \vee q$	B $p \wedge q$	$(A) \rightarrow (B)$
0	0	0	0	1
0	1	1	0	0
1	0	1	0	0
1	1	1	1	1

Since the proposition is sometimes true and sometimes false, the proposition is a contingency.

e)

p	$p \vee \neg p$
0	1
1	1

Since the proposition is always true, the proposition is a tautology.

f)

p	q	A $p \wedge q$	B $p \vee q$	$(A) \rightarrow (B)$
0	0	0	0	1
0	1	0	1	1
1	0	0	1	1
1	1	1	1	1

Since the proposition is always true, the proposition is a tautology.

4.

p	q	r	A $p \rightarrow q$	B $A \rightarrow r$	C $q \rightarrow r$	D $p \rightarrow D$
0	0	0	1	0	1	1
0	0	1	1	1	1	1
0	1	0	1	0	0	1
0	1	1	1	1	1	1
1	0	0	0	1	1	1
1	0	1	0	1	1	1
1	1	0	1	0	0	0
1	1	1	1	1	1	1

Since the truth tables for the expressions (columns B and D) are different, they are not equivalent. As our counterexample take for instance: $p = q = r = 0$. Thus $\rightarrow$ is not associative.
What about $\leftrightarrow$?

6. The four propositions are:

a) Galileo was not falsely accused and the Earth is the centre of the universe.
b) If the Earth moves then the Earth is not the centre of the universe.
c) The Earth moves if and only if the Earth is not the centre of the universe.
d) If the Earth moves the Galileo was falsely accused, but if the Earth is the centre of the universe then Galileo was not falsely accused.

Solutions 2.2

4. It is not. Take for example $p = q = 0$. In this case $\neg(p \leftrightarrow q)$ is false, but $(\neg p) \leftrightarrow (\neg q)$ is true. Try to see if you can find a more simplified expression that $(\neg p) \leftrightarrow (\neg q)$ is equivalent to.

10. Verify the following answers with truth tables yourself!

a) $q \rightarrow p$ **b)** $\mathbb{F}$ **c)** $\neg p$
d) $\neg p \wedge q$ **e)** $\mathbb{T}$ **f)** q

11. When translating try make the English sentences flow a bit without adding in more constraints. For example by using the word 'but' rather than 'and' in two of the examples below.

a) It is not sunny or it is not cold.
b) I will have neither stroopwafels nor appeltaart.
c) It is Tuesday today, but this is not Belgium.
d) You passed the final exam, but you did not pass the course.

5

Solutions 2.4

1.

a) $\exists x(P(x))$ **b)** $\forall x(\neg P(x) \vee \neg Q(x))$
c) $\exists z(P(z) \wedge \neg Q(z))$ **d)** $(\exists x(\neg P(x))) \vee (\exists y(\neg Q(y)))$
e) $\exists x \forall y \neg P(x, y)$ **f)** $\forall x(\neg R(x) \vee \exists y \neg S(x, y))$
g) $\forall y((\neg P(y) \wedge Q(y)) \vee (P(y) \wedge \neg Q(y)))$ **h)** $\exists x(P(x) \wedge (\forall y \neg Q(x, y)))$

8. We use the predicates $Ball(x)$ for x is a ball and $Have(x, y)$ for x must have y. We also use a constant *you* to represent you.

$$\exists x(Ball(x) \wedge \forall y(x \neq y \rightarrow \neg Ball(y)) \wedge Have(you, x))$$

13. Using the predicates $Person(x)$ for x is a person, $Question(x)$ for x is a question, $Answer(x)$ for x is an answer, and $Has(x, y)$ for x has y. Two different interpretations are:

$$\exists x(Person(x) \wedge \forall y(Question(y) \rightarrow (\exists z(Answer(z) \wedge Has(x, z)))))$$

In other words, there is a single person who has the answers to all questions.

$$\forall x(Question(x) \rightarrow \exists y \exists z(Person(y) \wedge Answer(z) \wedge Has(y, z)))$$

In other words, every question has an answer and some person knows it (but different people might know the answer to different questions).

Solutions 3.2

5. The claim is: $(r \mid s) \wedge (s \mid t) \rightarrow r \mid t$.

Proof. We need to show something is true for all integers r, s, t, so take arbitrary integers k, m, n such that: $k \mid m, m \mid n$.
Now we need to prove that $k \mid n$ holds.
Since $k \mid m$, we know that $m = ak$ for some integer a. Similarly $n = bm$ for some integer b.
Thus $n = bm = bak = ck$ with integer $c = ba$.
Thus $k \mid n$.
Since k, m, n were arbitrary, this holds for all integers. □

Solutions 3.3

3.

Proof. Assume that every hole has at most one pidgeon in it. This means that there are $< k$ pidgeons in total. Since $n > k$ this forms a contradiction. Therefore our assumption that eery hole has at most one pidgeon is incorrect and there must be at least one hole that has two or more pidgeons. □

(Take care to flip the quantifiers correctly when doing a proof by contradiction! $\neg\forall x(\dots)$ becomes $\exists x(\neg \dots)$.)

Solutions 2.4

8.

a) $\sum_{i=0}^{9} (2 * i + 1)$

b) $\sum_{i=0}^{6} (\frac{1}{3^i})$

c) $\sum_{i=50}^{100} (i)$

d) $\sum_{i=1}^{10} (i^2)$

e) $\sum_{i=2}^{99} (\frac{1}{2^i})$

Solutions 4.1

2.

a) $A \cup B = \{a, b, c\}; A \cap B = \emptyset; A \setminus B = \{a, b, c\}$
b) $A \cup B = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}; A \cap B = \emptyset; A \setminus B = \{1, 2, 3, 4, 5\}$
c) $A \cup B = \{a, b, c, d\}; A \cap B = \{a, b\}; A \setminus B = \emptyset$
d) $A \cup B = \{a, b, \{a\}, \{a, b\}\}; A \cap B = \{\{a, b\}\}; A \setminus B = \{a, b\}$

9. $A \cup A = A$ (remember that sets have no duplicates!)
$A \cap A = A$ (after all everything in A that is also in A, is everything!)
$A \setminus A = \emptyset$ (removing from A everything that is in A leaves us nothing!)

Couldn't find your answer here? Feel free to submit your own for future editions of the book here: `https://gitlab.ewi.tudelft.nl/reasoning_and_logic/book_solutions`. We will add your name to the list of contributors for the book if we accept your answers!

Contributors to Solutions

Your name here?

Further Reading

The books by Christian & Griffiths, Hofstadter and Smullyan are on recreational maths. Hofstadter's is a winner of the Pulitzer Prize. Besides Dowek, the other titles are textbooks; Hammack's book is an Open Textbook, just like this book is.

[1] Christian, B. and Griffiths, T. (2016). *Algorithms to Live By: The Computer Science of Human Decisions*. New York, NY: Henry Holt and Co.

[2] Critchlow, C. and Eck, D. (2011). *Foundations of Computation*, version 2.3.1. Geneva, NY: Hobart and William Smith Colleges. `math.hws.edu/FoundationsOfComputation/`

[3] Dowek, G. (2015). *Computation, Proof, Machine: Mathematics Enters a New Age*. New York, NY: Cambridge University Press. Original in French: *Les Métamorphoses du calcul* (2007).

[4] Epp, S. S. (2019). *Discrete Mathematics with Applications*, 5th edition. Boston, MA: Cengage Learning.

[5] Grassmann, W. K. and Tremblay, J.-P. (1996). *Logic and Discrete Mathematics*. Upper Saddle River, NJ: Prentice-Hall.

[6] Hammack, R. (2018). *Book of Proof*, 3rd edition. Richmond, VA: Virginia Commonwealth University. `www.people.vcu.edu/~rhammack/BookofProof3/`

[7] Hofstadter, D. (1979). *Gödel, Escher, Bach: An Eternal Golden Braid*. New York, NY: Basic Books.

[8] Huth, M. and Ryan, M. (2004). *Logic in Computer Science*, 2nd edition. Cambridge, UK: Cambridge University Press.

[9] Smullyan, R. M. (1979). *What is the Name of This Book?* Upper Saddle River, NJ: Prentice-Hall.

Index

Delftse Foundations of Computation

Stefan Hugtenburg and Neil Yorke-Smith

Delftse Foundations of Computation is a textbook for a one quarter introductory course in theoretical computer science. It includes topics from propositional and predicate logic, proof techniques, set theory and the theory of computation, along with practical applications to computer science. It has no prerequisites other than a general familiarity with computer programming.

Stefan Hugtenburg
TU Delft | Faculty of Electrical Engineering, Mathematics and Computer Science: Software Technology, Distributed Systems

Stefan Hugtenburg holds a MSc in Computer Science from the Delft University of Technology, where he now teaches in the undergraduate Computer Science and Engineering programme. He is involved in all courses of the Algorithmics track in the curriculum, starting with this book and the course Reasoning & Logic, up until the final year course Complexity Theory.

Neil Yorke-Smith
TU Delft | Faculty of Electrical Engineering, Mathematics and Computer Science: Software Technology, Algorithmics

Neil Yorke-Smith is an Associate Professor of Algorithmics in the Faculty of Electrical Engineering, Mathematics and Computer Science at the Delft University of Technology. His research focuses on intelligent decision making in complex socio-technical situations, with a particular current interest in agent-based methodologies and behavioural factors in automated planning and scheduling. He teaches Reasoning & Logic and graduate courses in Artificial Intelligence.

TU Delft

ISBN 978-94-6366-083-9
DOI https://doi.org/10.5074/t.isbn.9789463660839

textbooks.open.tudelft.nl