Course notes for CSC B36/236/240

INTRODUCTION TO THE THEORY OF COMPUTATION

Vassos Hadzilacos

Department of Computer Science
University of Toronto

To Stella Maria,
whose arrival I was eagerly anticipating
when I started writing these notes;
and to Pat,
who brought her.

ACKNOWLEDGEMENTS

Allan Borodin, Danny Heap and Charlie Rackoff read early drafts of these notes and provided very thoughtful and helpful feedback, as well as corrections. Over the years, I have also received corrections from Dimitris Achlioptas, Nick Cheng, George Giakkoupis, Puveindra Gunaratnam, Linh Ho, Zahra Kassam, Zongpeng Li, James MacLean, Lucia Moura, Gerald Penn, Eric Ruppert, and Anh Tran. Naturally, I am responsible (or perhaps I should say irresponsible) for all remaining errors.

My first exposure to formal languages and automata was through Hopcroft and Ullman's superb *Formal languages and their relation to automata* (Addison Wesley, 1969). My knowledge of the subject was later enhanced by two other excellent textbooks: Lewis and Papadimitriou's *Elements of the theory of computation* (Prentice Hall, 1981); and Sipser's *Introduction to the theory of computation* (PWS Publishing, 1997). All three of these sources have influenced the presentation of the material in Chapters 7 and 8.

These notes are an on-going project, and I will be grateful for feedback and criticism from readers. I can be contacted by email at `vassos@cs.utoronto.ca`.

Vassos Hadzilacos
Toronto, November 1998; July 2006

Preface

Beauty is truth, truth beauty, that is all
Ye know on earth, and all ye need to know
–John Keats

All intellectual pursuits seek to help us understand reality — i.e., the truth. Even endeavours intended to appeal to our aesthetic sense are ultimately after truth; this is the point of Keats' first assertion — that beauty is truth.

Mathematics is one of the most successful intellectual tools that human culture has developed in its quest for investigating reality. At first this appears paradoxical: unlike the physical and life sciences the subject matter of mathematics is not reality *per se*. Mathematics, on its own, cannot tell us that the mass of an object in motion affects the force it exerts or that the DNA is structured as a double helix, even though mathematics is crucial in the precise description of both these facts. Mathematics deals with abstract constructions of the human mind, such as numbers, sets, relations and functions.

Two elements of mathematics are the source of its success as a tool for the investigation of reality: its *techniques* and its *methodology*. Accordingly, mathematics is a tool in two interrelated but distinct ways. First, it is a tool in the straightforward sense in which *a hammer* is a tool. We can use mathematics to get a job done: calculate the volume of a complicated shape, determine the trajectory of an object given the forces acting on it, etc. In this role, mathematics provides us with techniques for performing computational tasks. Second, mathematics is also a tool in the less tangible sense in which *planning* is a tool. It is hard to point to a specific feature of a house attributable to the planning of its construction — certainly harder than pointing to a nail whose presence is attributable to the use of the hammer — but there is no doubt that without planning no house would be built. In this subtler role, mathematics provides us not with computational techniques but with a particular methodology.

The methodology of mathematics has many aspects; key among them are:

- An explicit and unambiguous acknowledgement of our assumptions.
- A small set of simple but powerful "rules of thought" through which we can reach conclusions that *necessarily* follow from our assumptions. The transition from assumptions to conclusions, through an unassailable process, is the essence of mathematical proof.
- An inclination to abstraction. In mathematics we ignore the coincidental and focus on the germane. Thus, for example, in mathematics we study numbers regardless of what quantity

they represent, and functions regardless of what relationship they capture. The virtue of abstraction is that it broadens the applicability of our results.
- An inclination to frugality. The mathematical methodology frowns upon unnecessary assumptions. A mathematical proof should be as direct and simple as possible. Mathematical constructions should be as efficient as possible, according to well-defined and meaningful notions of efficiency.

The two aspects of the "mathematical enterprise" — mathematics as a set of computational techniques and mathematics as a methodology of precise and elegant reasoning — are not counterposed. They go hand-in-hand. Neither can be properly appreciated and mastered without the other. In my opinion there has been a trend in the teaching of mathematics (at least in North America north of the Rio Grande) to overemphasise the former at the expense of the latter. This emphasis is often justified in the name of "practicality" or "relevance". In my view this choice, well-meaning as it may be, is seriously misguided. Learning computational techniques is, of course, necessary not only because of their immediate applications but also because without them there is no mathematics. But I believe that the ability for precise, rigorous and abstract reasoning is more important than the mastery of particular computational techniques. It is also relevant to a wider spectrum of tasks than computational techniques are.

The emphasis on computational techniques at the expense of exposing students to the methodology of mathematics is prevalent at all stages of mathematics education, with the exception of the way the subject is taught to mathematics majors at the university level. For reasons I outlined above, I believe that this is unfortunate for all students. It is particularly unfortunate for computer science students, the audience for which these notes are intended, because the methodology of mathematics is especially relevant to computer science. Abstraction, precise reasoning, generality, efficiency: these are the tools of the trade, and I know of no better way of sharpening one's mastery of them than through the study of mathematics.

In these notes I have tried to present the subject matter in what I believe is a balanced way, paying attention not only to the techniques that students need to master but also, and perhaps more importantly, to exposing them to what I have been calling the methodology of mathematics. It is my hope that the notes will help students gain an understanding, and even a working knowledge, of mathematics as a tool for investigating the truth. To the extent the notes are successful in this, I believe that they will also shed some light to the second point of Keats' couplet quoted at the start of the preface — that truth is beauty.

Contents

Chapter 0

PRELIMINARIES

In this chapter we have collected a number of basic mathematical concepts and definitions which we assume you have encountered before, in one form or another. It is probably wise to review this material so that you familiarise yourself with the terminology we will be using in this course.

0.1 Sets

A set is a collection of objects. Admittedly, this is not much of a definition: it defines the word ‘set’ in terms of another word, ‘collection’, which is no more precise or clear than the first one. Unfortunately, we can’t do better than this: the concept of a set is so elementary that it does not admit further definition. It is (hopefully) a self-evident concept that we will take for granted. The same is true about other elementary concepts that we encounter in various areas of mathematics. For example, in arithmetic we take the concept of number for granted; in geometry we take concepts such as point and line for granted.

The nature of the objects in a set is irrelevant: they may be concrete things (like players of a soccer club, items on sale in the supermarket, planets of the solar system, books in a library etc) or abstract things (like numbers, mathematical functions, or even sets).

The objects that comprise a set are the set’s ***elements***. If an object a is an element of set A, we say that a is ***in*** A, that a ***belongs*** to A, or that a is a ***member of*** A; we denote this fact as $a \in A$. If an object a does not belong to set A, we write $a \notin A$.

The collection that contains no elements at all is also considered a set, called the ***empty*** or ***null*** set, and denoted as $\emptyset$.

The number of elements in a set A, denoted $|A|$, is called its ***cardinality*** or its ***size***. If A has a finite number of elements, then $|A|$ is a nonnegative integer (that number); if A has an infinite number of elements, we write $|A| = \infty$. Regarding the symbol ∞, we adopt the convention that, for every integer k, $k < \infty$, and $\infty \leq \infty$.[1] The cardinality of the empty set

[1] The use of ∞ to denote the cardinalities of *all* infinite sets is an oversimplification. An important part of set theory is devoted to studying how to compare the cardinalities of different infinite sets. Roughly speaking, two sets A and B are said to have the same cardinality, if we can set up an ***one-to-one correspondence*** between

is 0; i.e., $|\emptyset| = 0$.

0.1.1 Describing sets

We can describe a set in one of two ways: by listing its elements explicitly (this is called an ***extensional*** description), or by stating a property that characterises its elements (this is called an ***intentional*** description).

To write an extensional description of a set, we list the elements of the set separated by commas, and enclose this list in curly brackets, as in: $\{1, 3, 5, 7, 9\}$. Since a set is simply a collection of objects, the only thing that matters about a set is which objects belong to it, and which do not. In particular, there is no such thing as "the first element of the set". Thus, the order in which objects are listed in an extensional description of a set is irrelevant; furthermore, there is no point in repeating an element more than once in the listing. For example, the above set is the same as $\{3, 9, 1, 7, 5\}$, and also the same as $\{9, 9, 1, 5, 7, 5, 3, 5\}$ (see set equality in the following subsection for more on this). There are other mathematical structures, called sequences and discussed in Section 0.7, where the order in which objects appear and the multiplicity with which they occur do matter.

To write an intensional description of a set we use the following conventions: we first identify a variable which will stand for a generic element of the set; we then write the symbol ':' (some people prefer '|'); we write the property that characterises the elements of the set (this property refers to the variable identified before the ':'); finally we enclose the entire expression in curly brackets. For example, an intensional description of the set $\{1, 3, 5, 7, 9\}$ is: $\{x : x \text{ is a positive odd integer less than } 10\}$. We read this as follows: "the set of all elements x such that x is a positive odd integer less than 10". A different intensional description of the same set is $\{i : i \text{ is an odd integer represented by a decimal digit}\}$. Note that here we use both a different variable to denote the generic element of the set, and a different property to describe precisely the same elements.

0.1.2 Fundamental relationships between sets

Let A and B be two sets. If every element of A is also an element of B, we say that A is a ***subset*** of B, denoted $A \subseteq B$, and that B is a ***superset*** of A, denoted $B \supseteq A$. If it is the case that both $A \subseteq B$ and $B \subseteq A$, we say that A is ***equal*** to B, denoted $A = B$. Finally, if $A \subseteq B$ and $A \neq B$, then we say that A is a ***proper subset*** of B, denoted $A \subset B$, and that B is a ***proper superset*** of A, denoted $B \supset A$.

The definition of equality between two sets A and B implies that $A = B$ if and only if A

the elements of A and the elements of B — in other words, a correspondence that associates to each element of A one and only one element of B, and vice versa. By this criterion, it turns out that the cardinality of the set of positive integers is exactly the same as the cardinality of the set of odd positive integers: to positive integer i we associate the odd positive integer $2i-1$; it is easy to see that this association is a one-to-one correspondence between the two sets. Perhaps more surprisingly, it turns out that there are as many rational numbers as there are integers, but there are more real numbers between 0 and 1 than there are integers! The seemingly esoteric pursuit of comparing cardinalities of infinite sets has turned out to have profound implications in such disparate (and applied) fields as probability theory, real analysis, and computer science.

and B have exactly the same elements: each element of A belongs to B and each element of B belongs to A. This is the precise meaning of the statement, made earlier, that the order in which the elements of a set are listed and the number of times an element is listed (in an extensional description of the set) are irrelevant.

The definition of ***proper*** subset, implies that $A \subset B$ if and only if every element of A is an element of B, and there is (at least) one element of B that is not an element of A.

Note that, by these definitions, the empty set is a subset of *every* set, and a proper subset of every set other that itself. That is, for any set A, $\emptyset \subseteq A$; and if $A \neq \emptyset$, then $\emptyset \subset A$.

0.1.3 Operations on sets

Let A and B be two sets. We define three binary operations on sets, i.e., ways of combining two sets to obtain another set.

- The ***union*** of A and B, denoted $A \cup B$, is the set of elements that belong to A or to B (or both).
- The ***intersection*** of A and B, denoted $A \cap B$, is the set of elements that belong to both A and to B. Note that this may be the empty set. If $A \cap B = \emptyset$, we say that A and B are ***disjoint*** sets.
- The ***difference*** of A and B, denoted $A - B$, is the set of elements that belong to A but do not belong to B. Note that $A - B = \emptyset$ if and only if $A \subseteq B$.

Above we defined the union and intersection of *two* sets. It is useful to also define the union and intersection of an arbitrary (even infinite) number of sets. Let I be a set of "indices" so that associated with each index $i \in I$ there is a set A_i. The union and intersection of all A_i's are defined, respectively, as

$$\cup_{i \in I} A_i = \{x : \text{ for some } i \in I,\ x \in A_i\}$$
$$\cap_{i \in I} A_i = \{x : \text{ for each } i \in I,\ x \in A_i\}$$

For example, I could be the set of nonnegative integers and, for each $i \in I$, A_i could be the set of all nonnegative integer powers of i — i.e., $\{i^0, i^1, i^2, i^3, \ldots\}$. For instance, $A_0 = \{0, 1\}$, $A_1 = \{1\}$, $A_2 = \{1, 2, 4, 8, 16, \ldots\}$, $A_3 = \{1, 3, 9, 27, 81, \ldots\}$ and so on. In this example, $\cup_{i \in I} A_i$ is the set of all nonnegative integers, and $\cap_{i \in I} A_i$ is the set $\{1\}$.

Another important operation on sets is the powerset operation. For any set A, the ***powerset*** of A, denoted $\mathcal{P}(A)$ (or sometimes 2^A), is the set of subsets of A. For example, if $A = \{a, b, c\}$, then $\mathcal{P}(A) = \{\emptyset, \{a\}, \{b\}, \{c\}, \{a, b\}, \{a, c\}, \{b, c\}, \{a, b, c\}\}$. Note that this operation can be applied to an infinite, as well as to a finite, set A. The powerset of the empty set is the set containing the empty set: $\mathcal{P}(\emptyset) = \{\emptyset\}$. Note that this is not the same thing as the empty set.

0.1.4 Partition of a set

If A is a set, a ***partition*** of A is a set of nonempty, pairwise-disjoint subsets of A whose union is A. In terms of the concepts we defined earlier, a partition of A is a set $\mathcal{X} \subseteq \mathcal{P}(A)$ such that (i) for each $X \in \mathcal{X}$, $X \neq \emptyset$; (ii) for each $X, Y \in \mathcal{X}$ such that $X \neq Y$, $X \cap Y = \emptyset$; and (iii) $\bigcup_{X \in \mathcal{X}} X = A$.

For example, the set of even integers and the set of odd integers is a partition of the set of integers. The set of integers less than -5, the set of integers between -5 and 5 (inclusive) and the set of integers greater than 5 is another partition of the set of integers.

0.2 Ordered pairs

Consider two objects a and b, not necessarily distinct. The ***ordered pair*** (a, b) is a mathematical construction that "bundles" these two objects together, *in a particular order* — in this case, first the object a and then the object b.

The fact that the order of the objects in an ordered pair matters is reflected in the following definition. Let a, b, c, d be objects. We say that the ordered pair (a, b) is equal to the ordered pair (c, d), denoted $(a, b) = (c, d)$, if and only if $a = c$ and $b = d$.

By this definition, the ordered pair (a, b) is different from the ordered pair (b, a), unless a and b are equal. Both of these ordered pairs are "beasts of a different nature" than the *set* $\{a, b\}$.

Strictly speaking, the above definition of "ordered pair" is not really satisfactory. What does it mean to say that a mathematical construction "bundles" two objects together "in a particular order"? Although these words are suggestive, they are not really rigorously defined. Is an ordered pair another instance of a fundamental concept that cannot be reduced to more primitive terms, just like a set? Perhaps surprisingly, the answer is no. We can use the concept of set (that we have already accepted as irreducibly primitive) to define rigorously an ordered pair.

Specifically, we can formally define the ordered pair (a, b) simply as the set $\{\{a\}, \{a, b\}\}$. This is a set of two elements, each of which is a set. One of them is a set that contains only one element, and that is to be viewed as the 'first' element of the ordered pair; the other element of the set that represents the ordered pair is a set that contains two elements; one of them is the first element of the ordered pair, and the other is the second element of the ordered pair. Note that, in describing how the elements of the set $\{\{a\}, \{a, b\}\}$ correspond to the two elements of the ordered pair (a, b) we are not allowed to talk about "the first element of the set" and the "second element of the set" — because a set's elements are not ordered. Instead, we can distinguish the two elements (and therefore the order of the pair's elements) by their cardinalities.

Let us verify that this representation of ordered pairs actually satisfies the definition of equality between ordered pairs given above. Consider the ordered pairs (a, b) and (c, d). The sets that represent these ordered pairs are, respectively, $X = \{\{a\}, \{a, b\}\}$, and $Y = \{\{c\}, \{c, d\}\}$. Under what conditions can these two sets be equal? By the definition of equality between sets we must have that every element of X is an element of Y and vice versa. Thus, either

1. $\{a\} = \{c\}$ and $\{a, b\} = \{c, d\}$, or

2. $\{a\} = \{c, d\}$ and $\{a, b\} = \{c\}$.

We consider each of these cases separately.

Case 1. $\{a\} = \{c\}$ and $\{a, b\} = \{c, d\}$. Then, by the definition of equality between sets, we have that $a = c$, and $b = d$.

Case 2. $\{a\} = \{c, d\}$ and $\{a, b\} = \{c\}$. Then, by the definition of equality between sets, we have that $a = c = d$ and $a = b = c$, i.e., that $a = b = c = d$.

In both cases, we have that $a = c$ and $b = d$. This is precisely the condition under which the ordered pairs (a, b) and (c, d) are equal. We have therefore shown that the formal definition of ordered pairs in terms of sets has the desired property regarding equality of ordered pairs. Although, strictly speaking, there is no need to introduce ordered pairs — since we can instead use sets as discussed — it is certainly more convenient and natural to be able to use the notation (a, b) instead of the more cumbersome $\{\{a\}, \{a, b\}\}$.

Having defined ordered pairs, we can also define ordered triples. Informally, an ordered triple (a, b, c) is a mathematical construction that "bundles" three objects a, b and c (not necessarily distinct) in a particular order. More formally, we can define an ordered triple (a, b, c) as the ordered pair $(a, (b, c))$ — an ordered pair the first element of which is the object a and the second element of which is the ordered pair (b, c). It is straightforward to verify that, by this definition, and the definition of equality between ordered pairs, two ordered triples (a, b, c) and (a', b', c') are equal if and only if $a = a'$, $b = b'$ and $c = c'$.

We can extend this to ordered quadruples, ordered quintuples and, in general, ordered n-tuples for any integer $n > 1$.

0.3 Cartesian product

Having defined the concept of ordered pairs, we can now define another important operation between sets. Let A and B be sets; the ***Cartesian product*** of A and B, denoted $A \times B$, is the set of ordered pairs (a, b) where $a \in A$ and $b \in B$. In other words, $A \times B$ is the set of ordered pairs the first element of which is an element of A, and the second of element of which is an element of B. If A or B (or both) is the empty set, then $A \times B$ is also empty.

It is easy to verify from this definition that if A and B are finite sets, then $|A \times B| = |A| \cdot |B|$; if at least one of A and B is infinite and the other is nonempty, then $A \times B$ is infinite. Also, if A, B are distinct nonempty sets, $A \times B \neq B \times A$.

Since we can generalise the notion of ordered pair to ordered n-tuple, for any $n > 1$, we can also talk about the Cartesian product of n sets. Specifically, the Cartesian product of $n > 1$ sets $A_1, A_2, \ldots, A_n$, denoted $A_1 \times A_2 \ldots A_n$, is the set of ordered n-tuples $(a_1, a_2, \ldots, a_n)$, where $a_i \in A_i$ for each i such that $1 \leq i \leq n$.

0.4 Relations

Let A and B be sets. Informally, a relation between A and B is an association between elements of the set A and elements of the set B. For example, suppose A is the set of persons and B is the set of universities. We can talk about the relation "graduate-of" between these two sets; this relation associates a person, say Ravi, with a university, say the University of Toronto, if and only if Ravi graduated from U of T. Notice that in such an association it is possible that a person is associated to no university (if that person has not graduated from university), or that a person is associated with multiple universities (if the person has degrees from different universities).

The above definition of a relation is not rigorous. What, after all, does it mean to "associate" elements of one set with elements of another? We can define the concept of a relation rigorously based on sets.

Formally, a ***relation*** R between A and B is a subset of the Cartesian product $A \times B$, i.e., $R \subseteq A \times B$. In our preceding example, the relation "graduate-of" is the subset of the Cartesian product of the set of people and the set of universities that contains, among other ordered pairs, $(\mathrm{Ravi}, \mathrm{U\ of\ T})$. As another example, suppose $A = B$ is the set of integers. We can define the relation *LessThan* between A and B as the set of ordered pairs of integers (a, b) such that $a < b$. For example, the ordered pair $(-2, 7)$ belongs to *LessThan*, while neither $(2, 2)$ nor $(5, 3)$ belong to this relation.

In general, we can have relations not just between two sets (as in the preceding examples), but between any number of sets. For example, consider the set A of students, the set B of possible marks, and the set C of courses. Suppose we want to talk about the relation which describes the mark that a student achieved in a course. Mathematically we can formalise this by defining this relation as the subset of the Cartesian product $A \times B \times C$ that consists of all ordered triples (s, c, m) such that student s received mark m in course c.

As another example, let A, B and C be the set of integers. We can define the relation *Sum* consisting of the set of ordered triples (a, b, c) such that $a = b + c$. For example, $(17, 7, 10)$ is in this relation, while $(17, 7, 5)$ is not.

Naturally, we can also define relations among four, five, or any number of sets. In general, let $n > 1$ be an integer, and $A_1, A_2, \ldots, A_n$ be sets. A relation among $A_1, A_2, \ldots, A_n$ is a subset of the Cartesian product $A_1 \times A_2 \times \ldots \times A_n$. The number of sets involved in a relation (n in this general definition) is called the ***arity*** of the relation. For example, relation *LessThan* is a relation of arity two (or a ***binary*** relation), while relation *Sum* is a relation of arity three (or a ***ternary*** relation). In the (common) special case where all n sets are the same, say A, we say that the relation is an n-ary relation ***over*** A. Thus, the relation *LessThan* is a binary relation over the integers.

Since relations are sets, we already have a notion of equality between relations: two relations are equal if the corresponding sets of tuples are equal. This fact has certain implications that must be clearly understood.

First, certain associations that might superficially appear to define the same relation are, in fact, different relations — at least mathematically speaking. For instance, compare the relation graduate-of between persons and universities mentioned above, and the relation alma-materof between universities and persons, which associates a university, say U of T, to a person, say Ravi, if and only if the person has graduated from that university. Superficially, it may appear that the two relations define the same "association". Yet, mathematically speaking, they are different relations, because they are different sets. The first contains ordered pairs like $(\mathrm{Ravi}, \mathrm{U\ of\ T})$, while the latter contains ordered pairs like $(\mathrm{U\ of\ T}, \mathrm{Ravi})$. This becomes a little clearer if we do something similar to the *LessThan* relation between integers; if we invert the order of the elements of each ordered pair in the relation we get the "greater-than" relation between integers — which is clearly a different relation than *LessThan*!

A second consequence of the definition of equality between relations is that the particular way of describing the association between elements is not important — what is important is the association itself. For example, consider the binary relation C over the real numbers consisting of the set of ordered pairs (x, y) that are the coordinates of points on the circumference of a circle whose centre lies at point $(0, 0)$ and whose radius has length 1. Also consider the binary relation C' over the real numbers consisting of the set of ordered pairs (x, y) such that $x^2 + y^2 = 1$. These two relations are equal because they contain exactly the same sets of ordered pairs. Note that we described these two relations in entirely different ways: C was described geometrically, and C' was described algebraically. Nevertheless — by a nontrivial, though elementary, theorem of analytic geometry — the two descriptions refer to exactly the same set of ordered pairs, and therefore the two relations are equal. The point of this example is that what matters in defining a relation is the set of ordered pairs it contains — not the particular manner in which we describe the association between the elements of the two sets over which the relation is defined.

0.5 Important types of binary relations

A binary relation between elements of the same set, can be represented graphically as a so-called ***directed graph***. A directed graph consists of a set of points and a set of arrows, each connecting two points. The points of a directed graph are also called ***nodes*** or ***vertices***, and the arrows are also called ***edges*** or ***arcs***. The directed graph that represents a binary relation $R \subseteq A \times A$ is constructed as follows. Each vertex corresponds to some element of A; thus we will think of the vertices as elements of A. The directed graph has an edge from vertex $a \in A$ to vertex $b \in A$ if and only if $(a, b) \in R$. For example, Figure 1 shows the directed graph that represents the relation $R = \{(a, b) : a, b \in \{1, 2, 3, 4\} \text{ and } a \leq b\}$. The vertices are drawn as circles in this figure, and each vertex is labeled with the element of A to which it corresponds.

We now define some important special types of binary relations that relate elements of the same set. Let A be a set and $R \subseteq A \times A$ be a relation.

R is a ***reflexive*** relation if for each $a \in A$, $(a, a) \in R$. For example, the relation $a \leq b$ between integers is reflexive, while $a < b$ is not. In the directed graph that represents a reflexive relation, for each vertex there is an edge that starts and ends at that vertex. This is the case in the graph of Figure 1.

R is a ***symmetric*** relation if for each $a, b \in A$, if $(a, b) \in R$ then $(b, a) \in R$. For example,

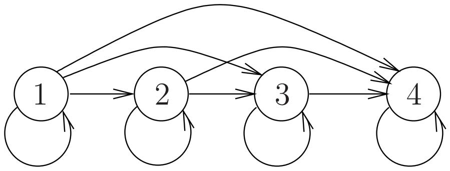


Figure 1: The directed graph of the relation $\{(a, b) : a, b \in \{1, 2, 3, 4\}$ and $a \leq b\}$

the relation

$$R_1 = \{(a, b) : a \text{ and } b \text{ are persons with at least one parent in common}\}$$

is symmetric. In the directed graph that represents a symmetric relation, whenever there is an arrow from a to b, there is also an arrow from b to a. To avoid the unnecessary visual clutter that results from having pairs of arrows running between the same two vertices in opposite directions, we can replace each such pair of arrows by a single line that connects the two vertices, without indicating the direction with an arrowhead. The resulting graph that consists of points and lines (still called vertices and edges) is referred to as an ***undirected graph***. Figure 2 shows the directed graph that represents a symmetric relation and, on its right, the undirected graph that represents the same relation.

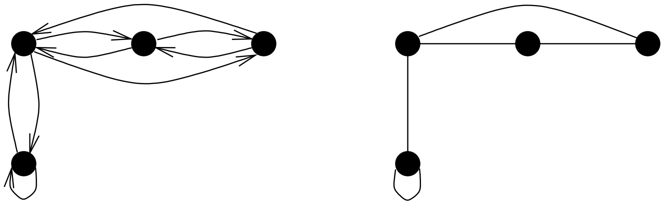


Figure 2: The directed and undirected graphs that correspond to a symmetric relation

R is a ***transitive*** relation if for each $a, b, c \in A$, if $(a, b) \in R$ and $(b, c) \in R$ then $(a, c) \in R$. For example, both relations defined below are transitive:

$$R_2 = \{(a, b) : a \text{ and } b \text{ are persons with the same parents}\}$$
$$R_3 = \{(a, b) : a \text{ and } b \text{ are persons and } a \text{ is an ancestor of } b\}.$$

R_2 is also reflexive and symmetric, while R_3 is neither. A ***path*** in a graph is a sequence of edges each of which ends at the vertex where the next edge, if one exists, starts. Such a path is said to be ***from*** vertex a ***to*** vertex b, if the first edge in the path starts at a and the last edge ends at b. In the graph that represents a transitive relation, whenever there is a path from a to b there is an edge that goes directly from a to b. This is the case in the graph of Figure 1.

R is an ***equivalence relation*** if it is reflexive, symmetric and transitive. For example, R_2 is an equivalence relation. R_1 is not an equivalence relation because, although it is reflexive and symmetric, it is not transitive. R_3 is not an equivalence relation because it is not reflexive or symmetric. Another example of an equivalence relation between integers is the so-called congruence modulo m relation: If m is a positive integer, integers a and b are ***congruent modulo*** m, written $a \equiv_m b$, if the divisions of a and b by m leave the same remainder. For example, $7 \equiv_5 17$. It is easy to see that the relation $\equiv_m$ is an equivalence relation, for every positive integer m.

Let R be an equivalence relation and a be an element of A. The ***equivalence class of*** a ***under*** R is defined as the set $R_a = \{b : (a,b) \in R\}$, i.e., the set of all elements that are related to a by R. The fact that R is reflexive implies that for any $a \in A$, $R_a \neq \emptyset$; and the fact that R is transitive implies that for any $a, b \in A$, if $R_a \neq R_b$ then $R_a \cap R_b = \emptyset$. Therefore, an equivalence relation R on A partitions A into a collection of equivalence classes. The elements of each equivalence class are related to each other by R, and elements of different equivalence classes are not related to each other by R. For example, since the division of an integer by a positive integer m can leave m possible remainders $(0, 1, \ldots, m-1)$, the equivalence relation $\equiv_m$ partitions the set of integers into m equivalence classes: the integers that have remainder 0 when divided by m, those that have remainder 1 when divided by m, and so on. The set of people is partitioned by the equivalence relation R_2 into a collection of equivalence classes, each consisting of siblings, i.e., persons with the same parents.

In view of this discussion, if R is an equivalence relation, the undirected graph that represents R is a collection of disconnected "clusters" as shown in Figure 3. Each cluster corresponds to one of the equivalence classes of A under R. Since every pair of elements in an equivalence class are related by R, there is an edge between every pair of nodes in each cluster (including an edge from a node to itself). And since elements in different equivalence classes are not related by R, there are no edges between nodes in different clusters. The equivalence relation represented in Figure 3 partitions a set into four equivalence classes, with one, four, two and five elements respectively.

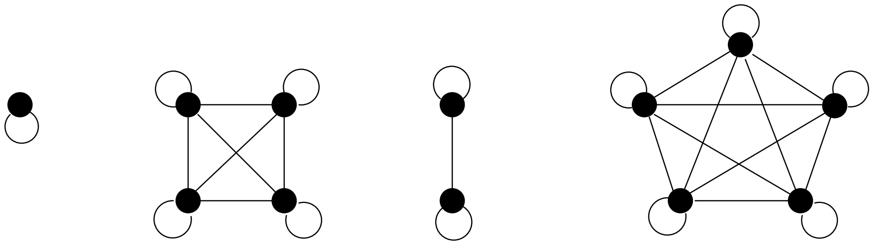


Figure 3: The undirected graph that corresponds to an equivalence relation of twelve elements

R is an ***antisymmetric*** relation if for each $a, b \in A$ such that $a \neq b$, if $(a,b) \in R$ then $(b,a) \notin R$. For example, the relations $a \leq b$ and $a < b$ between integers are antisymmetric. Relation R_3 defined above is antisymmetric. Note that a relation could be neither symmetric nor antisymmetric. For example,

$$R_4 = \{(a, b) : a \text{ and } b \text{ are persons with the same parents and } a \text{ is male}\}$$

is neither symmetric nor antisymmetric.

R is a ***partial order*** if it is antisymmetric and transitive. For example, the relation R_3 is a partial order. Note that there are persons neither one of which is an ancestor of the other. The qualification "partial" in the name "partial order" refers to the fact that, in general, a partial order need not specify an order between every two elements, A partial order that relates any two elements, in one way or the other, is called a ***total order***. More precisely, R is a total order if it is a partial order that satisfies the following property: for each $a, b \in A$, either $(a, b) \in R$ or $(b, a) \in R$. For example, if each person has a unique Social Insurance Number (SIN), the relation

$$R_5 = \{(a, b) : a \text{ and } b \text{ are persons and } a\text{'s SIN is smaller than } b\text{'s}\}$$

is a total order.

0.6 Functions

Let A and B be sets. A function f from A to B is a special kind of relation between A and B. Specifically, it is a relation with the property that, each element $a \in A$ is associated to exactly one element of B. More formally, the relation $f \subseteq A \times B$ is a ***function*** if for each $a \in A$ there is exactly one $b \in B$ such that $(a, b) \in f$. Note that, since a relation is actually a set (of ordered pairs), a function is also a set.

We usually write $f : A \to B$ to denote the fact that f is a function (and hence a relation) from A to B. The set A is called the ***domain*** of the function, and B is called the ***range*** of the function. If $a \in A$ and $b \in B$ are such that $(a, b) \in f$, we say that a ***maps to*** b (under f) and we write $f(a)$ to denote b. Note that, by definition of a function, such an element $b \in B$ is unique and therefore $f(a)$ is well-defined.

The function $f : A \to B$ is called an ***onto*** (or ***surjective***) function if for every $b \in B$ there is (at least) one $a \in A$ such that $f(a) = b$. In other words, every element of B is mapped onto by some element of A (under f). It is called ***one-to-one*** (or ***injective***) if for every element $b \in B$, there is at most one element $a \in A$ that maps onto b (under f). Finally, a function is called ***bijective*** if it is one-to-one and onto. If $f : A \to B$ is a bijection, then $|A| = |B|$.

The ***restriction*** of a function $f : A \to B$ to a subset A' of its domain, denoted $f \upharpoonright A'$, is the function $f' : A' \to B$ such that for every $a \in A'$, $f'(a) = f(a)$.

Since functions are relations and we already know what equality between relations means, we also know what equality between functions means. Let $f : A \to B$ and $f' : A \to B$ be functions. From the definitions, it is easy to check that $f = f'$ if and only if for every $a \in A$, $f(a) = f'(a)$.

The comments made at the end of the Section 0.4 regarding the implications of the definition of equality between relations apply to functions as well. To emphasise the point, consider two functions that map nonnegative integers to nonnegative integers. The first function, f is defined recursively as follows:

$$\begin{aligned}
&f(0) = 0\\
&f(1) = 1\\
&f(n) = f(n-1) + f(n-2), \text{ for any integer } n > 1.
\end{aligned}$$

So, for example,

$$\begin{aligned}
&f(2) = f(1) + f(0) = 1 + 0 = 1,\\
&f(3) = f(2) + f(1) = 1 + 1 = 2,\\
&f(4) = f(3) + f(2) = 2 + 1 = 3,\\
&f(5) = f(4) + f(3) = 3 + 2 = 5,\\
&f(6) = f(5) + f(4) = 5 + 3 = 8, \text{ and so on.}
\end{aligned}$$

The second function, f', is defined as follows: For any nonnegative integer n,

$$f'(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}.$$

It can be shown that, for any nonnegative integer n, $f(n) = f'(n)$. (In fact, we will show this later in the course. The proof is surprisingly simple, although the result is by no means obvious prima facie.) Thus, the two functions are equal. The point, once again, is that the manner in which we describe a function is not important: it is the set of ordered pairs that we define that matters. As this example shows, the same function can be described in extremely different ways, and it is not always immediately obvious whether two different descriptions in fact define the same function or different ones. We must therefore keep in mind that just because we defined two functions in different ways, it does not thereby follow that the two functions are themselves different. To prove that two functions f and f' from A to B are different, we must produce a particular element on which they differ — i.e., we must show that there is some element $a \in A$ such that $f(a) \neq f'(a)$.

0.7 Sequences

Informally, a sequence is an arrangement of objects in a particular order. Unlike a set, the order of the objects in the sequence is important, and an object may appear in the sequence several times. When we write down a sequence, we list the objects in the order they appear in the sequence, separating them with commas — e.g., a, b, c, b, f. Sometimes, the entire sequence is surrounded by angled brackets, as in $\langle a, b, c, b, f\rangle$. Naturally, if the sequence in question is infinite, we cannot write it down in this manner. In that case we rely in a combination of English and, perhaps, an initial fragment of the infinite sequence followed by ellipses. For example, we might talk about "the sequence of prime numbers in increasing order, i.e., $2, 3, 5, 7, \ldots$".

Another way or writing down an infinite sequence is reminiscent of intentional descriptions of sets. We use a variable, say x, subscripted by a nonnegative integer index, say i; a generic element of the sequence is denoted as x_i, and the sequence may be described by stating the property that x_i satisfies. For example, the sequence of prime numbers in increasing order can be written as: $\langle x_i :\ x_i$ is prime and there are exactly i primes less than $x_i \rangle$.

The preceding definition of sequence, although perhaps suggestive, is not rigorous since it relies on undefined terms like "arrangement of objects". We can define sequences formally by using the mathematical concepts we have introduced earlier.

Let $\mathbb{N}$ be the set of natural numbers, i.e. the set of nonnegative integers. An ***initial segment*** I of $\mathbb{N}$ is a subset of $\mathbb{N}$ with the following property: for any element $k \in I$, if $k > 0$, then $k-1 \in I$. Thus, an initial segment of $\mathbb{N}$ is either the empty set, or the set $\{0, 1, 2, \ldots, k\}$ for some nonnegative integer k, or the entire set $\mathbb{N}$.

Let A be a set. A ***sequence over*** A is a function $\sigma : I \to A$, where I is an initial segment of $\mathbb{N}$. Intuitively, $\sigma(0)$ is the first element in the sequence, $\sigma(1)$ is the second element in the sequence and so on. If $I = \emptyset$, then σ is the ***empty*** or ***null*** sequence, denoted ϵ. If $I = \mathbb{N}$ then σ is an infinite sequence; otherwise, it is a finite sequence. The ***length*** of σ is $|I|$ — i.e., the number of elements in the sequence. Note that the length of the empty sequence is 0, and the length of an infinite sequence (i.e., one whose domain is the entire set of nonnegative integers, $\mathbb{N}$) is ∞.

0.7.1 Operations on sequences

In this section we define two operations on sequences: concatenation and reversal.

Let $\sigma : I \to A$ and $\sigma' : I' \to A$ be sequences over the same set A, and suppose that σ is finite. Informally, the ***concatenation*** of σ and σ', denoted $\sigma \circ \sigma'$ (and sometimes as $\sigma\sigma'$), is the sequence over A that is obtained by juxtaposing the elements of σ' after the elements of σ. More precisely, this is defined as follows. If $I' = \mathbb{N}$ (i.e., σ' is infinite), then let $J = \mathbb{N}$; otherwise, let J be the initial segment $\{0, 1, \ldots, |I| + |I'| - 1\}$. Then $\sigma \circ \sigma' : J \to A$, where for any $i \in I$, $\sigma \circ \sigma'(i) = \sigma(i)$, and for any $i \in I'$, $\sigma \circ \sigma'(|I| + i) = \sigma'(i)$.

Let $\sigma : I \to A$ be a finite sequence over A. Informally, the ***reversal*** of σ, denoted σ^R, is the sequence of the elements of σ in reverse order. More precisely, $\sigma^R : I \to A$ is the sequence so that, for each $i \in I$, $\sigma^R(i) = \sigma(|I| - 1 - i)$.

0.7.2 Fundamental relationships between sequences

Since, strictly speaking, a sequence is a function of a special kind, and we know what it means for two functions to be equal, we also know what it means for two sequences to be equal. Let $\sigma : I \to A$ and $\sigma' : I \to A$ be two sequences over A of the same length. Then $\sigma = \sigma'$ if and only if, for every $k \in I$, $\sigma(k) = \sigma'(k)$. (Note that two sequences of different length cannot be equal, since the corresponding functions cannot possibly be equal.)

From the definitions of concatenation and equality of sequences, it is easy to verify the following facts (recall that concatenation of two sequences is defined only if the first of the sequences is finite): For any sequence σ, $\epsilon \circ \sigma = \sigma$; if σ is finite, $\sigma \circ \epsilon = \sigma$. Furthermore, for any sequences σ, τ over the same set, if τ is finite and $\tau \circ \sigma = \sigma$, then $\tau = \epsilon$; if σ is finite and $\sigma \circ \tau = \sigma$, then $\tau = \epsilon$.

A sequence σ is a ***subsequence*** of sequence τ if the elements of σ appear in τ and do so in the same order as in σ. For example, $\langle b, c, f\rangle$ is a subsequence of $\langle a, b, c, d, e, f, g\rangle$. Note that we do not require the elements of σ to be consecutive elements of τ — we only require that they appear in the same order as they do in σ. If, in fact, the elements of σ are consecutive elements of τ, we say that σ is a contiguous subsequence of τ. More formally, the definition of the subsequence relationship between sequences is as follows. Let A be a set, I and J be initial segments of $\mathbb{N}$ such that $|I| \leq |J|$, and $\sigma : I \to A$, $\tau : J \to A$ be sequences over A. The sequence σ is a subsequence of τ if there is an increasing function[2] $f : I \to J$ so that, for all $i \in I$, $\sigma(i) = \tau(f(i))$. If σ is a subsequence of τ and is not equal to τ, we say that σ is a ***proper subsequence*** of τ.

The sequence σ is a ***contiguous subsequence*** of sequence τ if there is some nonnegative integer j such that for all $i \in I$, $\sigma(i) = \tau(i + j)$. For example, $\langle c, d, e\rangle$ is a contiguous subsequence of $\langle a, b, c, d, e, f, \rangle$ (what is the value of j for this example?). Obviously, if σ is a contiguous subsequence of τ, then σ is a subsequence of τ. The converse, however, is not, in general, true: σ may be a subsequence of τ without being a contiguous subsequence of it.

The sequence σ is a ***prefix*** of sequence τ, if (i) σ is finite and there is a sequence σ' such that $\sigma \circ \sigma' = \tau$, or (ii) σ is infinite and $\sigma = \tau$. If σ is a prefix of τ, but is not equal to it, then σ is a ***proper prefix*** of τ. For example, $\langle a, b, c\rangle$ is a prefix (in fact, a proper prefix) of $\langle a, b, c, d\rangle$.

The sequence σ is a ***suffix*** of τ, if there is some (finite) sequence σ' so that $\sigma' \circ \sigma = \tau$. If σ is a suffix of τ but is not equal to it, then σ is a ***proper suffix*** of τ. For example, $\langle c, d, e\rangle$ is a suffix (in fact, a proper suffix) of $\langle a, b, c, d, e\rangle$.

It is an immediate consequence of these definitions that if σ is a (proper) prefix of τ, then σ is a contiguous (proper) subsequence of τ, but the converse is not necessarily true. Thus, the prefix relationship is a special case of the contiguous subsequence relationship. A similar fact holds if σ is a suffix of τ.

0.7.3 Strings

An ***alphabet*** is a nonempty set Σ; the elements of an alphabet are called its ***symbols***. A ***string*** (over alphabet Σ) is simply a finite sequence over Σ. By convention when we deal with strings we write the sequence without separating its elements by commas. Thus, if the alphabet is $\{0, 1\}$, we write 0100 instead of $\langle 0, 1, 0, 0\rangle$; if the alphabet is $\{a, b, \ldots, z\}$, we write *boy* instead of $\langle b, o, y\rangle$. The empty sequence is a string and is denoted, as usual, ϵ. The set of all strings over alphabet Σ is denoted Σ^*. Note that $\epsilon \in \Sigma^*$, for any alphabet Σ. Although each string is a *finite* sequence, it can be of arbitrary length and thus Σ^* is an infinite set.

Since strings are simply (finite) sequences over a specified set, various notions defined for sequences apply to strings as well. In particular, this is the case for the notion of length (which

[2] A function f whose domain is an initial segment I of $\mathbb{N}$ is called ***increasing*** if, for every $i, j \in I$ such that $i < j$, $f(i) < f(j)$.

must now be a natural number, and cannot be ∞), the operations concatenation and reversal, and the relationships equality, (proper) prefix and (proper) suffix. We use the term ***substring*** as synonymous to *contiguous subsequence* (*not* to subsequence); thus for any $\sigma, \tau \in \Sigma^*$, σ is a substring of τ if and only if there exist $\sigma', \sigma'' \in \Sigma^*$ (either or both of which may be empty) such that $\tau = \sigma'\sigma\sigma''$. If $\sigma' = \epsilon$ then σ is, in fact, a prefix of τ; and if $\sigma'' = \epsilon$ then σ is a suffix of τ.

0.8 Permutations

Let A be a finite set. A ***permutation*** of A is a sequence in which every element of A appears once and only once. For example, if $A = \{a, b, c, d\}$ then $\langle b, a, c, d\rangle$, $\langle a, c, d, b\rangle$ and $\langle a, b, c, d\rangle$ are permutations of A (there are 24 such permutations in total).

Sometimes we speak of permutations of a *sequence* (rather than a set). In this case, the definition is as follows: Let $\sigma : I \to A$ and $\tau : I \to A$ be finite sequences over A (note that the two sequences have the same length, $|I|$). The sequence τ is a permutation of σ if there is a bijective function $f : I \to I$ so that for every $i \in I$, $\tau(i) = \sigma(f(i))$.

Exercises

1. Let $A = \{1, 2, 3, \{1, 2\}, \{1, 2, 3\}\}$.

(a) Is $\{1, 2\}$ an element of A, a subset of A, both or neither?

(b) Is $\{2, 3\}$ an element of A, a subset of A, both or neither?

(c) Is 1 an element of A, a subset of A, both or neither?

2. What is $|\{\{1, 2, 3\}\}|$?

3. Prove that if A and B are finite sets such that $A \subseteq B$ and $|A| = |B|$, then $A = B$. Does the same result hold if A and B are not finite?

4. If $A \cup B = A$, what can we conclude about the sets A and B?

5. If $A \cap B = A$, what can we conclude about the sets A and B?

6. Write down the set that formally represents the ordered triple (a, b, c).

7. If $A \times B = B \times A$, what can we conclude about the sets and A and B? Justify your answer.

8. If $|A \times B| = |A|$, what can we conclude about the sets A and B? Justify your answer.

9. Let $A = \{1, 2\}$ and $B = \{a, b, c\}$. Write down all possible functions from A to B. (Each function should be written as a set of ordered pairs.) How many such functions are there?

10. Let $f : A \to B$ be a function, where A and B are finite sets. Recall that a function is a certain kind of relation, and a relation is a certain kind of set, so it makes sense to talk about the cardinality of the function, $|f|$.

(a) What can we say about $|f|$ in comparison to $|A|$?

(b) If f is surjective, what can we say about $|A|$ in comparison to $|B|$?

(c) If f is injective, what can we say about $|A|$ in comparison to $|B|$?

(d) If f is bijective, what can we say about $|A|$ in comparison to $|B|$?

11. Formally, a sequence is a special kind of function, a function is a special kind of a relation, and a relation is a special kind of set. Thus, ultimately a sequence is some sort of a set. Given two sequences, then, it makes perfect sense to ask whether one is a subset of the other. Is the subset relationship between sequences (viewed as sets) the same as either the subsequence or prefix relationship? In other words, suppose σ and τ are arbitrary sequences over some set A.

(a) Is it true that $\sigma \subseteq \tau$ if and only if σ is a subsequence of τ?

(b) Is it true that $\sigma \subseteq \tau$ if and only if σ is a prefix of τ?

Justify your answers.

12. Let $A = \{1, 2, 3\}$. Write down all the permutations of A. Suppose $B = \{1, 2, 3, 4\}$. How many permutations of B are there? How would you generate all permutations of B in a systematic way, given all the permutations of A? Based on your experience with this example, write a (recursive) program which takes as input a number $n \geq 0$, and returns, in some reasonable representation, the set of permutations of the set $\{1, 2, \ldots, n\}$ (it returns $\emptyset$, if $n = 0$).

13. In the text we defined permutations of a *set* and permutations of a *sequence*. Consider the following alternative definition of a permutation of a sequence $\sigma : I \to A$. Let S be the set of elements that appear in σ; more precisely, $S = \{x\text{: for some } i \in I,\ x = \sigma(i)\}$. Then a permutation of σ is a permutation of the set S. Is this definition equivalent to the definition of permutation of sequence given in the text? Justify your answer.

Chapter 1

INDUCTION

1.1 *Fundamental properties of the natural numbers*

The natural numbers are the nonnegative integers $0, 1, 2, \ldots$. The set of natural numbers is denoted $\mathbb{N}$; the set of all integers (including the negative ones) is denoted $\mathbb{Z}$. In this section we discuss three fundamental properties of $\mathbb{N}$: well-ordering, (simple) induction, and complete induction.

1.1.1 *Well-ordering*

basically shows there is a minimum number in Natural numbers

Principle of well-ordering: *Any nonempty subset A of $\mathbb{N}$ contains a minimum element; i.e., for any $A \subseteq \mathbb{N}$ such that $A \neq \emptyset$, there is some $a \in A$ such that for all $a' \in A$, $a \leq a'$.*

Note that this principle applies to *all* nonempty subsets of $\mathbb{N}$ and, in particular, to *infinite* subsets of $\mathbb{N}$. Also note that this principle does *not* apply to several other sets of numbers. For instance, it does not apply to $\mathbb{Z}$: the set of negative integers is a nonempty subset of $\mathbb{Z}$ that does not have a minimum element! Similarly, the well-ordering principle does not apply to the set of all rational numbers between 0 and 1: $\{\frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \frac{1}{5}, \ldots\}$ is a subset of this set that has no minimum element. The point of these examples is to illustrate that the well-ordering principle, although rather obvious, is not something completely trivial that is true of all sets of numbers. It is a property specific to the set $\mathbb{N}$.

1.1.2 *Simple induction*

Principle of (simple) induction: *Let A be any set that satisfies the following properties:*

(i) 0 is an element of A;

(ii) for any $i \in \mathbb{N}$, if i is an element of A then $i + 1$ is also an element of A.

Then A is a superset of $\mathbb{N}$.

Here is an informal justification of why this principle is valid.

- By (i), $0 \in A$.
- For $i = 0$, property (ii) states that if $0 \in A$ then $1 \in A$. Since we have just shown that $0 \in A$, it follows that $1 \in A$.
- For $i = 1$, property (ii) states that if $1 \in A$ then $2 \in A$. Since we have just shown that $1 \in A$, it follows that $2 \in A$.
- For $i = 2$, property (ii) states that if $2 \in A$ then $3 \in A$. Since we have just shown that $2 \in A$, it follows that $3 \in A$.

It seems perfectly clear that we can proceed in this manner to also prove that each of $4, 5, 6, \ldots$ — in fact, any natural number whatsoever — belongs to A; and therefore, that $\mathbb{N} \subseteq A$. This, however, is not a rigorous proof: We have not actually *shown* that this argument (the repeated application of (ii)) can be used to prove that *every* natural number is in A. In fact, it is intrinsically impossible to do so, since there are infinitely many cases to check. Nothing we have said so far can rule out a skeptic's concern that this argument will break down after some (perhaps incredibly large) number! Our *intuition* about the natural numbers strongly suggests that the argument does not break down, and so we accept this principle as "obviously true".

1.1.3 Complete induction

Principle of complete induction: *Let A be any set that satisfies the following property:*

($*$) *for any $i \in \mathbb{N}$, if every natural number less than i is an element of A then i is also an element of A.*

Then A is a superset of $\mathbb{N}$.

This principle seems quite similar to the principle of (simple) induction, although there are some differences. First, the requirement that $0 \in A$ (see property (i) in the principle of induction) appears to be missing. Actually, this is not true: the requirement that $0 \in A$ is implicit in ($*$). To see this, note that for $i = 0$, ($*$) states that if every natural number less than 0 belongs to A, then 0 belongs to A. Since there are no natural numbers less than 0, the hypothesis of this implication is (vacuously) true, and therefore so is the conclusion. In other words, 0 is an element of A.

vacuous truth: all member of an empty set have a certain property

The second, and less superficial, difference between complete and simple induction is this: In complete induction, we require i to be an element of A if *all* the numbers preceding i are in A. In contrast, in simple induction, we require i to be an element of A as long as just the *previous* number, $i - 1$, is in A.

We can give an informal justification of this principle, along the lines we used for simple induction:

- As we argued two paragraphs earlier, 0 belongs to A (by considering what ($*$) says when $i = 0$).

- When $i = 1$, $(*)$ states that if 0 belongs to A, then 1 also belongs to A. But we showed that 0 does belong to A; therefore, so does 1.
- When $i = 2$, $(*)$ states that if 0 and 1 belong to A, then 2 also belongs to A. But we showed that 0 and 1 do belong to A; therefore, so does 2.
- When $i = 3$, $(*)$ states that if 0, 1 and 2 belong to A, then 3 also belongs to A. But we showed that 0, 1 and 2 do belong to A; therefore, so does 3.

Proceeding in the same manner, we can show that each of $4, 5, 6, \ldots$ is in A. This is not a formal proof of the validity of complete induction, for the same reason given in the case of simple induction.

1.1.4 *Equivalence of the three principles*

The three properties of the natural numbers discussed in the preceding sections are so basic that they cannot be proved from more elementary facts. We accept them axiomatically, as "self-evident" truths. This may be disappointing, but it is perhaps comforting to know that we don't have to accept all three of them independently. This is because we can prove that the three principles are equivalent: any one of them implies the other two. Thus, if we think that just one of them is "self-evident" we must be prepared to accept the other two as well, since they follow from it.

Theorem 1.1 *The principles of well-ordering, induction, and complete induction are equivalent.*

PROOF. We prove this by establishing a "cycle" of implications. Specifically, we prove that (a) well-ordering implies induction, (b) induction implies complete induction, and (c) complete induction implies well-ordering.

(a) *Well-ordering implies induction:* Assume that the principle of well-ordering holds. We will prove that the principle of induction is also true. To that end, let A be any set that satisfies the following properties:

(i) 0 is an element of A;

(ii) for any $i \in \mathbb{N}$, if i is an element of A then $i + 1$ is also an element of A.

To prove the principle of induction we must show that $A \supseteq \mathbb{N}$. We do so using a proof by contradiction.

Suppose, for contradiction, that A is not a superset of $\mathbb{N}$. Then the set $\overline{A} = \mathbb{N} - A$ must be nonempty. (Note that, by the definition of $\overline{A}$, a natural number k belongs to A if and only if it does not belong to $\overline{A}$.) By the principle of well-ordering (which holds by assumption) $\overline{A}$ has a minimum element, say i^*. By (i), $i^* \neq 0$ (because $0 \in A$, while $i^* \notin A$). Thus, $i^* > 0$, and hence $i^* - 1$ is a natural number. Since i^* is the minimum element of $\overline{A}$, it follows that $i^* - 1 \notin \overline{A}$, and therefore, $i^* - 1 \in A$. By (ii) and the fact that $i^* - 1 \in A$, it follows that $i^* \in A$.

But then $i^* \notin \overline{A}$, contradicting the fact that i^* is an element of $\overline{A}$ — in fact, the minimum element of it.

This contradiction means that our original supposition, that A is not a superset of $\mathbb{N}$, is false. In other words, $A \supseteq \mathbb{N}$, as wanted.

(b) *Induction implies complete induction:* Assume that the principle of induction holds. Let A be any set that satisfies the following property:

($*$) for any $i \in \mathbb{N}$, if every natural number less than i is an element of A then i is also an element of A.

To prove that the principle of complete induction holds, we must prove that $A \supseteq \mathbb{N}$. To this end, define the set B as follows:

$$B = \{i \in \mathbb{N} : \text{every natural number less than or equal to } i \text{ is in } A\}.$$

In other words, B is the initial segment of $\mathbb{N}$ up to (but not including) the smallest number that is not in A. Obviously, $B \subseteq A$. Thus, to prove that $A \supseteq \mathbb{N}$ it suffices to prove that $B \supseteq \mathbb{N}$. Indeed, we have:

(1) $0 \in B$. (This is because ($*$), for $i = 0$, implies that 0 is in A.)

(2) For any $i \in \mathbb{N}$, if $i \in B$ then $i + 1 \in B$. (To see why, suppose $i \in B$. By definition of B, every natural number less than or equal to i is in A. By ($*$), $i + 1 \in A$. Thus, every natural number less than or equal to $i + 1$ is in A. By definition of B, $i + 1 \in B$.)

By (1), (2) and the principle of induction (which we assume holds), we get that $B \supseteq \mathbb{N}$, as wanted.

(c) *Complete induction implies well-ordering:* Assume that the principle of complete induction holds. Let A be any subset of $\mathbb{N}$ that has no minimum element. To prove that the principle of well-ordering holds it suffices to show that A is empty.

For any $i \in \mathbb{N}$, if every natural number less than i is *not* in A, it follows that i is *not* in A either — otherwise, i would be a minimum element of A, and we are assuming that A has no minimum element. Let $\overline{A} = \mathbb{N} - A$. (Thus, any natural number i is in A if and only if it is not in $\overline{A}$.) Therefore, for any $i \in \mathbb{N}$, if every natural number less than i is in $\overline{A}$, then i is also in $\overline{A}$. By the principle of complete induction (which we assume holds), it follows that $\overline{A}$ is a superset of $\mathbb{N}$. Therefore A is empty, as wanted. □

1.2 Mathematical induction

Mathematical induction is a proof technique which can often be used to prove that a certain statement is true for *all* natural numbers. Before we see how and why this proof technique works, we must first understand very clearly the *kind* of statements to which it can be applied. Consider the following statements:

- $P_1(n)$: The sum of the natural numbers up to and including n is equal to $n(n+1)/2$.
- $P_2(n)$: If a set A has n elements, then the powerset of A has 2^n elements.
- $P_3(n)$: The number n is prime or it is equal to the product of a sequence of primes.
- $P_4(n)$: n cents worth of postage can be formed using only 4-cent and 5-cent stamps.
- $P_5(n)$: n is a perfect square (i.e., there is some $a \in \mathbb{N}$ such that $n = a^2$).
- $P_6(n)$: n is a perfect cube and is the sum of two perfect cubes of positive numbers (i.e., there are $a, b, c \in \mathbb{N}$ such that $n = a^3$, $b, c > 0$ and $n = b^3 + c^3$).

Each of these is a *generic* statement about the variable n, which we assume ranges over the natural numbers. It becomes a *specific* statement once we choose a particular value for n; and that specific statement may be true or false. For instance, if we choose $n = 5$, $P_1(5)$ states that the sum $0+1+2+3+4+5$ is equal to $5 \cdot 6/2$; this is clearly a true statement. On the other hand, $P_5(5)$ states that 5 is a perfect square, which is a false statement. Technically, a statement like these is called a *predicate* of natural numbers.

In general, a predicate of a natural number n may be true for *all* values of n, for *some* values of n, or for *no* value of n. For example, $P_1(n)$ and $P_2(n)$ are true for all n; $P_3(n)$ is true for all $n \geq 2$; $P_4(n)$ is true for all $n \in \mathbb{N}$ except $0, 1, 2, 3, 6, 7$ and 11; $P_5(n)$ is true for an infinite number of natural numbers, and false for an infinite number of natural numbers; and $P_6(n)$ is true for no natural number whatsoever.

In mathematics and in computer science, we are often interested in proving that a predicate of a natural number n is true for *all* $n \in \mathbb{N}$. Obviously we can't hope to do this by checking each natural number in turn, since there are infinitely many such numbers and we'd never be done checking them all. Induction is an important technique (though not the only one) that can be used to prove such statements.

Thus, a proof by induction, is a method for proving the statement

$$\text{for each } n \in \mathbb{N},\ P(n) \text{ is true}$$

where $P(n)$ is a predicate of natural numbers. This method of proof consists of two steps:

BASIS: Prove that $P(0)$ is true, i.e., that the predicate $P(n)$ holds for $n = 0$.

INDUCTION STEP: Prove that, for each $i \in \mathbb{N}$, if $P(i)$ is true then $P(i+1)$ is also true.

The assumption that $P(i)$ holds in the induction step of this proof is called the ***induction hypothesis***.

The question now arises: Why does such a proof actually show that $P(n)$ is true for all $n \in \mathbb{N}$? The answer should be clear if we recall the principle of induction for $\mathbb{N}$ (see Section 1.1.2). To spell out exactly what is going on, let $A[P]$ be the set of natural numbers for which the predicate P is true; i.e., $A[P] = \{n \in \mathbb{N} : P(n) \text{ is true}\}$. By the basis of the induction proof,

(i) $0 \in A[P]$.

translation from principle of induction to techniques of inductionmethod

By the Induction Step,

(ii) for each $i \in \mathbb{N}$, if $i \in A[P]$ then $i + 1 \in A[P]$.

By the principle of induction, $A[P]$ is a superset of $\mathbb{N}$. Since (by definition) $A[P]$ contains only natural numbers, $A[P]$ is a subset of $\mathbb{N}$. Thus, $A[P] = \mathbb{N}$. From the definition of $A[P]$, this means that $P(n)$ is true for all $n \in \mathbb{N}$.

Before looking at some examples, it is important to dispell a common misconception about the induction step. The induction hypothesis is *not* the assumption that predicate P is true for all natural numbers. This is what we want to prove and so assuming it would be circular reasoning. Rather, the induction step requires us to prove, for each natural number, that if P holds for that number then it holds for the next number too. Thus, typically the induction step proceeds as follows:

- We let i be an arbitrary natural number.
- We assume that $P(i)$ is true (the induction hypothesis).
- Using the induction hypothesis, we prove that $P(i + 1)$ is also true.

The particulars of the last step depend on the predicate P. To carry out this step we must somehow relate what the predicate asserts for a number to what it asserts for the previous number.

1.2.1 Examples

Having seen, in general terms, what a proof by induction looks like and why it works, let us now turn our attention to some specific examples of inductive proofs.

Example 1.1 Let i and j be integers, and suppose that, for each integer t such that $i \leq t \leq j$, a_t is some number. We use the notation $\sum_{t=i}^{j} a_t$ to denote the sum $a_i + a_{i+1} + \cdots + a_j$. (If $i > j$, we adopt the convention that $\sum_{t=i}^{j} a_t$ is 0.) We will use induction to prove the following:

Proposition 1.2 *For any $n \in \mathbb{N}$, $\sum_{t=0}^{n} t = n(n+1)/2$; i.e. $0 + 1 + \cdots + n = n(n+1)/2$.*

(Note that in the above use of the $\sum$ notation, $a_t = t$.)

Whenever we do a proof by induction, it is crucial to be very clear about the predicate which we want to prove is true for all natural numbers. The best way to ensure this is to explicitly write down this predicate using a suitable variable to denote the generic natural number that the predicate is talking about, and to give the predicate a name. In this particular case, let us call the predicate in question $S(n)$ (so in this case we have chosen n as the variable name denoting a generic natural number). We have

$$S(n): \qquad \sum_{t=0}^{n} t = \frac{n(n+1)}{2}. \tag{1.1}$$

Be sure you understand the distinction between the *predicate* $S(n)$ and the *statement* "for every natural number n, $S(n)$ holds". The former (being a predicate of natural numbers) is a function that becomes true or false once we specify a particular natural number. It makes no sense to ask whether the predicate $S(n)$ is true *unless we have already specified a value for* n; it makes sense to ask whether $S(5)$ or $S(17)$ is true. On the other hand, it makes perfect sense to ask whether the statement "for all $n \in \mathbb{N}$, $S(n)$ holds" is true (and we will presently prove that it is).

predicate of natural number is proved to be true or false once we specify a value for n

Proof of Proposition 1.2. Let $S(n)$ be the predicate defined in (1.1). We use induction to prove that $S(n)$ is true for all $n \in \mathbb{N}$.

Basis: $n = 0$. In this case, $S(0)$ states that $0 = 0 \cdot 1/2$, which is obviously true. Thus $S(0)$ holds.

Induction Step: Let i be an arbitrary natural number and assume that $S(i)$ holds, i.e., $\sum_{t=0}^{i} t = i(i+1)/2$. We must prove that $S(i+1)$ holds, i.e., that $\sum_{t=0}^{i+1} t = (i+1)(i+2)/2$. Indeed, we have:

$$\begin{aligned}
\sum_{t=0}^{i+1} t &= \Big(\sum_{t=0}^{i} t\Big) + (i+1) && \text{[by associativity of addition]}\\
&= \frac{i(i+1)}{2} + (i+1) && \text{[by induction hypothesis]}\\
&= \frac{(i+1)(i+2)}{2} && \text{[by simple algebra]}
\end{aligned}$$

Therefore $S(i+1)$ holds, as wanted. □

End of Example 1.1

Example 1.2 Suppose A is a finite set with k elements. We would like to determine how many different subsets of A there are. For small values of k, it is easy to determine this directly. For example, if $k = 0$ (i.e., A is empty) then A has 1 subset, namely itself. If $k = 1$ then A has two subsets, namely the empty set and itself. If $k = 2$ then A has four subsets: the empty set, two subsets that contain one element each, and itself. If $k = 3$ then A has eight subsets: the empty set, three subsets that contain one element each, three subsets that contain two elements each, and itself. In general:

Proposition 1.3 *For any $k \in \mathbb{N}$, any set with k elements has exactly 2^k subsets.*

Proof. Define the predicate $P(k)$ as follows:

$P(k)$: any set with k elements has exactly 2^k subsets.

We will prove, using induction, that for every $k \in \mathbb{N}$, $P(k)$ holds.

Basis: $k = 0$. Then $P(0)$ states that any set with 0 elements has exactly $2^0 = 1$ subsets. But the only set with 0 elements is the empty set, and it has exactly one subset (namely itself). Thus, $P(0)$ holds.

Induction Step: Let i be an arbitrary natural number, and assume that $P(i)$ holds, i.e., any set with i elements has 2^i subsets. We must prove that $P(i+1)$ holds, i.e., that any set with $i+1$ elements has 2^{i+1} subsets.

Consider an arbitrary set with $i+1$ distinct elements, say $A = \{a_1, a_2, \ldots, a_{i+1}\}$. There are two (mutually exclusive) types of subsets of A: those that contain a_{i+1} and those that do not. Let $\mathcal{Y}$ (for "yes") be the set of subsets of A that contain a_{i+1}, and $\mathcal{N}$ (for "no") be the set of subsets of A that do not contain a_{i+1}.

First, we note that $\mathcal{Y}$ and $\mathcal{N}$ have the same number of elements. (Note that the elements of $\mathcal{Y}$ and $\mathcal{N}$ are *subsets* of A.) To see this, let $f : \mathcal{Y} \to \mathcal{N}$ be the function defined as follows: For any $B \in \mathcal{Y}$ (i.e., for any $B \subseteq A$ such that $a_{i+1} \in B$), let $f(B) = B - \{a_{i+1}\}$ (hence, $f(B) \in \mathcal{N}$). It is easy to see that this function is a bijection, and therefore $|\mathcal{Y}| = |\mathcal{N}|$.

Next, we note that $\mathcal{N}$ is (by definition) the set of subsets of $A' = \{a_1, a_2, \ldots, a_i\}$. Since A' has i elements, by induction hypothesis, it has 2^i subsets. Therefore, $\mathcal{N}$ has 2^i elements. Since, as shown before, $\mathcal{Y}$ has as many elements as $\mathcal{N}$, we have that $\mathcal{Y}$ also has 2^i elements.

Finally, the number of subsets of A is equal to the number of elements in $\mathcal{Y}$ plus the number of elements in $\mathcal{N}$ (because every subset of A must belong to one and only one of $\mathcal{Y}$ or $\mathcal{N}$). Therefore, the number of subsets of A is $2^i + 2^i = 2 \cdot 2^i = 2^{i+1}$. Hence, the number of subsets of an arbitrary set of $i+1$ elements is 2^{i+1}. This shows that $P(i+1)$ holds, as wanted. □

Recall that if A is a set, then the powerset of A, $\mathcal{P}(A)$, is the set of subsets of A. Thus, Proposition 1.3 can be stated as follows:

Corollary 1.4 *For any finite set A, $|\mathcal{P}(A)| = 2^{|A|}$.*

End of Example 1.2

Example 1.3 Our next example is not directly an illustration of induction, but a useful result that can be obtained as a corollary to Proposition 1.3. A ***binary string*** is a finite sequence of bits — i.e., a string over the alphabet $\{0, 1\}$ (cf. Section 0.7.3, page 13). For example, a byte is a binary string of length eight. Binary strings come up very frequently in computer science because information in computers is ultimately encoded, stored and manipulated as binary strings. The question we are interested in is: How many different binary strings are there of a specified length ℓ?

Proposition 1.3 immediately yields the answer to this question, if we observe that we can think of a binary string of length ℓ as the subset of the set $\{0, 1, \ldots, \ell - 1\}$ which contains the indices (positions) of all the bits in the string which are 1. For example, 0110 can be thought of as designating the set $\{1, 2\}$, and 1011 can be thought of as designating the set $\{0, 2, 3\}$. Since there are 2^ℓ subsets of $\{0, 1, \ldots, \ell - 1\}$ and (by the correspondence just mentioned) there are as many binary strings of length ℓ as subsets of this set, we have the answer to our question.

binary string of length l can be considered as subsets of {1,2, … , l-1} which contains the indices of position of all bits which are 1!!

We now state all this a little more carefully.

Proposition 1.5 *For any $\ell \in \mathbb{N}$, there are 2^ℓ binary strings of length ℓ.*

PROOF. Let $\mathcal{B}_\ell$ be the set of binary strings of length ℓ, and $A_\ell = \{0, 1, \ldots, \ell - 1\}$. Define the function $f : \mathcal{B}_\ell \to \mathcal{P}(A_\ell)$ as follows: For any string $b_0 b_1 \ldots b_{\ell-1} \in \mathcal{B}_\ell$ (where $b_i \in \{0, 1\}$, for each i such that $0 \leq i < \ell$),

$$f(b_0 b_1 \ldots b_{\ell-1}) = \{i : 0 \leq i < \ell \text{ and } b_i = 1\}.$$

It is easy to check that this function is a bijection, and therefore $|\mathcal{B}_\ell| = |\mathcal{P}(A_\ell)|$. By Proposition 1.3, $|\mathcal{B}_\ell| = 2^\ell$. Thus, there are 2^ℓ binary strings of length ℓ, as wanted. □

As an exercise, you should prove Proposition 1.5 by using induction directly, *without* using Proposition 1.3. **End of Example 1.3**

Example 1.4 Let A and B be finite sets. We are interested in determining the number of functions from A to B.

Proposition 1.6 *For any $m, n \in \mathbb{N}$, there are exactly n^m functions from any set of m elements to any set of n elements.*

We will prove this proposition by using induction. Our first task is to identify the predicate which we want to show holds for all natural numbers. The difficulty, in this example, is that the statement we want to prove has two parameters that range over $\mathbb{N}$, namely m and n. We have two natural choices; we can consider the predicate

$$P(m) : \quad \text{for any } n \in \mathbb{N}, \text{ there are exactly } n^m \text{ functions} \\ \text{from any set of } m \text{ elements to any set of } n \text{ elements}$$

and try using induction to prove that $P(m)$ holds for every $m \in \mathbb{N}$. Alternatively, we can consider the predicate

$$Q(n) : \quad \text{for any } m \in \mathbb{N}, \text{ there are exactly } n^m \text{ functions} \\ \text{from any set of } m \text{ elements to any set of } n \text{ elements}$$

and try using induction to prove that $Q(n)$ holds for every $n \in \mathbb{N}$. Proving either of these statements would be a proof of the desired proposition. As a matter of fact, the first of them is easily amenable to an induction proof, while the second is not.

PROOF OF PROPOSITION 1.6. We use induction to prove that $P(m)$ holds for all $m \in \mathbb{N}$.

BASIS: $m = 0$. $P(0)$ states that, for any integer $n \in \mathbb{N}$, there is exactly one (since $n^0 = 1$) function from any set A of zero elements to any set B of n elements. Since A has zero elements it is the empty set, and there is indeed exactly one function whose domain is empty, namely the empty function. (Recall that, formally speaking, a function from A to B is a set of pairs in $A \times B$; the empty function then, is the empty set.)

Induction Step: Let i be an arbitrary natural number, and assume that $P(i)$ holds; i.e., for any $n \in \mathbb{N}$, there are exactly n^i functions from any set of i elements to any set of n elements. We must prove that $P(i+1)$ holds as well; i.e., for any integer $n \in \mathbb{N}$, there are exactly n^{i+1} functions from any set of $i+1$ elements to any set of n elements.

Let A be an arbitrary set with $i+1$ elements, say $A = \{a_1, a_2, \ldots, a_{i+1}\}$, and B be an arbitrary set with n elements, say $B = \{b_1, b_2, \ldots, b_n\}$. Let us fix a particular element of A, say a_{i+1}, and let us group the functions from A to B according to the element of B into which a_{i+1} gets mapped. That is, define the following sets of functions from A to B:

$$\begin{aligned}
X_1 &= \text{the set of functions from } A \text{ to } B \text{ that map } a_{i+1} \text{ to } b_1\\
X_2 &= \text{the set of functions from } A \text{ to } B \text{ that map } a_{i+1} \text{ to } b_2\\
&\vdots\\
X_n &= \text{the set of functions from } A \text{ to } B \text{ that map } a_{i+1} \text{ to } b_n
\end{aligned}$$

Every function from A to B belongs to one and only one of these sets. Notice that, for any j such that $1 \leq j \leq n$, X_j contains exactly as many functions as there are functions from the set $A' = \{a_1, a_2, \ldots, a_i\}$ to B. (This is because X_j contains the functions from $\{a_1, a_2, \ldots, a_i, a_{i+1}\}$ to B where, however, we have specified the element to which a_{i+1} is mapped; the remaining elements of A, namely the elements of A', can get mapped to the elements of B in any possible way.) Therefore, by induction hypothesis, $|X_j| = n^i$, for each j such that $1 \leq j \leq n$. The total number of functions from A to B then is:

$$|X_1| + |X_2| + \ldots + |X_n| = \underbrace{n^i + n^i + \ldots + n^i}_{n \text{ times}} = n \cdot n^i = n^{i+1}$$

as wanted. □

You may find it instructive to try proving, using induction, that $Q(n)$ holds for all $n \in \mathbb{N}$ in order to discover the difficulty in carrying out the induction step. Basically, there seems to be no natural way to use the induction hypothesis that there i^m functions from any set of m elements to any set of i elements in order to prove that there are $(i+1)^m$ functions from any set of m elements to any set of $i+1$ elements. In contrast, we were able to find (in the proof given above) a way to successfully use the induction hypothesis that there n^i functions from any set of i elements to any set of n elements in order to prove that there are n^{i+1} functions from any set of $i+1$ elements to any set of n elements. You may wonder how we knew to try proving that $P(m)$ holds for all $m \in \mathbb{N}$, rather than to try proving that $Q(n)$ holds for all $n \in \mathbb{N}$. In fact, there was no *a priori* reason to favour the first of these alternatives over the other. We simply have to try them out. If one doesn't seem to work, then we can try the other. As you gain experience and acquire a deeper understanding of various mathematical techniques, you will, in many instances, have a better feeling as to which of the various choices facing you is likely to "pan out". But even the best mathematicians, when faced with difficult enough problems, have to resort to trial-and-error. **End of Example 1.4**

Example 1.5 Let m, n be natural numbers such that $n \neq 0$. The division of m by n yields a quotient and a remainder — i.e., unique natural numbers q and r such that $m = q \cdot n + r$ and $r < n$. How do we know this? Presumably we believe that this is true because (sometime long ago) we learned an algorithm for dividing numbers. But, although we all learned the division algorithm in elementary school, nobody really proved to us that this algorithm works! The algorithm for dividing m by n undoubtedly produces two natural numbers q and r. But how do we know that these numbers really are the quotient and remainder of the division, in the sense that they satisfy the property $m = q \cdot n + r$ and $r < n$? Perhaps our belief that given any two natural numbers m and n (such that $n \neq 0$), the quotient and remainder of the division of m by n exist (and are unique) is not well-founded. We will now prove that such numbers really do exist. (And rest assured that the division algorithm you learned in elementary school really does work — although proving that it does, is a somewhat nontrivial matter!)

Proposition 1.7 *For any $m, n \in \mathbb{N}$ such that $n \neq 0$, there are unique $q, r \in \mathbb{N}$ such that $m = q \cdot n + r$ and $r < n$.*

PROOF. Define the predicate $P(m)$ as follows:

$$P(m): \quad \text{for any natural number } n \neq 0, \text{there are } q, r \in \mathbb{N} \text{ such that } m = q \cdot n + r \text{ and } r < n$$

We use induction to prove that $P(m)$ holds for every $m \in \mathbb{N}$.

BASIS: $m = 0$. Let $q = r = 0$. Since, for any $n \in \mathbb{N}$, $0 = 0 \cdot n + 0$, $P(0)$ holds.

INDUCTION STEP: Let $i \geq 0$ be an arbitrary natural number, and suppose that $P(i)$ holds; i.e., for any natural number $n \neq 0$ there are $q, r \in \mathbb{N}$ such that $i = q \cdot n + r$ and $r < n$. We must prove that $P(i + 1)$ holds as well. That is, we must prove that for any natural number $n \neq 0$ there are $q', r' \in \mathbb{N}$ such that $i + 1 = q' \cdot n + r'$ and $r' < n$. Since $r < n$, there are two cases: either $r < n - 1$ or $r = n - 1$. why these 2 cases?

CASE 1. $r < n - 1$. In this case let $q' = q$ and $r' = r + 1$. Note that $r' < n$. We have:

$$\begin{aligned}
i + 1 &= (q \cdot n + r) + 1 && \text{[by induction hypothesis]} \\
&= q \cdot n + (r + 1) && \text{[by associativity of addition]} \\
&= q' \cdot n + r' && \text{[because } q' = q \text{ and } r' = r + 1\text{]}
\end{aligned}$$

as wanted. Thus, $P(i + 1)$ holds in this case.

there are 2 cases here because r could
1. increment by 1
2. become 0; and q increment by 1

Case 2. $r = n - 1$. In this case let $q' = q + 1$ and $r' = 0$. Note that $r' < n$, because $n > 0$ (since n is a natural number and $n \neq 0$). We have:

$$\begin{aligned}
i + 1 &= (q \cdot n + r) + 1 && \text{[by induction hypothesis]}\\
&= \big(q \cdot n + (n - 1)\big) + 1 && \text{[because } r = n - 1\text{]}\\
&= q \cdot n + n\\
&= (q + 1) \cdot n\\
&= q' \cdot n + r' && \text{[because } q' = q + 1 \text{ and } r' = 0\text{]}
\end{aligned}$$

as wanted. Thus, $P(i + 1)$ holds in this case as well. prove uniqueness of q and r

We have shown that for any $m, n \in \mathbb{N}$ such that $n \neq 0$, the division of m by n has a quotient and a remainder. It remains to show that these are unique. The proof of this fact does not require induction. Suppose that m and n are natural numbers such that $n \neq 0$. Let $q, q', r, r' \in \mathbb{N}$ be such that $m = q \cdot n + r = q' \cdot n + r'$, where $r, r' < n$. To prove the uniqueness of the quotient and remainder we must prove that $q = q'$ and $r = r'$.

Since $q \cdot n + r = q' \cdot n + r'$, it follows that

$$(q - q') \cdot n = r' - r \tag{1.2}$$

Without loss of generality, we may assume that $q \geq q'$, i.e., $q - q' \geq 0$.[1] If $q - q' > 0$, then $q - q' \geq 1$, so $(q - q') \cdot n \geq n$ and, by (1.2), $r' - r \geq n$; this is impossible, since $r' < n$ and $r \geq 0$, so that $r' - r \leq r' < n$. Therefore $q - q' = 0$. By (1.2), $r' - r = 0$ as well. In other words, $q = q'$ and $r = r'$, as wanted. □

By Proposition 1.7, there are functions **div** : $\mathbb{N} \times (\mathbb{N} - \{0\}) \to \mathbb{N}$ and **mod** : $\mathbb{N} \times (\mathbb{N} - \{0\}) \to \mathbb{N}$, where **div**$(m, n)$ is the (unique) quotient and **mod**(m, n) is the (unique) remainder of the division of m by n. Proposition 1.7 can be easily extended so that m, n are integers (not just natural numbers). Now the quotient may be negative (if exactly one of m, n is negative), but the remainder r is in the range $0 \leq r < |n|$. The functions **div** and **mod** can be extended accordingly. **End of Example 1.5**

[1] The expression "without loss of generality", sometimes abbreviated as "wlog", is an idiom frequently used in mathematical proofs. It means that the argument about to be made holds thanks to an assumption (in our case, that $q \geq q'$) which, however, can be easily discharged — for example, by renaming some variables, by uniformly changing a + sign to a − sign, or the direction of inequality sign from < to a >, and so on. This allows us to avoid tedious repetition in a proof. This idiom, however, should be used carefully. In particular, when it is used, the assumption made should truly not restrict the applicability of the argument. Furthermore, it should be clear (to the typical reader of the proof) *why* the assumption does not restrict the applicability of the argument. Sometimes, it may be helpful to provide a hint as to why the assumption does no harm to generality. For instance, in our case we might have added the following remark to help the reader see why the assumption that $q \geq q'$ is truly without loss of generality: "(otherwise, switch the roles of q and q' and the roles of r and r' in the foregoing argument)".

1.2.2 Bases other than zero

There are predicates of interest that are true not for *all* natural numbers, but for all *sufficiently large* ones — i.e., for all natural numbers greater than or equal to some constant c. For example, $2^n > 10n$ is true for all $n \geq 6$, but it is easy to see that it is false for $0 \leq n < 6$. Suppose $c \in \mathbb{N}$ is a constant, and $P(n)$ is a predicate of natural numbers. We can prove a statement of the form

$$\text{for each } n \in \mathbb{N} \text{ such that } n \geq c,\ P(n) \text{ is true.} \tag{1.3}$$

using the following variant of induction.

Basis: Prove that $P(c)$ is true, i.e., that the predicate $P(n)$ holds for $n = c$.

Induction Step: Prove that, for each $i \in \mathbb{N}$ such that $i \geq c$, if $P(i)$ is true then $P(i+1)$ is also true.

Note that statement (1.3) is just another way of saying

$$\text{for each } n \in \mathbb{N},\ P(n+c) \text{ is true.} \tag{1.4}$$

Thus, the basis and induction step given above are just the corresponding steps of a normal proof by induction of statement (1.4).

Example 1.6 Any exponential function, no matter how small the base, eventually gets larger than any linear function, no matter how large its slope. The following proposition is a specific instance of this general fact.

Proposition 1.8 *For all natural numbers $n \geq 6$, $2^n > 10n$.*

Proof. Let $P(n)$ be the predicate defined as follows:

$$P(n): \qquad 2^n > 10n$$

We will use induction to prove that $P(n)$ is true for all natural numbers $n \geq 6$.

Basis: $n = 6$. $2^6 = 64$, while $10 \cdot 6 = 60$. Thus, $2^6 > 10 \cdot 6$, and $P(6)$ holds.

Induction Step: Let i be an arbitrary integer such that $i \geq 6$, and suppose that $P(i)$ holds, i.e., $2^i > 10i$. We must show that $P(i+1)$ holds as well, i.e., that $2^{i+1} > 10(i+1)$. Indeed, we have

$$\begin{aligned} 2^{i+1} &= 2 \cdot 2^i \\ &> 2 \cdot 10i && \text{[by induction hypothesis]} \\ &= 10i + 10i \\ &> 10i + 10 && \text{[because } i \geq 6 \text{ and thus } 10i > 10\text{]} \\ &= 10(i+1) \end{aligned}$$

Thus, $2^{i+1} > 10(i+1)$, as wanted. □

Notice that the requirement $i \geq 6$ was used not only in the basis (to start the induction with 6, rather than 0), but also in the induction step (to argue that $10i > 10$).

End of Example 1.6

Example 1.7 Suppose you have an unlimited supply of postage stamps, where each stamp has a face value of 4 cents or a face value of 7 cents. Can you use your supply of stamps to make *exact* postage for a letter that requires \$1.03 worth of stamps? The answer is affirmative; one way of achieving the desired goal is to use 3 of the 4-cent stamps and 13 of the 7-cent stamps (since $3 \times 4 + 13 \times 7 = 103$). In fact, we can prove something much more general:

Proposition 1.9 *We can use an unlimited supply of 4-cent and 7-cent postage stamps to make (exactly)* any *amount of postage that is 18 cents or more.*

Proof. Let $P(n)$ be the predicate defined as follows:

$P(n)$: postage of exactly n cents can be made using only 4-cent and 7-cent stamps

We will use induction to prove that $P(n)$ holds for all $n \geq 18$.

Basis: $n = 18$. We can use one 4-cent stamp and two 7-cent stamps to make 18 cents worth of postage (since $1 \times 4 + 2 \times 7 = 18$).

Induction Step: Let i be an arbitrary integer such that $i \geq 18$, and suppose that $P(i)$ holds, i.e., we can make i cents of postage using only 4-cent and 7-cent stamps. Thus, there are $k, \ell \in \mathbb{N}$ such that $i = 4 \cdot k + 7 \cdot \ell$ (k is the number of 4-cent stamps and ℓ is the number of 7-cent stamps used to make i cents worth of postage). We will prove that $P(i+1)$ holds, i.e., that we can make $i+1$ cents of postage using only 4-cent and 7-cent stamps. In other words, we must prove that there are $k', \ell' \in \mathbb{N}$ such that $4 \cdot k' + 7 \cdot \ell' = i + 1$. There are two cases, depending on whether $\ell > 0$ or $\ell = 0$.

Case 1. $\ell > 0$. Then let $k' = k + 2$ and $\ell' = \ell - 1$ (note that since $\ell > 0$, $\ell' \in \mathbb{N}$). (Intuitively we trade in one 7-cent stamp — which we know we have, since $\ell > 0$ — for two 4-cent stamps; the result is to increase the total value of the postage by 1.) More precisely, we have:

$$\begin{aligned}
4 \cdot k' + 7 \cdot \ell' &= 4 \cdot (k+2) + 7 \cdot (\ell - 1) && \text{[by definition of } k' \text{ and } \ell'] \\
&= 4 \cdot k + 8 + 7 \cdot \ell - 7 \\
&= 4 \cdot k + 7 \cdot \ell + 1 \\
&= i + 1 && \text{[since, by induction hypothesis, } i = 4 \cdot k + 7 \cdot \ell]
\end{aligned}$$

Case 2. $\ell = 0$. Since $i \geq 18$, we have $4 \cdot k \geq 18$ and therefore $k \geq 5$. Then let $k' = k - 5$ and $\ell' = 3$ (note that since $k \geq 5$, $k' \in \mathbb{N}$). (Now we don't have a 7-cent stamp to trade in, but because $i \geq 18$, we have at least five 4-cent stamps, which we can trade in for three 7-cent stamps, again increasing the total value of the postage by 1.) More precisely, we have:

$$\begin{aligned}
4 \cdot k' + 7 \cdot \ell' &= 4 \cdot (k-5) + 7 \cdot 3 && \text{[by definition of } k' \text{ and } \ell'\text{]} \\
&= 4 \cdot k - 20 + 21 \\
&= 4 \cdot k + 1 \\
&= 4 \cdot k + 7 \cdot \ell + 1 && \text{[since } \ell = 0 \text{ in this case]} \\
&= i + 1 && \text{[since, by induction hypothesis, } i = 4 \cdot k + 7 \cdot \ell\text{]}
\end{aligned}$$

Thus, in both cases $P(i+1)$ holds, as wanted. □

End of Example 1.7

1.2.3 Pitfalls

Leaving out the basis

The basis of an induction proof is often (though not always) much easier to prove than the induction step. This, however, does not mean that we should feel free to leave out the basis of an induction proof. An induction "proof" in which the basis is omitted is *not* a sound way to prove that a predicate holds for all natural numbers. The following example should convince you of this.

Example 1.8

Conjecture 1.10 *For any $n \in \mathbb{N}$, $\sum_{t=0}^{n} 2^t = 2^{n+1}$.*

"PROOF". Define the predicate

$$P(n): \qquad \sum_{t=0}^{n} 2^t = 2^{n+1}.$$

We "prove" that $P(n)$ holds for all $n \in \mathbb{N}$ by "induction".

INDUCTION STEP: Let i be an arbitrary natural number, and suppose that $P(i)$ holds, i.e., $\sum_{t=0}^{i} 2^t = 2^{i+1}$. We must prove that $P(i+1)$ holds as well, i.e., that $\sum_{t=0}^{i+1} 2^t = 2^{i+2}$. Indeed, we have

$$\begin{aligned}
\sum_{t=0}^{i+1} 2^t &= \Big(\sum_{t=0}^{i} 2^t\Big) + 2^{i+1} && \text{[by associativity of addition]} \\
&= 2^{i+1} + 2^{i+1} && \text{[by induction hypothesis]} \\
&= 2 \cdot 2^{i+1} \\
&= 2^{i+2}
\end{aligned}$$

as wanted. □

Now, let's check what we have supposedly proved for, say, $n = 3$. $P(3)$ states that $2^0 + 2^1 + 2^2 + 2^3 = 2^4$. However, the left hand side of this evaluates to $1+2+4+8 = 15$, while the right hand side evaluation to 16. So something is wrong here.

What is wrong, of course, is that in the above inductive "proof" we left out the basis, $P(0)$. If we had tried to prove that $P(0)$ is true, we would have discovered the error, because $P(0)$ states that $2^0 = 2^1$, which is clearly false.

The correct fact is that for any $n \in \mathbb{N}$, $\sum_{t=0}^{n} 2^t = 2^{n+1} - 1$. As an exercise, prove this by using induction. **End of Example 1.8**

Changing the predicate in the induction step

A somewhat more subtle mistake is sometimes committed in carrying out the induction step. The mistake consists in proving that $P(i+1)$ holds using as the "inductive hypothesis" that $P'(i)$ holds, where P' is a predicate stronger than P. (In a correct proof the induction hypothesis must be that $P(i)$ holds.) We will illustrate this type of error in the two examples that follow.

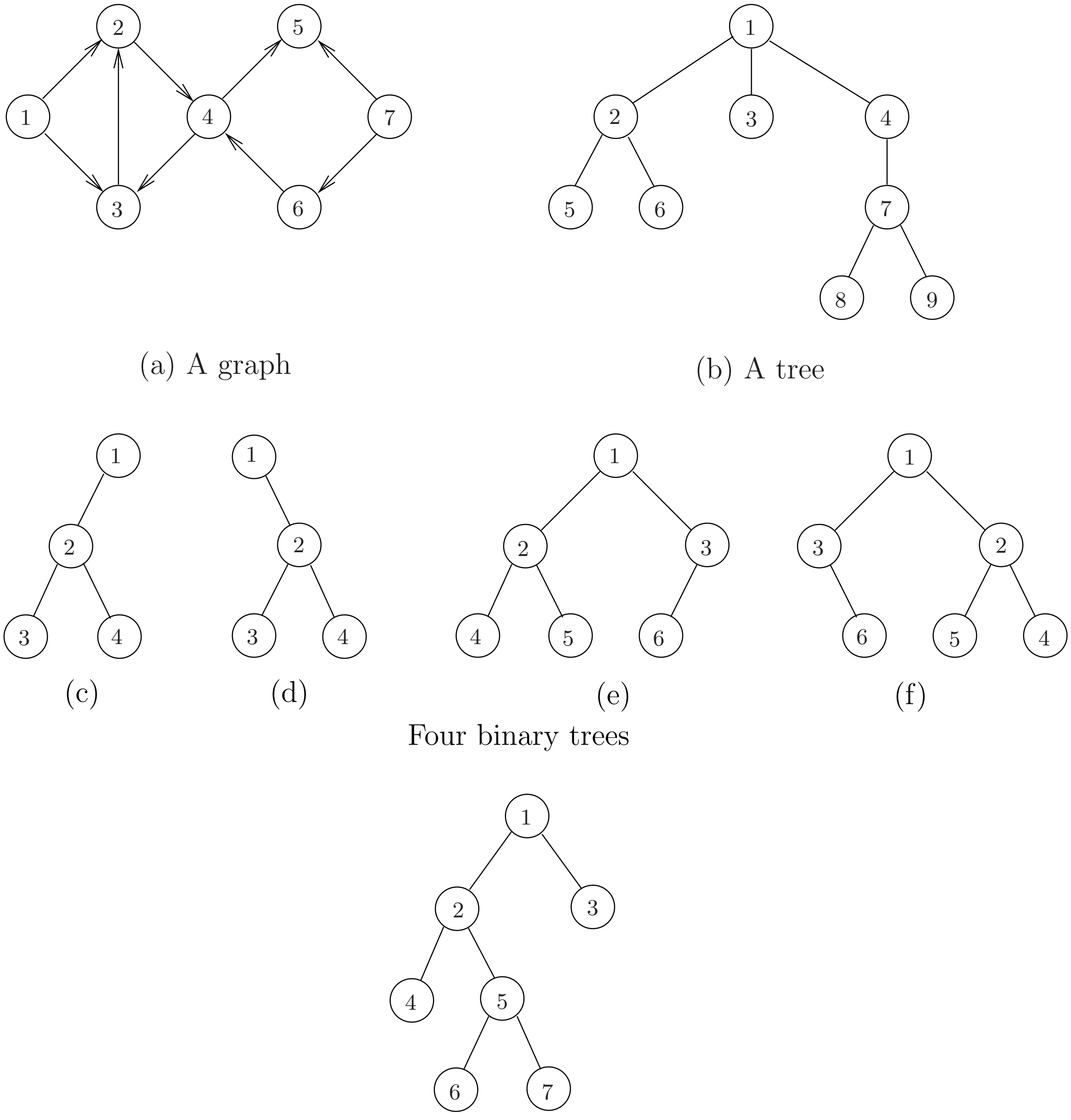
First we need to refresh our memory about binary trees. Recall that a (directed) graph G consists of a set of nodes N and a set of edges E, where $E \subseteq N \times N$. Thus, an edge e is an ordered pair of nodes (u, v); we say that e goes *from* u to v. A ***path*** in graph G is a sequence of nodes $\langle u_1, u_2, \ldots, u_k \rangle$, such that for each i, $1 \leq i < k$, (u_i, u_{i+1}) is an edge of G. A graph is shown diagrammatically in Figure 1.1(a); its nodes are drawn as circles, and its edges as arrows. Specifically, the edge (u, v) is drawn as an arrow with its tail in u and its head in v.

A ***tree*** is a directed graph G with the following property: If the set of nodes of G is nonempty, then G has a node r, called G's ***root*** such that, for every node u, there is one and only one path from r to u. If (u, v) is an edge of a tree, we say that v is a ***child*** of u, and that u is the ***parent*** of v.[2] Nodes with the same parent are called ***siblings***. A node that has no children is called a ***leaf***. A node that is not a leaf is called an ***internal node***. It is easy to see that a nonempty tree with finitely many nodes has at least one leaf. Furthermore, if from a tree we remove a leaf (and the edge, if any,[3] that connects it to its parent), the resulting object is also a tree. A tree is shown diagrammatically in Figure 1.1(b). It is customary to draw the root of the tree at the top, and to omit the direction of the edges, the convention being that edges are directed "away" from the root.

A ***binary tree*** is a tree every node of which has at most two children, together with a labeling of each edge as either "left" or "right", so that edges from a node to distinct children have different labels. If edge (u, v) is labeled "left" (respectively, "right"), then we say that v is the ***left child of*** u (respectively, v is the ***right child of*** u). When we draw a binary tree,

[2] A tree node cannot have two parents; if it did, there would be multiple distinct paths from the root to it: at least one through each of its parents. This fact justifies our talking about *the* parent (rather than *a* parent) of v.

[3] If the root is also a leaf, then the tree has just one node and no edges.



(a) A graph

(b) A tree

(c)

(d)

(e)

(f)

Four binary trees

(g) A full binary tree

Figure 1.1: Illustration of graphs and trees

the direction (left or right) of the edges emanating from u indicates their labels. Some binary trees are shown diagrammatically in Figures 1.1(c)–(g). Note that viewed as *binary trees*, the objects depicted in Figures 1.1(c) and (d) are different; viewed as trees, however, they are identical. The reason is that in binary trees it matters whether a node is the left or right child of another. In contrast, in trees only the parent-child relationships matter, not the ordering of siblings. Similar remarks apply to the pair of objects depicted in Figures 1.1(e) and (f).

A ***full*** binary tree is a binary tree in which every internal node has exactly two children. For example, Figure 1.1(g) is a full binary tree, while Figures 1.1(c)–(f) are not.

Example 1.9

Conjecture 1.11 *For any $n \geq 1$, any full binary tree with n nodes has more leaves than the square of the number of its internal nodes.*

"Proof". Define the predicate $P(n)$ as follows:

$P(n)$: any full binary tree with n nodes, has more leaves than the square of the number of its internal nodes.

We use induction to "prove" that $P(n)$ holds for all $n \geq 1$. a single root node is a leaf

Basis: $n = 1$. There is only one full binary tree with one node, consisting of just that node and no edges. Thus, a full binary tree has one leaf and zero internal nodes. Since $1 > 0^2$, $P(1)$ holds, as wanted.

Induction Step: Let $i \geq 1$ be an arbitrary integer, and assume that $P(i)$ holds i.e., any full binary tree with i nodes has more leaves than the square of the number of its internal nodes. We must prove that $P(i+1)$ holds as well, i.e., that any full binary tree with $i+1$ nodes has more leaves than the square of the number of its internal nodes.

Let T be an arbitrary full binary tree with $i+1$ nodes. Let v be a leaf of T, and let v' be the parent of v. (Note that v' exists, because v cannot be the only node of T, since T has $i+1 \geq 2$ nodes.) Let T' be the binary tree obtained by removing the leaf v from T. T' has i nodes.

Let ℓ be the number of leaves of T, and m be the number of internal nodes of T. Similarly, let ℓ' be the number of leaves of T', and m' be the number of internal nodes of T'. We must prove that $\ell > m^2$.

Since T is a full binary tree and v is a leaf of T, it follows that v' has another child in T, and therefore v' is not a leaf in T'. Hence, $\ell = \ell' + 1$ (T has one more leaf than T', namely v), and $m = m'$ (T has the same number of internal nodes as T', since the removal of v does not make v' a leaf in T'). Therefore we have:

$$\begin{aligned} \ell &= \ell' + 1 && \text{[as argued above]} \\ &> {m'}^2 + 1 && \text{[by induction hypothesis; recall that } T' \text{ has } i \text{ nodes]} \\ &= m^2 + 1 && \text{[since } m' = m\text{, as argued above]} \\ &> m^2 \end{aligned}$$

Thus, $\ell > m^2$, which means that $P(i+1)$ holds, as wanted. □

The binary tree shown in Figure 1.1(g) is a full binary tree with 7 nodes, 3 of which are internal nodes and 4 are leaves. However, it is *not* the case that $4 > 3^2$ — so $P(7)$ does not hold, as this counterexample shows. Evidently, something is wrong with the proof. *Before you continue reading you should carefully go over the "proof" and try to identify the mistake, if you have not already done so.*

The mistake lies in our use of the induction hypothesis. We said that T' is a binary tree with i nodes and therefore we can apply the induction hypothesis $P(i)$ to conclude that $\ell' > {m'}^2$. But the induction hypothesis applies to *full* binary trees with i nodes. T' is a binary tree with i nodes, but is it a full one? Some thought will convince you that it is not — therefore we had no right to apply the induction hypothesis to it.

In more general terms, the mistake we made can be stated as follows. Let $P'(n)$ be the predicate

$P'(n)$: any binary tree with n nodes, has more leaves than the square of the number of its internal nodes.

The difference between $P(n)$ and $P'(n)$ is that $P(n)$ talks about *full* binary trees, while $P'(n)$ talks about *all* binary trees (regardless of whether they are full). $P'(n)$ is stronger than $P(n)$, since it applies to a larger class of objects. In a correct induction proof, in the induction step we are supposed to prove that $P(i+1)$ holds assuming $P(i)$ holds. In the incorrect "proof" given above, we proved that $P(i+1)$ holds assuming that $P'(i)$ (a stronger property than $P(i)$) holds! It was this inappropriate use of induction that led us astray. **End of Example 1.9**

Example 1.10 In this example we examine a mistake similar to the one above, but with a different, and very important, moral.

Proposition 1.12 *For any $n \geq 1$, a full binary tree with n nodes has $n - 1$ edges.*

"PROOF". Define the predicate $P(n)$ as follows:

$P(n)$: any full binary tree with n nodes, has $n - 1$ edges

We use induction to "prove" that $P(n)$ holds for all $n \geq 1$.

Basis: $n = 1$. There is only one full binary tree with one node, consisting of just that node and no edges. Thus, $P(1)$ holds, as wanted.

Induction Step: Let $i \geq 1$ be an arbitrary integer, and assume that $P(i)$ holds; i.e., any full binary tree with i nodes has $i - 1$ edges. We must prove that $P(i + 1)$ holds as well; i.e., that any full binary tree with $i + 1$ nodes has i edges.

Let T be an arbitrary full binary tree with $i + 1$ nodes. We must prove that T has i edges.

Let v be a leaf of T and T' be the binary tree obtained by removing the leaf v from T. T' has i nodes (because T has $i + 1$ nodes and v is not in T'), and one edge less than T (because the edge connecting v to its parent is not in T'). We have:

$$\begin{aligned}
\#\text{ of edges in } T &= (\#\text{ of edges in } T') + 1 && \text{[as argued above]}\\
&= \big((\#\text{ of nodes in } T') - 1\big) + 1 && \text{[by induction hypothesis; recall that } T' \text{ has } i \text{ nodes]}\\
&= \#\text{ of nodes in } T'\\
&= i
\end{aligned}$$

use structural proof instead

Thus, T has i edges, as wanted. □

This proof is incorrect for exactly the same reason we saw in the previous example: We applied the induction hypothesis to a binary tree T' that is not full, while the predicate we are trying to prove is talking specifically about full binary trees. Unlike the previous example, however, hard as we may try, we will not succeed in finding a counterexample to Proposition 1.12. This is because, although our "proof" was incorrect, the proposition itself is actually true!

How do we prove it, then? We can use induction to prove a *stronger* proposition:

Proposition 1.13 *For any $n \geq 1$, a binary tree with n nodes has $n - 1$ edges.*

Notice that this proposition talks about *all* binary trees (not just full ones). It is easy to verify that this proposition is proved by the induction argument given above, modified by striking out all references to *full* binary trees and replacing them by references simply to binary trees.[4] Having proved Proposition 1.13, Proposition 1.12 follows immediately, since a full binary tree is a special case of a binary tree.

This example is a simple illustration of an important and somewhat paradoxical phenomenon regarding inductive proofs. Sometimes, when we use induction to prove that a predicate $P(n)$ holds for all $n \in \mathbb{N}$, we stumble in the induction step and realise that in order to prove that $P(i+1)$ holds we really need to assume $P'(i)$, rather than $P(i)$, as our induction hypothesis, where $P'(n)$ is a stronger predicate than $P(n)$.

[4] As an aside, an even stronger proposition holds and can be proved in a similar way; this relationship between the number of nodes and edges applies to all trees, whether binary or not.

Of course, this is not legitimate; if we did this we would be using an incorrect (and therefore unsound) induction proof along the lines of the two preceding examples. In such a case, however, it is advisable to try using induction in order to prove the stronger fact that $P'(n)$ holds for all $n \in \mathbb{N}$. If we succeed in doing so then, of course, the desired weaker fact — namely, that $P(n)$ holds for all $n \in \mathbb{N}$ — follows immediately. There is no guarantee that we will succeed; after all, it is possible that we are trying to prove a false statement, as in the previous example. Sometimes, however, this works — as in the current example.

It may seem counterintuitive that proving a *stronger* statement is easier than proving a weaker one. The apparent paradox is resolved if we think a little more carefully about the nature of induction proofs. In the induction step, we use the induction hypothesis that the predicate of interest holds for i, in order to prove that the predicate of interest holds for $i+1$. It is possible that, by strengthening the induction hypothesis, we get the leverage required to carry out the induction step, while a weaker induction hypothesis would not allow us to do so. **End of Example 1.10**

1.3 Complete induction

In carrying out the induction step of an inductive proof, we sometimes discover that to prove that $P(i+1)$ holds, it would be helpful to know that $P(j)$ holds for one or more values of j that are smaller than $i+1$, but are not necessarily i. For example, it might be that to prove that $P(i+1)$ holds, we need to use the fact that $P(\lfloor i/2 \rfloor)$ holds, rather than the fact that $P(i)$ holds.[5] Or, we might need to use the fact that both $P(\lfloor i/2 \rfloor)$ and $P(i)$ hold. The induction proofs we have seen so far, which are based on the principle of induction (see Section 1.1.2), are of no help is such situations. There is a different type of induction proof, based on the principle of complete induction (see Section 1.1.3), that is designed to address just this sort of situation. This type of proof is called ***complete induction***.[6]

Let $P(n)$ be a predicate of natural numbers. A complete-induction proof of the statement "$P(n)$ holds for all $n \in \mathbb{N}$" consists in proving the following statement:

$(*)$ For each $i \in \mathbb{N}$, if $P(j)$ holds for all natural numbers $j < i$ then $P(i)$ holds as well.

The assumption, in $(*)$, that $P(j)$ holds for all natural numbers $j < i$ is called the ***induction hypothesis***. Thus, a proof by complete induction proceeds as follows:

- We let i be an arbitrary natural number.
- We assume that $P(j)$ holds for all natural numbers $j < i$ (induction hypothesis).
- Using the induction hypothesis, we prove that $P(i)$ holds as well.

[5] If x is a real number, then $\lfloor x \rfloor$ (called the ***floor*** of x) denotes the largest integer that is smaller than or equal to x; similarly $\lceil x \rceil$ (the ***ceiling*** of x) denotes the smallest integer that is greater than or equal to x. Thus $\lfloor 17.3 \rfloor = 17$, $\lceil 17.3 \rceil = 18$, $\lfloor 17 \rfloor = \lceil 17 \rceil = 17$.

[6] Some mathematicians use the terms ***course-of-values induction*** and ***strong induction*** instead of complete induction.

Why does proving $(*)$ actually show that $P(n)$ holds for all $n \in \mathbb{N}$? To see this, let $A[P]$ be the set of natural numbers for which the predicate P is true; i.e., $A[P] = \{n \in \mathbb{N} : P(n) \text{ is true}\}$. Proving $(*)$ means that we prove that, for any $i \in \mathbb{N}$, if every natural number less than i is an element of $A[P]$ then i is also an element of $A[P]$. By the principle of complete induction (see Section 1.1.3), $A[P]$ is a superset of $\mathbb{N}$. By definition, $A[P]$ is a set of natural numbers, so it is also a subset of $\mathbb{N}$. Hence, $A[P] = \mathbb{N}$. By the definition of $A[P]$, this means that $P(n)$ holds for every $n \in \mathbb{N}$.

1.3.1 A second form of complete induction

Sometimes complete induction proofs take a somewhat different form. In that form, the proof that $P(n)$ holds for all $n \in \mathbb{N}$ consists of two steps, as in the case of simple induction:

Basis: Prove that $P(0)$ is true, i.e., that the predicate $P(n)$ holds for $n = 0$.

Induction Step: Prove that, for each natural number $i > 0$, if $P(j)$ holds for all natural numbers $j < i$ then $P(i)$ holds as well.

The second form of complete induction seems to differ from the previous form, in that it has an explicit basis case, for $n = 0$. As we saw in our discussion of the difference between the principles of (simple) induction and complete induction (see page 18, Section 1.1.3), the case $n = 0$ is implicit in $(*)$. Both forms of complete induction prove that $P(n)$ holds for all $n \in \mathbb{N}$. We can use whichever is more convenient for the specific predicate $P(n)$ of interest.

Speaking in general terms, induction lets us prove that a certain result holds for all numbers, by proving that

(i) it holds for certain "simplest" numbers, and

(ii) it holds for "more complex" numbers, if it holds for "simpler" ones.

In many instances, "simplest" just means "smallest" — i.e., 0 or some other single basis case. There are instances, however, where the meaning of "simplest" is different, and there are several "simplest" numbers that can be treated together as a single special case. In such instances, the first form of complete induction leads to more natural and more elegant proofs. (See Example 1.11.)

1.3.2 Bases other than zero

As with simple induction, sometimes we need to prove that a predicate holds, not for all natural numbers, but for all natural numbers greater than or equal to some constant, say c. We can prove such a statement using variants of complete induction.

The variant of the first form of complete induction consists in proving the following statement:

$(*)$ For each $i \in \mathbb{N}$ such that $i \geq c$, if $P(j)$ holds for all natural numbers j such that $c \leq j < i$ then $P(i)$ holds as well.

This proves that $P(n)$ holds for all $n \geq c$. (Why?)

The variant of the second from of complete induction consists of the following two steps:

BASIS: Prove that $P(c)$ is true.

INDUCTION STEP: Prove that, for each natural number $i > c$, if $P(j)$ holds for all natural numbers j such that $c \leq j < i$ then $P(i)$ holds as well.

This, too, proves that $P(n)$ holds for all $n \geq c$. (Why?)

1.3.3 Examples

Example 1.11 An excellent illustration of complete induction is afforded by the proof of a fundamental result in number theory: every natural number that is greater than or equal to 2 can be written as the product of prime numbers.

First we need to recall some definitions. If n, m are integers, we say that n is ***divisible*** by m if the division of n by m has no remainder, i.e., if n/m is an integer. An integer n is ***prime*** if $n \geq 2$ and the only positive integers that divide n are 1 and itself.[7] A ***prime factorisation*** of a natural number n is a sequence of primes whose product is n. Notice that a prime factorisation, being a *sequence* may contain the same (prime) number repeatedly. For example, $\langle 3, 2, 2, 7, 3\rangle$ is a prime factorisation of 252.

Proposition 1.14 *Any integer $n \geq 2$, has a prime factorisation.*

PROOF. Define the predicate $P(n)$ as follows:

$$P(n): \qquad n \text{ has a prime factorisation}$$

We use complete induction to prove that $P(n)$ holds for all integers $n \geq 2$.

Let i be an arbitrary integer such that $i \geq 2$. Assume that $P(j)$ holds for all integers j, such that $2 \leq j < i$. We must prove that $P(i)$ holds as well. There are two cases.

CASE 1. i is prime. Then $\langle i\rangle$ is a prime factorisation of i. Thus, $P(i)$ holds in this case.

CASE 2. i is not prime. Thus, there is a positive integer a that divides i such that $a \neq 1$ and $a \neq i$. Let $b = i/a$; i.e., $i = a \cdot b$. Since $a \neq 1$ and $a \neq i$, it follows that a, b are both integers such that $2 \leq a, b < i$. Therefore, by the induction hypothesis, $P(a)$ and $P(b)$ both hold. That is, there is a prime factorisation of a, say $\langle p_1, p_2, \ldots, p_k\rangle$, and there is a prime factorisation of b, say $\langle q_1, q_2, \ldots, q_\ell\rangle$. Since $i = a \cdot b$, it is obvious that the concatenation of the prime factorisations of a and b, i.e., $\langle p_1, p_2, \ldots, p_k, q_1, q_2, \ldots, q_\ell\rangle$, is a prime factorisation of i. Therefore, $P(i)$ holds in this case as well.

[7] Keep in mind that, according to this definition, 1 is *not* a prime number. Although the only positive integers that divide it are 1 and itself, it is not greater than or equal to 2. This exclusion of 1 from the set of prime numbers may appear arbitrary, but there is a good reason for it, as we will see shortly.

Therefore, $P(n)$ holds for all $n \geq 2$. □

Notice that there is no obvious way in which we could have done this proof by using simple induction, as opposed to complete induction. Just knowing the prime factorisation of i gives us absolutely no clue about how to find a prime factorisation of $i+1$. However, if we know the factorisation of *all* numbers less than i, then we can easily find a prime factorisation of i: if i is prime, then it is its own prime factorisation, and we are done; if i is not prime, then we can get a prime factorisation of i by concatenating the prime factorisations of two factors (which are smaller than i and therefore whose prime factorisation we know by induction hypothesis).

Also notice how convenient was the use of the first form of complete induction (the one with no explicit basis) in this proof. The reason is that there are actually infinitely many "simplest" numbers as far as prime factorisations go: Every prime number has a trivial prime factorisation, namely itself. Thus, there is nothing special about the basis case of $n = 2$; the argument used to prove that $P(2)$ holds is also used to prove that $P(m)$ holds for every prime number m. It is more succinct and elegant to treat all these cases at once, as in the preceding proof.

There is actually more to prime factorisations of natural numbers than Proposition 1.14 states. More specifically, not only does every natural number $n \geq 2$ have a prime factorisation, but the factorisation is essentially unique — up to permuting the prime numbers in the sequence. That is, if two sequences are prime factorisations of the same number, then they are permutations of each other. Proving the uniqueness of prime factorisations also involves complete induction, but requires a few basic results about number theory, and so we will not prove this fact in this course.

By the way, in view of this, we can see why 1 should not be considered as a prime number. If it were, prime factorisation would not be unique (up to reordering of the factors). Any prime factorisation of a number could be extended by introducing an arbitrary number of 1s in the sequence. This would not affect the product of the numbers in the sequence, but would result in a sequence that is not a permutation of the original one. We could allow 1 to be a prime, thus slightly simplifying the definition of primality, at the expense of complicating the sense in which a prime factorisation is unique. It turns out that doing so is not a good tradeoff, and this is the reason that 1 is by definition *not* prime. **End of Example 1.11**

Example 1.12 We reprove Proposition 1.9 (see Example 1.7) using complete induction.

Alternative proof of Proposition 1.9. Let $P(n)$ be the predicate defined as follows:

$$P(n): \quad \text{postage of exactly } n \text{ cents can be made using only 4-cent and 7-cent stamps}$$

We will use complete induction to prove that $P(n)$ holds for all $n \geq 18$.

Let i be an arbitrary integer such that $i \geq 18$, and assume that $P(j)$ holds for all j such that $18 \leq j < i$. We will prove that $P(i)$ holds as well.

Case 1. $18 \leq i \leq 21$. We can make postage for

- 18 cents using one 4-cent stamp and two 7-cent stamps ($18 = 1 \times 4 + 2 \times 7$);
- 19 cents using three 4-cent stamp and one 7-cent stamps ($19 = 3 \times 4 + 1 \times 7$);
- 20 cents using five 4-cent stamps ($20 = 5 \times 4$);
- 21 cents using three 7-cent stamps ($21 = 3 \times 7$).

Thus, $P(i)$ holds for $18 \leq i \leq 21$.

the idea is to set j and i 4 units apart… according to smallest postage difference.

Case 2. $i \geq 22$. Let $j = i - 4$. Thus, $18 \leq j < i$ and therefore, by induction hypothesis, $P(j)$ holds. This means that there are $k, \ell \in \mathbb{N}$ such that $j = 4 \cdot k + 7 \cdot \ell$. Let $k' = k + 1$ and $\ell' = \ell$. We have:

$$\begin{aligned} 4 \cdot k' + 7 \cdot \ell' &= 4 \cdot (k+1) + 7 \cdot \ell && \text{[by definition of } k' \text{ and } \ell'\text{]} \\ &= 4 \cdot k + 7 \cdot \ell + 4 \\ &= j + 4 && \text{[by induction hypothesis]} \\ &= i \end{aligned}$$

Thus, $P(i)$ holds in this case, as well.

Thus, in both cases $P(i)$ holds, as wanted. □

End of Example 1.12

Example 1.13 If you draw a few *full* binary trees, you may notice that each has an odd number of nodes and you may suspect that this is true for *all* full binary trees. You may therefore formulate the following

Conjecture 1.15 *For any positive integer n, if a full binary tree has n nodes then n is odd.*

Induction is a natural method to try using in order to prove this conjecture. Our attempts to prove things about full binary trees using simple induction (recall Examples 1.9 and 1.10) were fraught with difficulties. The basic problem was that the removal of a leaf from a full binary tree yields a binary tree with one less node which, however, is not full — and, therefore, to which we cannot apply the induction hypothesis.

In the case of Conjecture 1.15, we cannot possibly get around this difficulty by using the technique of Example 1.10 — i.e., strengthening the statement we wish to prove by applying it to all binary trees, not just full ones. This is because the strengthened statement is false: It is certainly possible to have binary trees with an even number of nodes! In this case, complete induction comes to the rescue.

Let T be a tree and u be an arbitrary node of T. The nodes of T that are reachable from u (i.e., the set of nodes to which there is a path from u) and the edges through which these nodes are reached from u form a tree, whose root is u. This is called ***the subtree of T rooted at u***. The term "subtree of T" (without reference to a specific node as its root) refers to a subtree rooted at one of the children of T's root. For example, the tree in Figure 1.1(b) has three subtrees: one contains nodes 2, 5 and 6; one contains just node 3; and one contains nodes 4, 7, 8 and 9.

Thus, trees have an interesting recursive structure: The subtrees of a tree are smaller trees, which can be further decomposed into yet smaller trees, and this decomposition process can continue recursively until we have arrived at the smallest possible trees. (Depending on what it is we are trying to prove about trees, the "smallest possible trees" may be trees with just one node, trees with no nodes, or some other "basis" case.) Thus, it is natural to prove things about trees inductively by applying the inductive hypothesis to subtrees, and using this to prove that the desired property holds for the entire tree. In general, this is an application of complete induction, since the subtrees are "smaller" (say, in number of nodes) than the tree, but they are not necessarily smaller by one.

It is easy to see that if we apply the decomposition described above to *binary* trees, the resulting subtrees are themselves binary. Similarly, if we apply it to *full* binary trees, the resulting subtrees are themselves full binary trees. Thus, we can use this idea to prove inductively things about binary trees or about full binary trees. We will illustrate this by proving the above conjecture.

PROOF OF CONJECTURE 1.15. Let $P(n)$ be the following predicate:

$$P(n): \qquad \text{if a full binary tree has } n \text{ nodes, then } n \text{ is an odd number.}$$

We will prove that $P(n)$ holds for all integers $n \geq 1$ using complete induction.

Let i be an arbitrary integer $i \geq 1$, and suppose that $P(j)$ holds for all positive integers $j < i$. That is, for any positive integer $j < i$, if a full binary tree has j nodes, then j is an odd number. We will prove that $P(i)$ holds as well. That is, we will prove that if a full binary tree has i nodes, then i is an odd number. Let T be an arbitrary full binary tree with i nodes; we must prove that i is odd. There are two cases, depending on whether $i = 1$ or $i > 1$.

CASE 1. $i = 1$. Obviously, i is odd in this case.

CASE 2. $i > 1$. Let T_1 and T_2 be the two subtrees of T, and let i_1 and i_2 be the number of nodes of T_1 and T_2, respectively. The nodes of T are: the root, the nodes of T_1 and the nodes of T_2. Thus,

$$i = i_1 + i_2 + 1 \tag{1.5}$$

Since T is a full binary tree, T_1 and T_2 are also full binary trees. Since T has more than one node (because $i > 1$) and is a full binary tree, each of T_1 and T_2 has at least one node, so $i_1, i_2 \geq 1$. Since i_1 and i_2 are nonnegative numbers, (1.5) implies that $i_1, i_2 < i$. Thus, $1 \leq i_1, i_2 < i$; by the induction hypothesis, $P(i_1)$ and $P(i_2)$ hold. Thus, i_1 and i_2 are odd numbers. By (1.5), i is an odd number.

□

End of Example 1.13

1.3.4 Pitfalls

In addition to the problems we discussed Section 1.2.3 (i.e., omitting the basis case or inadvertently strengthening the induction hypothesis in the induction step), in complete induction proofs we must watch out for another problem. Suppose we are using complete induction to prove that $P(n)$ holds for all integers $n \geq c$, where $c \in \mathbb{N}$ is some constant. In the induction step of this proof we consider an arbitrary natural number i, and we prove that $P(i)$ holds assuming that $P(j)$ holds for all integers j in the interval $c \leq j < i$. Thus, if the argument that $P(i)$ holds uses the fact that $P(j)$ holds, we must be careful to ensure that both (a) $j \geq c$, and (b) $j < i$. If not, then we have no right to assume that $P(j)$ holds as part of the induction hypothesis.

It is requirements (a) and (b) that dictate for which values of n we must prove that the predicate $P(n)$ holds with no recourse to the induction hypothesis — i.e., what should be the "basis cases" of the induction proof. To illustrate this point, suppose that we wish to prove that $P(n)$ holds for all $n \geq 18$, and in the induction step of this proof we are trying to prove that $P(i)$ holds using, as the induction hypothesis, that $P(i-4)$ holds, where $i \geq 18$. Of course, $i - 4 < i$, so (b) is automatically satisfied. Because of requirement (a), however, we can use the induction hypothesis $P(i-4)$ only if $i - 4 \geq 18$; i.e., only if $i \geq 22$. This means that this induction step is valid only if $i \geq 22$, and suggests that the truth of $P(18)$, $P(19)$, $P(20)$ and $P(21)$ should be handled separately, without recourse to the induction hypothesis. In fact, we have already seen an instance just like this in Example 1.12. You should review the proof in that example and make sure you understand why the proof would be incomplete if we had only proved that $P(18)$ holds in Case 1.

Now we consider a situation where it is the satisfaction of requirement (b) that imposes restrictions on the values of i for which the induction step can be carried out. Suppose that we wish to prove that $P(n)$ holds for all $n \geq 0$, and in the induction step we are trying to prove that $P(i)$ holds using, as the induction hypothesis, that $P(\lceil \frac{i+1}{2} \rceil)$ holds, where $i \geq 0$. It turns out that $\lceil \frac{i+1}{2} \rceil < i$, for all $i \geq 3$; however, $\lceil \frac{i+1}{2} \rceil \geq i$, for $0 \leq i < 3$. This means that the induction step in such a proof is valid *only* if $i \geq 3$, and suggests that the truth of $P(0)$, $P(1)$ and $P(2)$ must be handled separately, with no recourse to the induction hypothesis.

Exercises

1. Describe a set A that satisfies properties (i) and (ii) of the induction principle (see Section 1.1.2, page 17) and is a *proper* superset of $\mathbb{N}$.

2. When we do an induction proof, is it necessary to prove the basis before we prove the Induction Step, or could we prove the two steps in either order?

3. Let $P(n)$ be a predicate of natural numbers. Suppose we prove the following facts:

- $P(0)$ holds
- $P(1)$ holds
- for any $i \geq 0$, if $P(i)$ holds then $P(i+2)$ holds

Does this constitute a valid proof that $P(n)$ holds for all $n \in \mathbb{N}$? Justify your answer.

4. Let $P(n)$ be a predicate of the integers. Suppose we prove the following facts:

- $P(0)$ holds
- for any $i \geq 0$, if $P(i)$ holds then $P(i+1)$ holds
- for any $i \geq 0$, if $P(i)$ holds then $P(i-1)$ holds

Does this constitute a valid proof that $P(n)$ holds for all $n \in \mathbb{Z}$? Justify your answer.

5. Let $P(n)$ be a predicate of the integers. Suppose we prove the following facts:

- $P(0)$ holds
- for any $i \geq 0$, if $P(i)$ holds then $P(i+1)$ holds
- for any $i \leq 17$, if $P(i)$ holds then $P(i-1)$ holds

Does this constitute a valid proof that $P(n)$ holds for all $n \in \mathbb{Z}$? Justify your answer.

6. Let $P(n)$ be a predicate of the integers. Suppose we prove the following facts:

- $P(17)$ holds
- for any $i \geq 17$, if $P(i)$ holds then $P(i+1)$ holds
- for any $i \leq -17$, if $P(i)$ holds then $P(i-1)$ holds

Does this constitute a valid proof that $P(n)$ holds, for all integers n such that $n \geq 17$ or $n \leq -17$? Justify your answer.

7. Use induction to prove that, for any integers $m \geq 2$ and $n \geq 1$, $\sum_{t=0}^{n} m^t = \frac{m^{n+1}-1}{m-1}$.

8. Review the definitions of equality between sequences and the subsequence relationship between sequences. Then prove that any two *finite* sequences are equal if and only if each is a subsequence of the other. Does this result hold for *infinite* sequences? (Show that it does, or provide a counterexample proving that it does not.)

9. Prove that every nonempty finite set of natural numbers has a maximum element. Does the same hold for infinite sets of natural numbers? Compare this with the Well-Ordering principle.

10. Use Proposition 1.6 to prove Propositions 1.3 and 1.5 without using induction.

11. Each of the proofs of Proposition 1.9 (see Example 1.7, page 30, and Example 1.12, page 40) not only proves that we can make postage for any amount of $n \geq 18$ cents using only 4-cent and 7-cent stamps; it also (implicitly) provides a recursive algorithm for determining *how* to do so. Specifically, starting from any $n \geq 18$ and "unwinding the induction" backwards (either in steps of 1 — as in the first proof — or in steps of 4 — as in the second proof) we can see how to recursively compute the number of 4-cent and the number of 7-cent stamps for making exactly n cents of postage. Do the two algorithms produce the same answer for each amount of postage? That is, for any value of $n \geq 18$, do both algorithms yield the same number of 4-cent and 5-cent stamps with which to make n cents of postage? Justify your answer!

Note that, in principle, this need not be the case. For instance, postage of 56 cents can be produced in any of the following ways:

- seven 4-cent stamps and four 7-cent stamps
- fourteen 4-cent stamps
- eight 7-cent stamps.

12. Use complete induction to prove that for each nonempty full binary tree the number of leaves exceeds the number of internal nodes by exactly one.

13. Let $P(n)$ be the predicate:

$P(n)$: postage of exactly n cents can be made using only 4-cent and 6-cent stamps.

Consider the following complete induction "proof" of the statement "$P(n)$ holds for all $n \geq 4$".

Basis: $n = 4$. Postage of exactly 4 cents can be made using just a single 4-cent stamp. So $P(4)$ holds, as wanted.

Induction Step: Let $i \geq 4$ be an arbitrary integer, and suppose that $P(j)$ holds for all j such that $4 \leq j < i$. That is, for all j such that $4 \leq j < i$, postage of exactly j cents can be made using only 4-cent and 6-cent stamps. We must prove that $P(i)$ holds. That is, we must prove that postage of exactly i cents can be made using only 4-cent and 6-cent stamps.

Since $i - 4 < i$, by induction hypothesis we can make postage of exactly $i - 4$ cents using only 4-cent and 6-cent stamps. Suppose this requires k 4-cent stamps and ℓ 6-cent stamps; i.e., $i - 4 = 4 \cdot k + 6 \cdot \ell$. Let $k' = k + 1$ and $\ell' = \ell$. We have

$$\begin{aligned} 4 \cdot k' + 6 \cdot \ell' &= 4 \cdot (k+1) + 6 \cdot \ell && \text{[by definition of } k' \text{ and } \ell' \text{]} \\ &= 4 \cdot k + 6 \cdot \ell + 4 \\ &= (i-4) + 4 && \text{[by induction hypothesis]} \\ &= i \end{aligned}$$

Thus, $P(i)$ holds, as wanted.

Clearly, however, we can't make an odd amount of postage using only 4-cent and 6-cent stamps! Thus, the statement "$P(n)$ holds for all $n \geq 4$" is certainly false. Consequently, the above "proof" is incorrect. What is wrong with it?

14. Define the sequence of integers $a_0, a_1, a_2, \cdots$ as follows:

$$a_i = \begin{cases} 2, & \text{if } 0 \leq i \leq 2 \\ a_{i-1} + a_{i-2} + a_{i-3}, & \text{if } i > 2 \end{cases}$$

Use complete induction to prove that $a_n < 2^n$, for every integer $n \geq 2$.

15. An n-bit *Gray code* is a sequence of all 2^n n-bit strings with the property that any two successive strings in the sequence, as well as the first and last strings, differ in exactly one position. (You can think of the 2^n strings as arranged around a circle, in which case we can simply say that any two successive strings on the circle differ in exactly one bit position.) For example, the following is a 3-bit Gray code: $111, 110, 010, 011, 001, 000, 100, 101$. There are many other 3-bit Gray codes — for example, any cyclical shift of the above sequence, or reversal thereof, is also a 3-bit Gray code.

Prove that for every integer $n \geq 1$, there is an n-bit Gray code.

16. Let n be any positive integer. Prove that

(a) Every set S that contains binary strings of length n such that no two strings in S differ in *exactly* one position, contains no more than 2^{n-1} strings.

(b) There exists a set S that contains exactly 2^{n-1} binary strings of length n such that no two strings in S differ in exactly one position.

Chapter 2

CORRECTNESS OF ITERATIVE AND RECURSIVE PROGRAMS

2.1 Program correctness

When we say that a program is ***correct***, we mean that it produces a correct output on *every* acceptable input. Of course, what exactly is a "correct output" and what are deemed as "acceptable inputs" depends on the problem that the program is designed to solve — e.g., if it is a program to sort arrays, or a program to solve systems of linear equations. Thus, when we test that our program does the right thing by running it on some selected set of inputs we are not producing conclusive evidence that the program is correct. We are, at best, gaining confidence that the program is not incorrect. To *prove* that a program is correct we must give a sound mathematical argument which shows that the right output is produced on *every* acceptable input, not just those on which we chose to test the program.

Iterative programs (i.e., programs with loops) present a special challenge in this respect because, in general, the number of times that a loop is executed depends on the input. The same is true about recursive programs. Computer scientists have developed certain techniques for proving the correctness of such programs. These techniques rely heavily on induction. In this chapter we will examine these techniques.

2.2 Specification through preconditions and postconditions

First we need to discuss how to specify what are the acceptable inputs of a program and what are the correct outputs for each acceptable input. One convenient and popular way to do this is by using preconditions and postconditions. A ***precondition*** for a program is an assertion involving some of the variables of the program; this assertion states what must be true before the program starts execution — in particular, it can describe what are deemed as acceptable inputs for the program. A ***postcondition*** for a program is an assertion involving some of the variables of the program; this assertion states what must be true when the program ends — in particular, it can describe what is a correct output for the given input.

Given a precondition/postcondition specification and a program, we say that the program is

correct with respect to the specification (or the program ***meets*** the specification), if whenever the precondition holds before the program starts execution, then (a) the program terminates and (b) when it does, the postcondition holds. Following are some examples of precondition/postcondition pairs for some simple but useful specifications. In these notes we assume that the starting index of arrays is 1, and that $\mathit{length}(A)$ is the number of elements of array A.

Example 2.1 Suppose we are interested in a program which, given an array A and an element x, searches A for x and returns the index of a position of A that contains x; if no position of A contains x, then the program returns 0. Such a program can be specified by the following precondition/postcondition pair.

Precondition: A is an array.

Postcondition: Return an integer i such that $1 \leq i \leq \mathit{length}(A)$ and $A[i] = x$, if such an integer exists; otherwise return 0.

Note that this specification does not tell us *how* to search A for x. We can meet the specification with a program that searches A from the first position to the last, or from the last position to the first, or in any other order.

Strictly speaking, the postcondition in the above specification should also state that the values of A and x are not altered by the program. To see why, consider the program that sets $A[1] := x$ and returns 1. This program does, indeed, return the index of an element of A that contains x, but it does so by changing A, not by searching it! Because it becomes tedious to explicitly state that various variables are not changed by the program, we adopt the convention that, unless stated otherwise, all variables mentioned in the precondition and postcondition are not altered by the program. **End of Example 2.1**

Example 2.2 Some programs that search an array for an element work correctly *only if* the array is sorted.[1] To specify such a search program, the additional assumption that the array is sorted must be included in the precondition. Thus, for such a program, the precondition given above would be replaced by

Precondition: A is a *sorted* array.

The postcondition remains as in Example 2.1.

A program that satisfies this specification can exhibit arbitrary behaviour if the given array is not sorted: It may return an index that contains an element other than x, it may return a value that is not an index of the array, it may never return a value, or it may cause the computer to crash! **End of Example 2.2**

[1] An array is sorted if its elements appear in nondecreasing order. More precisely, A is ***sorted*** if, for each t such that $1 \leq t < \mathit{length}(A)$, $A[t] \preceq A[t+1]$, where $\preceq$ is a total order of the data type to which the elements of A belong. (When we speak of a sorted array we are, implicitly or explicitly, assuming a total order of the data types to which the array elements belong. For example, if the array elements contain integers, the total order could be the $\leq$ or $\geq$ relation on integers; if the array elements contain strings, the total order could be lexicographic ordering.) Note that, in general, a sorted array may contain duplicates.

Example 2.3 Suppose we are interested in a program which *sorts* a given array. Such a program can be specified by the following precondition/postcondition pair.

Precondition: A is an array.

Postcondition: A contains the same elements as before the invocation, but in sorted (nondecreasing) order.

Notice that this specification allows for A to be altered by the program — albeit in restricted ways, namely by rearranging its elements. **End of Example 2.3**

Specifications of the kind illustrated by these examples are very important in practice. The single most important element of good documentation for a program is to convey just what the program does (postconditions) and under what conditions it is expected to do it (preconditions). Thus, a well-documented program must include, at a minimum, a precondition/postconditions pair that constitutes its statement of correctness. Providing such a specification is crucial regardless of whether one intends to carry out a detailed proof of correctness for the program. This is because a programmer who wishes to use an existing program should be able to do so by reading just this specification, and without having to look at the code.

There are many formal notations that have been developed for expressing preconditions and postconditions, each with its advantages and disadvantages. Some of these notations rely heavily on predicate logic, a subject that we will encounter in Chapter 6. In these notes we will not be concerned with formal notations used to express preconditions and postconditions; we will express them informally — though hopefully clearly and precisely — in English.

2.3 Correctness of binary search

Binary search is an algorithm which, given a *sorted* array A and an element x determines whether x appears in A and if so it also determines an index of A that contains x. Thus, the specification of this algorithm is that given in Example 2.2. Informally, binary search works by comparing the target element x to the "middle" element of A. If the middle element is greater than or equal to x, then the search is confined to the first half of A; otherwise, the search is confined to the second half. This process is repeated until the search is confined to a subarray of size 1, in which case the information sought by the algorithm is trivial to determine. This algorithm can be expressed either iteratively, in which case each halving of the array corresponds to an iteration of a loop, or recursively, in which case each halving corresponds to a recursive call.

The iterative version of binary search is shown in Figure 2.1, which also contains the preconditions and postconditions relative to which we want to prove the correctness of the program. (The function **div** was defined on page 28.) The program uses two variables, f and ℓ that indicate, respectively, the "first" and "last" index of the range to which the search has been confined so far. If the range is nontrivial (i.e., has more than one element), the "middle" index of the range is computed as variable m, and the range is confined to the appropriate half.

```
BinSearch($A, x$)
    ▶ Precondition: $A$ is a sorted array of length at least 1.
    ▶ Postcondition: Return an integer $t$ such that $1 \le t \le \mathit{length}(A)$ and $A[t] = x$,
    ▶                if such a $t$ exists; otherwise return 0.
    $f := 1;\ \ \ell := \mathit{length}(A)$
    while $f \neq \ell$ do
        $m := (f + \ell)$ div 2
        if $A[m] \ge x$ then
            $\ell := m$
        else
            $f := m + 1$
        end if
    end while
    if $A[f] = x$ then
        return $f$
    else
        return 0
    end if
```

Figure 2.1: Iterative binary search

We will prove the correctness of this program with respect to the precondition/postcondition pair given in Figure 2.1. This can be stated as follows:

Theorem 2.1 *Suppose A is a sorted array of length at least 1.* BinSearch(A, x) *terminates and returns t such that $1 \le t \le \mathit{length}(A)$ and $A[t] = x$, if such a t exists; otherwise* BinSearch(A, x) *terminates and returns 0.*

The proof of this theorem will be given in two parts:

(a) *Partial Correctness:* Suppose A is a sorted array of length at least 1. If BinSearch(A, x) terminates then, when it does, it returns t such that $1 \le t \le \mathit{length}(A)$ and $A[t] = x$, if such a t exists; otherwise it returns 0.

(b) *Termination:* Suppose A is a sorted array of length at least 1. BinSearch(A, x) terminates.

You should make sure you understand why (a) and (b) together imply Theorem 2.1 — and why (a) alone does not. In the next two subsections we present the proofs of these two parts.

2.3.1 Partial correctness of BinSearch

To prove partial correctness of BinSearch, we will prove that if the preconditions hold before the program starts then the following is true *at the end of each iteration of the loop*:

$$1 \le f \le \ell \le \mathit{length}(A), \text{ and if } x \text{ is in } A \text{ then } x \text{ is in } A[f..\ell]. \tag{2.1}$$

(For any integers i and j such that $1 \leq i \leq j \leq \mathit{length}(A)$, the notation $A[i..j]$ denotes the subarray of A between indices i and j, inclusive.) This assertion captures the essence of the loop of BinSearch: The loop ensures that the element x being sought, if it is anywhere at all in the array, then it is in the part of the array that lies between indices f (as a lower bound) and ℓ (as an upper bound). This is just what (2.1) states, albeit in more mathematical terms.

A statement that is true at the end of each iteration of a loop, such as (2.1), is called a ***loop invariant***. More precisely, consider a program that contains a loop. Let P and Q be predicates of (some of) the variables of the program. We say that P is an ***invariant for the loop with respect to precondition*** Q if, assuming the program's variables satisfy Q before the loop starts, the program's variables also satisfy P before the loop starts as well as *at the end of each iteration* of the loop. From now on, we adopt the convention that the "end of the 0-th iteration of the loop" is the point in the program just before entering the loop, and that "each iteration of the loop" includes this 0-th iteration. Thus, we simply say that an invariant is true at the end of each iteration of the loop, without explicitly saying that it is true before the loop.

We will now prove that (2.1) is, in fact, an invariant for the loop in BinSearch with respect to that program's precondition.[2] Later, in the proof of Corollary 2.4, we will see the relevance of this fact for the partial correctness of BinSearch.

First we need to develop some notation, which we will also use in other proofs of partial correctness. Consider a variable v, whose value may change during an iteration of the loop. (BinSearch contains three such variables: f, ℓ and m.) We use the notation v_i to denote the value of variable v at the end of the i-th iteration of the loop — provided that there is such an iteration.

We will need the following lemma:

Lemma 2.2 *For any integers f, ℓ such that $f < \ell$, $f \leq (f+\ell)$ **div** $2 < \ell$.*

Proof. We have:

$$\begin{aligned} (f+\ell)\ \mathbf{div}\ 2 &\geq (f+f)\ \mathbf{div}\ 2 && [\text{since } \ell > f] \\ &= (2f)\ \mathbf{div}\ 2 \\ &= f && (2.2) \end{aligned}$$

and

$$\begin{aligned} (f+\ell)\ \mathbf{div}\ 2 &\leq (f+\ell)/2 && [\text{by definition of } \mathbf{div}] \\ &\leq \big((\ell-1)+\ell\big)/2 && [\text{since } f < \ell \text{ and so } f \leq \ell - 1] \\ &= \ell - \tfrac{1}{2} \\ &< \ell && (2.3) \end{aligned}$$

[2] Note that, in this case, the precondition of BinSearch involves variables that are not modified before executing the loop. Thus, if the precondition holds before the program starts, the precondition still holds before the loop starts.

The lemma follows from (2.2) and (2.3). □

We are now ready to prove that (2.1) is a loop invariant. More precisely,

Lemma 2.3 *Suppose the precondition of* BinSearch *holds before the program starts. For each $i \in \mathbb{N}$, if the loop of* BinSearch(A, x) *is executed at least i times, then (i) $1 \leq f_i \leq \ell_i \leq \mathit{length}(A)$, and (ii) if x is in A, then x is in $A[f_i..\ell_i]$.*[3]

Proof. Let $P(i)$ be the following predicate:

proving loop invariant

$P(i)$: if the loop is executed at least i times, then (i) $1 \leq f_i \leq \ell_i \leq \mathit{length}(A)$, and (ii) if x is in A, then x is in $A[f_i..\ell_i]$.

We will use induction to prove that $P(i)$ holds for all $i \in \mathbb{N}$.

Basis: $i = 0$. We have $f_0 = 1$ and $\ell_0 = \mathit{length}(A)$. Part (i) of $P(0)$ follows because, by the precondition, $\mathit{length}(A) \geq 1$. Part (ii) of $P(0)$ is trivially true.

by simple induction

Induction Step: Let j be an arbitrary natural number, and assume that $P(j)$ holds. We must prove that $P(j+1)$ holds as well. If there is no $(j+1)$-st iteration then $P(j+1)$ holds trivially. So, assume that there is such an iteration. By the loop exit condition, $f_j \neq \ell_j$. By induction hypothesis, $f_j \leq \ell_j$, and thus $f_j < \ell_j$. By Lemma 2.2,

$$f_j \leq m_{j+1} < \ell_j. \tag{2.4}$$

By the program, either $f_{j+1} = f_j$ and $\ell_{j+1} = m_{j+1}$, or $f_{j+1} = m_{j+1} + 1$ and $\ell_{j+1} = \ell_j$. In conjunction with the fact that $1 \leq f_j \leq \ell_j \leq \mathit{length}(A)$ (part (i) of the induction hypothesis), and (2.4), we have that $1 \leq f_{j+1} \leq \ell_{j+1} \leq \mathit{length}(A)$. This is part (i) of $P(j+1)$. It remains to prove part (ii) of $P(j+1)$. Suppose that x is in A. By part (ii) of the induction hypothesis, x is in $A[f_j..\ell_j]$. We must prove that x is in $A[f_{j+1}..\ell_{j+1}]$. There are three cases:

Case 1. $A[m_{j+1}] = x$. In this case, by the program, $f_{j+1} = f_j$ and $\ell_{j+1} = m_{j+1}$ and so it is obvious that x is in $A[f_{j+1}..\ell_{j+1}]$, as wanted.

same case

Case 2. $A[m_{j+1}] > x$. Since A is sorted, $A[t] > x$ for all t such that $m_{j+1} \leq t \leq \ell_j$. Since x is in $A[f_j..\ell_j]$ but it is not in $A[m_{j+1}..\ell_j]$, it follows that x is in $A[f_j..m_{j+1}]$. By the program, in this case $f_{j+1} = f_j$ and $\ell_{j+1} = m_{j+1}$. Thus, x is in $A[f_{j+1}..\ell_{j+1}]$, as wanted.

Case 3. $A[m_{j+1}] < x$. Since A is sorted, $A[t] < x$ for all t such that $f_j \leq t \leq m_{j+1}$. Since x is in $A[f_j..\ell_j]$ but it is not in $A[f_j..m_{j+1}]$, it follows that x is in $A[m_{j+1}+1..\ell_j]$. By the program, in this case $f_{j+1} = m_{j+1}+1$ and $\ell_{j+1} = \ell_j$. Thus, x is in $A[f_{j+1}..\ell_{j+1}]$, as wanted. □

We are now in a position to prove the partial correctness of BinSearch.

[3] We do not subscript A and x because these variables' values do not change in the loop.

Corollary 2.4 *Suppose the precondition of* BinSearch *holds before the program starts. If the program terminates then, when it does, the postcondition holds.*

Proof. Suppose the precondition holds and the program terminates. Since the program terminates, the loop is executed a finite number of times, say k. By the exit condition of the loop, $f_k = \ell_k$. By part (i) of Lemma 2.3, $1 \leq f_k \leq \mathit{length}(A)$. There are two cases:

Case 1. There is some t such that $1 \leq t \leq \mathit{length}(A)$ and $A[t] = x$. This means that x is in A and by part (ii) of Lemma 2.3, x is in $A[f_k..\ell_k]$. But since $f_k = \ell_k$, we have that $x = A[f_k]$ and the program returns f_k, as required by the postcondition in this case.

equivalent to meaning x not in A

Case 2. For each t such that $1 \leq t \leq \mathit{length}(A)$, $A[t] \neq x$. Since, by part (i) of Lemma 2.3, $1 \leq f_k \leq \mathit{length}(A)$, in particular, $A[f_k] \neq x$. So, in this case, the program returns 0, as required by the postcondition in this case. □

It is instructive to step back for a moment to consider the basic structure of the preceding corollary's proof: The loop invariant (x is in $A[f..\ell]$, if it is anywhere in A) is used in conjunction with the loop exit condition ($f = \ell$) to obtain the postcondition. This sheds some interesting light to the question of how we came up with (2.1) as the particular statement for our loop invariant.

This question is very relevant in view of the fact that there are many statements that are invariants for a loop. For example, the statement "A is a sorted array" is an invariant of the loop of BinSearch with respect to that program's precondition. This is simply because the precondition assures us that A is sorted and the loop does not modify A. So, the question arises: Why did we choose (2.1), rather than "A is sorted", as our loop invariant? The answer is that the former helps prove partial correctness while the latter does not. This is really the ultimate test for the claim we made earlier that (2.1) "captures the essence" of the loop. In contrast, the statement "A is sorted", although true at the end of each iteration of the loop, does not correspond to our intuition about the purpose of the loop in BinSearch. This is reflected by the fact that the statement "A is sorted", combined with the loop exit condition, is not enough to yield the postcondition of BinSearch.

2.3.2 Termination of BinSearch

To prove that BinSearch terminates it is sufficient to show that the loop in lines 2–9 terminates, since all other statements of the program obviously terminate. Proving that a loop terminates usually involves induction in the guise of the well-ordering principle (see Section 1.1.1). A sequence σ of numbers is ***decreasing*** if, for each i such that $0 \leq i < |\sigma|$, $\sigma(i+1) < \sigma(i)$. It is easy to see that the principle of well-ordering immediately implies:

Theorem 2.5 *Every decreasing sequence of natural numbers is finite.*

To prove the termination of a loop we typically proceed as follows: We associate with each iteration i of the loop a number k_i, defined in terms of the values of the variables in the i-th iteration, with the properties that (i) each k_i is a natural number and (ii) the sequence $\langle k_0, k_1, k_2, \ldots \rangle$ is decreasing. Then the loop must terminate, for otherwise we would have an infinite decreasing sequence of natural numbers, which Theorem 2.5 precludes.

In BinSearch we can associate with each iteration of the loop the value of the quantity $\ell_i - f_i$. This choice reflects the intuition that the reason the loop of BinSearch terminates is that the range of the array into which the search for x has been confined gets smaller and smaller with each iteration. The fact that $\ell_i - f_i$ is a natural number follows immediately from the fact that ℓ_i and f_i are natural numbers and, by part (i) of Lemma 2.3, $f_i \leq \ell_i$. It remains to show that the sequence $\langle \ell_0 - f_0, \ell_1 - f_1, \ell_2 - f_2, \ldots \rangle$ is decreasing. More precisely,

Lemma 2.6 *Consider the loop in lines 2–9 of Figure 2.1. For each $i \in \mathbb{N}$, if the loop is executed at least $i+1$ times then $\ell_{i+1} - f_{i+1} < \ell_i - f_i$.*

Proof. Suppose that the loop is executed at least $i+1$ times. By the loop termination condition, this means that $\ell_i \neq f_i$. By part (i) of Lemma 2.3, $f_i \leq \ell_i$, and so $f_i < \ell_i$. Therefore, by Lemma 2.2,

$$f_i \leq m_{i+1} < \ell_i. \tag{2.5}$$

There are two cases:

Case 1. $A[m_{i+1}] \geq x$. In this case, $f_{i+1} = f_i$ and $\ell_{i+1} = m_{i+1}$. By (2.5), $m_{i+1} < \ell_i$. Therefore, $\ell_{i+1} - f_{i+1} = m_{i+1} - f_i < \ell_i - f_i$.

Case 2. $A[m_{i+1}] < x$. In this case, $f_{i+1} = m_{i+1} + 1$ and $\ell_{i+1} = \ell_i$. By (2.5), $f_i \leq m_{i+1}$ and so $f_i < m_{i+1} + 1$. Therefore, $\ell_{i+1} - f_{i+1} = \ell_i - (m_{i+1} + 1) < \ell_i - f_i$.

In either case, $\ell_{i+1} - f_{i+1} < \ell_i - f_i$, as wanted. □

Since we can associate with each iteration of the loop a natural number, so that successive iterations correspond to a decreasing sequence of natural numbers, by Theorem 2.5, the loop terminates.

2.4 The correctness of a multiplication program

Consider the program in Figure 2.2 which, as it turns out, returns the product of its inputs m and n, whenever $m \in \mathbb{N}$ and $n \in \mathbb{Z}$. (The functions **div** and **mod** were defined on page 28.) Some observations about this program are in order. This is supposed to be a program for multiplying numbers, yet in line 7 it uses multiplication (and in line 6 it uses integer division)! This seems rather pointless; if multiplication is available then why bother with Mult in the first place? The integer division and multiplication of lines 6 and 7, however, are of a very special kind: they divide and multiply by 2. These operations can be accomplished in the

```
MULT$(m, n)$
    ▶ Precondition: $m \in \mathbb{N}$ and $n \in \mathbb{Z}$.
    ▶ Postcondition: Return $m \cdot n$.
        $x := m$
        $y := n$
        $z := 0$
        while $x \neq 0$ do
            if $x$ mod $2 = 1$ then $z := z + y$ end if
            $x := x$ div $2$
            $y := y * 2$
        end while
        return $z$
```

Figure 2.2: A multiplication program

following way: to multiply an integer by 2, append a 0 to its binary representation; to integer-divide a number by 2, drop the rightmost bit from its binary representation.[4] Thus, MULT could have easily been written without multiplication or division operations, using shift-left and shift-right operations instead. We chose to use multiplication and division to make things clearer.

Let us now return to the issue of correctness of MULT, with respect to the precondition/postcondition pair given in Figure 2.2. This can be stated as follows:

Theorem 2.7 *If $m \in \mathbb{N}$ and $n \in \mathbb{Z}$, then* MULT(m, n) *terminates and returns $m \cdot n$.*

As with the proof of correctness of BINSEARCH we will prove this theorem in two parts:

(a) *Partial Correctness:* If $m, n \in \mathbb{Z}$ and MULT(m, n) terminates, then it returns $m \cdot n$.

(b) *Termination:* If $m \in \mathbb{N}$ and $n \in \mathbb{Z}$, then MULT(m, n) terminates.

One interesting detail is that in (a) we only require that m and n be integers, while in (b) and the Theorem, m must be a *nonnegative* integer. Of course, (a) would still be true (*a fortiori*) if we required m to be nonnegative but, as it turns out, it is not necessary to do so. On the other hand, it is necessary to require m to be nonnegative for (b); as we will see, Termination would not hold otherwise!

2.4.1 Partial correctness of Mult

To prove partial correctness, we will prove that at the end of each iteration of the loop, $z = mn - xy$. In Section 2.4.3 we will explain *how* it ever occurred to us to prove that

[4] To help you see why, think of the corresponding operations in decimal notation: To multiply a number by 10, we append a 0 to its decimal representation; to integer-divide a number by 10, we drop the rightmost digit.

$z = mn - xy$ is a loop invariant. For now, we just concentrate on proving this fact. We will use the notation we developed in the proof of correctness of BinSearch: A variable v subscripted with a natural number i denotes the value of the variable v at the end of the i-th iteration of the loop (if such an iteration exists).

Lemma 2.8 *Suppose $m, n \in \mathbb{Z}$. For each $i \in \mathbb{N}$, if the loop of* Mult(m, n) *is executed at least i times, then $z_i = mn - x_i y_i$.*[5]

Proof. Let $P(i)$ be the predicate defined as follows:

$$P(i): \qquad \text{if the loop is executed at least } i \text{ times, then } z_i = mn - x_i y_i.$$

We will use induction to prove that $P(i)$ holds for all $i \in \mathbb{N}$.

Basis: $i = 0$. We have $z_0 = 0$, $x_0 = m$ and $y_0 = n$. Hence, $z_0 = mn - x_0 y_0$, and $P(0)$ holds.

Induction Step: Let j be an arbitrary natural number, and assume that $P(j)$ holds; i.e., if the loop is executed at least j times, then $z_j = mn - x_j y_j$. We must prove that $P(j + 1)$ holds as well; i.e., if the loop is executed at least $j + 1$ times, then $z_{j+1} = mn - x_{j+1} y_{j+1}$.

First note that if the loop is executed at least $j + 1$ times then

- $x_{j+1} = x_j \textbf{ div } 2$. Thus,
$$x_j = \begin{cases} 2x_{j+1}, & \text{if } x_j \textbf{ mod } 2 = 0 \\ 2x_{j+1} + 1, & \text{if } x_j \textbf{ mod } 2 = 1 \end{cases} \tag{2.6}$$
- $y_{j+1} = 2y_j$. Thus,
$$y_j = y_{j+1}/2. \tag{2.7}$$

Consider the effect of the $(j + 1)$-st iteration on z. This depends on whether $x_j \textbf{ mod } 2$ is 0 or 1. Thus, there are two cases:

Case 1. $x_j \textbf{ mod } 2 = 0$. Then

$$\begin{aligned} z_{j+1} &= z_j && \text{[because line 5 is not executed, in this case]} \\ &= mn - x_j y_j && \text{[by induction hypothesis]} \\ &= mn - (2x_{j+1})(y_{j+1}/2) && \text{[by (2.6) and (2.7) above]} \\ &= mn - x_{j+1} y_{j+1} \end{aligned}$$

as wanted.

[5] We do not subscript m and n because these variables' values do not change in the loop.

CASE 2. $x_j \textbf{ mod } 2 = 1$. Then

$$\begin{aligned} z_{j+1} &= z_j + y_j && \text{[because line 5 is executed, in this case]} \\ &= mn - x_j y_j + y_j && \text{[by induction hypothesis]} \\ &= mn - (2x_{j+1} + 1)(y_{j+1}/2) + y_{j+1}/2 && \text{[by (2.6) and (2.7) above]} \\ &= mn - x_{j+1} y_{j+1} \end{aligned}$$

as wanted. □

Corollary 2.9 *Suppose the precondition of* MULT *holds before the program starts. If the program terminates then, when it does, the postcondition holds.*

PROOF. Suppose the precondition holds and the program terminates. Since the program terminates, the loop is executed a finite number of times, say k. By the exit condition of the loop, $x_k = 0$. By Lemma 2.8, $z_k = mn - x_k y_k$. But since $x_k = 0$, we have that $z_k = mn$. Since the program returns the value of z when the loop terminates, i.e., z_k, the postcondition holds. □

There are many assertions that qualify as invariants for the loop of MULT, in the sense that they are true at the end of each iteration. For example, a straightforward induction shows that $y_i = 2^i \cdot n$ is true at the end of each iteration. What distinguishes $z_i = mn - x_i y_i$ from these, and the reason we chose *it* as our loop invariant, is that this assertion, in conjunction with the loop exit condition, immediately implies the postcondition. In contrast, for example, the (true) assertion that $y_i = 2^i \cdot n$, together with the loop exit condition, does not imply the postcondition.

important! we can infer the desired loop invariant by thinking about what, together with loop exit condition, yields postcondition

2.4.2 *Termination of* Mult

To prove that MULT terminates it is sufficient to show that the loop in lines 4–8 terminates, since all other statements of the program obviously do. We will prove that loop of MULT terminates by following the approach we used in Section 2.3.2 to prove that the loop of BINSEARCH terminates. Namely, we will associate with each iteration i of the loop a number k_i with the properties that (i) each k_i is a natural number, and (ii) the sequence $\langle k_0, k_1, k_2, \ldots \rangle$ is decreasing. Then the loop must terminate, by Theorem 2.5.

In MULT we can simply associate with iteration i the value x_i. This choice reflects the intuition that the loop of MULT terminates because the value of x gets smaller and smaller. The fact that x_i is a natural number can be shown by a straightforward induction.[6] It remains

[6] It is precisely in the base case of this induction where we need the precondition that $m \in \mathbb{N}$ (rather than the weaker assumption that $m \in \mathbb{Z}$). This fact, which is critical for the termination — and thus the correctness — of the program, is only needed in this seemingly humble step. This is another instance of the phenomenon illustrated in Example 1.8. By leaving out the basis case of this induction we could claim to have "proved" that MULT(m, n) is a correct multiplication program for all $m, n \in \mathbb{Z}$; in actuality it is correct only for all $m \in \mathbb{N}$ and $n \in \mathbb{Z}$.

to show that the sequence $\langle x_0, x_1, x_2, \ldots \rangle$ is decreasing. More precisely, we must show that if the loop is executed at least $i+1$ times, then $x_{i+1} < x_i$. This is true because $x_{i+1} = x_i$ **div** 2. We have already argued that x_i is a natural number; since $x_i \neq 0$ (otherwise, there would be no $(i+1)$-st iteration), it follows that x_i **div** $2 < x_i$, so $x_{i+1} < x_i$.

2.4.3 Deriving the invariant in Mult

We now return to the question: How on earth did we ever think of the invariant $z_i = mn - x_i \cdot y_i$ in our proof of partial correctness of Mult? The mystery vanishes as soon as we understand how this program works. Mult multiplies numbers by the standard elementary school algorithm, except that it views numbers in binary, rather than in decimal. How do we multiply numbers in binary? We consider the multiplier m bit-by-bit from right to left, and maintain a running total. If the i-th bit of the multiplier is 1, we add to this running total the multiplicand n padded with i 0s to the right; if the i-th bit is 0, we add nothing to the running total. In terms of the program in Figure 2.2, the padding of the multiplicand with 0s is accomplished by the multiplication in line 7, and the dropping of the multiplier's bits that have already been considered is accomplished by the integer division in line 6. The running total is accumulated in variable z.

With this in mind, it is not hard to see what is going on in Mult: At the end of the i-th iteration of the loop, the value of z is equal to n multiplied by the number formed by the i rightmost bits of m. That is, $z_i = n \cdot (\text{rightmost } i \text{ bits of } m)$.

Now, the rightmost i bits of m can be expressed as $m - (m \textbf{ div } 2^i) \cdot 2^i$. To see this, observe that we can obtain the rightmost i bits by shifting m i bits to the right (thus making the i leftmost bits 0s), then shifting the resulting number i bits to the left (thus restoring the bits from positions $i+1$ on to their proper places), and finally subtracting the resulting number from m (thus turning into 0s the bits to the left of the i rightmost ones). In other words, we have

$$\begin{aligned} z_i &= n \cdot (\text{rightmost } i \text{ bits of } m) \\ &= n \cdot \big(m - (m \textbf{ div } 2^i) \cdot 2^i\big) \\ &= nm - (m \textbf{ div } 2^i)(n \cdot 2^i) \end{aligned}$$

It is easy to see (and to prove by induction!) that $y_i = n \cdot 2^i$ and $x_i = m \textbf{ div } 2^i$. Putting everything together we have $z_i = mn - x_i y_i$, which was our invariant.

2.5 A more interesting proof of termination

Consider the loop shown in Figure 2.3. Does this loop terminate with respect to its precondition? (We are not really interested in what this loop does other than whether it terminates. For this reason the postcondition is the assertion "true" — or, if you prefer, something like $x = x$ — which is satisfied no matter what the loop does!)

We can informally describe what the program does as follows: It decrements the value of x until it becomes 0; it then sets x to 16 and decrements y once. This is repeated until both x

```
▶ Precondition: $x, y \in \mathbb{N}$.
▶ Postcondition: True.
while $x \neq 0$ or $y \neq 0$ do
    if $x \neq 0$ then
        $x := x - 1$
    else
        $x := 16$
        $y := y - 1$
    end if
end while
```

Figure 2.3: A loop with an interesting proof of termination

and y become 0. In view of this description, it appears plausible to conjecture that, if x and y are both nonnegative integers before the loop starts, then the loop does, in fact, terminate. Let's now prove this by following the methodology we used in Sections 2.3.2 and 2.4.2, to prove that the loops of BinSearch and Mult terminate. We want to associate with each iteration of the loop some *natural number* so that the sequence of numbers that correspond to successive iterations is *decreasing*. A moment's thought shows that the number we associate with an iteration cannot be the value of x: This is because iterations in which $x \neq 0$ cause x to decrease, but iterations in which $x = 0$ actually cause x to increase (from 0 to 16) — thus the value of x does not always decrease. Also, the number we associate with an iteration cannot be the value of y: This is because in iterations where $x \neq 0$, the value of y does not decrease.

However, an expression that involves both x and y will do the trick. Specifically, assuming that x, y are natural numbers before the loop starts, we can prove that the value of $17y + x$ is always a natural number that decreases in each iteration of the loop. The fact that $17y + x$ is always a natural number can be shown by a straightforward induction, which we omit. (The basis of this induction uses the precondition that $x, y \in \mathbb{N}$.) We now prove that the value of $17y + x$ decreases in each iteration of the loop. As usual, v_i denotes the value of variable v at the end of iteration i, if such an iteration exists.

Lemma 2.10 *For each $i \in \mathbb{N}$, if the loop in Figure 2.3 is executed at least $i + 1$ times, then $17y_{i+1} + x_{i+1} < 17y_i + x_i$.*

Proof. There are two cases:

Case 1. $x_i \neq 0$. By the algorithm, $x_{i+1} = x_i - 1$ and $y_{i+1} = y_i$. Thus, $17y_{i+1} + x_{i+1} = 17y_i + x_i - 1 < 17y_i + x_i$.

Case 2. $x_i = 0$. By the algorithm, $x_{i+1} = 16$ and $y_{i+1} = y_i - 1$. Thus, $17y_{i+1} + x_{i+1} = 17(y_i - 1) + 16 = 17y_i - 1 < 17y_i = 17y_i + x_i$.

In either case then, $17y_{i+1} + x_{i+1} < 17y_i + x_i$, as wanted. □

Where did the expression $17y + x$ come from? The informal description of the loop given earlier (decrement x until it becomes 0, then set it to 16, decrement y once and repeat until both x and y are 0) suggests the following interpretation for this loop. Think of x and y as the digits of a two-digit number in base 17, where x is the low-order digit and y is the high-order digit. (Thus, x and y are integers in the range 0 to 16.) The action of the loop corresponds to decrementing by 1 the base-17 number represented by the two digits. This number is simply $17y + x$. (To see why, think of the analogous situation in base 10: If x and y represent, respectively, the low-order and high-order digits of a number in base 10, the number represented by these two digits is $10y + x$.)

2.6 Comments on proving the correctness of iterative programs

The correctness proofs of BinSearch and Mult, detailed in Sections 2.3 and 2.4, exemplify some general points regarding correctness proofs of iterative programs. We first summarise these points, and then elaborate on them.

- Correctness proofs of iterative programs are typically divided into two parts: one proving partial correctness (i.e., that the program is correct *assuming it terminates*); and another proving termination (i.e., that the program does indeed terminate).
- The proof of termination typically involves associating a decreasing sequence of natural numbers with the iterations of the loop, and appealing to Theorem 2.5 (i.e., to the well-ordering principle).
- The proof of partial correctness of an iterative program is typically based on a ***loop invariant***. Proving that a statement is a loop invariant involves induction.

When proving termination it is generally a bad idea to try proving directly that the loop exit condition will eventually become true. It is best to do so indirectly, by exhibiting a quantity (expressed in terms of the program's variables) that (i) always takes on nonnegative integer values, and (ii) always decreases in each iteration of the loop. Informal arguments of the type "quantity such-and-such gets smaller and smaller in each iteration" may be informative but are not entirely convincing: A quantity can get smaller and smaller in each iteration but the loop may not terminate, if the quantity can take on negative or noninteger values. So, it is important to make sure that the quantity in question has property (i). Furthermore, it is important to make sure that the quantity *strictly* decreases in each iteration. (See Exercise 2, for example.)

When proving partial correctness, the most vexing question confronting people who are new to program correctness is: "How do I know what the loop invariant should be?" Unfortunately, there is no mechanical way to determine an appropriate loop invariant — that is, one that helps us establish partial correctness. Finding such an invariant is a creative process, and is usually the most difficult part of the proof. It is fair to say that to determine an appropriate loop invariant one must achieve a very clear understanding of what the loop does and be able to verbalise that understanding.

From this point of view, determining loop invariants is important not only for proving program correctness (an activity which, for better or worse, is not very common), but also for *understanding* programs and *documenting* them. Of course this implies that loop invariants are also important for *developing* programs, since one can't write a good program without understanding or documenting it! Indeed, some software development organisations require, as part of their programming style guidelines, that their programming staff annotate the loops of their programs with appropriate invariants. These invariants (as well as other things) are scrutinised by the team of quality control inspectors whose job is to ensure that the software produced by the programmers meets the organisation's standards of quality.

In some cases, programming style guidelines require loop invariants that are not merely manually inspected, but are actually executed. Here is how this works. Some program development environments incorporate something called an ***assert statement***. This is a statement of the form "**assert** B", where B is a Boolean expression involving (some of) the program's variables. Putting such a statement in a particular point of a program asserts that, at that point of the program, B is (supposed to be) true. (Thus, an assert statement at the beginning of a loop is an invariant for that loop.) If, at run time, the Boolean expression of an assert statement evaluates to false, an error message is generated to indicate that something went wrong: A fact that was supposed to be always true, was actually found to be false in some execution of the program. Assert statements are included in the code during the debugging and testing phase of software development, but are removed when the program is used in production to avoid the overhead of executing them. Some software development organisations have programming style guidelines which require the inclusion of appropriate assert statement in loops — i.e., loop invariants. The points raised in the last two paragraphs should convince you that loop invariants and program correctness are not just pie-in-the-sky theoretical flights of fancy. The associated techniques and concepts are practical and useful, and it is important that you should understand and be capable of using them.

Although it is not possible to *mechanically* derive a loop invariant that is sufficient to prove partial correctness, it is possible to do so through an iterative process of trial-and-error that involves the following steps:

(i) First we must try to understand what the loop does and how its function is related to the correctness of the program.

(ii) On the basis of our (perhaps incomplete) understanding in step (i), we formulate a candidate statement as our proposed loop invariant.

(iii) We check to see if the proposed loop invariant is sufficient to prove partial correctness. Typically, this involves combining the loop invariant with the exit condition of the loop and the preconditions of the program and seeing if these, taken together, imply the postcondition.

(iv) If the answer in step (iii) is negative, then we repeat the above three steps, trying to deepen our understanding of the loop. This should result in a more refined candidate for loop invariant. This process is repeated until we produce a proposed loop invariant that is sufficient to prove partial correctness.

(v) Finally, we prove that the proposed statement that meets the requirement of step (iii) really *is* a loop invariant. This proof involves induction. If we cannot carry out this induction, we must again revise the proposed loop invariant, repeating the above steps.

An important remark regarding step (v) is now in order. Suppose we have a candidate loop invariant $P(i)$, which we have determined is sufficient for partial correctness. We must now prove, using induction, that $P(i)$ really *is* an invariant. Sometimes, in the induction step of this proof we discover that $P(j)$ is not enough to let us prove $P(j+1)$, and a stronger induction hypothesis $P'(j)$ is needed. This is a classic instance of the phenomenon discussed at some length in Example 1.10 (see page 35).

We cannot, of course, simply use the stronger induction hypothesis $P'(j)$, to prove $P(j+1)$, as this would be an invalid proof. We can, however, revise our loop invariant from $P(i)$ to $P'(i)$, and attempt to prove that $P'(i)$ is a loop invariant. Note that strengthening the invariant is certainly not going to hurt its utility in proving the correctness of the program: Since $P'(i)$ holds at the end of each iteration, certainly $P(i)$ holds as well (that's what it means to say that $P'(i)$ is stronger than $P(i)$); and since we have already established that $P(i)$ is good enough to prove the correctness of the program, so is $P'(i)$. If we succeed in proving that $P'(i)$ is a loop invariant, then all is well and good. But we might not: it is possible that we strengthened the predicate too much and, in fact, $P'(i)$ is not an invariant for our loop. If that's the case, we can now try changing our candidate for a loop invariant to another predicate P'' which is stronger than P (and thus still useful for proving the correctness of the program) but weaker than P', and repeat this process.

The process of formulating candidate loop invariants and revising them (strengthening or weakening them) until we succeed in proving that our candidate is what we want — is quite common. For a reasonably complicated iterative program, even after achieving a fairly good grasp of what the loop does, it is rare that our first attempt at formulating a loop invariant will be just right. The iterative process outlined above, if carried out intelligently, will allow us to zero into the right invariant after a few trials.

2.7 Proof of correctness of recursive programs

In this section we illustrate, by means of an example, the process of proving the correctness of recursive programs. Consider the program shown in Figure 2.4, which is a recursive version of binary search. The program consists of two parts, a recursive program RecBinSearch which does all the work, and the main program MainBinSearch which merely calls RecBinSearch with the right parameters.

Informally, the recursive program works as follows. It takes as parameters: the array A to be searched, two indices f and ℓ which delimit the portion of A to be searched, and the element x for which $A[f..\ell]$ is to be searched. If x is in the subarray $A[f..\ell]$, then the program returns the index of an element in the subarray that contains x; otherwise it returns 0. Thus,

```
MainBinSearch($A, x$)
    ▶ Precondition: $A$ is a sorted array of length at least 1.
    ▶ Postcondition: Return $t$ such that $1 \leq t \leq length(A)$ and $A[t] = x$, if such a $t$ exists;
    ▶                otherwise return 0.
        return RecBinSearch($A, 1, length(A), x$)
```

```
RecBinSearch($A, f, \ell, x$)
if $f = \ell$ then
    if $A[f] = x$ then
        return $f$
    else
        return 0
    end if
else
    $m := (f + \ell)$ div 2
    if $A[m] \geq x$ then
        return RecBinSearch($A, f, m, x$)
    else
        return RecBinSearch($A, m + 1, \ell, x$)
    end if
end if
```

Figure 2.4: Recursive binary search

to find an index of A that contains x (if x appears in A), all the main program has to do is call RecBinSearch($A, 1, length(A), x$).

The correctness proof of MainBinSearch(A, x) involves the following three steps:

- First, we give a careful specification for the correctness of the recursive program RecBinSearch. This specification formalises the informal description given above.
- We then prove that RecBinSearch is correct with respect to this specification.
- Finally, we prove that MainBinSearch meets its specification using the correctness of RecBinSearch established in the previous step.

We now carry out these three steps.

The specification of RecBinSearch(A, f, ℓ, x) consists of the following precondition/postcondition pair.

Precondition: $1 \leq f \leq \ell \leq length(A)$ and $A[f..\ell]$ is sorted.

Postcondition: Return t such that $f \leq t \leq \ell$ and $A[t] = x$, if such a t exists; otherwise, return 0.

Next we prove that RECBINSEARCH(A, f, ℓ, x) is correct with respect to this specification. More precisely,

Lemma 2.11 *Suppose that f and ℓ are integers such that $1 \leq f \leq \ell \leq \mathit{length}(A)$, and that $A[f..\ell]$ is sorted when* RECBINSEARCH(A, f, ℓ, x) *is called. Then this call terminates and returns t such that $f \leq t \leq \ell$ and $A[t] = x$, if such a t exists; otherwise it returns 0.*

PROOF. We will prove this lemma using induction. But induction on what exactly? The call RECBINSEARCH(A, f, ℓ, x) searches for x in the subarray $A[f..\ell]$. The induction will be on the length of this subarray. Intuitively this means that we will prove that the recursive algorithm works correctly on subarrays of length 1 (the basis of the induction); and that is works correctly on subarrays of arbitrary length $i > 1$, *assuming that it works correctly on subarrays of length less than* i (the induction step). Notice that the recursive call made by RECBINSEARCH(A, f, ℓ, x) is for searching a subarray whose length is roughly *half* the length of $A[f..\ell]$. It is therefore necessary to use *complete* induction.

We now show the proof in detail. Notice that if $1 \leq f \leq \ell \leq \mathit{length}(A)$, then $\mathit{length}(A[f..\ell]) = \ell - f + 1$. Let $P(k)$ be the following predicate:

$P(k)$: if f, ℓ are integers such that $1 \leq f \leq \ell \leq \mathit{length}(A)$ and $\mathit{length}(A[f..\ell]) = k$,
and $A[f..\ell]$ is sorted when RECBINSEARCH(A, f, ℓ, x) is called,
then this call terminates and returns some t such that $f \leq t \leq \ell$ and $A[t] = x$,
if such a t exists; otherwise it returns 0.

Intuitively $P(k)$ states that RECBINSEARCH correctly searches subarrays of length k. Using complete induction, we will prove that $P(k)$ holds for all integers $k \geq 1$. This means that RECBINSEARCH correctly searches subarrays of *any* length.

Let i be an arbitrary integer such that $i \geq 1$. Assume that $P(j)$ holds for all j such that $1 \leq j < i$. We must prove that $P(i)$ holds as well.

CASE 1. $i = 1$. Let f, ℓ be integers such that $1 \leq f \leq \ell \leq \mathit{length}(A)$ and $\mathit{length}(A[f..\ell]) = 1$. This means that the subarray $A[f..\ell]$ has only one element. Thus, $\ell = f$, and the "if" branch in lines 1–6 is executed. Therefore, RECBINSEARCH(A, f, ℓ, x) terminates by returning in line 3 or 5. Since there is only one element in $A[f..\ell]$, if x is in $A[f..\ell]$ it must be that $A[f] = x$ and so RECBINSEARCH(A, f, ℓ, x) returns f in line 3; if, on the other hand, x is not in $A[f..\ell]$, $A[f] \neq x$ and so RECBINSEARCH(A, f, ℓ, x) returns 0 in line 5. In either case, the call returns the right value. Thus, $P(1)$ holds.

CASE 2. $i > 1$. Let f, ℓ be integers such that $1 \leq f \leq \ell \leq \mathit{length}(A)$ and $\mathit{length}(A[f..\ell]) = i$, and suppose that $A[f..\ell]$ is sorted when RECBINSEARCH(A, f, ℓ, x) is called. Since $\mathit{length}(A[f..\ell]) = i$ and $i > 1$, it follows that $f < \ell$. Therefore the "else" branch in lines 7–14 is executed. Let $m = (f + \ell)$ **div** 2 (cf. line 8). By Lemma 2.2,

$$f \leq m < \ell \tag{2.8}$$

There are two subcases, depending on the outcome of the comparison in line 9:

Subcase 2(a). $A[m] \geq x$. In this case, RecBinSearch(A, f, m, x) is called in line 10. Let $j = \mathit{length}(A[f..m])$. By (2.8), $1 \leq j < i$. Therefore, by induction hypothesis, $P(j)$ holds.

must first verify precondition of induction hypothesis

Furthermore, $1 \leq f \leq m \leq \mathit{length}(A)$ (by (2.8) and the assumption that $1 \leq f \leq \ell \leq \mathit{length}(A)$), and $A[f..m]$ is sorted when RecBinSearch(A, f, m, x) is called in line 10 (because, by assumption, $A[f..\ell]$ is sorted when RecBinSearch(A, f, ℓ, x) is called, and RecBinSearch does not change A). Therefore, we know from $P(j)$ that the call RecBinSearch(A, f, m, x) terminates, and hence so does RecBinSearch(A, f, ℓ, x). It remains to prove that RecBinSearch(A, f, ℓ, x) returns the right value.

First suppose x is in $A[f..\ell]$. Then x must be in $A[f..m]$. This is obvious if $A[m] = x$. To see why it is also true if $A[m] \neq x$ note that then $A[m] > x$ (since, by the hypothesis of the subcase, $A[m] \geq x$). But then, since $A[f..\ell]$ is sorted, $A[t] > x$ for all t such that $m + 1 \leq t \leq \ell$. Since x is in $A[f..\ell]$ but is not in $A[m+1..\ell]$, x must be in $A[f..m]$. Since x is in $A[f..m]$, we know from $P(j)$ that the call RecBinSearch(A, f, m, x) returns some t such that $f \leq t \leq m$ and $A[t] = x$. By line 10, the same t is returned by RecBinSearch(A, f, ℓ, x). So, in this case RecBinSearch(A, f, ℓ, x) returns the right value.

Finally, suppose x is not in $A[f..\ell]$. Obviously, x is not in $A[f..m]$. Therefore, from $P(j)$ we know that the call RecBinSearch(A, f, m, x) returns 0. So, RecBinSearch(A, f, ℓ, x) also returns 0 in line 10. In this case too, RecBinSearch(A, f, ℓ, x) returns the right value.

Subcase 2(b). $A[m] < x$. This is similar to the previous subcase and is omitted. □

Finally, we use the correctness of RecBinSearch to prove that MainBinSearch is correct with respect to its own specification.

Corollary 2.12 *Suppose that when the program* MainBinSearch(A, x) *is started A is a sorted array of length at least 1. Then the program terminates and returns some t such that $1 \leq t \leq \mathit{length}(A)$ and $A[t] = x$, if such a t exists; otherwise it returns 0.*

Proof. By the hypothesis, when MainBinSearch calls RecBinSearch$(A, 1, \mathit{length}(A), x)$, A is a sorted array of length at least 1. By Lemma 2.11, this call terminates, and thus so does MainBinSearch. Also by Lemma 2.11, this call returns t such that $1 \leq t \leq \mathit{length}(A)$ and $A[t] = x$, if such a t exists; otherwise, it returns 0. MainBinSearch(A, x) returns the same value, as wanted. □

It is perhaps instructive to compare the overall structure of this proof with the correctness proof of the iterative version of binary search, discussed in Section 2.3. The first step of the proof (formulation of the correctness specification for RecBinSearch) is analogous to formulating an invariant for the loop of BinSearch. (Note the similarity of the loop in lines 2–9 in Figure 2.1, and lines 7–14 in Figure 2.4.)

The second step (proving that RecBinSearch is correct with respect to its specification) has many similarities to the proof that the statement that we came up as a loop invariant for

BinSearch is, in fact, an invariant. Both proofs essentially show that each recursive call — or each iteration of the loop — "does the right thing", and they do so by using induction. There are some differences, however. The proof of the loop invariant uses simple induction: iteration $i+1$ does the right thing assuming iteration i did the right thing. The correctness proof of the recursive program may need to use complete induction: a recursive call on inputs of size i does the right thing assuming recursive calls on inputs of *any* size less than i (not necessarily inputs of size $i-1$) do the right thing. Also note that in the correctness proof of the recursive program we need to show that the recursive calls made by the program are on inputs of smaller size than the original call. This is similar to proving termination of iterative programs where we have to show that some nonnegative integer quantity (here, some measure of the input size) decreases.

Finally, the third step (proving that the overall program is correct using the fact that the recursive program is) is analogous to proving that the loop invariant and the loop exit condition yield the postcondition of binary search.

2.8 Correctness of a recursive sorting program

Let A be an array of numbers that we wish to sort. That is, we want to rearrange the elements of A so that, when the algorithm is done,

$$A[i] \leq A[i+1] \qquad \text{for each } i \in \mathbb{N} \text{ such that } 1 \leq i < \mathit{length}(A).$$

(Recall that arrays are indexed starting at position 1, and that $\mathit{length}(A)$ denotes the number of elements of array A.) We will now introduce a recursive sorting algorithm called ***Mergesort*** and will prove its correctness. Mergesort is very important in practice, as it is the basis of almost all utilities for sorting large files stored in secondary storage (i.e., on disk or tape).

2.8.1 How Mergesort works

The basic idea behind Mergesort is quite simple. To sort the array A we proceed as follows:

1. If $\mathit{length}(A) = 1$ then we are done: the array is already sorted!

2. Otherwise, let A_1 and A_2 be the two halves of array A, as suggested in the figure below

Recursively sort A_1 and A_2. Notice that each of A_1 and A_2 has about $\mathit{length}(A)/2$ elements. More precisely, one of them has $\lceil \mathit{length}(A)/2 \rceil$ and the other has $\lfloor \mathit{length}(A)/2 \rfloor$ elements.

3. Merge the two (now sorted) subarrays A_1 and A_2 into a single sorted array.

Consider the third step of this algorithm, where we must merge two sorted subarrays. Recall that, for any integers i and j such that $1 \leq i \leq j \leq \mathit{length}(A)$, $A[i..j]$ denotes the subarray of A between indices i and j, inclusive. Note that $\mathit{length}(A[i..j]) = j - i + 1$.

In the case of Mergesort, the two sorted subarrays that we want to merge are consecutive subarrays of A. That is, the low index of the second (A_2, in the previous figure) is one more than the high index of the first (A_1, in the previous figure). Suppose that

(a) f, m, ℓ are integers such that $1 \leq f \leq m < \ell \leq \mathit{length}(A)$ (f stands for "first", m for "middle", and ℓ for "last"); and

(b) $A[f..m]$ and $A[m+1..\ell]$ are both sorted.

Then the algorithm MERGE(A, f, m, ℓ) shown in Figure 2.5 merges these two subarrays, placing the resulting sorted array in $A[f..\ell]$. The idea is to scan the two subarrays simultaneously, considering the elements of each in increasing order; indices i and j keep track of where we are in subarrays $A[f..m]$ and $A[m+1..\ell]$, respectively. We compare $A[i]$ and $A[j]$, outputting the smaller of the two into an auxiliary array and advancing the corresponding index. We continue in this fashion until we have exhausted one of the two subarrays, at which point we simply transfer the remaining elements of the other subarray to the auxiliary array. Finally, we transfer the elements from the auxiliary array back to $A[f..\ell]$.

Using this procedure, we can now write the Mergesort algorithm as shown in Figure 2.6. The procedure MERGESORT(A, f, ℓ) sorts subarray $A[f..\ell]$, where $1 \leq f \leq \ell \leq \mathit{length}(A)$. To sort the entire array A, we simply call MERGESORT$(A, 1, \mathit{length}(A))$.

2.8.2 The correctness of Mergesort

We now address the correctness of MERGESORT. There are two parts to proving this program correct. First, we prove that the MERGE algorithm in Figure 2.5 is correct. Using this, we then prove that MERGESORT is correct.

First we briefly look at the correctness of MERGE(A, f, m, ℓ). This requires us to prove that *if* the preconditions of the algorithm hold before it is invoked (i.e., if $1 \leq f \leq m < \ell \leq \mathit{length}(A)$ and $A[f..m]$, $A[m+1..\ell]$ are sorted) *then* the algorithm (a) terminates and (b) when it does, $A[f..m]$ is sorted and all other elements of A are unchanged. Since MERGE is an iterative program, this proof can be carried out by using the techniques discussed in Section 2.1. We leave this proof as an interesting exercise.

Next we address the correctness of MERGESORT(A, f, ℓ). The methodology is similar to that of the proof of correctness of the recursive version of binary search, discussed in Section 2.7. We must prove that *if* the preconditions of the algorithm hold before it is invoked (i.e., if $1 \leq f \leq \ell \leq \mathit{length}(A)$) *then* the algorithm (a) terminates, and (b) when it does, $A[f..\ell]$ is sorted[7] and all other elements of A are unchanged. We will prove this fact using complete induction.

[7] We use the phrase "$A[f..\ell]$ is sorted" as an abbreviation for the longer but more precise statement "$A[f..\ell]$ contains the same elements as before the invocation, in nondecreasing order".

```
MERGE($A, f, m, \ell$)
  ▶ Preconditions: (a) $1 \leq f \leq m < \ell \leq length(A)$, and
  ▶                (b) $A[f..m]$ and $A[m+1..\ell]$ are sorted
  ▶ Postcondition: $A[f..\ell]$ has the same elements as before invocation, in sorted order; and
  ▶                all other elements of $A$ are unchanged
    $i := f$; $j := m + 1$                    ▶ indices into $A$
    $k := f$                                  ▶ index into $aux$
    ▶ Merge subarrays until one of the two is exhausted
    while $i \leq m$ and $j \leq \ell$ do
        if $A[i] < A[j]$ then
            $aux[k] := A[i]$
            $i := i + 1$
        else
            $aux[k] := A[j]$
            $j := j + 1$
        end if
        $k := k + 1$
    end while
    ▶ Determine bounds of the rest of the unexhausted subarray
    if $i > m$ then                           ▶ first subarray was exhausted
        $low := j$
        $high := \ell$
    else                                      ▶ second subarray was exhausted
        $low := i$
        $high := m$
    end if
    ▶ Copy the rest of the unexhausted subarray into $aux$
    for $t := low$ to $high$ do
        $aux[k] := A[t]$
        $k := k + 1$
    end for
    ▶ Copy $aux$ back to $A[f..\ell]$
    for $t := f$ to $\ell$ do
        $A[t] := aux[t]$
    end for
```

Figure 2.5: The MERGE algorithm

```
Mergesort($A, f, \ell$)
   ▶ Precondition: $1 \leq f \leq \ell \leq length(A)$
   ▶ Postcondition: $A[f..\ell]$ has the same elements as before invocation, in sorted order; and
   ▶                all other elements of $A$ are unchanged
1     if $f = \ell$ then                        ▶ Subarray is already sorted
2         return
3     else
4         $m := (f + \ell)$ div 2               ▶ $m$: index of the middle element of $A[f..\ell]$
5         Mergesort($A, f, m$)                  ▶ sort first half
6         Mergesort($A, m + 1, \ell$)           ▶ sort second half
7         Merge($A, f, m, \ell$)                ▶ merge the two sorted parts
8     end if
```

Figure 2.6: Mergesort

Let $P(k)$ be the predicate:

$P(k)$: if f, ℓ are integers such that $1 \leq f \leq \ell \leq length(A)$ and $length(A[f..\ell]) = k$ then Mergesort(A, f, ℓ) terminates and, when it does, $A[f..\ell]$ is sorted and all other elements of A are unchanged

Thus, $P(k)$ states that Mergesort correctly sorts subarrays of length k. Using complete induction, we will prove that $P(k)$ holds for all integers $k \geq 1$. This implies that Mergesort correctly sorts subarrays of *any* length, i.e., that Mergesort is correct.

Let i be an arbitrary integer such that $i \geq 1$. Assume that $P(j)$ holds for all j such that $1 \leq j < i$. We must prove that $P(i)$ holds as well.

Case 1. $i = 1$. Let f, ℓ be integers such that $1 \leq f \leq \ell \leq length(A)$ and $length(A[f..\ell]) = 1$. This means that the subarray $A[f..\ell]$ has only one element. Recall that $length(A[f..\ell]) = \ell - f + 1$; so in this case we have $\ell = f$. By lines 1 and 2 (all lines in this proof refer to Figure 2.6 on page 69), the algorithm terminates and, trivially, $A[f..\ell]$ is sorted and all elements of A are unchanged. Thus, $P(1)$ holds.

Case 2. $i > 1$. Let f, ℓ be integers such that $1 \leq f \leq \ell \leq length(A)$ and $length(A[f..\ell]) = i$. We must prove that Mergesort(A, f, ℓ) terminates and, when it does, $A[f..\ell]$ is sorted and all other elements of A are unchanged. Since $length(A[f..\ell]) = i$ and $i > 1$, we have that $\ell - f + 1 > 1$, i.e., $f < \ell$. Let $m = (f + \ell)$ **div** 2 (cf. line 4). By Lemma 2.2, we have

$$f \leq m < \ell \tag{2.9}$$

Using (2.9), as well as the facts that $length(A[f..m]) = m - f + 1$ and $length(A[m+1..\ell]) = \ell - m$, we can conclude that

$$1 \leq length(A[f..m]) < i \qquad \text{and} \qquad 1 \leq length(A[m + 1..\ell]) < i$$

By induction hypothesis, then, $P(\mathit{length}(A[f..m]))$ and $P(\mathit{length}(A[m{+}1..\ell]))$ both hold. From (2.9) and the fact that $1 \le f \le \ell \le \mathit{length}(A)$, it follows that

$$1 \le f \le m \le \mathit{length}(A) \qquad \text{and} \qquad 1 \le m+1 \le \ell \le \mathit{length}(A)$$

Therefore,

- Mergesort(A, f, m) terminates and, when it does, $A[f..m]$ is sorted and all other elements of A are unchanged; and
- Mergesort$(A, m{+}1, \ell)$ terminates and, when it does, $A[m{+}1..\ell]$ is sorted and all other elements of A are unchanged.

Thus, lines 5 and 6 terminate and, just before line 7 is executed, $1 \le f \le m < \ell \le \mathit{length}(A)$, and $A[f..m]$ and $A[m+1..\ell]$ are sorted and all other elements of A are unchanged.[8] By the correctness of the Merge(A, f, m, ℓ) algorithm, line 7 terminates and, when it does, $A[f..\ell]$ is sorted and all other elements of A are unchanged, as wanted.

This completes the proof that $P(k)$ holds for every integer $k \ge 1$, and therefore that Mergesort is correct.

[8] Notice that at this point we are making use of the fact that each recursive call leaves the other elements of A unchanged. The facts that Mergesort(A, f, m) sorts $A[f..m]$ and Mergesort$(A, m{+}1, \ell)$ sorts $A[m{+}1..\ell]$ do *not* (by themselves) imply that after both of these calls are made, the entire subarray $A[f..\ell]$ is sorted. For example, this would not be true if the second call, Mergesort$(A, m+1, \ell)$, messed up the results of the first call, Mergesort(A, f, m), by rearranging elements of $A[f..m]$. It is for precisely this reason that we needed to include in the predicate $P(k)$ the property that each recursive call leaves the other elements of A unchanged.

Exercises

1. Prove that Theorem 2.5 is equivalent to the principle of well-ordering (see Section 1.1.1); i.e., each implies the other.

2. Consider the binary search program in Figure 2.1. Suppose that we change line 7 to $f := m$ (instead of $f := m + 1$). Is the resulting program correct? If so, prove that it is, by explaining what changes need to be made to the proof given in Section 2.3. If the program is not correct, state whether it is partial correctness or termination (or both) that is violated. If you say that a property (partial correctness or termination) is violated, show an input that causes the violation of the property. If you say that a property is not violated, explain why.

3. The binary search program in Figure 2.1 compares the middle element, $A[m]$, of $A[f..\ell]$ to x. The comparison is two-way: If $A[m] \geq x$ then the search is confined to $A[f..m]$; if $A[m] < x$ the search is confined to $A[m+1..\ell]$. Suppose that, instead we do a three-way comparison: We first test if $A[m] = x$ and, if so, we immediately return m; otherwise, if $A[m] > x$ we confine the search to $A[f..m-1]$, and if $A[m] < x$ we confine the search to $A[m+1..\ell]$. Write a program that implements this version of binary search and prove that your program is correct.

4. The algorithm Occur(A, f, ℓ, x) below returns the number of occurrences of x in array A between positions f and ℓ. More precisely, it satisfies the following precondition/postcondition pair:

Precondition: A is an array, and f and ℓ are integers such that $1 \leq f \leq \ell \leq \mathit{length}(A)$.

Postcondition: The integer returned by the algorithm is $|\{j : f \leq j \leq \ell \text{ and } A[j] = x\}|$.

```
Occur($A, f, \ell, x$)
$t := 0$
$i := f$
while $i \leq \ell$ do
    if $A[i] = x$ then $t := t + 1$ end if
    $i := i + 1$
end while
return $t$
```

Prove that Occur is correct with respect to the given specification.

5. The algorithm Mod(x, m) below returns the remainder of the division of x by m. More precisely, it satisfies the following precondition/postcondition pair:

Precondition: $x, m \in \mathbb{N}$ and $m \neq 0$.

Postcondition: The integer r returned by the algorithm is such that (a) $0 \leq r < m$ and (b) there is some $q \in \mathbb{N}$ such that $x = qm + r$.

```
Mod($x, m$)
$r := x$
```

```
while $r \geq m$ do
    $r := r - m$
end while
return $r$
```

Prove that Mod is correct with respect to the given specification.

6. Change the Boolean condition on line 4 of Mult from $x \neq 0$ to $x > 0$. Does the program remain correct? Justify your answer by either giving a counterexample (an input on which the program does not terminate or returns an incorrect output) or describing the parts in the proof of correctness given in Section 2.4 that must be changed as a result of this modification.

7. Prove that the program Mult3 below is correct with respect to its specification.
Hint: This is similar to the program discussed in Section 2.4, except that instead of viewing m and n in binary, here we view them in ternary (i.e., base-3).

```
Mult3$(m, n)$
▶ Precondition: $m \in \mathbb{N}$ and $n \in \mathbb{Z}$.
▶ Postcondition: Return $m \cdot n$.
    $x := m$;    $y := n$;    $z := 0$
    while $x \neq 0$ do
        if $x$ mod $3 = 1$ then
            $z := z + y$
            $x := x$ div $3$
        else if $x$ mod $3 = 0$ then
            $x := x$ div $3$
        else
            $z := z - y$
            $x := (x + 1)$ div $3$
        end if
        $y := y * 3$
    end while
    return $z$
```

8. Prove that the loop below terminates if the precondition holds before the loop starts.

```
▶ Precondition: $x, y \in \mathbb{N}$ and $x$ is even.
while $x \neq 0$ do
    if $y \geq 1$ then
        $y := y - 3$; $x := x + 2$
    else
        $x := x - 2$
    end if
end while
```

Repeat the same exercise if the assignment to y in line 3 is changed to $y := y - 1$ (instead of $y := y - 3$).

9. Prove that the following program halts for every input $x \in \mathbb{N}$.

```
▶ Precondition: $x \in \mathbb{N}$.
$y := x * x$
while $y \neq 0$ do
    $x := x - 1$
    $y := y - 2 * x - 1$
end while
```

Hint: Derive (and prove) a loop invariant whose purpose is to help prove termination.

Chapter 3

FUNCTIONS DEFINED BY INDUCTION

Induction can be used not only to prove mathematical statements but also to define mathematical objects. In this chapter we will see how induction can be used to define functions and to prove properties of functions defined inductively. In Chapter 4 we will see how induction can be used to define other mathematical objects and to prove properties of objects defined in this manner. Inductive definitions are "constructive": in addition to defining an object, they effectively provide a recursive algorithm to construct it. For this reason, they are also called ***recursive*** definitions. Because of their "constructive" nature, recursive definitions are especially common and useful in computer science.

3.1 Recursively defined functions

We begin with an example and then give a general form for recursive definition of functions.

Example 3.1 Let $f : \mathbb{N} \to \mathbb{N}$ be the function defined as follows:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ f(n-1) + 2n - 1 & \text{if } n > 0 \end{cases} \tag{3.1}$$

This is a recursive definition. It defines the value of the function at some natural number n in terms of the function's value at the previous number, $n - 1$; and it defines the function at 0 explicitly, since there is no natural number "previous" to 0. We will refer to the first part of the definition as the "basis" and to the second part as the "inductive step" of the recursion.

We can compute $f(n)$ for an arbitrary value n by "unwinding" the recursion. For example,

- to compute $f(4)$ we need to first compute $f(3)$,
- for which we need to first compute $f(2)$,
- for which we need to first compute $f(1)$,

- for which we need to first compute $f(0)$, which is given explicitly.

Now working forward, we get:

- $f(0) = 0$
- $f(1) = f(0) + 2 \cdot 1 - 1 = 0 + 2 - 1 = 1$
- $f(2) = f(1) + 2 \cdot 2 - 1 = 1 + 4 - 1 = 4$
- $f(3) = f(2) + 2 \cdot 3 - 1 = 4 + 6 - 1 = 9$
- $f(4) = f(3) + 2 \cdot 4 - 1 = 9 + 8 - 1 = 16$

Given these calculations, it looks as though the function f simply computes the square of its argument! We prove that this is, in fact, the case.

Proposition 3.1 *For any $n \in \mathbb{N}$, $f(n) = n^2$.*

PROOF. Let $P(n)$ be the predicate defined by:

$$P(n) : \qquad f(n) = n^2$$

We use induction to prove that $P(n)$ holds for all $n \in \mathbb{N}$.
BASIS: We have,

$$\begin{aligned} f(0) &= 0 && \text{[by definition of } f\text{]} \\ &= 0^2 \end{aligned}$$

Thus, $P(0)$ holds.
INDUCTION STEP: Let i be an arbitrary natural number, and suppose that $P(i)$ holds; i.e., suppose that $f(i) = i^2$. We will prove that $P(i+1)$ holds as well; i.e., that $f(i+1) = (i+1)^2$. Indeed we have,

$$\begin{aligned} f(i+1) &= f(i) + 2(i+1) - 1 && \text{[by definition of } f\text{, since } i+1 > 0\text{]} \\ &= i^2 + 2(i+1) - 1 && \text{[by induction hypothesis]} \\ &= i^2 + 2 \cdot i + 1 \\ &= (i+1)^2 \end{aligned}$$

as wanted. □

This example illustrates a general phenomenon. When an object (like the function f) is defined recursively, it is natural to try proving properties of the object (in this case that f is the square function) by using induction. **End of Example 3.1**

Why exactly do we accept Equation (3.1) as a bona fide definition of a function? This is not an idle question — there are certainly other equations that look very much like (3.1) but which are not proper definitions of functions. For instance, the equation

$$h(n) = h(n-1) + 5, \quad \text{for all } n > 0 \tag{3.2}$$

does not define a function. Intuitively, the reason is that this equation does not specify a basis case, so we can't tell what $h(0)$ is — or, for that matter, what $h(k)$ is for any $k \in \mathbb{N}$. Indeed, there are infinitely many functions from $\mathbb{N}$ to $\mathbb{N}$ that satisfy (3.2): we obtain a different one for each choice of a value for the basis.

The equation

$$h'(n) = \begin{cases} 0, & \text{if } n = 0 \\ h'(n-2) + 5, & \text{if } n > 0 \end{cases}$$

also fails to properly define a function. Now the problem is not the lack of a basis in the definition. Rather, the induction step does not properly define the value of a function when $n = 1$ and, more generally, for any odd value of n.

Similarly, the equation

$$h''(n) = \begin{cases} 0, & \text{if } n = 0 \\ h''(\lceil n/2 \rceil) + 5, & \text{if } n > 0 \end{cases}$$

fails to properly define a function, because in the induction step, for $n = 1$, $h''(1)$ is defined in terms of $h''(1)$, which is circular.

How about the following equation?

$$f'(n) = \begin{cases} 0, & \text{if } n = 0 \\ f'(n+1) - 2n - 1, & \text{if } n > 0 \end{cases} \tag{3.3}$$

This also seems to not define a function properly. Superficially, the reason appears to be that it defines $f'(n)$ in terms of the value of f' at a larger argument, $f'(n+1)$. It is true that (3.3) does not define a function, but our reasoning as to why that is so is not adequate: A seemingly minor modification of this equation (which also appears to define $f(n)$ in terms of $f(n+1)$) is really just a different way of writing (3.1)

$$f(n) = \begin{cases} 0, & \text{if } n = 0 \\ f(n+1) - 2n - 1, & \text{if } n \geq 0 \end{cases} \tag{3.4}$$

To see why (3.4) is the same thing as (3.1), note that the induction step of (3.1) can be rewritten as

$$f(n-1) = f(n) - 2n + 1, \quad \text{if } n > 0$$

By replacing $n - 1$ by n (and, therefore, n by $n + 1$) this is the same as

$$\begin{aligned} f(n) &= f(n+1) - 2(n+1) + 1, && \text{if } n + 1 > 0 \\ &= f(n+1) - 2n - 1, && \text{if } n \geq 0 \end{aligned}$$

which is just the "induction step" of (3.4). Since we are happy with (3.1) as defining a function, we ought to be just as happy with (3.4).

The conclusion to be drawn from these examples is that there is something rather subtle going on with recursive definitions in the style of (3.1), and we need a firm grounding on which to base such definitions. The following Principle provides just that.

Principle of function definition by recursion: *Let $b \in \mathbb{Z}$, and $g : \mathbb{N} \times \mathbb{Z} \to \mathbb{Z}$ be a function. Then there is a unique function $f : \mathbb{N} \to \mathbb{Z}$ that satisfies the following equation:*

$$f(n) = \begin{cases} b, & \text{if } n = 0 \\ g(n, f(n-1)), & \text{if } n > 0 \end{cases} \tag{3.5}$$

The definition of $f(n)$ in Example 3.1 follows this principle, where $b = 0$, and $g(n, m) = m + 2n - 1$. Therefore, assuming this principle is valid, $f(n)$ is properly defined. It is possible to prove that the principle of induction implies the principle of function definition by recursion. Thus, if we are prepared to accept the validity of the the principle of induction (or any of the other two that are equivalent to it: complete induction or well-ordering) we must thereby also accept the validity of the principle of function definition by recursion.

Our next example shows a recursive definition of a function based on *complete*, rather than simple, induction.

Example 3.2 Consider the function $F : \mathbb{N} \to \mathbb{N}$ defined as follows:

$$F(n) = \begin{cases} 0, & \text{if } n = 0 \\ 1, & \text{if } n = 1 \\ F(n-1) + F(n-2), & \text{if } n > 1 \end{cases} \tag{3.6}$$

Notice that the basis in this definition encompasses two numbers, 0 and 1, while the induction step defines the value of F at point n in terms of the values of the function at the previous *two* points: $n-1$ and $n-2$. The function defined by (3.6) is called the ***Fibonacci function***, and the infinite sequence of integers $\langle F(0), F(1), F(2), \ldots \rangle$ is called the ***Fibonacci sequence***. It comes up in a surprising number of settings.

Perhaps surprisingly, we can give a "closed-form formula" for $F(n)$, i.e., a simple formula to compute $F(n)$ given n and without having to explicitly compute $F(n-1)$ and $F(n-2)$.[1]

[1] A closed-form formula is one that allows us to calculate a function by applying only a *fixed* number of *"basic"* operations to its argument(s) — where by "basic" operations we mean ones such as addition, subtraction, multiplication, division, and exponentiation. An expression such as "$1 + 2 + \ldots + n$" is not considered closed-form, because although the operations are basic (just addition), their number is not fixed. An expression such as $F(n-1) + F(n-2)$ is not considered closed-form, because although the number of operations is fixed (two applications of F and an addition), one of the operations (applying F) is not "basic". The term "closed-form formula" is not entirely precise because there is some ambiguity as to what constitutes a "basic" operation.

closed form has to be
1. basic operation
2. fixed number

Let $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$. The quantity ϕ is approximately equal to $1.6180\ldots$ and is known as the "golden ratio".[2]

Theorem 3.2 *For any $n \in \mathbb{N}$, $F(n) = \frac{\phi^n - (\hat{\phi})^n}{\sqrt{5}}$.*

PROOF. Let $P(n)$ be the predicate defined as:

$$P(n): \qquad F(n) = \frac{\phi^n - (\hat{\phi})^n}{\sqrt{5}}.$$

We use complete induction to prove that $P(n)$ holds for all $n \in \mathbb{N}$.

Let i be an arbitrary natural number, and assume that $P(j)$ holds for all $j < i$; i.e., for any j such that $0 \leq j < i$, $F(j) = \frac{\phi^j - (\hat{\phi})^j}{\sqrt{5}}$. We will prove that $P(i)$ holds as well; i.e., that $F(i) = \frac{\phi^i - (\hat{\phi})^i}{\sqrt{5}}$.

CASE 1. $i = 0$. We must prove that $F(0) = \frac{\phi^0 - (\hat{\phi})^0}{\sqrt{5}}$. The left-hand side is 0 by the definition of the Fibonacci function, and straightforward arithmetic shows that the right-hand side is 0 as well. Thus, the desired equation holds in this case.

CASE 2. $i = 1$. We must prove that $F(1) = \frac{\phi^1 - (\hat{\phi})^1}{2}$. The left-hand side is 1 by the definition of the Fibonacci function. For the right-hand side we have:

$$\frac{\phi^1 - (\hat{\phi})^1}{\sqrt{5}} = \frac{1}{\sqrt{5}} \cdot (\frac{1+\sqrt{5}}{2} - \frac{1-\sqrt{5}}{2}) = \frac{1}{\sqrt{5}} \cdot \frac{2\sqrt{5}}{2} = 1.$$

Thus, the desired equation holds in this case.

CASE 3. Let $i > 1$ be an arbitrary natural number. Then $0 \leq i-1, i-2 < i$. Therefore, by

[2] This quantity has been of interest to mathematicians since antiquity. It has the following interesting geometric interpretation: Suppose we are given a line segment and we wish to divide it into two parts, so that the ratio of the larger part to the smaller is the same as the ratio of the entire segment to the larger. Then, ϕ is that ratio. Euclid's *Elements* (*ci.* 300 BCE) contain a geometric construction (using only straight edge and compass) to achieve such a division of a line segment, and evidence suggests that this construction was already known to Pythagoras (*ci.* 500 BCE). The architects and sculptors of classical Greece were fond of rectangular shapes in which the ratio of the longer side to the shorter side is equal to ϕ, believing these to be the most aesthetically pleasing rectangles. Indeed, the association of the letter ϕ with this quantity is in honour of Φειδίας (Phidias), the greatest sculptor of classical Greece.

induction hypothesis, $P(i-1)$ and $P(i-2)$ hold. We have,

$$\begin{aligned}
F(i) &= F(i-1)+F(i-2) && \text{[by (3.6), since } i>1\text{]}\\
&= \frac{\phi^{i-1}-(\hat{\phi})^{i-1}}{\sqrt{5}}+\frac{\phi^{i-2}-(\hat{\phi})^{i-2}}{\sqrt{5}} && \text{[by induction hypothesis]}\\
&= \frac{1}{\sqrt{5}}\cdot\big(\phi^{i-2}(\phi+1)-(\hat{\phi})^{i-2}(\hat{\phi}+1)\big)\\
&= \frac{1}{\sqrt{5}}\cdot(\phi^{i-2}\phi^2-(\hat{\phi})^{i-2}(\hat{\phi})^2) && \text{[because } \phi^2=\phi+1 \text{ and } (\hat{\phi})^2=\hat{\phi}+1,\\
& && \text{by simple arithmetic]}\\
&= \frac{\phi^i-(\hat{\phi})^i}{\sqrt{5}}
\end{aligned}$$

as wanted.

End of Example 3.2

Definition (3.6) does not quite fit the principle of function definition by recursion, since it has multiple base cases. There is a more general principle that allows us to define function by recursion that encompasses (3.6).

Principle of function definition by complete recursion: *Let k,ℓ be positive integers, $b_0,b_1,\ldots,b_{k-1}$ be arbitrary integers, $h_1,h_2,\ldots,h_\ell:\mathbb{N}\to\mathbb{N}$ be functions such that $h_i(n)<n$ for each i, $1\le i\le\ell$ and each $n\ge k$, and $g:\mathbb{N}\times\mathbb{Z}^\ell\to\mathbb{Z}$ be a function ($\mathbb{Z}^\ell$ denotes the ℓ-fold Cartesian product of set $\mathbb{Z}$). Then there is a unique function $f:\mathbb{N}\to\mathbb{Z}$ that satisfies the following equation:*

$$f(n)=\begin{cases} b_n, & \text{if } 0\le n<k\\ g(n,f(h_1(n)),f(h_2(n)),\ldots,f(h_\ell(n))), & \text{if } n\ge k\end{cases} \tag{3.7}$$

To see how the Fibonacci function fits this pattern of definition, notice that we can obtain (3.6) from (3.7) by taking $k=\ell=2$, $b_0=0$, $b_1=1$, $h_1(n)=n-1$, $h_2(n)=n-2$, and $g(n,i,j)=i+j$. Also, it is easy to see that the recursive definition (3.5) is the special case of (3.7) where $k=1$ (i.e., only one basis case), $\ell=1$ and $h_1(n)=n-1$. The validity of the principle of function definition by complete recursion can be shown by induction.

Function definitions that fit this general pattern are also called ***recurrence equations***, ***recurrence relations*** or simply ***recurrences***. The latter two terms are more general in that they also refer to recursive formulas in which the equality is replaced by an inequality ($\le$, $<$, $\ge$ or $>$). In this case, the recurrence does not define a single function but the family of functions that satisfy the specified inequality.

3.2 *Divide-and-conquer recurrences*

Inductively defined functions arise naturally in the analysis of many recursive algorithms — especially, though not exclusively, the time complexity analysis of such algorithms. The general idea is that by examining the recursive structure of the algorithm we can come up with an inductive definition of a function that describes the quantity we wish to analyse — say, the time complexity of the algorithm.

There is an important class of recursive algorithms, called ***divide-and-conquer*** algorithms, that includes binary search and mergesort. The time complexity of such algorithms is described by recurrence relations of a particular form. We will study these so-called ***divide-and-conquer recurrences*** in this section. We will see how, from the structure of a divide-and-conquer algorithm, we can extract a recurrence relation that describes its time complexity. Furthermore, we will see how to "solve" this recurrence relation — that is, how to obtain the function in closed-form (see footnote 1). We do this first through a specific example of a divide-and-conquer algorithm and then by considering a general form of divide-and-conquer recurrences.

3.2.1 *The time complexity of* MergeSort

Recall the mergesort algorithm discussed in Section 2.8 and shown in Figure 2.6. For any positive integer n, define $T(n)$ to be the maximum number of steps executed by a call to $\textsc{MergeSort}(A, f, \ell)$, where n is the size of the subarray being sorted, i.e., $n = \ell - f + 1$. The function T describes the (worst-case) time complexity of $\textsc{MergeSort}(A, f, \ell)$ as a function of the size of the subarray $A[f..\ell]$ being sorted. We claim that the following recurrence relation describes $T(n)$, where c and d are positive real constants (i.e., quantities independent of n):

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ T(\lceil n/2 \rceil) + T(\lfloor n/2 \rfloor) + dn, & \text{if } n > 1 \end{cases} \tag{3.8}$$

The basis of this recurrence reflects the fact that when the subarray being sorted has length 1 the algorithm executes some number of steps, which we designate c. The exact value of c depends on what exactly we count as a step, a detail that is not particularly relevant to our analysis. The induction step reflects the fact that to sort a subarray of length $n > 1$, we must

- Recursively sort the first "half" of the subarray. By definition of T, this takes $T(\lceil n/2 \rceil)$ steps.
- Recursively sort the second "half" of the subarray. By definition of T, this takes $T(\lfloor n/2 \rfloor)$ steps.
- Merge the two sorted "halves", using the $\textsc{Merge}$ procedure. It is not hard to see that the merging algorithm takes time proportional to n. Again, the exact constant of proportionality depends on how one would count steps, an unimportant detail that we sidestep by using an unspecified constant d.

We must verify that equation (3.8) is a proper recursive definition. For this we need to ensure that for any integer $n > 1$, $1 \leq \lceil n/2 \rceil < n$ and $1 \leq \lfloor n/2 \rfloor < n$. The following lemma shows that this is the case.

Lemma 3.3 *For any integer $n > 1$, $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil < n$.*

Proof. If n is odd then $\lceil n/2 \rceil = \frac{n+1}{2}$; if n is even then $\lceil n/2 \rceil = n/2 < \frac{n+1}{2}$. Thus, for any $n \in \mathbb{N}$, $\lceil n/2 \rceil \leq \frac{n+1}{2}$. By hypothesis, $n > 1$; by adding n to both sides of this inequality we get $2n > n + 1$, and therefore $\frac{n+1}{2} < n$. Combining with the previous inequality we get that, for any integer $n > 1$, $\lceil n/2 \rceil < n$. It remains to show that $1 \leq \lfloor n/2 \rfloor \leq \lceil n/2 \rceil$. This follows immediately from the definition of the floor and ceiling functions, and the fact that $n > 1$. □

Our task now is to solve (3.8), i.e., to find a closed-form formula for $T(n)$. Let us simplify our task by eliminating the floors and ceilings in the induction step of (3.8). The reason for the floors and ceilings, in the first place, is to handle the case where n is odd — and therefore the two "halves" into which we divide the subarray are not of length $n/2$ each but, rather, of length $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$. Let us then assume that n is a power of 2, so that if we keep dividing n into two halves we always get an even number — except in the very last step when we divide a subarray of length 2 into two parts of length 1. So when n is a power of 2, we can simplify (3.8) as follows:

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ 2T(n/2) + dn, & \text{if } n > 1 \end{cases} \tag{3.9}$$

To find a closed-form formula for $T(n)$ we can use a technique called ***repeated substitution***. We "unwind" the recursive definition of $T(n)$ by repeatedly applying the induction step of the definition to smaller and smaller arguments of T. We keep doing this until we discover a pattern that will help us obtain a closed-form formula. In our particular case, we proceed as follows. Let n be an arbitrary power of 2; thus $n/2^i$ is also a power of 2 for every natural number $i \leq \log_2 n$. We can therefore legitimately substitute $n/2^i$ for n in (3.9) for any such i. We have,

$$\begin{aligned}
T(n) &= 2T(n/2) + dn && \text{[by (3.9)]} \\
&= 2\Big(2T(n/2^2) + dn/2\Big) + dn && \text{[by substituting } n/2 \text{ for } n \text{ in (3.9)]} \\
&= 2^2T(n/2^2) + 2dn \\
&= 2^2\Big(2T(n/2^3) + dn/2^2\Big) + 2dn && \text{[by substituting } n/2^2 \text{ for } n \text{ in (3.9)]} \\
&= 2^3T(n/2^3) + 3dn \\
&\;\vdots \\
&= 2^iT(n/2^i) + idn && \text{[after } i \text{ applications of the same substitution]}
\end{aligned}$$

Of course, the last step in this derivation is not rigorously justified. After unwinding the recursion for a couple of steps *it looks as though* we have discovered a pattern that will persist if we continue this process, but we don't really have a convincing proof that it does. So really, at this point we have only formulated a plausible conjecture regarding the value of $T(n)$. Not surprisingly, we can verify our conjecture by using induction. Specifically:

Lemma 3.4 *If n is a power of 2, for any natural number $i \leq \log_2 n$, the function defined by (3.9) satisfies the following equation: $T(n) = 2^i T(n/2^i) + idn$.*

Proof. Let $P(i)$ be the predicate

$P(i)$: For each $n \in \mathbb{N}$ that is a power of 2 such that $i \leq \log_2 n$, $T(n) = 2^i T(n/2^i) + idn$

We will use induction to prove that $P(i)$ holds for all $i \in \mathbb{N}$.

Basis: $i = 0$. In this case, the right-hand side of the equation is $2^0 T(n/2^0) + 0 \cdot dn$, which is obviously equal to $T(n)$. Thus $P(0)$ holds.

Induction Step: Let j be an arbitrary natural number. Assume that $P(j)$ holds; i.e., for each $n \in \mathbb{N}$ that is a power of 2 such that $j \leq \log_2 n$, $T(n) = 2^j T(n/2^j) + jdn$. We must show that $P(j+1)$ holds as well. Consider an arbitrary $n \in \mathbb{N}$ that is a power of 2 such that $j + 1 \leq \log_2 n$. We must prove that $T(n) = 2^{j+1} T(n/2^{j+1}) + (j+1)n$. Since n is a power of 2 and $j < \log_2 n$, it follows that $n/2^j$ is a power of 2 greater than 1. Therefore,

$$\begin{aligned} T(n) &= 2^j T(n/2^j) + jdn && \text{[by induction hypothesis]} \\ &= 2^j \Big(2T(n/2^{j+1}) + dn/2^j \Big) + jdn && \text{[by (3.9), since $n/2^j$ is a power of 2 greater than 1]} \\ &= 2^{j+1} T(n/2^{j+1}) + (j+1)dn \end{aligned}$$

as wanted. □

Using this lemma we can now obtain a closed-form formula for $T(n)$, when n is a power of 2.

Theorem 3.5 *If n is a power of 2, the function defined by (3.9) satisfies the following equation: $T(n) = cn + dn \log_2 n$.*

Proof. We have,

$$\begin{aligned} T(n) &= 2^{\log_2 n} T(n/2^{\log_2 n}) + (\log_2 n)dn && \text{[by Lemma 3.4, for $i = \log_2 n$]} \\ &= nT(1) + dn \log_2 n \\ &= cn + dn \log_2 n, \end{aligned}$$

as wanted. □

What can we say about $T(n)$ if n is not a power of 2? Although the simple closed-form formula we derived for $T(n)$ in Theorem 3.5 does not apply exactly in that case, we will prove that the result is "essentially" valid for arbitrary n. Specifically, we will show that there is a positive constant κ so that for all integers $n \geq 2$,

$$T(n) \leq \kappa n \log_2 n \tag{3.10}$$

(Readers familiar with the so-called "big-oh" notation will observe that this means precisely that $T(n)$ is in $O(n \log n)$.) The value of κ depends on the constants c and d. Although the inequality (3.10) does not tell us exactly what $T(n)$ is, it gives us valuable information about it: Within the constant factor κ, $T(n)$ grows no faster than the function $n \log_2 n$. In fact, using techniques very similar to those we will use to prove 3.10 it is also possible to derive a corresponding lower bound, namely, that there is a positive real number κ' so that for all integers $n \geq 2$, $T(n) \geq \kappa' n \log_2 n$.

First we need a preliminary result. A function f on numbers is called ***nondecreasing*** if the function's value never decreases as its argument increases. More precisely, for every numbers m, n in the range of f, if $m < n$ then $f(m) \leq f(n)$.

Lemma 3.6 *The function $T(n)$ defined by recurrence (3.8) is nondecreasing.*

PROOF. Let $P(n)$ be the following predicate:

$$P(n): \qquad \text{for every positive integer } m\text{, if } m < n \text{ then } T(m) \leq T(n)$$

We use complete induction to prove that $P(n)$ holds for all integers $n \geq 1$. Let k be an arbitrary integer such that $k \geq 1$. Suppose that $P(\ell)$ holds for all integers ℓ such that $1 \leq \ell < k$. We must prove that $P(k)$ holds as well.

CASE 1. $k = 1$. $P(1)$ holds trivially since there is no positive integer $m < 1$.

CASE 2. $k = 2$. Using (3.8) we compute the values of $T(1)$ and $T(2)$:

$$\begin{aligned} T(1) &= c \\ T(2) &= T(1) + T(1) + 2d = 2c + 2d \end{aligned}$$

Since $c, d \geq 0$, $T(1) \leq T(2)$ and so $P(2)$ holds.

CASE 3. $k > 2$. In this case, $1 \leq k-1 < k$ and, by Lemma 3.3, $1 \leq \lfloor k/2 \rfloor \leq \lceil k/2 \rceil < k$. Therefore, by induction hypothesis, $P(k-1)$, $P(\lfloor k/2 \rfloor)$ and $P(\lceil k/2 \rceil)$ all hold. Since $P(k-1)$ holds, to prove that $P(k)$ holds as well, it suffices to prove that $T(k-1) \leq T(k)$. Indeed we have,

$$\begin{aligned} T(k-1) &= T(\lfloor \tfrac{k-1}{2} \rfloor) + T(\lceil \tfrac{k-1}{2} \rceil) + d(k-1) && [\text{by (3.8), since } k > 2 \text{ and so } k-1 > 1] \\ &\leq T(\lfloor k/2 \rfloor) + T(\lceil k/2 \rceil) + dk && [\text{by } P(\lfloor k/2 \rfloor), P(\lceil k/2 \rceil) \text{ and the fact that } d \geq 0] \\ &= T(k) && [\text{by (3.8)}] \end{aligned}$$

Therefore, $T(k-1) \leq T(k)$, as wanted. □

We are now ready to prove the desired bound on the function defined by recurrence (3.8).

Theorem 3.7 *There is a constant $\kappa \geq 0$ (whose value depends on c and d), so that the function $T(n)$ defined by recurrence (3.8) satisfies: $T(n) \leq \kappa n \log_2 n$, for all integers $n \geq 2$.*

PROOF. Let n be an arbitrary positive integer such that $n \geq 2$, and let $\hat{n} = 2^{\lceil \log_2 n \rceil}$; i.e., $\hat{n}$ is the smallest power of 2 that is greater than or equal to n. Thus,

$$\frac{\hat{n}}{2} < n \leq \hat{n} \tag{3.11}$$

For every integer $n \geq 2$, we have:

$$\begin{aligned}
T(n) &\leq T(\hat{n}) && \text{[by (3.11) and Lemma 3.6]}\\
&= c\hat{n} + d\hat{n}\log_2 \hat{n} && \text{[by Theorem 3.5, since } \hat{n} \text{ is a power of 2]}\\
&\leq c(2n) + d(2n)\log_2(2n) && \text{[since } \hat{n} < 2n\text{, by (3.11)]}\\
&= 2cn + 2dn + 2dn\log_2 n &&\\
&\leq 2cn\log_2 n + 2dn\log_2 n + 2dn\log_2 n && \text{[since } n \geq 2 \text{ and so } \log_2 n \geq 1]\\
&= \kappa n \log_2 n
\end{aligned}$$

where $\kappa = 2c + 4d$. Note that since $c, d \geq 0$, we also have that $\kappa \geq 0$. Therefore, there is a $\kappa \geq 0$ so that for all $n \geq 2$, $T(n) \leq \kappa n \log_2 n$, as wanted. □

3.2.2 General form of divide-and-conquer recurrences

Divide-and-conquer algorithms

Mergesort and Binary Search are examples of a general problem-solving technique known as ***divide-and-conquer***. This technique can be used to solve efficiently many different problems. We now informally describe the general structure of divide-and-conquer algorithms. Suppose we are given a computational problem such as sorting an array or multiplying two numbers. An ***instance*** of the problem is a legitimate input for that problem (in the case of sorting, an array). A ***solution*** of an instance is an output for that instance (in the case of sorting, the given array with its elements rearranged in nondecreasing order). Each instance has a ***size***, expressed as a natural number. What exactly is the size of an instance depends on the problem in question. For example, in the case of sorting, the size of an instance might be taken to be the number of elements that must be sorted; in the case of multiplying two numbers, the size of an instance might be taken to be the total number of bits needed to represent them. With this terminology in mind, we can describe a divide-and-conquer algorithm for a problem as follows: To solve a "large" instance of the problem, we

(a) Divide up the given instance of size n into a smaller instances *of the same problem*, each of size roughly n/b.

(b) Recursively solve each of the smaller instances (this can be done because they are instances of the same problem).

(c) Combine the solutions to the smaller instances into the solution of the given "large" instance.

If the given instance is "small enough" then we solve it directly using an ad hoc method. In some cases this is completely trivial — e.g., as in sorting an array of one element.

In this general scheme, a represents the number of subinstances into which we divide up the given one, and b represents the factor by which we reduce the input size in each recursive step. Therefore, a and b are positive integers, and $b > 1$ so that $n/b < n$.

In the above description of divide-and-conquer algorithms we said that we divide up a large instance of size n into a instances each of size roughly n/b. The qualification "roughly" is necessary because n/b might not be a natural number. Typically, some of the subinstances, say a_1 of them, are of size $\lceil n/b \rceil$ and the remaining, say a_2, are of size $\lfloor n/b \rfloor$, where a_1 and a_2 are natural numbers so that $a_1 + a_2 = a \geq 1$.

A general divide-and-conquer recurrence

Consider a divide-and-conquer algorithm of the type just described and let $T(n)$ be the maximum number of steps that the algorithm requires for instances of size n. Assume that the number of steps required for parts (a) and (c) of the algorithm — i.e., dividing up the given instance and combining the solutions to the smaller instances — is given by the polynomial dn^ℓ, where d and ℓ are nonnegative real constants (i.e., quantities that do not depend on n). Then the following recurrence relation describes $T(n)$:

$$T(n) = \begin{cases} c, & \text{if } 1 \leq n < b \\ a_1 T(\lceil \frac{n}{b} \rceil) + a_2 T(\lfloor \frac{n}{b} \rfloor) + dn^\ell, & \text{if } n \geq b \end{cases} \tag{3.12}$$

where, as discussed before, a_1 and a_2 are natural numbers such that $a = a_1 + a_2 \geq 1$, b is a natural number such that $b > 1$, and c, d, ℓ are nonnegative reals. For example the mergesort recurrence (3.8) is an instance of the above general form with $a_1 = a_2 = 1$, $b = 2$ and $\ell = 1$.

To keep the recurrence relation relatively simple, we assume that all base cases take the same number of steps, c. We have chosen the base cases to be the values of n such that $1 \leq n < b$. In this way, and because b is an integer greater than 1 (so $b \geq 2$), in the induction step the values of n are such that $1 \leq \lfloor n/b \rfloor \leq \lceil n/b \rceil < n$, and so (3.12) is well-defined.

The induction step of the definition reflects the number of steps required by the three parts of the divide-and-conquer paradigm: The recursive solution of the smaller problem instances requires $a_1 T(\lceil \frac{n}{b} \rceil) + a_2 T(\lfloor \frac{n}{b} \rfloor)$ steps: this is the number of steps required to solve the a_1 instances of size $\lceil \frac{n}{b} \rceil$, and the a_2 instances of size $\lfloor \frac{n}{b} \rfloor$. The term dn^ℓ is the number of steps required to divide up the input and to combine the solutions to the smaller instances.

Solving the divide-and-conquer recurrence

Our task now is to find a closed-form formula for the function $T(n)$ defined by (3.12). Actually, the recurrence is complex enough that it is not possible to obtain a simple closed-form formula, even for values of n that are powers of b. We can, however, derive tight upper and lower bounds for $T(n)$ that are expressed as simple closed-form formulas.

As with the solution of the MERGESORT recurrence (cf. Section 3.2.1) the first step is to simplify our task by considering only values of n that are powers of b. In this case, (3.12) can be simplified as follows. For any n that is a power of b,

$$T(n) = \begin{cases} c, & \text{if } n = 1 \\ aT(\frac{n}{b}) + dn^\ell, & \text{if } n > 1 \end{cases} \tag{3.13}$$

where $a = a_1 + a_2$. The following theorem gives upper bounds for the function $T(n)$ defined by (3.13), when n is a power of b. The form of the upper bound depends on whether a is smaller, equal to, or greater than b^ℓ.

Theorem 3.8 *There is a constant $\kappa \geq 0$ (that depends on a, b, c, d and ℓ) so that, for all integers $n \geq b$ that are powers of b, the function $T(n)$ defined by recurrence (3.13) satisfies the following inequality:*

$$T(n) \leq \begin{cases} \kappa n^\ell, & \textit{if } a < b^\ell \\ \kappa n^\ell \log_b n, & \textit{if } a = b^\ell \\ \kappa n^{\log_b a}, & \textit{if } a > b^\ell \end{cases}$$

PROOF. Since n is a power of b, by applying repeated substitution to (3.13) we obtain:

$$\begin{aligned}
T(n) &= aT(\tfrac{n}{b}) + dn^\ell \\
&= a\Big(aT(\tfrac{n}{b^2}) + d(\tfrac{n}{b})^\ell\Big) + dn^\ell \\
&= a^2T(\tfrac{n}{b^2}) + \big(\tfrac{a}{b^\ell}\big)dn^\ell + dn^\ell \\
&= a^2\Big(aT(\tfrac{n}{b^3}) + d(\tfrac{n}{b^2})^\ell\Big) + \big(\tfrac{a}{b^\ell}\big)dn^\ell + dn^\ell \\
&= a^3T(\tfrac{n}{b^3}) + \big(\tfrac{a}{b^\ell}\big)^2dn^\ell + \big(\tfrac{a}{b^\ell}\big)dn^\ell + dn^\ell \\
&\vdots \\
&= a^iT(\tfrac{n}{b^i}) + dn^\ell \sum_{j=0}^{i-1} (\tfrac{a}{b^\ell})^j \qquad\qquad [\text{after } i \text{ applications, where } i \leq \log_b n]
\end{aligned}$$

(The last step in this sequence of equations is, of course, informal; it can justified rigorously using induction as in the proof of Lemma 3.4.) Taking $i = \log_b n$ we have:

$$T(n) = a^{\log_b n}T(1) + dn^\ell \sum_{j=0}^{\log_b n-1} (\tfrac{a}{b^\ell})^j = cn^{\log_b a} + dn^\ell \sum_{j=0}^{\log_b n-1} (\tfrac{a}{b^\ell})^j \tag{3.14}$$

(In the last step of (3.14) we used the following property of logarithms: for any $x, y > 0$, $x^{\log_b y} = y^{\log_b x}$. It is easy to verify this identity by taking base-b logarithms of both sides.)

For convenience, let $\alpha = a/b^\ell$. Using the formula for the geometric series, we have

$$\sum_{j=0}^{\log_b n-1} \alpha^j = \begin{cases} \frac{1-\alpha^{\log_b n}}{1-\alpha}, & \text{if } \alpha \neq 1 \\ \log_b n, & \text{if } \alpha = 1 \end{cases}$$

Therefore, by (3.14),

$$T(n) = \begin{cases} cn^{\log_b a} + dn^\ell \Big(\frac{1-\alpha^{\log_b n}}{1-\alpha} \Big), & \text{if } a \neq b^\ell \text{ (i.e., } \alpha \neq 1) \\ cn^\ell + dn^\ell \log_b n, & \text{if } a = b^\ell \text{ (i.e., } \alpha = 1) \end{cases} \tag{3.15}$$

(For the second case note that since $a = b^\ell$, $\log_b a = \ell$ and so $n^{\log_b a} = n^\ell$.) Now consider each of the three cases in the statement of the theorem, i.e. $a < b^\ell$, $a = b^\ell$ and $a > b^\ell$. In each case we will manipulate the expression on the right-hand side of (3.15), resulting in a simple upper bound for $T(n)$.

Case 1. $a < b^\ell$ (i.e., $\alpha < 1$). In this case we have

$$\begin{aligned} T(n) &= cn^{\log_b a} + dn^\ell \Big(\tfrac{1-\alpha^{\log_b n}}{1-\alpha} \Big) && \text{[by (3.15)]} \\ &\leq cn^{\log_b a} + d\big(\tfrac{1}{1-\alpha}\big)n^\ell && \text{[because } 1-\alpha^{\log_b n} \leq 1] \\ &\leq cn^\ell + \big(\tfrac{d}{1-\alpha}\big)n^\ell && \text{[because } a < b^\ell \Rightarrow \log_b a < \ell] \\ &= \kappa n^\ell \end{aligned}$$

where $\kappa = c + \frac{d}{1-\alpha}$. Since $c, d \geq 0$ and $1 - \alpha > 0$, we have that $\kappa \geq 0$. So, in this case, $T(n) \leq \kappa n^\ell$ for some constant $\kappa \geq 0$ that depends on a, b, c, d and ℓ.

Case 2. $a = b^\ell$ (i.e., $\alpha = 1$). In this case we have

$$\begin{aligned} T(n) &= cn^\ell + dn^\ell \log_b n && \text{[by (3.15)]} \\ &\leq cn^\ell \log_b n + dn^\ell \log_b n && \text{[because } n \geq b \Rightarrow \log_b n \geq 1] \\ &= \kappa n^\ell \log_b n \end{aligned}$$

where $\kappa = c + d$. Since $c, d \geq 0$, we have that $\kappa \geq 0$. So, in this case, $T(n) \leq \kappa n^\ell \log_b n$ for some constant $\kappa \geq 0$ that depends on c and d.

Case 3. $a > b^\ell$ (i.e., $\alpha > 1$). In this case we have

$$\begin{aligned}
\frac{1-\alpha^{\log_b n}}{1-\alpha} &= \frac{\alpha^{\log_b n}-1}{\alpha-1} \\
&= \beta(n^{\log_b \alpha}-1) && [\text{where } \beta = 1/(\alpha-1); \text{ recall that } x^{\log_b y} = y^{\log_b x}] \\
&= \beta(n^{\log_b (a/b^\ell)}-1) && [\text{recall that } \alpha = a/b^\ell] \\
&= \beta(n^{\log_b a - \log_b b^\ell}-1) \\
&= \beta(n^{\log_b a - \ell}-1) \\
&\le \beta n^{\log_b a - \ell} && [\text{since } \beta > 0]
\end{aligned}$$

Therefore, by (3.15),

$$\begin{aligned}
T(n) &\le cn^{\log_b a} + dn^\ell \beta n^{\log_b a - \ell} \\
&= cn^{\log_b a} + d\beta n^{\log_b a} \\
&= \kappa n^{\log_b a}
\end{aligned}$$

where $\kappa = c+d\beta$. Since $c, d \ge 0$ and $\beta > 0$, we have that $\kappa \ge 0$. So, in this case, $T(n) \le \kappa n^{\log_b a}$ for some constant $\kappa \ge 0$ that depends on a, b, c, d and ℓ. □

We can prove that the same result actually holds for *all* $n \ge b$, not only for values of n that are powers of b.

Theorem 3.9 *There is a real constant $\lambda \ge 0$ (that depends on a, b, c, d and ℓ) so that, for all integers $n \ge b$, the function $T(n)$ defined by recurrence (3.12) satisfies the following inequality:*

$$T(n) \le \begin{cases} \lambda n^\ell, & \text{if } a < b^\ell \\ \lambda n^\ell \log_b n, & \text{if } a = b^\ell \\ \lambda n^{\log_b a}, & \text{if } a > b^\ell \end{cases}$$

This can be proved in a way similar to that we used in Section 3.2.1 to extend the applicability of Lemma 3.4 to all values of n, not only powers of 2. We leave the proof of this theorem as an exercise (see Exercise 10).

Theorem 3.9 gives simple closed-form formulas that bound from above the function defined by the recurrence (3.12). How tight are these bounds? It turns out it that they are quite tight, because we can also establish corresponding lower bounds. In particular, the following theorem holds:

Theorem 3.10 *There is a real constant $\lambda' > 0$ (that depends on a, b, c, d and ℓ) so that, for all integers $n \ge b$, the function $T(n)$ defined by recurrence (3.12) satisfies the following inequality:*

$$T(n) \ge \begin{cases} \lambda' n^\ell, & \text{if } a < b^\ell \text{ and } d > 0 \\ \lambda' n^\ell \log_b n, & \text{if } a = b^\ell \text{ and } d > 0 \\ \lambda' n^{\log_b a}, & \text{if } a > b^\ell \text{ or } d = 0 \end{cases}$$

Notice that in the upper bound (Theorem 3.9) we require the constant λ to be nonnegative, while here we require λ' to be strictly positive. Otherwise, Theorem 3.10 would be trivial by taking $\lambda' = 0$. Also note the additional constraints regarding the value of d (we now require $d > 0$ in the first two cases). This is related to the requirement that λ' be positive. The proof of this theorem is similar to the proof of the upper bound and is left as an exercise (see Exercise 11).

Other divide-and-conquer recurrences

There are divide-and-conquer algorithms whose time complexity can't be described as an instance of recurrence (3.12). For example, in some algorithms the factor $b > 1$ by which we reduce the size of the problem in each recursive call is not an integer — e.g., it could be $3/2$. (Our proof crucially depended on the fact that b is an integer since in Theorem 3.8 we assumed that powers of b are positive integers, so that they are legitimate arguments of T.)

Another reason why recurrence (3.12) is not completely general is that there are divide-and-conquer algorithms in which the smaller subproblems are not of "roughly" the same size. For example, an input of size n can be divided up into two smaller inputs, of sizes $\lceil n/5 \rceil$ and $\lfloor 4n/5 \rfloor$.

Although (3.12) does not apply to all divide-and-conquer algorithms, it applies to many, and so Theorems 3.9 and 3.10 are important tools for analysing the time complexity of algorithms. In addition, the techniques discussed in this chapter can be used to solve other types of divide-and-conquer recurrences. Exercises 8 and 9 provide examples of such recurrences.

3.3 Another recurrence

Consider the recurrence

$$T(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T(\lfloor \frac{n}{2} \rfloor) + \lfloor \log_2 n \rfloor, & \text{if } n > 1 \end{cases} \tag{3.16}$$

This does not quite fit the general divide-and-conquer recurrence (3.12) because the term $\lfloor \log_2 n \rfloor$ in the induction step is not a polynomial.

Actually, we can apply the results of the previous section to obtain an upper (and lower) bound for the function $T(n)$ defined by (3.16). To see how, consider the recurrence

$$T'(n) = \begin{cases} 1, & \text{if } n = 1 \\ 2T'(\lfloor \frac{n}{2} \rfloor) + \sqrt{n}, & \text{if } n > 1 \end{cases} \tag{3.17}$$

This is a special case of recurrence (3.12), where $a = b = 2$, $c = d = 1$, and $\ell = 1/2$ (recall that ℓ was a nonnegative *real*, and so it can take on the value $1/2$). So, in this case $a > b^\ell$ and by Theorem 3.9 we conclude that there is a constant $\lambda \geq 0$ so that $T'(n) \leq \lambda n$, for all $n \geq 1$. Using the definitions of $T(n)$ and $T'(n)$, it is easy to show (by complete induction) that for all $n \geq 1$, $T(n) \leq T'(n)$. Therefore, we get that $T(n) \leq \lambda n$, for all $n \geq 1$.

With this background, let us now try to prove the same bound for $T(n)$ from "first principles", i.e., without explicitly relying on results of the previous section. In other words, we will prove that there is a constant $c \geq 0$ such that $T(n) \leq cn$, for all integers $n \geq 1$. It is natural to try proving this by induction on n. Let $P(n)$ be the predicate

$$P(n): \qquad T(n) \leq cn$$

for some, as yet unspecified, constant $c \geq 0$. We will attempt to prove that $P(n)$ holds for every integer $n \geq 1$ using complete induction. In the course of this proof we will try to determine the unspecified nonnegative constant c.

Basis: $n = 1$. We want to prove that $T(1) \leq c\,1$. By (3.16), $T(1) = 1$, so the basis holds as long as $c \geq 1$.

Induction Step: Let i be an arbitrary integer such that $i > 1$. Assume that $P(j)$ holds for all integers j such that $1 \leq j < i$; i.e., $T(j) \leq cj$, for all $1 \leq j < i$. We must prove that $P(i)$ holds as well, i.e., that $T(i) \leq ci$. We have:

$$\begin{aligned}
T(i) &= 2T(\lfloor i/2 \rfloor) + \lfloor \log_2 i \rfloor && \text{[by (3.16), since } i > 1\text{]}\\
&\leq 2\big(c\lfloor i/2 \rfloor\big) + \lfloor \log_2 i \rfloor && \text{[by induction hypothesis; note that } 1 \leq \lfloor i/2 \rfloor < i\text{, since } i > 1\text{]}\\
&\leq 2\big(c(i/2)\big) + \log_2 i && \text{[because } \lfloor i/2 \rfloor \leq i/2 \text{ and } \lfloor \log_2 i \rfloor \leq \log_2 i\text{, by definition of}\\
& && \text{the floor function]}\\
&\leq ci + \log_2 i
\end{aligned}$$

So, to prove the desired inequality $T(i) \leq ci$, it would suffice to show that $ci + \log_2 i \leq ci$. This, however, is obviously false because $\log_2 i > 0$, for all $i > 1$. So, this particular tack gets us nowhere.

Notice, however, that we are not too far off from what we want: We wanted to prove that $T(i) \leq ci$, and in the induction step we were able to show that $T(i) \leq ci + \log_2 i$; the term $\log_2 i$ is small, compared to ci. Of course, this does not mean that we can simply drop it, but this observation raises the possibility of circumventing the difficulty by strengthening the predicate we are trying to prove. (Recall the discussion on page 35 in Chapter 1.) In view of our failed first attempt, a reasonable next step is to consider the following predicate

$$P'(n): \qquad T(n) \leq cn - d\log_2 n$$

for some, as yet unspecified, constants $c, d > 0$. We will now prove that $P'(n)$ holds for all $n \geq 1$ using complete induction, in the course of which we will also determine the unspecified positive constants c and d. Note that if we prove this, it follows immediately that $P(n)$ holds for all n, since $P'(n)$ is stronger than $P(n)$.

Basis: $n = 1$. We want to prove that $T(1) \leq c\,1 - d\log_2 1 = c$. By (3.16), $T(1) = 1$, so the basis holds as long as

$$c \geq 1 \tag{3.18}$$

Induction Step: Let i be an arbitrary natural number such that $i > 1$. Assume that $P'(j)$ holds for all integers j such that $1 \leq j < i$; i.e., $T(j) \leq cj - d\log_2 j$, for all $1 \leq j < i$. We must prove that $P'(i)$ holds as well, i.e., that $T(i) \leq ci - d\log_2 i$. Indeed, we have:

$$\begin{aligned}
T(i) &= 2T(\lfloor \tfrac{i}{2} \rfloor) + \lfloor \log_2 i \rfloor && \text{[by (3.16), since } i > 1\text{]}\\
&\leq 2(c\lfloor \tfrac{i}{2} \rfloor - d\log_2 \lfloor \tfrac{i}{2} \rfloor) + \lfloor \log_2 i \rfloor && \text{[by induction hypothesis; note that}\\
& && 1 \leq \lfloor \tfrac{i}{2} \rfloor < i, \text{ since } i > 1\text{]}\\
&\leq 2c\tfrac{i}{2} - 2d\log_2 \tfrac{i-1}{2} + \log_2 i && \text{[because } \tfrac{i-1}{2} \leq \lfloor \tfrac{i}{2} \rfloor \leq \tfrac{i}{2}\text{, by definition of}\\
& && \text{the floor function]}\\
&= ci - 2d\big(\log_2(i-1) - \log_2 2\big) + \log_2 i\\
&= ci - 2d\log_2(i-1) + 2d + \log_2 i\\
&\leq ci - 2d\big((\log_2 i) - 1\big) + 2d + \log_2 i && \text{[because } i > 1\text{, so } \log_2(i-1) \geq (\log_2 i) - 1\text{]}\\
&= ci - 2d\log_2 i + 4d + \log_2 i
\end{aligned}$$

Therefore, for the desired inequality, $T(i) \leq ci - d\log_2 i$, to hold, it suffices to show that $ci - 2d\log_2 i + 4d + \log_2 i \leq ci - d\log_2 i$. For $\log_2 i > 4$, i.e., $i > 16$, this is equivalent to

$$d \geq \frac{\log_2 i}{(\log_2 i) - 4} \tag{3.19}$$

It is easy to see that the right-hand side of (3.19) decreases as i increases, and since we must have $i > 16$, the right-hand side of (3.19) attains its maximum value at $i = 17$. Using a calculator we can compute this value to be $\frac{\log_2 17}{(\log_2 17)-4} = 46.7337\ldots$ To satisfy (3.19) we can therefore pick $d = 47$.

This choice of d ensures that the induction step of the proof can be carried out *provided* $i \geq 17$. We do not know that it works for values of i in the range $1 \leq i \leq 16$. Therefore, we must make sure that, for these values of i too, $T(i) \leq ci - 47\log_2 i$ (note that 47 is the value we have chosen for d). This means that c should be chosen so that

$$c \geq \frac{T(i) + 47\log_2 i}{i} \qquad \text{for all } i \text{ such that } 1 \leq i \leq 16 \tag{3.20}$$

To find a value of c that satisfies (3.20), we compute the value of $T(i)$ for $1 \leq i \leq 16$ using the definition (3.16). Using a simple program or calculator, we can then compute the value of $\big(T(i) + 47\log_2 i\big)/i$, for every i such that $1 \leq i \leq 16$. If we do this, we will find that the maximum value of this quantity in this interval is $25.4977\ldots$ (attained when $i = 3$). To satisfy constraint (3.20) we can then choose any value of c greater than that — for example, $c = 26$. Note that this choice automatically satisfies the other constraint we need on c, namely (3.18).

We have therefore shown that, for $c = 26$ and $d = 47$, $T(n) \leq cn - d\log_2 n$, for all $n \geq 1$. Hence there is a constant $c \geq 0$ (namely, $c = 26$) so that for all $n \geq 1$, $T(n) \leq cn$, as wanted.

Exercises

1. Let $f : \mathbb{N} \to \mathbb{N}$ be defined by

$$f(n) = \begin{cases} 3, & \text{if } n = 0 \\ 6, & \text{if } n = 1 \\ 5f(n-1) - 6f(n-2) + 2, & \text{if } n > 1 \end{cases}$$

Prove that $f(n) = 1 + 2^n + 3^n$ for all $n \geq 0$.

2. Let $f : \mathbb{N} \to \mathbb{N}$ be the function defined recursively as follows.

$$f(n) = \begin{cases} 1, & \text{if } n = 0 \\ 4, & \text{if } n = 1 \\ f(n-1) + 12f(n-2), & \text{if } n > 1 \end{cases}$$

Prove that for every $n \in \mathbb{N}$, $f(n) = 4^n$.

3. Let $f : \mathbb{N} \to \mathbb{N}$ be the function defined by

$$f(n) = \begin{cases} 1, & \text{if } n = 0 \\ 2, & \text{if } n = 1 \\ 4f(n-2) + 2^n, & \text{if } n > 1 \end{cases}$$

Prove that for every integer $n \geq 3$, $f(n) \leq 3 \cdot n \cdot 2^{n-2}$.

4. Let $f : \mathbb{N} \to \mathbb{N}$ be defined as follows:

$$f(n) = \begin{cases} 6, & \text{if } n = 0 \\ 5f(n-1) + 3^{n-1}, & \text{if } n > 0 \end{cases}$$

Use induction to prove that for some positive constant c and for all $n \in \mathbb{N}$, $f(n) \leq c \cdot 5^n$.
Hint: Prove first the stronger statement that, for all $n \in \mathbb{N}$, $f(n) \leq 7 \cdot 5^n - 3^n$.

5. Consider the following recurrence defining a function $f : \mathbb{N} \to \mathbb{N}$:

$$f(n) = \begin{cases} 10, & \text{if } n = 0 \\ 100\, f(\lfloor n/5 \rfloor) + n^2 3^n & \text{if } n > 0 \end{cases}$$

Prove that there is a constant $c > 0$ so that for all sufficiently large integers n $f(n) \leq cn^2 3^n$. ("All sufficiently large integers n" means "all integers larger than or equal to some constant n_0", which you should determine.)

6. Consider the following recurrence defining a function $f : \mathbb{N} \to \mathbb{N}$:

$$f(n) = \begin{cases} 4, & \text{if } n = 0 \\ 7f(\lfloor \frac{n}{3} \rfloor) + 5n^2, & \text{if } n > 0 \end{cases}$$

Without using Theorem 3.9, prove that there is a constant $c > 0$ such that for all sufficiently large integers n, $f(n) \leq cn^2$.

7. The ***height*** of a binary tree is the number of edges on the longest path from the root to a leaf. (Thus, the height of a binary tree that consists of a single node, its root, is zero.) A ***height-balanced tree*** is a binary tree where every node x satisfies the following property: If x has only one child, then the subtree rooted at that child has height 0; if x has two children, then the heights of the subtrees rooted at those children differ by at most one. Height-balanced trees play an important role in data structures.
Define $g(h)$ to be the smallest possible number of nodes in a height-balanced tree of height h.

(a) Prove that g is strictly increasing; that is for $h, h' \in \mathbb{N}$, if $h < h'$ then $g(h) < g(h')$.

(b) Prove that

$$g(h) = \begin{cases} 1, & \text{if } h = 0 \\ 2, & \text{if } h = 1 \\ g(h-1) + g(h-2) + 1, & \text{if } h \geq 2 \end{cases}$$

(**Hint:** Use part (a).)

(c) Prove that $g(h) \geq 1.6^h$ for every $h \geq 0$. (**Hint:** Use part (b).)

Note: This result implies that the height of a height-balanced tree is at most logarithmic in its size, an important fact for algorithms operating on such trees.

(d) It turns out that 1.6^h is not quite the best exponential lower bound for $g(h)$. What is the largest real number c such that for every $h \geq 0$, $g(h) \geq c^h$? Justify your answer.

8. Consider the following recurrence:

$$T(n) = \begin{cases} c, & \text{if } 1 \leq n \leq 3 \\ T(\lceil \frac{3}{4}n \rceil) + T(\lceil \frac{1}{5}n \rceil) + dn, & \text{if } n > 3 \end{cases}$$

Prove that there are constants $\lambda', \lambda > 0$ so that for all sufficiently large values of n, $\lambda' n \leq T(n) \leq \lambda n$.

Note: This recurrence actually comes up in the time complexity analysis of a divide-and-conquer algorithm for finding the median of a list of elements. The algorithm divides up the input into two parts one of which is about three-quarters the original size and the other is about one-fifth the original size. Note that the sum of the sizes of the two subproblems is strictly less than the size of the original problem. This is crucial in yielding a linear solution.

9. Following is a general form of a recurrence describing the time complexity of some divide-and-conquer algorithms that divide up the input into inputs of unequal size.

$$T(n) = \begin{cases} 1, & \text{if } 0 \le n \le n_0 \\ \sum_{i=1}^{k} a_i T(\lfloor n/b_i \rfloor) + c \cdot n^d, & \text{if } n > n_0. \end{cases}$$

In this recurrence $n_0 \ge 0$ is a constant representing the number of base cases and $k \ge 1$ is a constant representing the number of different sizes of smaller inputs into which the given input is divided up. For each $i = 1, 2, \ldots, k$, the ith input size is shrunk by a factor of b_i where $b_i > 1$, and the divide-and-conquer algorithm requires solving a_i instances of the ith size where $a_i \ge 1$. (Note that the recurrence of Exercise 8 is a special case of this recurrence where $n_0 = 3$, $k = 2$, $a_1 = a_2 = 1$, $b_1 = 4/3$ and $b_2 = 5$.)

Prove the following generalisation of Exercise 8: If $\sum_{i=1}^{k} a_i b_i^d < 1$, then there are constants $\lambda', \lambda > 0$ so that for all sufficiently large values of n, $\lambda' n \le T(n) \le \lambda n$.

10. Prove Theorem 3.9. (Hint: Prove that the function $T(n)$ defined by (3.12) is nondecreasing, and use Theorem 3.8.)

11. Prove Theorem 3.10.

Chapter 4

SETS DEFINED BY INDUCTION AND STRUCTURAL INDUCTION

4.1 Defining sets recursively

Induction can also be used to define sets. The idea is to define a set of objects as follows:

(i) We define the "smallest" or "simplest" object (or objects) in the set.

(ii) We define the ways in which "larger" or "more complex" objects in the set can be constructed out of "smaller" or "simpler" objects in the set.

Clause (i) of such a recursive definition is called the basis, and clause (ii) is called the induction step.

Example 4.1 Suppose we want to define the set of "well-formed", fully parenthesised algebraic expressions involving a certain set of variables and a certain set of operators. To be more concrete, let us say that the variables in question are x, y and z, and the operators are $+$, $-$, $\times$ and $\div$. We want to define an infinite set $\mathcal{E}$ of strings over alphabet $\Sigma = \{x, y, z, +, -, \times, \div, (,)\}$. Not every string in Σ^* should be in this set, of course. A well-formed expression, like $((x + y) + (z \div (y \times z)))$ should be in the set, while ill-formed expressions, such as $(x + y$ and $x + (\div y)$ should not be. We can achieve this by using the following recursive definition.

Definition 4.1 *Let $\mathcal{E}$ be the smallest set such that*

BASIS: $x, y, z \in \mathcal{E}$.

INDUCTION STEP: *If $e_1, e_2 \in \mathcal{E}$ then the following four expressions are also in $\mathcal{E}$: $(e_1 + e_2)$, $(e_1 - e_2)$, $(e_1 \times e_2)$, and $(e_1 \div e_2)$.*

Note the requirement that $\mathcal{E}$ be the *smallest* set that satisfies the conditions spelled out in the basis and induction step of the definition. We should clarify that when we say that a set X is the smallest set to satisfy a property P, we mean that X is a subset of every set that satisfies

P — or, equivalently, that X is the intersection of all sets that satisfy P. Why do we need to say this about $\mathcal{E}$? Without this proviso Definition 4.1 would not specify a unique set, but an infinite number of sets, all of which satisfy the basis and induction step clauses and are supersets of the set $\mathcal{E}$ we are interested in.

To see this consider the set $\mathcal{F}$ defined as above except that the basis case also stipulates that variable w is in $\mathcal{F}$. Consider as well the set $\mathcal{G}$ defined similarly except that the induction step also stipulates that if e is $\mathcal{G}$ then so is $-e$. Both $\mathcal{F}$ and $\mathcal{G}$ (and infinitely many other sets) satisfy the basis and induction step clauses of the recursive Definition 4.1. By requiring $\mathcal{E}$ to be the *smallest* set that satisfies these properties, we avoid this indeterminacy. Another way of achieving the same effect is to add a third clause to the recursive definition stating "Nothing else is in the defined set, except as obtained by a finite number of applications of the basis and induction step". **End of Example 4.1**

We will now state the general principle that justifies recursive definitions of sets, such as Definition 4.1. First we need some terminology. Let $\mathcal{S}$ be a set. A ***k-ary operator*** on $\mathcal{S}$ is simply a function $f : \mathcal{S}^k \to \mathcal{S}$. ($\mathcal{S}^k$ is the k-fold Cartesian product of $\mathcal{S}$.) We say that a subset A of $\mathcal{S}$ is ***closed under f*** if, for all $a_1, a_2, \ldots, a_k \in A$, $f(a_1, a_2, \ldots, a_k) \in A$ as well. For example, addition and subtraction are binary operators on the set of integers $\mathbb{Z}$. The subset $\mathbb{N}$ of $\mathbb{Z}$ is closed under addition but is not closed under subtraction. We have:

Theorem 4.2 *Let $\mathcal{S}$ be a set, B be a subset of $\mathcal{S}$, m be a positive integer, and $f_1, f_2, \ldots, f_m$ be operators on $\mathcal{S}$ of arity $k_1, k_2, \ldots, k_m$, respectively. Then, there is a unique subset S of $\mathcal{S}$ such that*

(a) S contains B;

(b) S is closed under $f_1, f_2, \ldots, f_m$; and

(c) any subset of $\mathcal{S}$ that contains B and is closed under $f_1, f_2, \ldots, f_m$ contains S.

This theorem can be proved without using induction. We leave the proof as an exercise, with the hint that the set S which the theorem asserts that exists is the intersection of all subsets of $\mathcal{S}$ that contain B and are closed under $f_1, f_2, \ldots, f_m$.

As a result of this theorem we know that Definition 4.1 (and others like it) is proper. To see why, let us cast that definition in the terms of Theorem 4.2. The set $\mathcal{S}$ is the set of all strings over the alphabet $\Sigma = \{x, y, z, (,), +, -, \times, \div\}$. The set B is $\{x, y, z\}$. The integer m is 4. There are four binary operators: *Plus*, *Minus*, *Times* and *Div*, where $\mathit{Plus}(e_1, e_2) = (e_1 + e_2)$, and similarly for the other three operators.[1] Definition 4.1 defines $\mathcal{E}$ to be the smallest subset of $\mathcal{S}$ that contains $\{x, y, z\}$ and is closed under *Plus*, *Minus*, *Times* and *Div*. Theorem 4.2 assures us that this set exists.

[1] Note that $\mathit{Plus}(e_1, e_2)$ is *not* the addition operator on integers. It takes as arguments two *strings*, e_1 and e_2, and produces as output a *string*, by concatenating a left parenthesis, followed by its first argument e_1, followed by the symbol +, followed by its second argument e_2, followed by a right parenthesis.

The connection of this style of definition to induction is due to a different characterisation of the "smallest subset of $\mathcal{S}$ that contains B and is closed under a set of operators". The idea is as follows. Suppose $\mathcal{S}$ is a set, and $f_1, f_2, \dots, f_m$ are operators on $\mathcal{S}$. Let B be a subset of $\mathcal{S}$ and define an infinite sequence of sets $S_0, S_1, S_2, \dots$ as follows:

$$\begin{aligned}
S_0 &= B \\
S_1 &= S_0 \cup \text{the set obtained by applying } f_1, \dots, f_m \text{ to elements of } S_0 \\
S_2 &= S_1 \cup \text{the set obtained by applying } f_1, \dots, f_m \text{ to elements of } S_1 \\
S_3 &= S_2 \cup \text{the set obtained by applying } f_1, \dots, f_m \text{ to elements of } S_2 \\
&\vdots \\
S_i &= S_{i-1} \cup \text{the set obtained by applying } f_1, \dots, f_m \text{ to elements of } S_{i-1} \\
&\vdots
\end{aligned}$$

The infinite union of all the S_i's

$$S_0 \cup S_1 \cup S_2 \cup \cdots \cup S_i \cup \cdots$$

is the smallest subset of $\mathcal{S}$ that contains B and is closed under $f_1, f_2, \dots, f_m$. More precisely,

Principle of set definition by recursion: *Let $\mathcal{S}$ be a set, B be a subset of $\mathcal{S}$, m be a positive integer, and $f_1, f_2, \dots, f_m$ be operators on $\mathcal{S}$ of arity $k_1, k_2, \dots, k_m$, respectively. Let*

$$S_i = \begin{cases} B, & \text{if } i = 0 \\ S_{i-1} \cup \left(\bigcup_{j=1}^{m} \{ f_j(a_1, a_2, \dots, a_{k_j}) : a_1, a_2, \dots, a_{k_j} \in S_{i-1} \} \right), & \text{if } i > 0 \end{cases}$$

Then $\cup_{i \in \mathbb{N}} S_i$ is the smallest subset of $\mathcal{S}$ that contains B and is closed under $f_1, f_2, \dots, f_m$.

This principle can be proved by using the principle of induction (or of well-ordering).

4.2 Structural induction

Once we have provided a recursive definition of a set it is natural to prove properties of its elements by using induction. A variant of induction, called ***structural induction***, is especially well-suited to this purpose. Suppose we have defined a set X using induction, and we now wish to prove that every element of X has a certain property P. A proof by structural induction proceeds in two steps.

BASIS: We prove that every "smallest" or "simplest" element of X (i.e., every element that is in the set by virtue of the basis of the recursive definition) satisfies P.

INDUCTION STEP: We also prove that each of the (finitely many) ways of constructing "larger" or "more complex" elements out of "smaller" or "simpler" ones (i.e., each of the constructions in the induction step of the recursive definition) *preserves* property P. That is, if all the "smaller" elements that are being combined by the construction have property P, then the "larger" element that results from the construction will necessarily have property P, too.

If we prove these two facts, then structural induction allows us to conclude that all elements in X have property P. We now illustrate this proof technique and explain why it is valid.

Example 4.2 Consider the set $\mathcal{E}$ of well-formed algebraic expressions defined recursively in Definition 4.1. By following the rules of that definition we can form the expression

$$(((x + x) - (x \div y)) + z)$$

In this expression the variable x appears three times, while y and z appear once each. Similarly, the operator $+$ appears twice, while $-$ and $\div$ appear once each. The total number of variable occurrences is 5 and the total number of operator occurrences is 4. That is, in this expression the number of variable occurrences is one more than the number of operator occurrences. This is no accident: the same relationship between the number of variable occurrences and the number of operator occurrences holds in *any* expression that belongs to $\mathcal{E}$. Let us prove this fact using structural induction. To simplify the presentation we introduce the following notation: If e is a string, $\mathbf{vr}(e)$ and $\mathbf{op}(e)$ denote the number of variable and operator occurrences, respectively, in e.

Proposition 4.3 *For any $e \in \mathcal{E}$, $\mathbf{vr}(e) = \mathbf{op}(e) + 1$.*

PROOF. Let $P(e)$ be the following predicate (of strings):

$$P(e): \qquad \mathbf{vr}(e) = \mathbf{op}(e) + 1$$

We prove that $P(e)$ holds for each $e \in \mathcal{E}$ by using structural induction.

BASIS: There are three cases: $e = x$, $e = y$ and $e = z$. In each case, $\mathbf{vr}(e) = 1$ and $\mathbf{op}(e) = 0$, so $P(e)$ holds for the basis.

INDUCTION STEP: Let e_1, e_2 be arbitrary elements of $\mathcal{E}$ such that $P(e_1)$ and $P(e_2)$ hold; i.e., $\mathbf{vr}(e_1) = \mathbf{op}(e_1) + 1$ and $\mathbf{vr}(e_2) = \mathbf{op}(e_2) + 1$. We will prove that $P(e)$ also holds for any $e \in \mathcal{E}$ that can be constructed from e_1 and e_2. There are four cases, depending on how e is constructed from e_1 and e_2: $e = (e_1 + e_2)$, $e = (e_1 - e_2)$, $e = (e_1 \times e_2)$ and $e = (e_1 \div e_2)$. In each case we have

$$\mathbf{vr}(e) = \mathbf{vr}(e_1) + \mathbf{vr}(e_2) \tag{4.1}$$
$$\mathbf{op}(e) = \mathbf{op}(e_1) + \mathbf{op}(e_2) + 1 \tag{4.2}$$

Thus,

$$\begin{aligned}
\mathbf{vr}(e) &= \mathbf{vr}(e_1) + \mathbf{vr}(e_2) && \text{[by (4.1)]}\\
&= (\mathbf{op}(e_1) + 1) + (\mathbf{op}(e_2) + 1) && \text{[by induction hypothesis]}\\
&= \big(\mathbf{op}(e_1) + \mathbf{op}(e_2)\big) + 2 &&\\
&= \big(\mathbf{op}(e) - 1\big) + 2 && \text{[by (4.2)]}\\
&= \mathbf{op}(e) + 1
\end{aligned}$$

as wanted. □

A mistake that is sometimes committed in structural induction proofs such as the one just presented, is that the form of the expressions being considered in the induction step is not the most general possible. For instance, instead of considering expressions of the form $(e_1 \odot e_2)$, where $e_1, e_2 \in \mathcal{E}$ and $\odot$ is one of the symbols $+, -, \times, \div$, one might (erroneously) consider only expressions of the form $(e_1 \odot x)$ and $(x \odot e_2)$. This is not sufficient, because in this case the induction step covers only a limited number of expressions, not the entire set $\mathcal{E}$.

Why does a structural induction proof like the one of Proposition 4.3 prove that every $e \in \mathcal{E}$ satisfies the property $P(e)$ in question? To understand this, recall the alternative characterisation of a recursively defined set that is given by the principle of set definition by recursion. According to that characterisation, the set of algebraic expressions $\mathcal{E}$ is equal to the union of the infinitely many sets in the sequence $\langle \mathcal{E}_0, \mathcal{E}_1, \mathcal{E}_2, \ldots \rangle$ where $\mathcal{E}_0$ is the set of simplest algebraic expressions (i.e., the set $\{x, y, z\}$) and for each $i > 0$, $\mathcal{E}_i$ is the set obtained by applying the four operators (*Plus*, *Minus*, *Times* and *Div*) to the algebraic expressions in $\mathcal{E}_{i-1}$. It is easy to see that the basis and induction step in the above structural induction proof amount to proving, respectively:

Basis: Every $e \in \mathcal{E}_0$ satisfies $P(e)$.

Induction Step: Let i be an arbitrary natural number. If every $e \in \mathcal{E}_i$ satisfies $P(e)$, then every $e \in \mathcal{E}_{i+1}$ satisfies $P(e)$.

Thus, a proof by structural induction amounts to a proof (by simple induction) of the statement:

$$\text{For all } i \in \mathbb{N}, \text{ every } e \in \mathcal{E}_i \text{ satisfies } P(e). \tag{4.3}$$

Since $\mathcal{E} = \mathcal{E}_0 \cup \mathcal{E}_1 \cup \cdots \cup \mathcal{E}_i \cup \cdots$, a proof of (4.3) is a proof that every $e \in \mathcal{E}$ satisfies $P(e)$ — which is the desired goal. **End of Example 4.2**

structural induction is a type of simple induction

Example 4.3 To prove that an element or a set of elements belongs to a recursively defined set X all we have to do is specify a way of constructing the element(s) in question using the rules in the recursive definition, and then prove that the specified construction yields the desired element(s). But how do we prove the *opposite*, i.e., that an element or set of elements does *not* belong to X? Interestingly, structural induction can be used to this end. The idea is to identify a property P which all the elements of X must have, but which the specified element(s) do not. Structural induction is used to prove that, in fact, all elements of X have property P.

For example, consider the string $e = (x + (\div y))$. We can prove that e does not belong to the set $\mathcal{E}$ of well-formed algebraic expressions, by appealing to Proposition 4.3. Obviously, $\mathbf{vr}(e) = \mathbf{op}(e) = 2$. Thus, $\mathbf{vr}(e) \neq \mathbf{op}(e) + 1$ and, by Proposition 4.3, $e \notin \mathcal{E}$.

Here is another, perhaps more interesting, example in which we show that $\mathcal{E}$ does not contain any of *infinitely* many strings that have a particular form.

Proposition 4.4 *For any natural number $i \neq 0$, the expression consisting of i left parentheses, followed by the variable x, followed by i right parentheses, does not belong to $\mathcal{E}$. That is, for all natural numbers $i \neq 0$,*

$$\underbrace{((\ldots(}_{i \text{ times}} x \underbrace{)\ldots))}_{i \text{ times}} \notin \mathcal{E}$$

Proof. Let $\mathbf{pr}(e)$ be the number of parentheses in string e, and $Q(e)$ be the predicate

$$Q(e): \qquad \mathbf{pr}(e) = 2 \cdot \mathbf{op}(e).$$

We use structural induction to prove that $Q(e)$ holds for every $e \in \mathcal{E}$.

Basis: There are three cases: $e = x$, $e = y$ and $e = z$. In each case, $\mathbf{pr}(e) = \mathbf{op}(e) = 0$, so $Q(e)$ holds for the basis.

Induction Step: Let e_1, e_2 be arbitrary elements of $\mathcal{E}$ such that $Q(e_1)$ and $Q(e_2)$ hold; i.e., $\mathbf{pr}(e_1) = 2 \cdot \mathbf{op}(e_1)$ and $\mathbf{pr}(e_2) = 2 \cdot \mathbf{op}(e_2)$. We will prove that $Q(e)$ also holds for any $e \in \mathcal{E}$ that can be constructed from e_1 and e_2. There are four cases, depending on how e is constructed from e_1 and e_2: $e = (e_1 + e_2)$, $e = (e_1 - e_2)$, $e = (e_1 \times e_2)$ and $e = (e_1 \div e_2)$. In each case we have

$$\mathbf{pr}(e) = \mathbf{pr}(e_1) + \mathbf{pr}(e_2) + 2 \tag{4.4}$$
$$\mathbf{op}(e) = \mathbf{op}(e_1) + \mathbf{op}(e_2) + 1 \tag{4.5}$$

Thus,

$$\begin{aligned}
\mathbf{pr}(e) &= \mathbf{pr}(e_1) + \mathbf{pr}(e_2) + 2 && \text{[by (4.4)]} \\
&= 2 \cdot \mathbf{op}(e_1) + 2 \cdot \mathbf{op}(e_2) + 2 && \text{[by induction hypothesis]} \\
&= 2 \cdot \big(\mathbf{op}(e_1) + \mathbf{op}(e_2) + 1\big) && \\
&= 2 \cdot \mathbf{op}(e) && \text{[by (4.5)]}
\end{aligned}$$

as wanted.

We have therefore proved

$$\text{for any } e \in \mathcal{E},\ \mathbf{pr}(e) = 2 \cdot \mathbf{op}(e). \tag{4.6}$$

Now, let e be any string of the form $\overbrace{((\ldots(}^{i \text{ times}} x \overbrace{)\ldots))}^{i \text{ times}}$, for some natural number $i \neq 0$. We want to prove that $e \notin \mathcal{E}$. Clearly, $\mathbf{pr}(e) = 2 \cdot i$, while $\mathbf{op}(e) = 0$. Since $i \neq 0$, $\mathbf{pr}(e) \neq 2 \cdot \mathbf{op}(e)$. Thus, by (4.6), $e \notin \mathcal{E}$, as wanted. □

End of Example 4.3

4.3 An alternative to structural induction proofs

Instead of using structural induction to prove properties of recursively defined sets it is possible to use complete induction. The idea is to associate with each object in the recursively defined set a measure of its "size". This measure is related to the number of applications of the rules needed to construct the object, and there may be several different natural choices for it. In our example of the set of well-formed algebraic expressions $\mathcal{E}$, natural choices include the length of an expression (as a string), the number of operators in the expression, or the maximum level of nesting of parentheses. Instead of proving that every $e \in \mathcal{E}$ satisfies a predicate $P(e)$ by structural induction, we can accomplish the same thing by using complete induction to prove that for any value n of the measure of "size", every $e \in \mathcal{E}$ of "size" n, satisfies the predicate $P(e)$.

Example 4.4 It is intuitively clear that every $e \in \mathcal{E}$ has an equal number of left and right parentheses. We can prove this by using structural induction. In this example, we use complete induction on the length of e instead, To simplify the presentation we introduce some notation. For any string e, let $\mathbf{lp}(e)$ and $\mathbf{rp}(e)$ denote the number of left and right parentheses (respectively) in e. Recall that $|e|$ denotes the length of string e, i.e., the number of symbols in it.

Proposition 4.5 *For any* $e \in \mathcal{E}$, $\mathbf{lp}(e) = \mathbf{rp}(e)$.

Proof. Let $R(n)$ be the predicate (on natural numbers) defined as

$$R(n): \qquad \text{for any } e \in \mathcal{E} \text{ such that } |e| = n,\ \mathbf{lp}(e) = \mathbf{rp}(e).$$

We use complete induction to prove that $R(n)$ holds for every $n \in \mathbb{N}$.

Case 1. $n \leq 1$. There are only three strings in $\mathcal{E}$ that have length at most one: x, y and z. In each case, the string has equally many left and right parentheses, namely zero. Thus, $R(n)$ holds when $n \leq 1$.

Case 2. Let $i > 1$ be an arbitrary natural number, and suppose that $R(j)$ holds for each integer j such that $0 \leq j < i$. That is, for any such j, and any $e \in \mathcal{E}$ such that $|e| = j$, $\mathbf{lp}(e) = \mathbf{rp}(e)$. We must prove that $R(i)$ holds as well.

Consider an arbitrary $e \in \mathcal{E}$ such that $|e| = i$; We must prove that $\mathbf{lp}(e) = \mathbf{rp}(e)$. Since $|e| > 1$, e does not satisfy the basis case in the definition of $\mathcal{E}$. Therefore, there must exist $e_1, e_2 \in \mathcal{E}$ such that e is of the form $(e_1 \odot e_2)$, where $\odot$ is one of the four operator symbols $+, -, \times, \div$. Since e has strictly more symbols than either e_1 or e_2, $0 \leq |e_1|, |e_2| < i$. Thus, by induction hypothesis, $\mathbf{lp}(e_1) = \mathbf{rp}(e_1)$ and $\mathbf{lp}(e_2) = \mathbf{rp}(e_2)$. We have:

$$\begin{aligned}
\mathbf{lp}(e) &= \mathbf{lp}(e_1) + \mathbf{lp}(e_2) + 1 && [\text{because } e = (e_1 \odot e_2)] \\
&= \mathbf{rp}(e_1) + \mathbf{rp}(e_2) + 1 && [\text{by induction hypothesis}] \\
&= \mathbf{rp}(e) && [\text{because } e = (e_1 \odot e_2)]
\end{aligned}$$

as wanted.

Therefore, $R(n)$ holds for all $n \in \mathbb{N}$. Since every string has a length that is a natural number, the Proposition follows. □

End of Example 4.4

As an exercise, you should prove Proposition 4.5 using structural induction. Whenever we need to prove that all objects of a recursively defined set satisfy some property, structural induction and complete induction are both natural approaches. The choice between the two is strictly a matter of convenience and taste.

4.4 *Elements of recursively defined sets as trees*

Objects that belong to recursively defined sets can be represented as trees, and this representation can sometimes be very useful. A "simplest" object (those defined in the basis clause of the recursive definition) is represented by either an empty tree, or a single node (depending on what most naturally captures the situation). A "compound" object constructed out of "simpler" objects (as described in the induction step clause of the definition) is represented as a tree whose subtrees are precisely the trees representing these "simpler" objects.

Example 4.5 Consider the tree in Figure 4.1. The entire tree, i.e., the tree rooted at the node labeled 'a', corresponds to the algebraic expression $((x+y)+(z \div (y \times z))) \in \mathcal{E}$. Its two subtrees, rooted at nodes 'b' and 'c', correspond to the expressions $(x+y)$ and $(z \div (y \times z))$, out of which the entire expression is constructed.

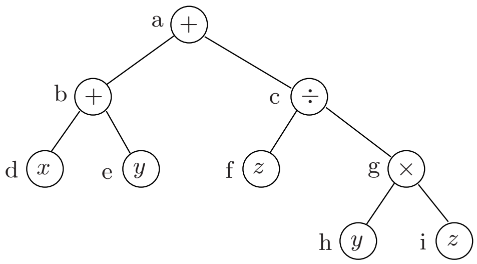


Figure 4.1: Binary tree of an algebraic expression

The subtrees of the tree rooted at 'b' (which corresponds to $(x+y)$), i.e., the trees rooted at nodes 'd' and 'e' correspond to the expressions x and y, out of which $(x+y)$ is made. These two expressions are leaves, because they are among the "simplest" ones, defined in the basis of the recursive definition.

Similarly, the subtrees of the tree rooted at 'c' (which, recall, corresponds to the expression $(z \div (y \times z)))$, i.e., the trees rooted at nodes 'f' and 'g' correspond to the expressions z and $(y \times z)$, out of which $((z \div (y \times z)))$ is made.

We can continue this decomposition of the expression working from to top of the tree downward. The careful reader will surely have noticed the similarity between this process and the recursive structure of trees that we observed in Example 1.13 (see page 42).

The trees that correspond to the algebraic expressions in $\mathcal{E}$ (as defined in Definition 4.1) are *binary* trees, in fact *full binary trees*. Notice that the nodes labeled with an operator (+, −, × or ÷) are the internal nodes of the binary tree, while the nodes labeled with variables are leaves. In view of these observations, the total number of nodes in the tree that represents an expression $e \in \mathcal{E}$ is $\mathbf{vr}(e) + \mathbf{op}(e)$. By Proposition 4.3, the total number of nodes in the (full binary) tree representing e is $2 \cdot \mathbf{op}(e) + 1$, i.e., an odd number. Compare this to Proposition 1.15. **End of Example 4.5**

The tree representation of a recursively defined object gives us a great deal of information about its structure and about how it was constructed. It is helpful both as a visual aid and also as a way of representing the object in a computer's memory. For example, when a compiler or interpreter processes, say, a Java program, it builds a tree that represents the structure of that program. Similarly, symbolic algebra systems such as Maple or Mathematica, store the mathematical expressions they manipulate as trees. The precise definitions of the objects manipulated by such programs (e.g., the syntax of Java programs or of Maple expressions) are actually specified recursively. Many of the algorithms that create and manipulate these trees "mimic" the recursive definitions. Many of the mathematical results underlying the correctness of these algorithms are proved by using various forms of induction. In other words, the techniques and ideas discussed in this section are very relevant in practice.

4.5 Inductive definition of binary trees

In Chapter 1 we reviewed the conventional, graph-based definition of binary trees (see page 32). According to that definition, a binary tree is a tree (i.e., a directed graph which, if nonempty, has a designated node r called the root, so that for every node u there is exactly one path from r to u) every node of which has at most two children and every edge of which has a specified direction ("left" or "right"). This definition is, of course, very useful but it obscures an important feature of binary trees, namely their recursive structure: Each (nonempty) binary tree is made up of smaller binary trees. In this section we provide an alternative characterisation of binary trees that highlights this feature. We will give an inductive definition of the set $\mathcal{T}$ of binary trees; at the same time, for every element T of this set we will define its set of nodes, $\mathbf{nodes}(T)$, its set of edges $\mathbf{edges}(T)$, and if T is nonempty, its root, $\mathbf{root}(T)$.

Definition 4.6 *Let $\mathcal{T}$ be the smallest set such that*

BASIS: *Λ is in $\mathcal{T}$. We use the symbol Λ to represent the empty binary tree. We define $\mathbf{nodes}(\Lambda) = \mathbf{edges}(\Lambda) = \emptyset$; $\mathbf{root}(\Lambda)$ is undefined.*

INDUCTION STEP: *Let T_1 and T_2 be elements of $\mathcal{T}$ such that $\mathbf{nodes}(T_1) \cap \mathbf{nodes}(T_2) = \emptyset$, and $r \notin \mathbf{nodes}(T_1) \cup \mathbf{nodes}(T_2)$. Then the ordered triple $T = (r, T_1, T_2)$ is also in $\mathcal{T}$. Furthermore,* *we define*

$$\begin{aligned}\mathbf{root}(T) &= r,\\ \mathbf{nodes}(T) &= \{r\} \cup \mathbf{nodes}(T_1) \cup \mathbf{nodes}(T_2),\\ \mathbf{edges}(T) &= E \cup \mathbf{edges}(T_1) \cup \mathbf{edges}(T_2),\end{aligned}$$

where E is the set that contains $(r, \mathbf{root}(T_1))$ if $T_1 \neq \Lambda$, $(r, \mathbf{root}(T_2))$ if $T_2 \neq \Lambda$, and nothing else.

For example, using this definition, the binary tree shown diagrammatically in Figure 4.2 is the following element of $\mathcal{T}$:

$$\Big(a, (b, \Lambda, \Lambda), \big(c, \big(d, \Lambda, (e, \Lambda, \Lambda)\big), \Lambda\big)\Big) \tag{4.7}$$

To see this note that, by the basis of Definition 4.6, Λ is an element of $\mathcal{T}$. Also, according to the basis of the definition, $\mathbf{nodes}(\Lambda) = \emptyset$, and so $\mathbf{nodes}(\Lambda) \cap \mathbf{nodes}(\Lambda) = \emptyset$ and $b \notin \mathbf{nodes}(\Lambda) \cup \mathbf{nodes}(\Lambda)$. So, by the induction step of the definition, (b, Λ, Λ) is also an element of $\mathcal{T}$. Similarly, we can argue that (e, Λ, Λ) is an element of $\mathcal{T}$. From the definition of **nodes** in the induction step, we have that $\mathbf{nodes}\big((e, \Lambda, \Lambda)\big) = \{e\}$. Thus, $\mathbf{nodes}(\Lambda) \cap \mathbf{nodes}\big((e, \Lambda, \Lambda)\big) = \emptyset$, and $d \notin \mathbf{nodes}(\Lambda) \cup \mathbf{nodes}\big((e, \Lambda, \Lambda)\big)$. So, by the induction step of the definition, $\big(d, \Lambda, (e, \Lambda, \Lambda)\big)$ is also an element of $\mathcal{T}$. Continuing in the same way, we can see that every triple that appears as a well-formed parenthesised subexpression in (4.7) is an element of $\mathcal{T}$ and corresponds, in a natural way, to a subtree of the binary tree shown in Figure 4.2.

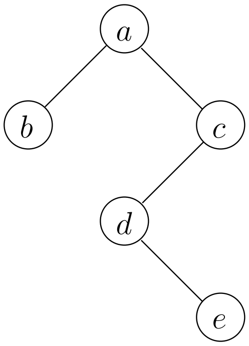


Figure 4.2: A binary tree

Although this example makes plausible the claim that Definition 4.6 is an alternative characterisation of the set of binary trees, it is certainly not a proof of this fact. When we say that Definition 4.6 is "an alternative characterisation of the set of binary trees", we mean that there is a one-to-one correspondence between the objects of the set $\mathcal{T}$ defined in that definition, and the kinds of directed graphs that we have defined as binary trees. We now formalise this correspondence. Let T^* is a conventionally defined binary tree with node set V, edge set E, and root r (if T^* is nonempty), and let T be an element of $\mathcal{T}$. We say that T^* and T ***correspond*** to each other if and only if $\mathbf{nodes}(T) = V$, $\mathbf{edges}(T) = E$, and (if T^* is nonempty) $\mathbf{root}(T) = r$.[2]

So, to say that $\mathcal{T}$ is an alternative characterisation of the set of binary trees means precisely that:

(a) for every binary tree T^* there is a corresponding (in the above sense) $T \in \mathcal{T}$; and

(b) for every $T \in \mathcal{T}$ there is a corresponding binary tree T^*.

We now sketch the arguments that prove these two facts.

To prove (a) we argue by contradiction: Suppose that there is at least one binary tree for which there is no corresponding element in $\mathcal{T}$. Thus, by the well-ordering principle, there is such a binary tree, say T^*, with the *minimum possible number of nodes*. T^* is not the empty binary tree because there *is* an element of $\mathcal{T}$, namely Λ, that corresponds to the empty binary tree. Since T^* is nonempty, it has a root, say r. We will now define the left and right subtrees of T^*, denoted T_1^* and T_2^*.

If T^* has no edge (r, u) labeled "left", then T_1^* is the empty binary tree. If T^* has an edge (r, u) labeled "left", the T_1^* is defined as follows: Let V_1 be the subset of nodes of T^* that includes every node v of T^* such that there is a (possibly empty) path from u to v; and let E_1 be the subset of edges of T^* that connect nodes in V_1. It is easy to verify that in the graph (V_1, E_1) there is one and only one path from u to every node in V_1. So we let T_1^* be the binary tree with node set is V_1, edge set is E_1, and root u. In a similar way we define the right subtree T_2^* of T^*.

Since each of T_1^* and T_2^* has fewer nodes than T^*, it follows from the definition of T^* that there are elements T_1 and T_2 in $\mathcal{T}$ that correspond to T_1^* and T_2^*, respectively. By the induction step of Definition 4.6, $T = (r, T_1, T_2)$ is in $\mathcal{T}$. It is easy to check that T corresponds to T^*. This contradicts the definition of T^*. So, for every binary tree T^* there is a corresponding element of $\mathcal{T}$.

To prove (b) we proceed as follows. We first prove the following lemma.

Lemma 4.7 *For every* $T = (r, T_1, T_2) \in \mathcal{T}$*, there is no* $(u, v) \in \mathbf{edges}(T)$ *such that* $u \in \mathbf{nodes}(T_1)$ *and* $v \in \mathbf{nodes}(T_2)$ *or vice-versa.*

This can be proved easily by structural induction based on Definition 4.6. The proof, which is omitted, makes use of the requirements, in the induction step of that definition, that $\mathbf{nodes}(T_1) \cap \mathbf{nodes}(T_2) = \emptyset$ and $r \notin \mathbf{nodes}(T_1) \cup \mathbf{nodes}(T_2)$.

[2] Our notion of correspondence between binary trees and elements of $\mathcal{T}$ makes no reference to the direction of edges. We could fix this by defining the direction of every edge in $\mathbf{edges}(T)$, using the recursive definition of $\mathcal{T}$. This would make the subsequent arguments that establish the correspondance between binary trees and elements of $\mathcal{T}$ more complete but more cumbersome without illuminating any important points. For this reason we opt for the simpler notion of correspondence presented here.

Next we use structural induction to prove that the predicate

$$P(T): \qquad \text{there is a binary tree that corresponds to } T$$

holds for every $T \in \mathcal{T}$. We sketch this proof. The basis is obvious (the empty binary tree corresponds to Λ). For the induction step, assume that T_1^*, T_2^* are binary trees that correspond to $T_1, T_2 \in \mathcal{T}$ respectively. We must prove that if $\mathbf{nodes}(T_1) \cap \mathbf{nodes}(T_2) = \emptyset$ and $r \notin \mathbf{nodes}(T_1) \cup \mathbf{nodes}(T_2)$, then there is a binary tree T^* that corresponds to $T = (r, T_1, T_2)$.

Let T^* be the graph with node set $\mathbf{nodes}(T)$ and edge set $\mathbf{edges}(T)$, and designate r as its root. We will prove shortly that T^* is a binary tree. It is then straightforward to check that T^* corresponds to T, and so $P(T)$ holds and (b) is established.

To prove that T^* is a binary tree it suffices to show that for each $u \in \mathbf{nodes}(T)$, there is a unique path from r to u in T^*. The existence of such a path is easy to show: If $u = r$, the empty path will do. If $u \in \mathbf{nodes}(T_1)$ then, by the induction hypothesis, T_1^* is a binary tree that corresponds to T_1. So, there is a path from $\mathbf{root}(T_1)$ to u that uses edges of T_1^*. Therefore, there is a path in T^* that goes from r to u, which first uses the edge $(r, \mathbf{root}(T_1))$ and then uses edges of T_1^* to go from $\mathbf{root}(T_1)$ to u. Similarly if $u \in \mathbf{nodes}(T_2)$. The uniqueness of such a path can be shown as follows: If $u = r$ we note that there is no edge in $\mathbf{edges}(T)$ from any node to r, and so the empty path is the only path from r to itself. If $u \in \mathbf{nodes}(T_1)$, by Lemma 4.7, any path from r to u must first use the edge $(r, \mathbf{root}(T_1))$ and cannot use any edge in $\mathbf{edges}(T_2)$. Therefore, if there are multiple paths from r to u in T^*, there must be multiple paths from $\mathbf{root}(T_1)$ to u in T_1^* which contradicts the assumption that T_1^* is a binary tree. Similarly if $u \in \mathbf{nodes}(T_2)$.

This completes the justification of the claim that Definition 4.6 is an alternative characterisation of the set of binary trees. As mentioned at the start of this section, this characterisation lays bare the recursive structure of binary trees. It also opens the door to convenient recursive definitions and structural induction proofs of various properties of binary trees. We illustrate these in the following example.

Example 4.6 First we give a recursive definition of the height of a binary tree.

Definition 4.8 *The height of any $T \in \mathcal{T}$, denoted $h(T)$, is defined as follows:*
BASIS: *If $T = \Lambda$ then $h(T) = -1$.*
INDUCTION STEP: *If $T = (r, T_1, T_2)$ then $h(T) = \max\big(h(T_1), h(T_2)\big) + 1$.*

We can now use structural induction to prove the following important fact that relates the number of nodes of a binary tree to its height:

Proposition 4.9 *For any binary tree T, $|\mathbf{nodes}(T)| \leq 2^{h(T)+1} - 1$.*

PROOF. We use structural induction to prove that

$$P(T): \qquad |\mathbf{nodes}(T)| \leq 2^{h(T)+1} - 1$$

holds for every $T \in \mathcal{T}$.

BASIS: $T = \Lambda$. Then $|\mathbf{nodes}(T)| = 0$, and $h(T) = -1$, so $|\mathbf{nodes}(T)| = 2^{h(T)+1} - 1$ and $P(T)$ holds in this case.

INDUCTION STEP: Let $T_1, T_2 \in \mathcal{T}$ and suppose that $P(T_1), P(T_2)$ hold. We must prove that if $\mathbf{nodes}(T_1) \cap \mathbf{nodes}(T_2) = \emptyset$ and $r \notin \mathbf{nodes}(T_1) \cup \mathbf{nodes}(T_2)$, then $P(T)$ also holds, where $T = (r, T_1, T_2)$. Indeed we have:

$$
\begin{aligned}
|\mathbf{nodes}(T)| &= |\mathbf{nodes}(T_1)| + |\mathbf{nodes}(T_2)| + 1 && \text{[by Def. 4.6 and the fact that]}\\
&&& \mathbf{nodes}(T_1), \mathbf{nodes}(T_2) \text{ are disjoint]}\\
&&& \text{and do not contain } r\text{]}\\
&\le \bigl(2^{h(T_1)+1} - 1\bigr) + \bigl(2^{h(T_2)+1} - 1\bigr) + 1 && \text{[by induction hypothesis]}\\
&\le 2^{\max(h(T_1),h(T_2))+1} + 2^{\max(h(T_1),h(T_2))+1} - 1 && \text{[because } h(T_1), h(T_2) \le \max\bigl(h(T_1), h(T_2)\bigr)\text{]}\\
&\le 2^{h(T)} + 2^{(T)} - 1 && \text{[by the def. of } h(T)\text{]}\\
&\le 2^{h(T)+1} - 1
\end{aligned}
$$

so $P(T)$ holds. □

End of Example 4.6

Exercises

1. Consider the set $\mathcal{E}$ defined in Definition 4.1. Show that for *any* strings x and y over the alphabet $\{x, y, z, +, -, \times, \div, (,)\}$, the string $(x)(y)$ is *not* a member of $\mathcal{E}$.

2. Consider the following recursive definition for the set $\mathcal{B}$ of binary strings (i.e., finite strings of 0s and 1s): $\mathcal{B}$ is the smallest set such that

BASIS: The empty binary string, denoted ϵ, is in $\mathcal{B}$.

INDUCTION STEP: If x is in $\mathcal{B}$ then $x0$ and $x1$ (i.e., x followed by 0 or 1, respectively), is in $\mathcal{B}$.

A binary string has *odd parity* if it has and odd number of 0s; it has *even parity* if it has an even number of 0s. (Thus, the empty binary string has even parity.)
The *concatenation* of two binary strings x and y is the string xy obtained by appending y to x. Concatenation is a function of two binary strings that returns a binary string.

(a) Give a *recursive* definition of the set of odd parity binary strings, and the set of even parity binary strings. (**Hint:** Define both sets simultaneously!)

(b) Prove that your recursive definitions in (a) are equivalent to the non-recursive definitions given previously (involving the number of 0s in the string).

(c) Give a more formal, recursive, definition of the concatenation function. (**Hint:** Use induction on y.)

(d) Using your inductive definitions in (a) and (c), show that the concatenation of two odd parity binary strings is an even parity binary string. (**Hint:** You may find it useful to prove something else simultaneously.)

Chapter 5

PROPOSITIONAL LOGIC

5.1 Introduction

Propositional logic is the formalisation of reasoning involving propositions. A proposition is a statement that is either true or false, but not both. Examples of propositions are "Ottawa is the capital of Canada" (a true statement), "Ottawa is the largest city in Canada" (a false statement), and "it rained in Paris on September 3, 1631" (a statement that is true or false, although probably nobody presently alive knows which). There are statements that are not propositions because they do not admit a truth value; examples are statements of opinion ("the government must resign") and interrogative statements ("how are you?").

We can combine propositions to form more complex ones in certain well-defined ways. For example, consider the propositions: "Mary studies hard", "Mary understands the material" and "Mary fails the course". The ***negation*** (or ***denial***) of the first proposition is the proposition "Mary does **not** study hard". The ***conjunction*** of the first two propositions is the proposition "Mary studies hard **and** Mary understands the material". The ***disjunction*** of the first and third propositions is the proposition "Mary studies hard **or** Mary fails the course". We can also form the ***conditional*** of the second proposition on the first; this is the proposition: "**if** Mary studies hard **then** Mary understands the material". Finally, we can form the ***biconditional*** of the first two propositions: "Mary studies hard **if and only if** Mary understands the material".

Since these ways of combining propositions produce new propositions, nothing can stop us from applying them again on the constructed, more complex propositions, to produce even more complex ones, such as: "**if** Mary studies hard **and** Mary understands the material, **then** Mary does **not** fail the course". Propositional logic allows us to recognise the fact that this particular proposition states exactly the same thing as the following one: "it is **not** the case that: Mary studies hard **and** Mary understands the material **and** Mary fails the course". In fact, here are two other propositions that also state the same thing:

- "**If** Mary studies hard, **then if** Mary fails the course **then** Mary does **not** understand the material"; and
- "Mary does **not** study hard **or** Mary does **not** understand the material **or** Mary does **not** fail the course".

It is often important to be able to tell whether two propositions express the same thing. As the propositions get more and more complex, by combining simpler ones using negations, conjunctions, conditionals and so on, it becomes increasingly difficult to tell whether two propositions express the same thing. In this chapter we will define precisely what it means for two propositions to "express the same thing"; we will also learn how to determine if they do. In addition, we will examine the relationship between propositional logic and mathematical proofs. Finally, we will explore some important connections of propositional logic to the design of hardware.

5.2 Formalising propositional reasoning

5.2.1 Propositional formulas

First, we shall define a set of so-called ***propositional formulas***. These are formal expressions (i.e., strings over some alphabet) each of which is intended to represent a proposition. The definition is recursive. We start with a set PV of ***propositional variables***. These are intended to represent the most primitive propositions that are relevant to our purposes. For instance, in our previous example we might introduce three propositional variables: S for "Mary studies hard", U for "Mary understands the material", and F for "Mary fails the course". When we don't have a specific reason to use propositional variables whose names have some mnemonic value, we use $x, y, z, \ldots$ for propositional variables. The set of propositional variables may be infinite — e.g., $\{x_0, x_1, x_2, \ldots\}$.

Definition 5.1 *Let PV be a set of propositional variables. The set of* ***propositional formulas*** *(relative to PV), denoted $\mathcal{F}_{PV}$ (or simply $\mathcal{F}$, if PV is clear from the context), is the smallest set such that:*

BASIS: *Any propositional variable in PV belongs to $\mathcal{F}_{PV}$.*

INDUCTION STEP: *If P_1 and P_2 belong to $\mathcal{F}_{PV}$ then so do the following expressions: $\neg P_1$, $(P_1 \wedge P_2)$, $(P_1 \vee P_2)$, $(P_1 \rightarrow P_2)$ and $(P_1 \leftrightarrow P_2)$.*

Thus, propositional formulas are strings made up of the following symbols: propositional variables, left and right parentheses, $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$. The last five symbols are called (***Boolean*** or ***propositional***) ***connectives***. The connective $\neg$ is a ***unary*** connective, because it is applied to one subformula; the other four are ***binary*** connectives, because each of them is applied to two subformulas. Of course, not every string made up of such symbols is a propositional formula. For instance, $(x \neg y)$ and $((\rightarrow x)$ are not propositional formulas. The strings that are propositional formulas are those that can be constructed by applying the basis and induction step of Definition 5.1 a finite number of times.

The Boolean connectives correspond to the different constructions of more complex propositions out of simpler ones described in the previous section. More specifically, the connective $\neg$ corresponds to negation, $\wedge$ to conjunction, $\vee$ to disjunction, $\rightarrow$ to conditional, and $\leftrightarrow$ to biconditional. Informally, if P and P' are formulas that represent two particular propositions, then $(P \wedge P')$ is a formula that represents the conjunction of those propositions. Similarly,

$\neg P$ is a formula that represents the negation of the proposition represented by P. Analogous remarks apply to the other Boolean connectives. So, for example, if S, U and F are propositional variables (and therefore formulas) that represent the propositions "Mary studies hard", "Mary understands the material", and "Mary fails the course", then $(S \wedge U)$ is a formula that represents the proposition "Mary studies hard and understands the material", and $((S \wedge U) \rightarrow \neg F)$ is a formula that represents the proposition "If Mary studies hard and understands the material, then she does not fail the course".

As we discussed in Section 4.4, recursively defined objects can be fruitfully viewed as trees. Thus, we can view each propositional formula as a tree. The subformulas from which the formula is constructed correspond to the subtrees of the tree. The leaves of the tree correspond to occurrences of propositional variables, and the internal nodes correspond to the Boolean connectives. An internal node that corresponds to a binary connective has two children, while an internal node that corresponds to a unary connective has only one child. Figure 5.1 shows the tree that corresponds to a propositional formula.

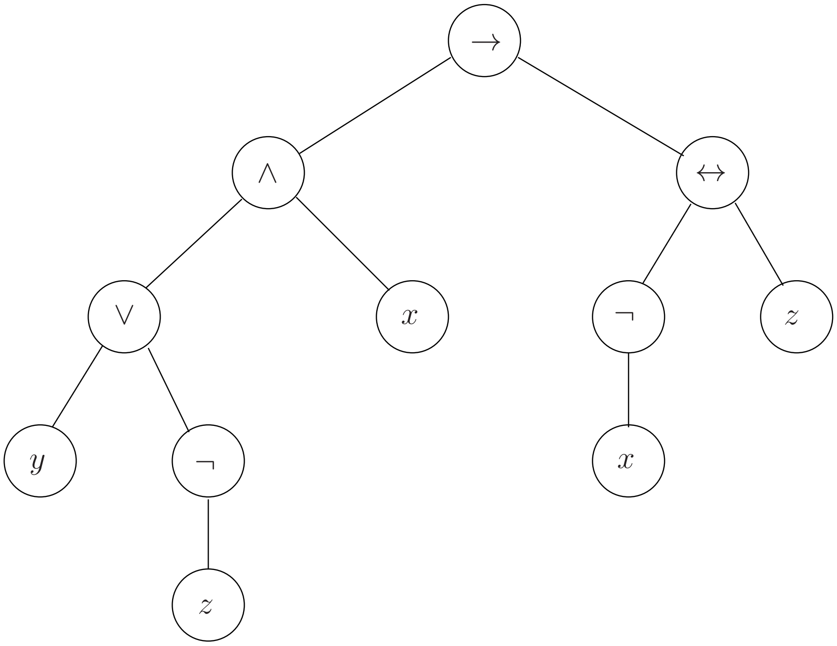


Figure 5.1: The parse tree of $\Big(\big((y \vee \neg z) \wedge x\big) \rightarrow (\neg x \leftrightarrow z)\Big)$

5.2.2 The meaning of propositional formulas

It is important to understand the distinction between a *propositional formula*, which is simply a syntactic object, and a *proposition*, which is an actual statement that is true or false. Given a propositional formula, and no other information at all, it makes no sense to ask whether the formula is true or false. To answer this question, we must be told whether the pieces out of which the formula is constructed represent propositions that are true or false. Since, ultimately, a formula is made up of propositional variables (combined by connectives), to determine whether a propositional formula is true or false, we need to be told, for each propositional variable that appears in the formula, whether it represents a true proposition or a false one. This leads us to the concept of a truth assignment.

Definition 5.2 *Let PV be a set of propositional variables. A **truth assignment** (to PV) is a function $\tau : PV \to \{0, 1\}$ — i.e., a function that assigns to each propositional variable a truth value. (We use the binary values 0 and 1 to represent the truth values false and true, respectively.)*

Thus, a truth assignment tells us, for each propositional variable, whether it represents a proposition that is true or false. Another way of thinking about a truth assignment is that it describes a possible "state of the world" by asserting what is true and what is false in that state. This interpretation assumes that the propositional variables capture all the aspects of the world that we are interested in.

Example 5.1 Consider our example with propositional variables S, U and F. Here the only aspects of the world that we are interested in are Mary's study habits, her understanding of the material and her success in the course. One possible truth assignment, τ_1 is $\tau_1(S) = \tau_1(F) = 0$ and $\tau_1(U) = 1$. This describes the state of affairs where Mary doesn't study hard but somehow she understands that material and does not fail the course. Another possible truth assignment is $\tau_2(S) = \tau_2(F) = 1$, $\tau_2(U) = 0$. This describes the state of affairs where Mary studies hard but nevertheless she does not understand the material and fails the course.

Since we have three propositional variables in this example, there are eight distinct truth assignments: one for each way of assigning the value true or false to each of the three variables. You should explicitly write down the remaining six truth assignments in this example, and interpret the state of affairs that each of them describes as regards Mary's studying, understanding the material, and passing the course. **End of Example 5.1**

A truth assignment τ specifies directly the truth values of the propositional variables. As we will now show, it can be used to obtain the truth value of *any* propositional formula. In mathematical terms, we wish to extend a truth assignment τ, which assigns truth values to the propositional variables, to a function τ^* that assigns truth values to *all* propositional formulas that can be built up from the propositional variables. The intention is that $\tau^*(P)$ is the truth value of formula P when the truth values of the propositional variables are those specified by τ. Formally, τ^* is defined by structural induction on the form of P (see Definition 5.1).

Definition 5.3 *Let $\tau : PV \to \{0, 1\}$. We define a function $\tau^* : \mathcal{F}_{PV} \to \{0, 1\}$ by structural induction.*

Basis: *$P \in PV$ (i.e., the formula P consists of just a propositional variable). In this case, $\tau^*(P) = \tau(P)$.*

Induction Step: *$P \notin PV$. Then there are $Q_1, Q_2 \in \mathcal{F}_{PV}$ such that P is one of the following formulas: $\neg Q_1$, $(Q_1 \wedge Q_2)$, $(Q_1 \vee Q_2)$, $(Q_1 \rightarrow Q_2)$, or $(Q_1 \leftrightarrow Q_2)$. We assume (inductively) that we have determined $\tau^*(Q_1)$ and $\tau^*(Q_2)$. In each of these cases, $\tau^*(P)$ is defined as shown below:*

$$\tau^*\big(\neg Q_1\big) = \begin{cases} 1, & \text{if } \tau^*(Q_1) = 0 \\ 0, & \text{otherwise} \end{cases}$$

$$\tau^*\big((Q_1 \wedge Q_2)\big) = \begin{cases} 1, & \text{if } \tau^*(Q_1) = \tau^*(Q_2) = 1 \\ 0, & \text{otherwise} \end{cases}$$

$$\tau^*\big((Q_1 \vee Q_2)\big) = \begin{cases} 0, & \text{if } \tau^*(Q_1) = \tau^*(Q_2) = 0 \\ 1, & \text{otherwise} \end{cases}$$

$$\tau^*\big((Q_1 \rightarrow Q_2)\big) = \begin{cases} 0, & \text{if } \tau^*(Q_1) = 1 \text{ and } \tau^*(Q_2) = 0 \\ 1, & \text{otherwise} \end{cases}$$

$$\tau^*\big((Q_1 \leftrightarrow Q_2)\big) = \begin{cases} 1, & \text{if } \tau^*(Q_1) = \tau^*(Q_2) \\ 0, & \text{otherwise} \end{cases}$$

We refer to $\tau^*(P)$ as ***the truth value of formula** P **under the truth assignment** τ*. If $\tau^*(P) = 1$, we say that τ ***satisfies*** P; if $\tau^*(P) = 0$, we say that τ ***falsifies*** P.

The dependence of a formula's truth value on the truth value(s) of its subformula(s) can be conveniently summarised in the following table:

Q_1	$\neg Q_1$
0	1
1	0

Q_1	Q_2	$(Q_1 \wedge Q_2)$	$(Q_1 \vee Q_2)$	$(Q_1 \rightarrow Q_2)$	$(Q_1 \leftrightarrow Q_2)$
0	0	0	0	1	1
0	1	0	1	1	0
1	0	0	1	0	0
1	1	1	1	1	1

The truth value of P under a truth assignment τ is given in the entry at the intersection of the column that corresponds to the form of P and the row that corresponds to the truth value(s) of the constituent part(s) of P under the truth assignment τ. For instance if P is of the form $(Q_1 \rightarrow Q_2)$, and the truth values of Q_1 and Q_2 under τ are 1 and 0 respectively, we look at the third row of the column labeled $(Q_1 \rightarrow Q_2)$ and find that the truth value of P under τ is 0.

Example 5.2 Using Definition 5.3, we can calculate the truth value of the formula $((S \vee U) \wedge \neg F)$ under the truth assignment τ_1 described in Example 5.5.1. Since $\tau_1(U) = 1$, we have that $\tau_1^*(U) = 1$ and therefore $\tau_1^*\big((S \vee U)\big) = 1$. Similarly, since $\tau_1(F) = 0$, we have that $\tau_1^*(F) = 0$ and therefore $\tau_1^*(\neg F) = 1$. Finally, since $\tau_1^*\big((S \vee U)\big) = \tau^*(\neg F) = 1$, it follows that $\tau_1^*\big(((S \vee U) \wedge \neg F)\big) = 1$. Therefore, τ_1 satisfies $((S \vee U) \wedge \neg F)$.

We can also determine the truth value of the same formula under the truth assignment τ_2 described in Example 5.5.1: Since $\tau_2(F) = 1$, it follows that $\tau_2^*(\neg F) = 0$. But then, $\tau_2^*\big((P \wedge \neg F)\big) = 0$, for any formula P. In particular, $\tau_2^*\big(((S \vee U) \wedge \neg F)\big) = 0$. **End of Example 5.2**

5.2.3 Some subtleties in the meaning of logical connectives

According to Definition 5.3, the formula $(Q_1 \vee Q_2)$ is true (under τ) provided that *at least* one of Q_1, Q_2 is true. In particular, the formula is true if *both* Q_1 and Q_2 are true. This is sometimes, but not always, consistent with the way in which the word "or" is used in English. Consider, for example, the assertion that an individual is entitled to Canadian citizenship if the individual's father *or* mother is a Canadian citizen. This is understood as implying that a person *both* of whose parents are Canadian is entitled to Canadian citizenship. In this case, the word "or" is said to be ***inclusive***. On the other hand, if mother tells Jimmy that he may have cake *or* ice cream, we would probably consider Jimmy to be in violation of this instruction if he were to have both. In this case, "or" is said to be ***exclusive***. The definition of truth value for the symbol $\vee$ reflects the inclusive interpretation of "or". There is a different Boolean connective (discussed later, in Section 5.11) that represents the exclusive meaning of "or".

The definition of the truth value for conditional formulas, i.e., formulas of the form $(Q_1 \rightarrow Q_2)$, also requires some clarification. First, some terminology. The subformula Q_1 is called the ***antecedent***, and the subformula Q_2 is called the ***consequent***, of the conditional. Some people find it counterintuitive (or at least not obvious) that a conditional statement is considered to be true if both the antecedent and the consequent are false. The following example might help convince you that this way of defining the truth value of conditional statements is sensible. Suppose M stands for the proposition "the patient takes the medicine", and F stands for the proposition "the fever will subside". Consider now the doctor's assertion: "if the patient takes the medicine then the fever will subside" — this is represented by the formula $(M \rightarrow F)$. Suppose the patient does *not* take the medicine and the fever does *not* subside — this corresponds to the truth assignment τ, where $\tau(M) = \tau(F) = 0$. Most people would agree that the doctor's assertion cannot be viewed as false in this case. Since the assertion is not false, it must be true. In other words, $\tau^*\big((M \rightarrow F)\big) = 1$.

5.2.4 Unique readability and conventions for omitting parentheses

Our definition of propositional formulas requires that every time we construct a larger formula by applying a binary connective we enclose the constructed formula inside parentheses. Without these parentheses, the manner in which the formula was constructed from other formulas is not uniquely determined. To see this suppose, for a moment, that we changed Definition 5.1 so that in the induction step we did not require parentheses for the binary connectives. Under this modified definition the expression $x \wedge y \vee z$ would be a propositional formula. But is this the formula obtained by *first* taking the conjunction of x with y, and *then* the disjunction of the resulting formula with z — i.e., the formula written $((x \wedge y) \vee z)$ in the original definition that requires parentheses? Or is it the formula obtained by *first* taking the disjunction of y with z and *then* the conjunction of x with the resulting formula — i.e., the formula written

$(x \wedge (y \vee z))$ in the original definition?

These two formulas are not only different in form — they have different meanings. To see why, consider the truth assignment in which x and y are false, and z is true. Under this truth assignment, the first formula is true, but the second formula is false. In other words, the order of construction of $x \wedge y \vee z$ matters for the meaning of the formula. Without an explicit indication of which of the two orders is intended, the meaning of the formula is ambiguous. It is to avoid this sort of ambiguity that we require the parentheses.

The fact that the use of parentheses avoids ambiguous parsing of propositional formulas can be stated precisely as the following so-called "Unique Readability Theorem".

Theorem 5.4 ***(Unique Reabability Theorem)*** *For any propositional formulas P_1, P_2, Q_1, Q_2 and binary connectives $\oplus$ and $\ominus$ (i.e., $\oplus, \ominus \in \{\wedge, \vee, \rightarrow, \leftrightarrow\}$), if $(P_1 \oplus P_2) = (Q_1 \ominus Q_2)$ then $P_1 = Q_1$, $\oplus = \ominus$ and $P_2 = Q_2$.*

Intuitively, this says that if there are two ways of constructing a formula, then the pieces out of which it is constructed are identical — in other words, there is really only one way of constructing a formula. As we discussed earlier, without the use of parentheses this is not true: For instance, if $P_1 = x$, $\oplus = \wedge$, $P_2 = y \vee z$, $Q_1 = x \wedge y$, $\ominus = \vee$, $P_2 = z$, it is certainly true that $P_1 \oplus P_2 = Q_1 \ominus Q_2$, but it is not true that $P_1 = Q_1$, $\oplus = \ominus$ and $P_2 = Q_2$.

The proof of the Unique Readability Theorem is an interesting application of structural induction and is left as an exercise (see Exercise 13).

Although the use of parentheses eliminates ambiguity in parsing propositional formulas, it is true that including all the parentheses required by the definition leads to formulas that are long and unnecessarily awkward to read. To avoid this problem, we adopt some conventions regarding the precedence of the connectives that allow us to leave out some of the parentheses without introducing ambiguities. Here are the conventions that we will use.

(a) We will leave out the outermost pair of parentheses (if one exists). For instance, we can write $x \wedge (y \vee z)$ as shorthand for $(x \wedge (y \vee z))$. Note that if we want to later use this formula to construct a larger formula, we have to introduce the parentheses we left out, since these are no longer the outermost parentheses! For example, if we want to form the negation of the formula $x \wedge (y \vee z)$, we have to write $\neg(x \wedge (y \vee z))$ — not $\neg x \wedge (y \vee z)$, which is a different formula, with a different meaning!

(b) $\wedge$ and $\vee$ have precedence over $\rightarrow$ and $\leftrightarrow$. Thus, $x \wedge y \rightarrow y \vee z$ is short for $((x \wedge y) \rightarrow (y \vee z))$.

(c) $\wedge$ has precedence over $\vee$. Thus, $x \wedge y \vee z$ is short for $((x \wedge y) \vee z))$. Combining with the previous rule, we have that $x \vee y \leftrightarrow x \vee y \wedge z$ is short for $((x \vee y) \leftrightarrow (x \vee (y \wedge z)))$.

(d) When the same binary connective is used several times in a row, grouping is assumed to be to the right. Thus, $x \rightarrow y \rightarrow z$ is short for $(x \rightarrow (y \rightarrow z))$.

Note that according to Definition 5.1, no parentheses are used when we apply the $\neg$ connective. This has the implicit consequence of assigning to this connective higher precedence than all the others. For instance, $\neg x \wedge y \vee z$ is short for $((\neg x \wedge y) \vee z)$. This is quite different from the formula $\neg(x \wedge y \vee z)$, which is short for $\neg((x \wedge y) \vee z)$.

These conventions are analogous to those used to omit parentheses in arithmetic expressions without introducing ambiguities: For instance, $-3 \times 5 + 10 \div 5$ is short for $(-3 \times 5) + (10 \div 5)$. Thus, multiplication and division have precedence over addition. Note that the unary $-$ (to indicate negative numbers) is similar to the $\neg$ symbol.

5.3 Truth tables

The truth table of a propositional formula P is a convenient representation of the truth value of P under each possible truth assignment to the propositional variables that appear in P. As we saw in Definition 5.3, to determine these truth values we must determine the truth values of the subformulas out of which P is constructed; for this we must, in turn, determine the truth values of the subformulas' subformulas — and so on, down to the most primitive subformulas, i.e., propositional variables, whose truth values are given directly by the truth assignment.

For example, consider the formula $x \vee y \to \neg x \wedge z$. The truth table of this formula (and its constituent subformulas) is shown below.

x	y	z	$x \vee y$	$\neg x$	$\neg x \wedge z$	$x \vee y \to \neg x \wedge z$
0	0	0	0	1	0	1
0	0	1	0	1	1	1
0	1	0	1	1	0	0
0	1	1	1	1	1	1
1	0	0	1	0	0	0
1	0	1	1	0	0	0
1	1	0	1	0	0	0
1	1	1	1	0	0	0

A slightly more succinct way of doing the truth table, that avoids rewriting the subformulas in separate columns, is to write the truth value of each subformula underneath the last connective applied in the construction of that subformula, starting from the "innermost" subformulas and working our way out. For instance, in the example we did above, we would proceed as follows. First we write all the truth assignments to the propositional variables x, y and z.

x	y	z	$x \vee y$	$\to$	$\neg x$	$\wedge$	z
0	0	0					
0	0	1					
0	1	0					
0	1	1					
1	0	0					
1	0	1					
1	1	0					
1	1	1					

The "innermost" subformulas are $x \vee y$ and $\neg x$, so we can calculate their truth tables by writing their truth values underneath the $\vee$ and $\neg$ symbols respectively. When we are done, we have:

x	y	z	$x \vee y$	$\to$	$\neg x$	$\wedge$	z
0	0	0	0		1		
0	0	1	0		1		
0	1	0	1		1		
0	1	1	1		1		
1	0	0	1		0		
1	0	1	1		0		
1	1	0	1		0		
1	1	1	1		0		

Now that we have computed the truth values of $\neg x$ we can compute the truth values of $\neg x \wedge z$, so we can now fill the column underneath the $\wedge$ connective.

x	y	z	$x \vee y$	$\to$	$\neg x$	$\wedge$	z
0	0	0	0		1	0	
0	0	1	0		1	1	
0	1	0	1		1	0	
0	1	1	1		1	1	
1	0	0	1		0	0	
1	0	1	1		0	0	
1	1	0	1		0	0	
1	1	1	1		0	0	

Finally, having computed the truth values of $x \vee y$ and $\neg x \wedge z$, we can compute the truth values of $x \vee y \to \neg x \wedge z$, we can can now fill the column underneath the $\to$ connective — this column gives the truth values of the formula we are interested in:

x	y	z	$x \vee y$	$\to$	$\neg x$	$\wedge$	z
0	0	0	0	1	1	0	
0	0	1	0	1	1	1	
0	1	0	1	0	1	0	
0	1	1	1	1	1	1	
1	0	0	1	0	0	0	
1	0	1	1	0	0	0	
1	1	0	1	0	0	0	
1	1	1	1	0	0	0	

Even so, however, it can be very tedious to write down the truth table of formulas of even moderate size, because of the number of rows. There has to be one row for each possible truth assignment, and if there are n propositional variables, then there are 2^n truth assignments. (This is because a truth assignment to n propositional variables can be viewed as a binary string of length n, and we have already seen that there are 2^n such strings.) Thus, the truth table of a propositional formula with only five variables has $2^5 = 32$ rows — about a page long. The truth table of a propositional formula with 10 variables has over 1,000 rows — out of the question for most people to do manually. The truth table of a propositional formula with just 30 variables has over 1 billion rows, and this would tax the abilities of even powerful computers. For small formulas, however (up to five variables or so), truth tables can be very useful.

5.4 Tautologies and satisfiability

If the truth value of a propositional formula P under truth assignment τ is 1, we say that τ ***satisfies*** P; otherwise, we say that τ ***falsifies*** P.

Definition 5.5 *Let P be a propositional formula.*

- *P is a **tautology** (or **a valid formula**) if and only if every truth assignment satisfies P.*
- *P is **satisfiable**, if and only if there is a truth assignment that satisfies P.*
- *P is **unsatisfiable** (or a **contradiction**) if and only if it is not satisfiable (i.e., every truth assignment falsifies P).*

For example, $(x \vee y) \wedge \neg x \rightarrow y$ is a tautology; $(x \vee y) \rightarrow x$ is a satisfiable formula but it is not a tautology; and $x \wedge \neg(x \vee y)$ is unsatisfiable. These facts can be readily verified by looking at the truth tables of these formulas.

It is immediate from Definition 5.5 that P is a tautology if and only if $\neg P$ is unsatisfiable.

5.5 Logical implication and logical equivalence

In this section we introduce two extremely important concepts: logical implication and logical equivalence.

Definition 5.6 *A propositional formula P **logically implies** propositional formula Q if and only if every truth assignment that satisfies P also satisfies Q.*

For example, $x \wedge y$ logically implies x; x logically implies $x \vee y$; $x \wedge (x \rightarrow y)$ logically implies y. Each of these logical implications can be verified by looking at the truth tables of the two formulas involved.

It is important to understand properly the relationship between $P \rightarrow Q$ and "P logically implies Q". The former is a propositional formula. The latter is a statement *about* two propositional formulas; it is *not* a propositional formula. Although $P \rightarrow Q$ and "P logically implies Q" are beasts of a different nature, there is a close relationship between the two. The following theorem states this relationship.

Theorem 5.7 *P logically implies Q if and only if $P \rightarrow Q$ is a tautology.*

PROOF.

	P logically implies Q
if and only if	every truth assignment that satisfies P also satisfies Q

[by the definition of "logically implies"]
if and only if every truth assignment satisfies $P \rightarrow Q$
[by the definition of satisfying $P \rightarrow Q$]
if and only if $P \rightarrow Q$ is a tautology
[by the definition of "tautology"]

□

From the definition of logical implication it follows immediately that a contradiction (an unsatisfiable formula) P logically implies *any* formula Q whatsoever! This is because there is no truth assignment that satisfies P (by definition of contradiction); vacuously, then, any truth assignment that satisfies P satisfies Q — no matter what Q is. In view of Theorem 5.7, the same observation can be stated by saying that, if P is a contradiction, then the formula $P \rightarrow Q$ is a tautology, regardless of what Q is.

Definition 5.8 *A propositional formula P* ***is logically equivalent to*** *propositional formula Q if and only if P logically implies Q and Q logically implies P.*

From this definition we have that P is logically equivalent to Q if and only if, for each truth assignment τ, the truth value of P under τ is the same as the truth value of Q under τ. Another way of saying the same thing is that any truth assignment that satisfies P also satisfies Q, and any truth assignment that falsifies P also falsifies Q.

For example, $x \wedge y$ is logically equivalent to $\neg(\neg x \vee \neg y)$. In English, we can paraphrase the former as "x and y are both true" and the latter as "it is not the case that either x or y is false" — and indeed, the two statements say exactly the same thing. The fact that the two formulas can be rendered in English by two expressions that "say the same thing" is only an informal justification that they are logically equivalent. After all, how do we know that the English "translations" of these formulas (especially of the second one) are really accurate? We can, in any event, verify that the two formulas are equivalent by constructing the truth tables of the two formulas and discovering that they are identical.

The following three properties of logical equivalence follow immediately from Definition 5.8:

Reflexivity: P is logically equivalent to P.

Symmetry: If P is logically equivalent to Q then Q is logically equivalent to P.

Transitivity: If P is logically equivalent to Q, and Q is logically equivalent to R, then P is logically equivalent to R.

Remarks similar to those we made regarding the relationship between the formula $P \rightarrow Q$ and the assertion that P logically implies Q apply to the relationship between the formula $P \leftrightarrow Q$ and the assertion that P is logically equivalent to Q. This can be stated as the following theorem, whose proof is very similar to the proof of Theorem 5.7, and is left as an exercise.

Theorem 5.9 *P is logically equivalent to Q if and only if $P \leftrightarrow Q$ is a tautology.*

5.6 Some important logical equivalences

There are some logical equivalences that are important enough to have achieved the exalted status of "laws"! Some of them, and their names, are listed below, where P, Q and R are arbitrary propositional formulas ("LEQV" is short for "is logically equivalent to"):

Law of double negation:	$\neg\neg P$	LEQV	P
De Morgan's laws:	$\neg(P \wedge Q)$	LEQV	$\neg P \vee \neg Q$
	$\neg(P \vee Q)$	LEQV	$\neg P \wedge \neg Q$
Commutative laws:	$P \wedge Q$	LEQV	$Q \wedge P$
	$P \vee Q$	LEQV	$Q \vee P$
Associative laws:	$P \wedge (Q \wedge R)$	LEQV	$(P \wedge Q) \wedge R$
	$P \vee (Q \vee R)$	LEQV	$(P \vee Q) \vee R$
Distributive laws:	$P \wedge (Q \vee R)$	LEQV	$(P \wedge Q) \vee (P \wedge R)$
	$P \vee (Q \wedge R)$	LEQV	$(P \vee Q) \wedge (P \vee R)$
Identity laws:	$P \wedge (Q \vee \neg Q)$	LEQV	P
	$P \vee (Q \wedge \neg Q)$	LEQV	P
Idempotency laws:	$P \wedge P$	LEQV	P
	$P \vee P$	LEQV	P
$\rightarrow$ *law:*	$P \rightarrow Q$	LEQV	$\neg P \vee Q$
$\leftrightarrow$ *law:*	$P \leftrightarrow Q$	LEQV	$P \wedge Q \vee \neg P \wedge \neg Q$

Each of these logical equivalences can be readily verified by constructing the truth tables of the two formulas involved.

In the Identity laws, note that the propositional formula $(Q \vee \neg Q)$ is a tautology, while the propositional formula $(Q \wedge \neg Q)$ is a contradiction. Thus, these laws say that a tautology can be "cancelled" from a conjunction, and that a contradiction can be "cancelled" from a disjunction.

Suppose we have a formula R and a formula S that appears as part of R. For example, R might be $x \wedge (\neg y \vee \neg z) \rightarrow w$, and S might be $(\neg y \vee \neg z)$:

$$\underbrace{x \wedge \overbrace{(\neg y \vee \neg z)}^{S} \rightarrow w}_{R}$$

By De Morgan's laws, the formula $S' = \neg(y \wedge z)$ is logically equivalent to S. Suppose now we replace S by S' in R, obtaining a new formula R':

$$\underbrace{x \wedge \overbrace{\neg(y \wedge z)}^{S'} \rightarrow w}_{R'}$$

This formula is logically equivalent to R. We can verify this by truth tables, but we can also reason in a simpler and more general way: The truth value of a formula under a truth assignment τ depends only on the truth values of its subformulas under τ. If S and S' are logically equivalent, they have the same truth value under τ. This means that, replacing S by S' in R will not affect the truth value of the resulting formula R'. In other words, for any truth assignment τ, the truth value of R under τ is the same as the truth value of R' under τ; i.e., R and R' are logically equivalent.

Since this argument does not depend on the specific form of R, S and S' — but only on the fact that S is a subformula of R and that S' is logically equivalent to S — we can generalise our observation as the following theorem.

Theorem 5.10 *Let R be any propositional formula, S be any subformula of R, S' be any formula that is logically equivalent to S, and R' be the formula that results from R by replacing S with S'. Then R' is logically equivalent to R.*

This theorem, in conjunction with the logical equivalences stated earlier, can be used to prove that two propositional formulas are logically equivalent *without having to go through truth table constructions*. Here is an example. Suppose we want to prove that $\neg((x \wedge y) \rightarrow \neg z)$ is logically equivalent to $x \wedge y \wedge z$.

We proceed as follows. First, note that $(x \wedge y) \rightarrow \neg z$ is logically equivalent to $\neg(x \wedge y) \vee \neg z$. This follows from the the $\rightarrow$ law, $P \rightarrow Q$ LEQV $\neg P \vee Q$, by taking $P = (x \wedge y)$ and $Q = \neg z$. By Theorem 5.10, where $R = \neg((x \wedge y) \rightarrow \neg z)$, $S = (x \wedge y) \rightarrow \neg z$ and $S' = \neg(x \wedge y) \vee \neg z$, it follows that

$$\neg((x \wedge y) \rightarrow \neg z) \quad \text{LEQV} \quad \neg(\neg(x \wedge y) \vee \neg z) \tag{5.1}$$

In addition,

$$\neg(\neg(x \wedge y) \vee \neg z) \quad \text{LEQV} \quad \neg\neg(x \wedge y) \wedge \neg\neg z \tag{5.2}$$

This follows by DeMorgan's law, $\neg(P \vee Q)$ LEQV $(\neg P \wedge \neg Q)$, by taking $P = \neg(x \wedge y)$ and $Q = \neg z$. By (5.1), (5.2) and transitivity of logical equivalence,

$$\neg((x \wedge y) \rightarrow \neg z) \quad \text{LEQV} \quad \neg\neg(x \wedge y) \wedge \neg\neg z \tag{5.3}$$

By the law of double negation, $\neg\neg(x \wedge y)$ is logically equivalent to $x \wedge y$, and $\neg\neg z$ is logically equivalent to z. By applying Theorem 5.10 twice (once to replace $\neg\neg(x \wedge y)$ by its logically equivalent $x \wedge y$, and once again to replace $\neg\neg z$ by its logically equivalent z), we get that

$$\neg\neg(x \wedge y) \wedge \neg\neg z \quad \text{LEQV} \quad x \wedge y \wedge z \tag{5.4}$$

By (5.3), (5.4) and transitivity of logical equivalence we get the desired logical equivalence:

$$\neg((x \wedge y) \rightarrow \neg z) \quad \text{LEQV} \quad x \wedge y \wedge z$$

We can present this argument in the following more convenient form, where we omit any reference to applications of Theorem 5.10 and to the transitivity of logical equivalence, and simply note which law we use in each step:

$$\begin{array}{lll}
 & \neg((x \wedge y) \rightarrow \neg z) & \\
\textsc{leqv} & \neg(\neg(x \wedge y) \vee \neg z) & [\text{by} \rightarrow \text{law}] \\
\textsc{leqv} & \neg\neg(x \wedge y) \wedge \neg\neg z) & [\text{by DeMorgan's law}] \\
\textsc{leqv} & x \wedge y \wedge z & [\text{by double negation law (applied twice)}]
\end{array}$$

5.7 *Conditional statements*

In mathematics and computer science we often encounter conditional statements which assert that if some hypothesis P holds, then some consequence Q follows — i.e., statements of the form $P \rightarrow Q$. There are two other conditional statements that are related to $P \rightarrow Q$: its ***converse***, i.e., the statement $Q \rightarrow P$; and its ***contrapositive***, i.e., the statement $\neg Q \rightarrow \neg P$. Many logical errors and fallacious arguments have their roots in confusing a conditional, its converse and its contrapositive. Thus, it is very important to understand the relationship among these three statements.

A conditional statement is logically equivalent to its contrapositive; i.e., $P \rightarrow Q$ is logically equivalent to $\neg Q \rightarrow \neg P$. A conditional statement is *not* logically equivalent to its converse; i.e., in general, $P \rightarrow Q$ is *not* logically equivalent to $Q \rightarrow P$. You can verify the first of these assertions by looking at the truth tables of the two statements, or by applying the following sequence of logical equivalences from Section 5.6:

$$\begin{array}{lll}
 & P \rightarrow Q & \\
\textsc{leqv} & \neg P \vee Q & [\text{by} \rightarrow \text{law}] \\
\textsc{leqv} & \neg P \vee \neg\neg Q & [\text{by double negation law}] \\
\textsc{leqv} & \neg\neg Q \vee \neg P & [\text{by commutative law}] \\
\textsc{leqv} & \neg Q \rightarrow \neg P & [\text{by} \rightarrow \text{law}]
\end{array}$$

The fact that $P \rightarrow Q$ is not, in general, logically equivalent to $Q \rightarrow P$ can be seen by noting that a truth assignment that satisfies P and falsifies Q, falsifies $P \rightarrow Q$ but satisfies $Q \rightarrow P$ — thus, in general, there is a truth assignment under which the two formulas do not have the same truth value.

To see a more concrete example of this point, consider the statement "if the battery is dead, then the car does not run". This is a true statement. The converse of this statement is "if the car does not run, then the battery is dead". This is a false statement, because there are other reasons why a car may not run, in addition to having a dead battery — for instance, it may have run out of gas. The contrapositive of the original statement is "if the car runs, then the battery is not dead". This, like the original statement (to which it is logically equivalent), is a true statement.

5.8 *Analysis of three proof techniques*

Three types of arguments that appear frequently in mathematics are indirect proofs, proofs by contradiction, and proofs by cases. You have seen instances of such arguments in other mathematics courses in high school and university — and, indeed, in proofs we have done earlier in this course. The reasoning that underlies each of these proof techniques is a logical equivalence in propositional logic. It may be illuminating to reexamine these proof techniques in more abstract terms now that we understand the concept of logical equivalence.

5.8.1 Indirect proof (or proof by contrapositive)

Many of the statements we wish to prove are conditionals. In general, we have to show that if a certain hypothesis H holds, then a certain conclusion C follows. Symbolically, we have to show that the statement $H \to C$ holds. In an indirect proof, we do so by proving that if the conclusion C does not hold, then the hypothesis H is false. Symbolically, we prove that the statement $\neg C \to \neg H$ holds. In other words, instead of proving the desired conditional statement, we prove its contrapositive. This is legitimate because the two statements, $H \to C$ and $\neg C \to \neg H$, are logically equivalent. They express exactly the same thing, and thus a proof of one constitutes a proof of the other.

An example of an indirect proof follows.

Theorem 5.11 *For any natural number n, if n^2 is even then n is even.*

Proof. We prove the contrapositive, i.e., if n is not even, then n^2 is not even.

Suppose that n is not even. Hence, n is odd, and we have that $n = 2k+1$, where k is some natural number. Hence, $n^2 = (2k+1)^2 = 2(2k^2+2k)+1$, which is an odd number. Therefore, n^2 is not even. □

It is very important to understand that in an indirect proof of we prove the *contrapositive* of the desired (conditional) statement, *not* its converse. A proof of the converse of a conditional statement does not constitute a proof of that conditional statement. This is because, as we saw earlier, a conditional and its converse are not logically equivalent.

This point is perhaps obscured in the preceding example because the converse of Theorem 5.11 happens to be true as well. But consider the following example:

Theorem 5.12 *For any natural number n, if n^2 **mod** $16 = 1$ then n is odd.*

Proof. We prove the contrapositive, i.e., if n is not odd, then $n^2 \mathbf{~mod~} 16 \neq 1$.

Suppose that n is not odd. Hence, n is even and so there is some $k \in \mathbb{N}$ so that $n = 2k$. Therefore, $n^2 = 4k^2$. By the definition of the **mod** and **div** functions (see page 28),

$$4k^2 = n^2 = 16(n^2 \mathbf{~div~} 16) + (n^2 \mathbf{~mod~} 16)$$

and thus

$$n^2 \mathbf{~mod~} 16 = 4k^2 - 16(n^2 \mathbf{~div~} 16) = 4\big(k^2 - 4(n^2 \mathbf{~div~} 16)\big)$$

This means that n^2 **mod** 16 is a multiple of 4, and so it cannot be equal to 1. □

Notice that the converse of Theorem 5.12 is *not* true: The converse asserts that if n is odd then n^2 **mod** $16 = 1$. It is easy to see that $n = 3$ is a counterexample to this assertion. Here it is perfectly clear that the proof of the contrapositive is not proof of the converse — since, in this case, the converse is not true!

5.8.2 Proof by contradiction

In this type of proof we are asked to show that some proposition P holds. Instead, we assume that P is false and derive a contradiction. From this we can conclude that the proposition P must be true. In other words, instead of proving P we prove that a statement of the form $\neg P \rightarrow (Q \wedge \neg Q)$ holds. (Note that $Q \wedge \neg Q$ is a contradiction.) This is legitimate because P is logically equivalent to $\neg P \rightarrow (Q \wedge \neg Q)$, and thus a proof of the latter constitutes a proof of the former.

We can show this logical equivalence as follows:

$$\begin{array}{lll} & \neg P \rightarrow (Q \wedge \neg Q) & \\ \text{LEQV} & \neg\neg P \vee (Q \wedge \neg Q) & [\text{by} \rightarrow \text{law}] \\ \text{LEQV} & P \vee (Q \wedge \neg Q) & [\text{by double negation law}] \\ \text{LEQV} & P & [\text{by Identity law}] \end{array}$$

A beautiful and very famous example of proof by contradiction follows. Recall that a rational number is one that can be expressed as a ratio of two integers, and an irrational number is one that is not rational.

Theorem 5.13 $\sqrt{2}$ *is an irrational number.*

Proof. Suppose, for contradiction, that $\sqrt{2}$ is a rational number. Thus, for two integers m and n, $\sqrt{2} = m/n$. Without loss of generality, we may assume that m and n have no common prime factors — i.e., the fraction m/n is irreducible. (The reason why we may assume this is that, if m and n had a common prime factor, and their highest common factor was h, we could replace m and n by $m' = m/h$ and $n' = n/h$ to get two numbers m' and n' whose ratio is $\sqrt{2}$ and which have no common factors.)

Since $\sqrt{2} = m/n$, we have that $2 = m^2/n^2$ and hence $m^2 = 2n^2$. This means that m^2 is even and, by Theorem 5.11, m is even. Thus, for some integer k, $m = 2k$, and therefore $m^2 = 4k^2$. Since $m^2 = 2n^2$, we get $4k^2 = 2n^2$, and hence $n^2 = 2k^2$. This means that n^2 is even and, by Theorem 5.11, n is even. But since both m and n are even, they have a common prime factor, namely 2. This contradicts the fact that m and n have no common factors.

Since we arrived at a contradiction, our original hypothesis that $\sqrt{2}$ is rational must be false. Thus, $\sqrt{2}$ is irrational. □

5.8.3 Proof by cases

Sometimes when we want to prove that a statement P holds, it is convenient to consider two cases that cover all possibilities — that Q holds (for some proposition Q) and that Q does not hold — and prove that in each case P holds. In other words, instead of proving P, we prove a statement of the form $(Q \to P) \land (\neg Q \to P)$. This is a legitimate proof of the original statement P because $(Q \to P) \land (\neg Q \to P)$ is logically equivalent to P. Thus, a proof of the former constitutes a proof of the latter.

We can show that P is logically equivalent to $(Q \to P) \land (\neg Q \to P)$ as follows:

	$(Q \to P) \land (\neg Q \to P)$	
LEQV	$(\neg Q \lor P) \land (\neg\neg Q \lor P)$	[by $\to$ law (applied twice)]
LEQV	$(\neg Q \lor P) \land (Q \lor P)$	[by double negation law]
LEQV	$(P \lor \neg Q) \land (P \lor Q)$	[by commutative law (applied twice)]
LEQV	$P \lor (\neg Q \land Q)$	[by distributive law]
LEQV	$P \lor (Q \land \neg Q)$	[by commutative law]
LEQV	P	[by Identity law]

Following is an example of a proof by cases.

Theorem 5.14 *For any natural number n, $\lfloor (n+1)/2 \rfloor = \lceil n/2 \rceil$.*

PROOF. There are two cases.

CASE 1. *n is odd.* Then $n = 2k+1$, for some natural number k. We have,

$$\lfloor (n+1)/2 \rfloor = \lfloor (2k+2)/2 \rfloor = \lfloor k+1 \rfloor = k+1$$

and

$$\lceil n/2 \rceil = \lceil (2k+1)/2 \rceil = \lceil k + \frac{1}{2} \rceil = k+1.$$

Thus, in this case $\lfloor (n+1)/2 \rfloor = \lceil n/2 \rceil$, as wanted.

CASE 2. *n is not odd.* Then n is even, and hence $n = 2k$, for some natural number k. We have,

$$\lfloor (n+1)/2 \rfloor = \lfloor (2k+1)/2 \rfloor = \lfloor k + \frac{1}{2} \rfloor = k$$

and

$$\lceil n/2 \rceil = \lceil 2k/2 \rceil = \lceil k \rceil = k.$$

Thus, in this case $\lfloor (n+1)/2 \rfloor = \lceil n/2 \rceil$, as wanted. □

We can generalise the technique of proof by cases. Instead of considering only two cases — those determined by a proposition Q and its negation $\neg Q$ — we can instead consider an arbitrary number of cases, determined by the propositions $Q_1, Q_2, \ldots, Q_{n-1}$, and the negation of their disjunction $\neg(Q_1 \vee Q_2 \vee \ldots \vee Q_{n-1})$. More specifically, to prove that the statement P holds, we prove that P holds if Q_1 is the case, that P holds if Q_2 is the case, ..., that P holds if Q_{n-1} is the case, as well as that P holds if "none of the above" is the case — i.e., that P holds if $\neg(Q_1 \vee Q_2 \vee \ldots \vee Q_{n-1})$. This is a legitimate way to prove P, because it can be shown that the formula

$$(Q_1 \rightarrow P) \wedge (Q_2 \rightarrow P) \wedge \ldots \wedge (Q_{n-1} \rightarrow P) \wedge (\neg(Q_1 \vee Q_2 \vee \ldots \vee Q_{n-1}) \rightarrow P)$$

is logically equivalent to P. (Prove this fact by induction on $n \geq 1$.)

5.9 Normal forms for propositional formulas

In this section we introduce two ***normal forms*** for propositional formulas. A normal form is a set of propositional formulas that are syntactically restricted in some way but, despite the syntactic restrictions, they can "represent" all propositional formulas. More precisely, for each propositional formula P, there is some formula $\hat{P}$ that has the required syntactic form and is logically equivalent to P. First we need some terminology.

Definition 5.15 *We define various restricted types of propositional formulas.*

- *A **literal** is a propositional formula that consists only of a propositional variable or the negation of a propositional variable.*
- *A **minterm** (or simply **term**) is a formula that is a literal, or a conjunction of two or more literals.*
- *A **maxterm** (or **clause**) is a formula that is literal, or a disjunction of two or more literals.*
- *A propositional formula is in **disjunctive normal form** (DNF) if it is minterm, or a disjunction of two or more minterms.*
- *A propositional formula is in **conjunctive normal form** (CNF) if it is a maxterm, or a conjunction of two or more maxterms.*

Example 5.3 The formulas x, $\neg x$, y, $\neg y$ are literals; the formulas $\neg\neg x$, $x \wedge y$, $\neg(x \wedge y)$, $x \wedge \neg y$ are not literals.

The formula $x \wedge \neg y \wedge \neg z \wedge w$ is a minterm, and $x \vee \neg y \vee \neg z \vee w$ is a maxterm. The formula $x \wedge \neg y \vee \neg z \wedge w$ is neither a minterm nor a maxterm. The formulas x and $\neg y$ are both a minterm and a maxterm.

The formula $(x \wedge y \wedge \neg z) \vee (\neg x \wedge \neg r) \vee \neg z$ is in DNF, but is not in CNF. The formula $(x \vee y \vee \neg z) \wedge (\neg x \vee \neg r) \wedge \neg z$ is in CNF, but it is not in DNF. The formulas $(x \wedge y \wedge \neg z) \rightarrow (x \wedge y)$ and $((x \vee y) \wedge \neg z) \vee (x \wedge z)$ are neither in DNF nor in CNF. The formulas $x \wedge \neg y \wedge \neg z$ and $\neg x \vee \neg y \vee \neg z$ are both in DNF and in CNF. **End of Example 5.3**

Notice that being in DNF or CNF is a *syntactic* property of a formula — it has to do with the particular format of the formula, not with what the formula says. In fact, *every* formula can be rewritten, equivalently, in each of these forms. More precisely, given any propositional formula P, there is a DNF formula P^D that is logically equivalent to P, and an CNF formula P^C that is logically equivalent to P. Hence, whenever we are dealing with a propositional formula, it is always possible to replace it by one that says exactly the same thing (i.e., is logically equivalent to it), and has the special form required by DNF or CNF. Being able to focus only on formulas with this special format is useful for a variety of reasons. We will see some applications in Sections 5.10 and 5.11.

5.9.1 DNF formulas

Let us first examine, by means of an example, how we can find a DNF formula that is logically equivalent to a given one. Say that the given formula is $P = x \vee y \rightarrow \neg x \wedge z$. Following is the truth table of P.

x	y	z	$x \vee y \rightarrow \neg x \wedge z$
0	0	0	1
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	0

Viewed in this manner we can think of P as asserting that it is a formula satisfied by exactly three truth assignments to x, y and z — the ones that correspond to rows 1, 2 and 4 in the truth table above. Thus P is logically equivalent to a formula that asserts:

I am satisfied by the truth assignment in row 1,
or I am satisfied by the truth assignment in row 2,
or I am satisfied by the truth assignment in row 4.

The truth assignment in row 1 makes all of x, y and z false; so the propositional formula $\neg x \wedge \neg y \wedge \neg z$ is satisfied by this truth assignment and by no other truth assignment. Similarly, the truth assignment in row 2 makes x, y false, and z true; so the propositional formula $\neg x \wedge \neg y \wedge z$ is satisfied by this truth assignment and by no other. Finally, the truth assignment in row 4 makes x false, and y, z true; so the propositional formula $\neg x \wedge y \wedge z$ is satisfied by this truth assignment and by no other. Note that each of these formulas is a conjunction of literals — i.e., a minterm. A formula that is satisfied by one of these three truth assignments and by no other truth assignment is the disjunction of the three formulas that correspond to the three truth assignments:

$$(\neg x \wedge \neg y \wedge \neg z) \vee (\neg x \wedge \neg y \wedge z) \vee (\neg x \wedge y \wedge z)$$

Note that this is a disjunction of minterms, i.e., a DNF formula. Since, by construction, this formula is satisfied by precisely the same truth assignments as P, it is a DNF formula that is logically equivalent to P.

We can generalise from this example to arrive at the following theorem.

Theorem 5.16 *For every propositional formula P, there is a propositional formula $\hat{P}$ in DNF that is logically equivalent to P.*

PROOF. (This is an example or proof by cases.) There are two cases to consider:

CASE 1. *No truth assignment satisfies P:* Then let $\hat{P} = x_1 \wedge \neg x_1$. Clearly $\hat{P}$ is in DNF and it is easy to verify that all truth assignments falsify $\hat{P}$, so that P and $\hat{P}$ are logically equivalent, as wanted.

CASE 2. *At least one truth assignment satisfies P:* Let $\tau_1, \tau_2, \ldots, \tau_k$ be all the truth assignments to the propositional variables $x_1, x_2, \ldots, x_n$ of P that satisfy P.

For any truth assignment τ to the propositional variables $x_1, x_2, \ldots, x_n$, let C_τ be the formula

$$C_\tau = \ell_1 \wedge \ell_2 \wedge \ldots \wedge \ell_n$$

where each ℓ_i, $1 \leq i \leq n$, is the literal defined as follows:

$$\ell_i = \begin{cases} x_i, & \text{if } \tau(x_i) = 1 \\ \neg x_i, & \text{if } \tau(x_i) = 0 \end{cases}$$

Note that C_τ, being a conjunction of literals, is a minterm. It is straightforward to see that

$$C_\tau \text{ is satisfied by } \tau \text{ and only by } \tau \qquad (*)$$

Let $\hat{P} = C_{\tau_1} \vee C_{\tau_2} \vee \ldots \vee C_{\tau_k}$. $\hat{P}$ is in DNF, because every C_{τ_i} is a minterm. (Note that, by the assumption of Case 2, we know that $k \geq 1$, so that $\hat{P}$ is really a formula!) It remains to show that P and $\hat{P}$ are logically equivalent. That is, that any truth assignment that satisfies P must also satisfy $\hat{P}$ and, conversely, any truth assignment that satisfies $\hat{P}$ must also satisfy P. These two facts are shown in the next two paragraphs.

Let τ be any truth assignment τ that satisfies P. Therefore, τ must be one of $\tau_1, \tau_2, \ldots, \tau_k$. By $(*)$, τ satisfies C_τ. Since $\hat{P}$ is a disjunction of minterms one of which (namely C_τ) is satisfied by τ, it follows that $\hat{P}$ is satisfied by τ.

Conversely, let τ be any truth assignment that satisfies $\hat{P}$. Since $\hat{P}$ is the disjunction of $C_{\tau_1}, C_{\tau_2}, \ldots, C_{\tau_k}$, τ must satisfy at least one of these formulas. Say it satisfies C_{τ_i}, where $1 \leq i \leq k$. By $(*)$, the only truth assignment that satisfies C_{τ_i} is τ_i. Therefore, $\tau = \tau_i$. But, by definition, each of $\tau_1, \tau_2, \ldots, \tau_k$ satisfies P. Thus, τ satisfies P. □

5.9.2 CNF formulas

Now suppose we wanted to find a CNF (rather than a DNF) formula equivalent to the formula $P = x \vee y \rightarrow \neg x \wedge z$. Consider its truth table, which was given earlier. We arrived at a logically equivalent DNF formula, by thinking of P as asserting that it is satisfied by exactly three truth assignments. Now, we will arrive at a logically equivalent CNF formula, by thinking of P, instead, as asserting that it is *falsified* — i.e., it is *not* satisfied — by exactly five truth assignments — those corresponding to rows 3, 5, 6, 7 and 8 in its truth table. This is a formula that is falsified by exactly the same truth assignments that falsify P, and hence it is logically equivalent to P. The formula we have in mind will say, effectively,

I am not satisfied by the truth assignment in row 3,
and I am not satisfied by the truth assignment in row 5,
and I am not satisfied by the truth assignment in row 6,
and I am not satisfied by the truth assignment in row 7,
and I am not satisfied by the truth assignment in row 8.

We have already seen that the propositional formula $\neg x \wedge y \wedge \neg z$ says "I am satisfied by the truth assignment in row 3". The negation of this, $\neg(\neg x \wedge y \wedge \neg z)$, says, "I am *not* satisfied by the truth assignment in row 3". By applying DeMorgan's law, and the law of double negation, we can see that the above formula is logically equivalent to $x \vee \neg y \vee z$. This formula is a maxterm (a disjunction of literals), that says "I am not satisfied by the truth assignment in row 3". We can obtain, in a similar fashion, a maxterm that says "I am not satisfied by the truth assignment in row ...", for each of the remaining rows (5, 6, 7 and 8). By taking the conjunction of these maxterms we get a CNF formula,

$$(x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z) \wedge (\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

which says "I am not satisfied by any of the truth assignments is row 3, 5, 6, 7 or 8". As we argued earlier, this is logically equivalent to P.

We can generalise from this example to arrive at the following theorem.

Theorem 5.17 *For every propositional formula P, there is a propositional formula $\hat{P}$ in CNF that is logically equivalent to P.*

The proof of this theorem is analogous to the proof of Theorem 5.16 (with the modifications suggested by the preceding example), and is left as an exercise.

5.10 Boolean functions and the design of digital circuits

A ***Boolean function*** of n inputs (or arguments) $x_1, x_2, \ldots, x_n$ is a function $f : \{0,1\}^n \rightarrow \{0,1\}$ — i.e., a formula that takes n binary values as input and produces one binary value as output. Here are some examples of Boolean functions of three inputs x, y and z: $\mathit{Majority}(x, y, z)$ is a Boolean function whose output is the majority value among its inputs. $\mathit{Agreement}(x, y, z)$

is a Boolean function whose output is 1 if all three inputs are identical (all 0 or all 1), and 0 otherwise. *Parity*(x, y, z) outputs the parity of its three input bits. That is,

$$\mathit{Parity}(x,y,z) = \begin{cases} 0 & \text{if the number of inputs that are 1 is even} \\ 1 & \text{otherwise} \end{cases}$$

Definition 5.18 *A propositional formula P with propositional variables $x_1, x_2, \ldots, x_n$* ***represents*** *a Boolean function f of n arguments $x_1, x_2, \ldots, x_n$, if and only if for any truth assignment τ, τ satisfies P when $f(\tau(x_1), \ldots, \tau(x_n)) = 1$ and τ falsifies P when $f(\tau(x_1), \ldots, \tau(x_n)) = 0$.*

For example, the propositional formula $(x \leftrightarrow y) \wedge (y \leftrightarrow z)$ represents the Boolean function *Agreement*(x, y, z) defined above. The same function is also represented by any propositional formula that is logically equivalent to the above, such as, $(x \wedge y \wedge z) \vee \neg(x \vee y \vee z)$.

In fact, ***any*** Boolean function is represented by some propositional formula. To see this, note that a Boolean function f of n inputs can be viewed as a truth table with $n + 1$ columns (one for each input x_i, and one for the output $f(x_1, \ldots, x_n)$, and 2^n rows (one for each possible assignment of binary values to $x_1, x_2, \ldots, x_n$). In the first n columns of the each row we write some combination of values for the inputs $x_1, x_2, \ldots, x_n$, respectively, and in the $(n + 1)$st column we write the value of $f(x_1, \ldots, x_n)$ for the specified input values.

For example, the Boolean function *Majority*(x, y, z) can be written, in truth-table form, as follows:

x	y	z	*Majority*(x, y, z)
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

We just learned how to write down a DNF (or CNF) propositional formula whose truth table is identical to a given truth table. For instance, the *Majority*(x, y, z) function is represented by the following DNF formula

$$P_1 = (\neg x \wedge y \wedge z) \vee (x \wedge \neg y \wedge z) \vee (x \wedge y \wedge \neg z) \vee (x \wedge y \wedge z)$$

as well as by the following CNF formula

$$P_2 = (x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (\neg x \vee y \vee z).$$

Thus, a general procedure for obtaining a propositional formula that represents a given Boolean function is:

1. Write down the Boolean function in truth-table form.

2. Write a CNF (or DNF) formula for that truth table.

There are many different formulas that represent a given Boolean function, and some of them are not in DNF or CNF. For instance, in addition to P_1 and P_2, the *Majority* function, is also represented by the following two propositional formulas, neither of which is in DNF or CNF:

$$P_3 = (\neg x \rightarrow (y \wedge z)) \wedge (x \rightarrow (y \vee z)) \qquad \text{and} \qquad P_4 = (\neg x \wedge (y \wedge z)) \vee (x \wedge (y \vee z)).$$

From the definitions of logical equivalence and of what it means for a formula to represent a Boolean function, we can immediately conclude:

Theorem 5.19 *Let P be a propositional formula that represents a Boolean function f. A propositional formula Q also represents f if and only if Q is logically equivalent to P.*

We have just shown that every Boolean function can be represented by a DNF or CNF propositional formula. Since DNF and CNF formulas only contain the connectives $\neg$, $\wedge$ and $\vee$ we have the following:

Theorem 5.20 *For each Boolean function f there is a propositional formula that represents f and uses only connectives in $\{\neg, \wedge, \vee\}$.*

This simple and somewhat arcane fact is actually the basis of digital circuit design — an extremely practical matter!

In digital circuit design, we are given a ***functional specification*** of a desired circuit, and we are required to construct a ***hardware implementation*** of that circuit. We now explain what we mean by "functional specification" and by "hardware implementation". A specification of a digital circuit consists of three things: (a) the number n of binary inputs to the circuit, (b) the number m of binary output from the circuit, and (c) a description of what the circuit does — i.e., what the outputs must be for each setting of the inputs. Part (c) of the specification can be given as a collection of m Boolean functions, telling us how each of the output bits depends on the input bits.

The required hardware implementation of the circuit consists of electronic devices called ***logical gates***. The inputs to these devices are wires whose voltage level represents a binary value: voltage above a certain threshold represents the value 1 (true), while voltage below that threshold represents the value 0 (false). Each logical gate is capable of performing a logical operation on its input wire(s). An ***inverter***, implements the $\neg$ operation. It has a single wire as input, and its output is 1 (i.e., high voltage on the output wire) if its input is 0 (i.e., low voltage on the input wire); the output of the inverter is 0 if its input is 1. Similarly, an ***and gate*** implements the $\wedge$ operation. This gate has two wires as input, and its output is 1 if both its inputs are 1; otherwise, its output is 0. Finally, an ***or gate*** implements the $\vee$ operation. This gate also has two wires as input; its output is 1 if at least 1 of its inputs are 1; otherwise, its output is 0. These three logical gates have a diagrammatic representation, shown in Figure 5.2.

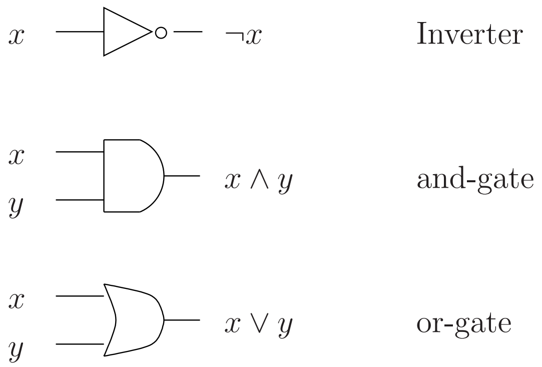


Figure 5.2: Diagrammatic representation of logical gates

By hooking up logical gates together appropriately, we can "implement" any propositional formula that contains only the connectives ¬, ∧ and ∨. For example, the diagram in Figure 5.3 shows how to connect gates together in order to create a circuit that implements formula P_4. Since P_4 represents the Boolean function *Majority*(x, y, z), the circuit shown diagrammatically in Figure 5.3 is a hardware implementation of the *Majority* function. As we can see, this circuit has three inputs, labeled x, y and z, and produces as its output the function *Majority*(x, y, z).

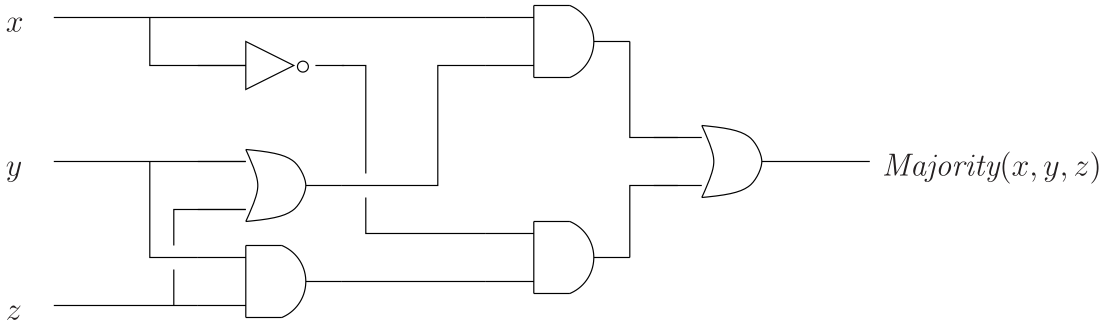


Figure 5.3: A circuit that computes *Majority*(x, y, z)

The overall process of designing a digital circuit from a functional specification then, consists of the following steps:

1. Convert the specification of the circuit (i.e., a given Boolean function) into a propositional formula that represents that function and uses only ¬, ∧ and ∨ connectives.
2. Convert the propositional formula into a circuit.

The process of constructing a circuit for a formula is an inductive one. We (recursively) construct circuits for the subformulas that make up the formula, and then connect these together by the appropriate gate. In the example shown in Figure 5.3 every gate has only one outgoing wire (the "fan-out" of each gate is one). This need not be the case, in general. If the same subformula appears several times in a formula, we need only compute its value once, and use multiple output wires to carry the computed value to all the gates at which it is needed. This is illustrated in Figure 5.4, which shows a circuit for the formula $((w \wedge x) \vee \neg y) \vee (((w \wedge x) \wedge z) \wedge \neg y)$. Note that the subformula $(w \wedge x)$ appears twice, and so there are two wires carrying the output of the gate that computes this value. The same observation applies regarding the subformula $\neg y$.

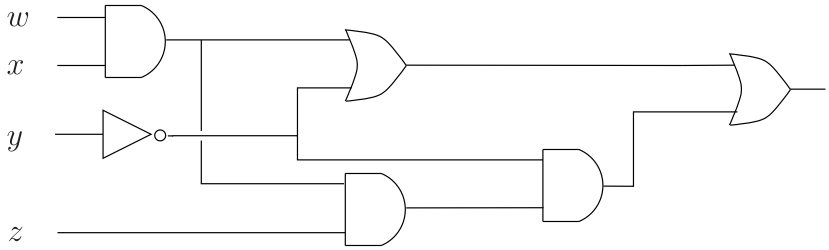


Figure 5.4: A circuit that reuses the outputs of common subexpressions

5.11 Complete sets of connectives

From Theorem 5.20 we know that any Boolean function can be represented by a formula that contains only the connectives $\neg$, $\wedge$ and $\vee$. Thus, in a sense, the other connectives, $\rightarrow$ and $\leftrightarrow$ are redundant. We can leave them out without loss of expressive power. Whenever we have to use $\rightarrow$ or $\leftrightarrow$ we can "paraphrase" it by using the other connectives instead. In particular, we can express $P \rightarrow Q$ as $\neg P \vee Q$; and we can express $P \leftrightarrow Q$ as $(P \wedge Q) \vee (\neg P \wedge \neg Q)$ (recall the $\rightarrow$ and $\leftrightarrow$ laws from Section 5.6). The idea that some set of connectives is sufficient to represent all others leads us to the notion of a "complete" set of connectives.

Definition 5.21 *A set of connectives is **complete** if and only if every Boolean function can be represented by a propositional formula that uses only connectives from that set.*

In view of this definition, Theorem 5.20 can be stated as follows:

Theorem 5.22 $\{\neg, \wedge, \vee\}$ *is a complete set of connectives.*

In fact, an even stronger result holds:

Theorem 5.23 $\{\neg, \wedge\}$ *is a complete set of connectives.*

The intuition as to why this theorem is true is as follows. First, we know from Theorem 5.22 that any Boolean function f can be represented by a propositional formula P_f that uses only connectives in $\{\neg, \wedge, \vee\}$. Next, we observe that by DeMorgan's law and the law of double negation, $(P \vee Q)$ is logically equivalent to $\neg(\neg P \wedge \neg Q)$. Thus, we can systematically replace each $\vee$ connective in P_f by $\neg$ and $\wedge$ connectives, while preserving logical equivalence. The end result of this process will be a formula that represents f (since it is logically equivalent to P_f — cf. Theorem 5.19) and uses only connectives in $\{\neg, \wedge\}$. The proof below makes this informal argument more rigorous. Notice how induction makes precise the process of "systematically replacing" all $\vee$ by $\neg$ and $\wedge$, and how the proof immediately suggests a simple recursive algorithm to carry out this systematic replacement.

Proof of Theorem 5.23. We define the set $\mathcal{G}$ of propositional formulas that use only connectives in $\{\neg, \wedge, \vee\}$, and the set $\mathcal{G}'$ of propositional formulas that use only connectives in $\{\neg, \wedge\}$. $\mathcal{G}$ is the smallest set that satisfies the following properties:

Basis: Any propositional variable is in $\mathcal{G}$.

Induction Step: If P_1 and P_2 are in $\mathcal{G}$, then so are $\neg P_1$, $(P_1 \wedge P_2)$ and $(P_1 \vee P_2)$.

$\mathcal{G}'$ is defined similarly, except that the induction step allows only for the first two constructions (i.e., $\neg P_1$ and $(P_1 \wedge P_2)$, but not for $(P_1 \vee P_2)$).

Let $S(P)$ be the following predicate about propositional formulas:

$$S(P): \qquad \text{there is a propositional formula } \hat{P} \in \mathcal{G}' \text{ that is logically equivalent to } P$$

We will use structural induction to prove that $S(P)$ holds for every formula $P \in \mathcal{G}$.

Basis: P is a propositional variable. By the basis of the definition of $\mathcal{G}'$, $P \in \mathcal{G}'$. Since P is logically equivalent to itself, it follows that by taking $\hat{P} = P$, $S(P)$ holds, in this case.

Induction Step: Let P_1 and P_2 be formulas in $\mathcal{G}$ such that $S(P_1)$ and $S(P_2)$ hold. That is, there are formulas $\hat{P}_1$ and $\hat{P}_2$ in $\mathcal{G}'$ that are logically equivalent to P_1 and P_2, respectively. We must show that $S(P)$ holds for each of the ways in which a formula P in $\mathcal{G}$ may be constructed out of P_1 and P_2. From the induction step of the definition of $\mathcal{G}$, there are three ways in which P can be constructed from P_1 and P_2.

Case 1. P is of the form $\neg P_1$. Let $\hat{P} = \neg\hat{P}_1$. Since $\hat{P}_1$ is logically equivalent to P_1 (by induction hypothesis), $\neg\hat{P}_1$ is logically equivalent to $\neg P_1$. Thus, $\hat{P}$ is logically equivalent to P. Furthermore, since $\hat{P}_1$ is in $\mathcal{G}'$ (also by induction hypothesis), it follows (by the induction step of the definition of $\mathcal{G}'$) that $\neg\hat{P}_1$, i.e., $\hat{P}$, is in $\mathcal{G}'$ as well. Therefore, $\hat{P}$ has the desired properties and so $S(P)$ holds, in this case.

Case 2. P is of the form $(P_1 \wedge P_2)$. Let $\hat{P} = (\hat{P}_1 \wedge \hat{P}_2)$. Since $\hat{P}_1$ and $\hat{P}_2$ are logically equivalent to P_1 and P_2, respectively (by induction hypothesis), $(\hat{P}_1 \wedge \hat{P}_2)$ is logically equivalent to $(P_1 \wedge P_2)$. Thus, $\hat{P}$ is logically equivalent to P. Furthermore, since $\hat{P}_1$ and $\hat{P}_2$ are in $\mathcal{G}'$ (also by induction hypothesis), it follows that $(\hat{P}_1 \wedge \hat{P}_2)$, i.e., $\hat{P}$, is in $\mathcal{G}'$ as well. Therefore, $\hat{P}$ has the desired properties and so $S(P)$ holds, in this case.

Case 3. P is of the form $(P_1 \vee P_2)$. Let $\hat{P} = \neg(\neg\hat{P}_1 \wedge \neg\hat{P}_2)$. By DeMorgan's law and the law of double negation, $\neg(\neg\hat{P}_1 \wedge \neg\hat{P}_2)$ is logically equivalent to $(\hat{P}_1 \vee \hat{P}_2)$. Since $\hat{P}_1$ and $\hat{P}_2$ are logically equivalent to P_1 and P_2, respectively (by induction hypothesis), $(\hat{P}_1 \vee \hat{P}_2)$ is logically equivalent to $(P_1 \vee P_2)$. Thus, $\hat{P}$ is logically equivalent to P. Furthermore, since $\hat{P}_1$ and $\hat{P}_2$

are in $\mathcal{G}'$ (also by induction hypothesis), it follows that $\neg(\neg\hat{P}_1 \wedge \neg\hat{P}_2)$, i.e., $\hat{P}$, in in $\mathcal{G}'$ as well. Therefore, $\hat{P}$ has the desired properties and so $S(P)$ holds, in this case.

We have thus shown that for every propositional formula that uses only connectives in $\{\neg, \wedge, \vee\}$, there is a logically equivalent formula that uses only connectives in $\{\neg, \wedge\}$. From this, and the fact that $\{\neg, \wedge, \vee\}$ is complete, it follows that $\{\neg, \wedge\}$ is complete as well. □

By a similar proof (left as an exercise), it can be shown that:

Theorem 5.24 $\{\neg, \vee\}$ *is a complete set of connectives.*

Theorems 5.23 and 5.24 suggest that we can get away with surprisingly few connectives: We only need $\neg$ and only one of $\wedge$ or $\vee$. Are there other small sets of connectives that are complete? It is not difficult to show, using similar techniques, that $\{\neg, \rightarrow\}$ is also complete. (Prove this!) On the other hand, it turns out that $\{\wedge, \vee\}$ is *not* a complete set of connectives. To prove this, we need to show that there is some Boolean function that cannot be represented using only $\wedge$ and $\vee$. Equivalently, we need to show that there is some propositional formula P (involving any connectives we like) for which there is *no* logically equivalent formula that uses only $\wedge$ and $\vee$.

Theorem 5.25 $\{\wedge, \vee\}$ *is not a complete set of connectives.*

The intuition as to why this theorem is true is as follows. Consider the truth assignment that satisfies all propositional variables. The conjunction of true propositions is a true proposition, and the disjunction of true propositions is also a true proposition. Therefore, any formula that is constructed using only conjunctions and disjunctions will be satisfied by this truth assignment. We know, however, that there are formulas that are not satisfied by the truth assignment that satisfies all propositional variables: for example, the negation of a propositional variable such as $\neg x$; or, more dramatically, any unsatisfiable formula such as $x \wedge \neg x$. Therefore a formula such as $\neg x$ cannot be equivalent to *any* formula that is constructed using only conjunctions and disjunctions, since their truth values will differ in at least one truth assignment (namely the one that satisfies all propositional variables). This intuition is made more precise in the proof below.

Proof of Theorem 5.25. Fix the set of propositional variables. We define, inductively, the set $\mathcal{H}$ of propositional formulas with these propositional variables that use only connectives in $\{\wedge, \vee\}$. $\mathcal{H}$ is the smallest set so that:

Basis: Every propositional variable is in $\mathcal{H}$.

Induction Step: If P_1 and P_2 are in $\mathcal{H}$, then so are $(P_1 \wedge P_2)$ and $(P_1 \vee P_2)$.

Let τ be the truth assignment that makes every propositional variable true. That is, $\tau(x) = 1$ for every propositional variable x. Let $S(P)$ be the following predicate about propositional formulas:

$$S(P): \qquad \tau \text{ satisfies } P$$

We will use structural induction to prove that $S(P)$ holds for every $P \in \mathcal{H}$.

Basis: P is a propositional variable. Obviously, τ satisfies P. So $S(P)$ holds in this case.

Induction Step: Assume that P_1 and P_2 are formulas in $\mathcal{H}$ such that $S(P_1)$ and $S(P_2)$ hold; i.e., τ satisfies both P_1 and P_2. We must show that $S(P)$ holds for each of the ways in which a formula P in $\mathcal{H}$ may be constructed out of P_1 and P_2. From the induction step of the definition of $\mathcal{H}$, there are two ways in which P can be constructed from P_1 and P_2.

Case 1. P is of the form $(P_1 \wedge P_2)$. Since τ satisfies P_1 and P_2 (by induction hypothesis), τ satisfies $(P_1 \wedge P_2)$, i.e., it satisfies P. So $S(P)$ holds in this case.

Case 2. P is of the form $(P_1 \vee P_2)$. Since τ satisfies P_1 and P_2 (by induction hypothesis), τ satisfies $(P_1 \vee P_2)$, i.e., it satisfies P. So, $S(P)$ holds in this case.

We have therefore shown that no formula in $\mathcal{H}$ is falsified by τ. Now consider the formula $\neg x$, where x is any propositional variable. This formula is falsified by τ. Since every formula in $\mathcal{H}$ is satisfied by τ and $\neg x$ is falsified by τ, it follows that no formula in $\mathcal{H}$ (i.e., no formula that uses only connectives in $\{\wedge, \vee\}$) can be logically equivalent to $\neg x$. Hence, $\{\wedge, \vee\}$ is not a complete set of connectives. □

We now introduce a new propositional connective, denoted $|$, known as ***nand*** or, more classically, ***Sheffer's stroke***. This is a binary connective, so if P and Q are propositional formulas we also define $(P \mid Q)$ to be a propositional formula. Informally, this is intended to say that P and Q are not both true. More precisely, the table below gives the truth value of $(P \mid Q)$ under a truth assignment τ, given the truth values of P and Q under τ.

P	Q	$(P \mid Q)$
0	0	1
0	1	1
1	0	1
1	1	0

This is a very interesting connective because, as it turns out, it is complete all by itself!

Theorem 5.26 $\{\,|\,\}$ *is a complete set of connectives.*

We sketch the basic idea of the proof for this theorem. By Theorem 5.23, it is enough to show that for any formula that uses only connectives in $\{\neg, \wedge\}$ there is an equivalent one that uses only $|$. Informally, it is enough to express $\neg P$ and $(P \wedge Q)$ using only $|$. But this is possible since (as it can be readily verified) $\neg P$ is logically equivalent to $(P \mid P)$, and $(P \wedge Q)$ is logically equivalent to $(P \mid Q) \mid (P \mid Q)$. You are encouraged to produce a more rigorous proof of Theorem 5.26 along the lines of the proof of Theorem 5.23, where the key ideas for the induction step are embodied in the two equivalences above.

Theorem 5.26 has interesting practical implications. It turns out that it is easy to construct a logical gate that implements the nand operation. This theorem then implies that we can implement *any* Boolean function with circuits that only have that one kind of gate. We can do away with $\neg$, $\wedge$ and $\vee$ gates and only manufacture $|$ gates.

For reasons that have to do with electronics, not mathematics, it turns out that some logical operations (such as $|$, $\neg$, $\wedge$ and $\vee$) can be easily implemented directly, while others (such as $\rightarrow$ and $\leftrightarrow$) are not so easy to implement directly. It takes about three times more microelectronic components out of which gates are made (transistors, diodes etc) to make a $\rightarrow$ or $\leftrightarrow$ gate, than to make a $|$, or $\vee$ gate. Of course, we can implement these other operations indirectly, by simple circuits of the easy-to-implement operations.

Another binary propositional connective is $\downarrow$, known as ***nor***. Informally, $P \downarrow Q$ is intended to say that P and Q are both false (i.e., that neither is true). The table below gives the truth value of $(P \downarrow Q)$.

P	Q	$(P \downarrow Q)$
0	0	1
0	1	0
1	0	0
1	1	0

Like $|$, this connective is also complete (see Exercise 8).

Another binary propositional connective is $\oplus$, called ***exclusive or***. $(P \oplus Q)$ is intended to mean that *exactly* one of P and Q is true. Unlike $\vee$ (which is sometimes called the ***inclusive or***, to underscore the difference) the exclusive or of two formulas is false if both formulas are true. The table below gives the truth value of $(P \oplus Q)$.

P	Q	$(P \oplus Q)$
0	0	0
0	1	1
1	0	1
1	1	0

Exercises 9-11 give some additional examples of complete and incomplete sets of connectives involving $\oplus$.

Exercises

1. Does the associativity law apply for the connectives $\rightarrow$ and $\leftrightarrow$? That is, are the formulas $((P \rightarrow Q) \rightarrow R)$ and $(P \rightarrow (Q \rightarrow R))$ logically equivalent (where P, Q and R are arbitrary propositional formulas)? How about the formulas $((P \leftrightarrow Q) \leftrightarrow R)$ and $(P \leftrightarrow (Q \leftrightarrow R))$?

2. Suppose P and Q are two *unsatisfiable* propositional formulas, and R is a *tautology.* For each of the statements below, determine whether it is true or false, and *justify your answer.*

(a) $P \leftrightarrow Q$ is a tautology.

(b) P logically implies R.

(c) R logically implies P.

(d) $P \leftrightarrow R$ is unsatisfiable.

3. Using only logical equivalences from Section 5.6 (and, in particular, using no truth tables), prove that

(a) $(x \leftrightarrow \neg y) \rightarrow z$ is logically equivalent to $(x \wedge y) \vee (\neg x \wedge \neg y) \vee z$.

(b) $(x \leftrightarrow \neg y) \rightarrow \neg(x \rightarrow y)$ is logically equivalent to $y \rightarrow x$.

(c) $x \wedge \neg y \rightarrow \neg z$ is logically equivalent to $x \wedge z \rightarrow y$.

4. For each of the following assertions state if it is true or false and justify your answer:

(a) $(x \rightarrow y) \wedge (x \rightarrow z)$ is logically equivalent to $x \rightarrow (y \wedge z)$.

(b) $(y \rightarrow x) \wedge (z \rightarrow x)$ is logically equivalent to $(y \wedge z) \rightarrow x$

5. Let $P_1, P_2, \ldots, P_n$ be arbitrary propositional formulas. Prove that, for any $n \geq 2$, the formula $P_1 \rightarrow (P_2 \rightarrow (\cdots \rightarrow (P_{n-1} \rightarrow P_n) \cdots))$ is logically equivalent to $(P_1 \wedge P_2 \wedge \cdots \wedge P_{n-1}) \rightarrow P_n$. (Hint: Use induction on n.)

6. True or false?

(a) A maxterm is both a CNF formula and a DNF formula.

(b) A minterm is both a CNF formula and a DNF formula.

(c) If a formula is both in CNF and in DNF, then it is either a minterm or a maxterm.

7. Write DNF and CNF formulas that are logically equivalent to $((x \rightarrow y) \vee (\neg x \wedge z)) \leftrightarrow (y \vee z)$.

8. Prove that $\{\downarrow\}$ is a complete set of connectives.

9. Prove that $\{\neg, \oplus, \leftrightarrow\}$ is not a complete set of connectives.

10. Prove that $\{\oplus, \rightarrow\}$ is a complete set of connectives.

11. Is $\{\oplus, \wedge, \vee\}$ a complete set of connectives? Prove your answer.

12. Let x and y be three-bit natural numbers. We can represent the value of x by using three propositional variables: x_2 (to denote the value of the most significant bit of x), x_1 (to represent the middle bit of x), and x_0 (to denote the value of the least significant bit of x). More precisely, a truth assignment τ to x_0, x_1 and x_2 represents the (three-bit) number $\tau(x_0) + 2\tau(x_1) + 4\tau(x_2)$. Similarly, we can represent the value of y by using three propositional variables y_2, y_1, and y_0.

The sum $x + y$ is a four-bit natural number. For each of the four bits in the sum $x + y$, write a propositional formula (with propositional variables $x_0, x_1, x_2, y_0, y_1, y_2$) which represents the value of that bit. You should write four formulas F_0, F_1, F_2, F_3 where F_0 represents the low-order bit of the sum and F_3 represents the high-order bit. Explain how you obtained your propositional formulas. Your formulas may contain any of the propositional connectives we have discussed ($\neg, \wedge, \vee, \rightarrow, \leftrightarrow, \oplus, \mid, \downarrow$).

Example: Consider the truth assignment which gives x the value 6 and y the value 5 That is, the variables $x_2, x_1, x_0, y_2, y_1, y_0$ get, respectively, the values $1, 1, 0, 1, 0, 1$. Since the sum $6 + 5 = 11$, the values of the formulas F_3, F_2, F_1, F_0 under this truth assignment should be (respectively) $1, 0, 1, 1$.

Hint: This can certainly be done with a truth table that involves six propositional variables (and therefore has 64 rows). This is too much boring work. Instead, think about how to add numbers in binary, and represent the algorithm in terms of logical operations. This will naturally lead you to the four formulas, via a probably faster (and surely intellectually more interesting) path.

13. In this exercise we will develop a proof of the Unique Readability Theorem (see Theorem 5.4) in three steps.

(a) The key to this proof is to establish some simple syntactic properties of propositional formulas. Using structural induction on the definition of propositional formulas, prove the following fact: For any propositional formula P,

  (i) P has the same number of '(' and ')' symbols.

  (ii) Every proper prefix of P that has at least one '(' symbol contains more '(' than ')' symbols.

  (iii) Every proper prefix of P that has no '(' symbol is a string of zero or more '$\neg$' symbols.

  Hint: Prove the three parts simultaneously in the same induction.

(b) Use Part (a) to prove the following fact: For any propositional formula P, no proper prefix of P is a propositional formula.

(c) Use Part (b) to prove the Unique Readability Theorem.

Chapter 6

PREDICATE LOGIC

6.1 Introduction

6.1.1 Predicates and relations

Predicate logic is the formalisation of reasoning that involves predicates. It is a generalisation of propositional logic because, as we will see shortly, a proposition is a very special kind of predicate.

A ***predicate*** is a Boolean-valued function. The set D of possible values for a predicate's arguments is called its ***domain of discourse***. The number $n > 0$ of a predicate's arguments is called its ***arity***. Thus, an n-ary predicate with domain of discourse D is simply a function

$$P : \underbrace{D \times \cdots \times D}_{n \text{ times}} \to \{0, 1\}$$

For example, "x is a sibling of y" is a binary (arity 2) predicate whose domain of discourse is the set of all people. This predicate is true if x is Groucho and y is Harpo (two of the Marx brothers), or if x is Emily and y is Charlotte (the Brontë sisters), or if x is Apollo and y is Artemis (stretching our domain of discourse to include mythological figures). On the other hand, this predicate is false if x is Cleopatra and y is Anthony, because these two were not a sister-and-brother pair. Two unary (arity 1) predicates with the same domain of discourse are "x is female" and "x is male", and a ternary (arity 3) predicate with the same domain of discourse is "x and y are the parents of z".

Similarly, "$x \leq y$" is a binary predicate whose domain of discourse is the set of, say, natural numbers; "x is a prime number" is a unary predicate, and "$x + y < z$" is a ternary predicate with the same domain of discourse.

We can think of a predicate as a relation, and vice-versa. Recall that, mathematically speaking, an n-ary ***relation*** over domain D is a subset of the n-fold Cartesian product $D \times \cdots \times D$. For example, we can view the predicate "x is a sibling of y" as the subset of the Cartesian product $P \times P$ (where P is the set of people), consisting of all pairs (x, y) such that x is a sibling of y. So, $(\text{Groucho}, \text{Harpo})$ and $(\text{Emily}, \text{Charlotte})$ belong to this set, but $(\text{Cleopatra}, \text{Anthony})$ does not.

In general, we can view an n-ary predicate $A : D \times \cdots \times D \to \{0,1\}$ as the relation $R_A = \{(d_1, \ldots, d_n) : A(d_1, \ldots, d_n) = 1\}$. Conversely, we can view an n-ary relation $R \subseteq D \times \cdots \times D$ as the predicate $A_R : D \times \cdots \times D \to \{0,1\}$, where $A_R(d_1, \ldots, d_n) = 1$ if and only if $(d_1, \ldots, d_n) \in R$.

6.1.2 Combining predicates to form more complex ones

Let us write $S(x,y)$, $M(x)$, $F(x)$, and $P(x,y,z)$ to denote the sibling-of, is-male, is-female and parents-of predicates described earlier. We can combine these predicates using the connectives of propositional logic to express more complex predicates. For instance, we can write the predicate

$$S(x,y) \wedge M(x)$$

which holds true if x is a sibling of y and x is male (for instance, it holds true if x is Groucho and y is Harpo; it is false if x is Artemis and y is Apollo; it is also false if x is Anthony and y is Cleopatra). As an other example, we can write the predicate

$$S(x,y) \wedge (M(x) \leftrightarrow F(y))$$

which is true of opposite-sex siblings x and y.

In addition to using our familiar propositional connectives, there is another way to construct new predicates, namely by using the ***existential*** and ***universal quantifiers***. Suppose A is a predicate of x (and possibly other variables). We can construct two new predicates as follows:

- $\exists x\, A$ — this predicate is true if there is *some* x so that A holds; the symbol $\exists$ is called the ***existential quantifier***.
- $\forall x\, A$ — this predicate is true if A holds for *all* x; the symbol $\forall$ is called the ***universal quantifier***.

Take, for example, the predicate $S(x,y)$ — i.e., x is a sibling of y. We can use the existential quantifier and this predicate to express the predicate "x has a sibling". To say that x has a sibling is to say that someone is x's sibling; or, to put in yet another way, that there is someone, call that person y, so that x is a sibling of y. Thus, we can express the predicate "x has a sibling" as

$$\exists y\, S(x,y).$$

The variable y that appears immediately after the existential quantifier names an object (in our example, x's sibling), whose existence the predicate asserts. Notice that the predicate $\exists y\, S(x,y)$ is a predicate of *only one* argument, namely x. This is clear if we think of this predicate as expressing "x has a sibling" (which is obviously a predicate of just x). However, the symbolic representation $\exists y\, S(x,y)$ may at first appear to obscure this fact since it contains two variables, x and y. As we will see later (Sections 6.2.3 and 6.3.3, and Theorem 6.7) x and y play quite different roles in this formula. x is the variable which this predicate is "about" while y is a "dummy" variable whose purpose is to name an object to which a quantifier refers. The predicate "x has a sibling" is expressed equally well as $\exists z\, S(x,z)$: it is immaterial which particular dummy variable, y or z or something else, we use to name the quantified object.

Let $L(x, y)$ be the binary predicate "x loves y". We can use the universal quantifier and this predicate to express the predicate "everybody loves x" as $\forall y\, L(y, x)$. This is a predicate of only one argument, x. It is true of any individual x whom everybody loves, and false of any other individual. The variable y that appears in the symbolic expression for this predicate, $\forall y\, L(y, x)$, is a dummy variable. As in the example of the previous paragraph, its role is merely to name the object to which the quantifier refers.

The result of applying a quantifier to a predicate is another predicate. Thus, we can keep applying repeatedly any of the available constructs (propositional connectives and quantifiers) to create more and more complex predicates. Following are some examples involving the predicates introduced earlier, along with a description, in English, of what the resulting predicate expresses.

- $\exists y\, (S(x, y) \wedge F(y))$ — x has a sister.
- $\forall y\, (S(x, y) \rightarrow F(y))$ — x has only sisters (or no siblings at all).
- $\exists y\, S(x, y) \wedge \forall y\, (S(x, y) \rightarrow F(y))$ — x has at least one sibling and has only sisters.
- $\exists v \exists w\, (M(x) \wedge S(x, v) \wedge P(v, w, y))$ — x is y's uncle.
- $\exists u \exists v \exists w\, (P(x, u, v) \wedge P(v, w, y))$ — x is y's grandparent.

6.2 The syntax of predicate logic

Expressions such as those we saw in the previous section are called ***predicate formulas***, or ***first-order formulas***. We will now give the precise rules for correctly forming such expressions.

6.2.1 First-order languages

A ***first-order language*** consists of:

- An infinite set of ***variables*** $\{x, y, \ldots\}$[1]
- A set of ***predicate symbols*** $\{A_1, A_2, \ldots\}$; associated with each predicate symbol is a positive integer indicating its arity.
- A set of ***constant symbols*** $\{\boldsymbol{c}_1, \boldsymbol{c}_2, \ldots\}$

The symbols of a first-order language, together with the propositional connectives ($\neg$, $\wedge$, $\vee$, $\rightarrow$ etc), quantifiers, parentheses and comma constitute the basic "vocabulary" of first-order formulas.

For example, to create a vocabulary for expressing predicates such as those we discussed in the previous section we might define the following first-order language:

[1] There are technical reasons why it is convenient to assume that we have an unlimited supply of variables at our disposal.

- Variables: An infinite set that includes u, v, w, x, y, z.
- Predicate symbols: P (of arity 3), S (of arity 2), L (of arity 2), M (of arity 1) and F (of arity 1).
- Constant symbols: none.

We will subsequently refer to this particular first-order language as the ***language of familial relationships***, and denote it as $\mathcal{LF}$.

As another example, suppose we want to express various facts concerning natural numbers that involve the predicates $x+y=z$, $x \cdot y = z$, $x < y$, $x = y$, x is even, and x is odd. To create the appropriate vocabulary to express such facts, we might define the following first-order language:

- Variables: An infinite set that includes x, y, z
- Predicate symbols: S (of arity 3), P (of arity 3), L (of arity 2) and $\approx$ (of arity 2).
- Constant symbols: $\boldsymbol{0}$, $\boldsymbol{1}$.

Here the intention is that S is a predicate symbol to denote the predicate "the sum of the first two arguments is equal to the third"; P denotes the predicate "the product of the first two arguments is equal to the third"; L denotes the predicate "the first argument is less than the second"; and $\approx$ denotes the predicate "the first argument is equal to the second".[2] Also, the intention is that the constant symbol $\boldsymbol{0}$ stands for the natural number zero, and the constant symbol $\boldsymbol{1}$ stands for the natural number one. We will subsequently refer to this particular first-order language as the ***language of arithmetic***, and denote it as $\mathcal{LA}$.

A ***first-order language with equality*** is simply one that includes, in its set of predicate symbols, the binary predicate symbol $\approx$. As we will see later (in Section 6.3), this predicate symbol has a special status. $\mathcal{LA}$ is a first-order language with equality, while $\mathcal{LF}$ is not.

It is important to emphasise that the choice of a first-order language amounts to the choice of certain sets of symbols. These symbols are not endowed with any particular meaning, although when we choose the symbols we may have in mind some particular interpretation for them. For example, in $\mathcal{LF}$ we had the predicate "loves" in mind for the (binary) predicate symbol L; while in $\mathcal{LA}$ we had the predicate "less than" in mind for the (binary) predicate symbol L. We will discuss in detail how we give meanings to the predicate and constant symbols of a first-order language in Section 6.3.

6.2.2 Formulas

Let us now fix a first-order language $\mathcal{L}$. A ***term*** (of $\mathcal{L}$) is a variable or a constant symbol of $\mathcal{L}$. An ***atomic formula*** (of $\mathcal{L}$) is an expression of the form $A(t_1, t_2, \ldots, t_n)$, where A is an n-ary predicate symbol of $\mathcal{L}$ and each t_i is a term of $\mathcal{L}$.

For example, $S(u, v)$ is an atomic formula of $\mathcal{LF}$; $L(x, y)$ and $P(x, \boldsymbol{1}, z)$ are atomic formulas of $\mathcal{LA}$.

Atomic formulas are the most primitive first-order formulas — just like propositional vari-

[2] We use the symbol $\approx$, rather than $=$, in order to emphasise the distinction between the formal symbol that is used in first-order formulas to express the predicate "is equal to", and the predicate itself which is usually written as $=$.

ables are the most primitive *propositional* formulas. The entire set of such formulas is defined inductively.

Definition 6.1 *The set of* ***first-order formulas*** *(of $\mathcal{L}$) is the smallest set such that:*

BASIS: *Any atomic formula (of $\mathcal{L}$) is in the set.*

INDUCTION STEP: *If F_1 and F_2 are in the set, and x is a variable of $\mathcal{L}$ then the following are also in the set: $\neg F_1$, $(F_1 \wedge F_2)$, $(F_1 \vee F_2)$, $(F_1 \rightarrow F_2)$, $(F_1 \leftrightarrow F_2)$, $\forall x\, F_1$, and $\exists x\, F_1$.*

In this definition, every time a binary propositional connective is used we are required to enclose the resulting formula in parentheses. This is necessary in order to ensure that there is only one way in which a complex formula is constructed from its constituent subformulas. The reason for this "unique readability" was discussed in the context of propositional formulas. We will adopt conventions analogous to those we used for propositional formulas which allow us to omit parentheses without creating ambiguities. We draw special attention to the following issue that arises from the fact that we often drop the outermost pair of parentheses from a formula: The formula $\exists x\, F_1 \vee F_2$, is an abbreviation of $(\exists x\, F_1 \vee F_2)$ (by omitting the outer parentheses) and is formed as the disjunction of $\exists x\, F_1$ and F_2. This is different from the formula $\exists x\, (F_1 \vee F_2)$ which is formed by existentially quantifying the disjunction $(F_1 \vee F_2)$. In the first formula the quantifier $\exists x$ applies only to F_1, while in the second it applies to both F_1 and F_2. Analogous remarks apply concerning the pair of formulas $\forall x\, F_1 \rightarrow F_2$ and $\forall x\, (F_1 \rightarrow F_2)$, and other similar constructs.

As with propositional formulas, it can be useful to view first-order formulas as trees. Figure 6.1 shows the tree that corresponds to a first-order formula of $\mathcal{LF}$. The entire formula F is represented by a tree whose subtrees represent the subformulas out of which F is constructed. The leaves of the tree correspond to atomic formulas that occur in F, and the internal nodes correspond to Boolean connectives and quantifiers. Note that the quantifiers, like negation, are unary connectives: a node that corresponds to a quantifier has a single child.

6.2.3 Free variables

A first-order formula is an expression that is supposed to represent a predicate. But a predicate of what? Let us look again at the formula $\exists y\, S(x,y)$ of the first-order language $\mathcal{LF}$. As we have seen, this formula represents the predicate "x has a sibling", and so it is about the variable x — but it is *not* about the variable y. In general, the variables that appear in a formula are of two kinds: those that name objects to which a quantifier refers (such as y in our example) and those that do not (such as x in our example). The latter are called ***free*** variables. In general, *a formula represents a predicate of its free variables.*

The precise definition of the set of free variables of a first-order formula F, *free*(F), is by structural induction. This definition is, in effect, a recursive program for computing the set *free*(F).

BASIS: F is an atomic formula, say $A(t_1, \ldots, t_n)$, where A is an n-ary predicate symbol and each t_i is a term. In this case, *free*(F) is the set of terms t_i, $1 \leq i \leq n$, that are variables. (Recall that terms are either variables or constant symbols.)

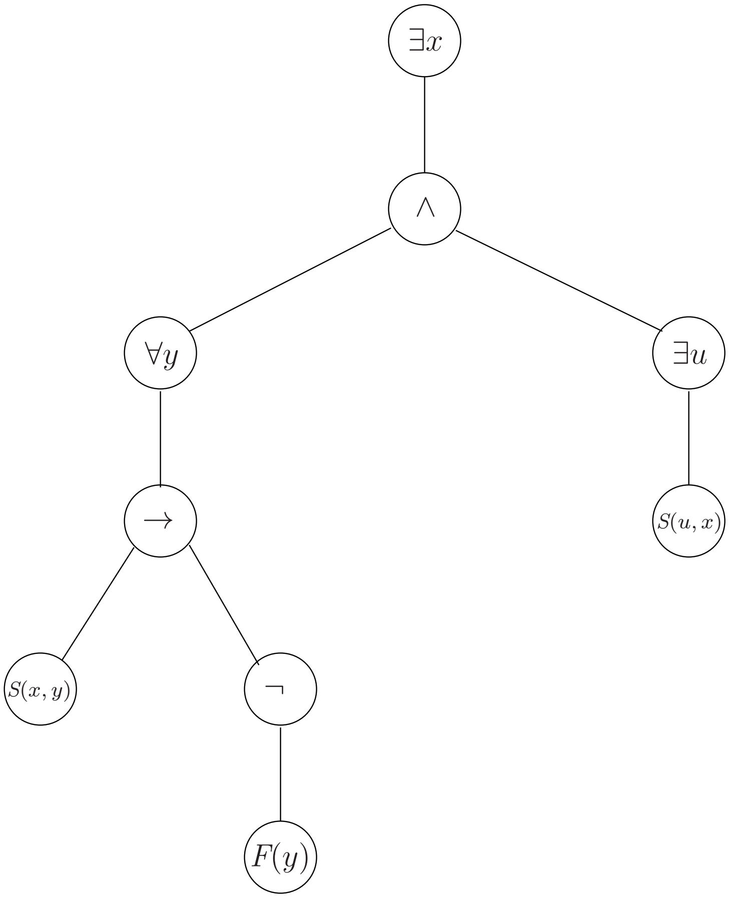


Figure 6.1: The parse tree of $\exists x \Big(\forall y \left(S(x,y)\rightarrow\neg F(y)\right) \wedge \exists u\, S(u,x)\Big)$

Induction Step: F is not an atomic formula. Then F is constructed from one or two formulas F_1, F_2, using a propositional connective or a quantifier. Assume, by induction, that we have determined the set of free variables of F_1 and F_2, *free*(F_1) and *free*(F_2). The set of free variables of F are as follows, depending on how F is obtained from F_1 and F_2.

- $F = \neg F_1$. Then *free*(F) = *free*(F_1).
- $F = (F_1 \wedge F_2)$ or $F = (F_1 \vee F_2)$ or $F = (F_1 \rightarrow F_2)$ or $F = (F_1 \leftrightarrow F_2)$. In each of these cases, *free*(F) = *free*$(F_1) \cup$ *free*(F_2).
- $F = \exists x\, F_1$ or $F = \forall x\, F_1$. In each of these cases, *free*(F) = *free*$(F_1) - \{x\}$.

If we apply this inductive definition to

$$\forall z \Big(P(x,y,z)\rightarrow\exists y \left(S(z,y) \wedge F(y)\right)\Big)$$

we will determine that the set of free variables of this formula is $\{x, y\}$.[3] What is interesting

[3] As we did in this formula, we will sometimes use different size parentheses as a visual aid to help identify

about this particular formula is that y is free in it, although it is not free in the subformula $\exists y\,(S(z, y) \wedge F(y))$. The y that appears in the antecedent of the conditional refers to a different object than the y that appears in the consequent. The former is free while the latter is not. Thus, we can speak not only about whether a variable is free in a formula, but whether a *particular occurrence of a variable* is free in a formula. More precisely,

Definition 6.2 *An* occurrence *of variable x is **free** in F if and only if it does* not *occur within a subformula of F of the form $\forall x\, E$ or $\exists x\, E$.*

A formula F that has no free variables (i.e., *free*$(F) = \emptyset$) is called a ***sentence***.

In terms of the tree representation of formulas, an occurrence of a variable x is free if the path from the node that contains the occurrence of the variable to the root contains no quantifier $\forall x$ or $\exists x$.

6.3 Semantics of predicate logic

So far we have defined formulas (of a first-order language $\mathcal{L}$) as strings of characters from a certain vocabulary that are constructed according to certain syntactic rules. We have not yet officially defined the meaning of these strings of characters. In this section, we define what it means for a first-order formula to be true or false.

As in propositional logic, we cannot determine the truth value of a formula, unless we are given some information that gives meaning to the basic vocabulary of our logic. In the case of propositional logic the necessary information is a truth assignment which gives meaning to the propositional variables. Now the situation is a little more complicated, because the basic vocabulary of a first-order language is richer.

6.3.1 An example

Suppose we are working in a first-order language consisting of some set of variables (including x and y), binary predicate symbols A and B, a unary predicate symbol C, and a constant symbol $\boldsymbol{c}$. Consider the formula $\exists y\,(A(x, y) \wedge B(\boldsymbol{c}, y) \wedge C(y))$ in this first-order language. To determine whether this formula is true or false we need some information:

1. We need to know what sorts of objects this formula is talking about; i.e., what are the possible values of the variables. The set of all such values is called the ***domain (of discourse)***.

2. We need to know the predicates to which the predicate symbols A, B and C correspond.

3. We need to know the domain element to which the constant symbol $\boldsymbol{c}$ refers.

4. Finally, we need to know the domain element to which x, the free variable of the formula, refers.

matching pairs of parentheses. Keep in mind, however, that in the first-order languages over which formulas are defined, there are *only two* parentheses symbols, '(' and ')' — not many pairs of symbols of different sizes.

Suppose, for instance, that we are given the following information:

- The domain of discourse is the set of people.
- The predicate symbol $A(x, y)$ stands for the predicate "x is a sibling of y"; the predicate symbol $B(x, y)$ stands for the predicate "x is a friend of y"; and the predicate symbol $C(x)$ stands for the predicate "x is female".
- The constant symbol $\boldsymbol{c}$ refers to Groucho.

At this point we can interpret the formula $\exists y\,(A(x, y) \wedge B(\boldsymbol{c}, y) \wedge C(y))$ as: there is some person y, who is x's sibling, who is Groucho's friend and who is female. In other words, as the predicate: "x has a sister who is Groucho's friend". This is a predicate which is true of certain x's and false of others. So, if we are now given the final piece of needed information, namely who is the individual x, we can determine the truth value of the formula for the specified x.

The same formula would have an entirely different meaning if we were given different information about the domain of discourse, the meaning of the predicate symbols or of the constant symbol. For example, suppose we were told the following:

- The domain of discourse is the set of natural numbers.
- The predicate symbol $A(x, y)$ stands for the predicate "x divides y"; the predicate symbol $B(x, y)$ stands for the predicate "$x \geq y$"; and the predicate symbol $C(x)$ stands for the predicate "x is odd".
- The constant symbol $\boldsymbol{c}$ refers to the number 100.

Now the formula $\exists y\,(A(x, y) \wedge B(\boldsymbol{c}, y) \wedge C(y))$ is interpreted as: there is some number y, so that x divides y, $100 \geq y$ and y is odd. In other words, x divides an odd number that does not exceed 100. This is a predicate that is true of some numbers and false of others. For instance, it is true if $x = 3$; but it is false if $x = 50$.

6.3.2 Structures, valuations and interpretations

The example in the previous subsection shows that to determine the truth value of a formula in some first-order language $\mathcal{L}$ we need four pieces of information. The first three pieces of information define a "structure" for $\mathcal{L}$. More precisely,

Definition 6.3 *Let $\mathcal{L}$ be a first-order language. A **structure** $\mathcal{S}$ for $\mathcal{L}$ consists of*

1. *A* nonempty *set D, called the **domain** of $\mathcal{S}$.*
2. *For each n-ary predicate symbol A of $\mathcal{L}$, an n-ary relation $A^{\mathcal{S}} \subseteq D \times \cdots \times D$.*
3. *For each constant symbol $\boldsymbol{c}$ of $\mathcal{L}$, an element $\boldsymbol{c}^{\mathcal{S}} \in D$.*

An important note is in order if $\mathcal{L}$ is a first-order language with equality — i.e., it contains the binary predicate symbol $\approx$. In this case we require that, in any structure $\mathcal{S}$ for $\mathcal{L}$, the binary relation that is associated with this symbol be the equality relation for the domain D of $\mathcal{S}$, i.e., the relation $\{(d,d) : d \in D\}$. Thus, we can construct structures that assign meanings to the various predicate symbols in any way we wish, but we are not free to assign any meaning to the symbol $\approx$. This symbol must be assigned the specific meaning "the first argument is the same as the second argument". The significance of this restriction will become evident later in Section 6.5, when we discuss logical implication and logical equivalence.

To determine the truth value of a formula of $\mathcal{L}$ we need to be given, in addition to a structure for $\mathcal{L}$, the values of the formula's free variables. Since we want to define the truth value of an arbitrary formula we will actually specify the values of *all* variables. This is accomplished by something called "valuation".

Definition 6.4 *Given a structure $\mathcal{S}$ for $\mathcal{L}$, a **valuation** of $\mathcal{S}$ is a function σ that maps each variable of $\mathcal{L}$ to some element of the structure's domain D.*

If σ is a valuation of structure $\mathcal{S}$ and v is an element of the structure's domain D, $\sigma|_v^x$ denotes another valuation of $\mathcal{S}$, namely the one defined by

$$\sigma|_v^x(y) = \begin{cases} \sigma(y) & \text{if } y \neq x \\ v & \text{if } y = x \end{cases}$$

That is, $\sigma|_v^x$ maps x to v, and agrees with σ on all other variables.

A structure together with a valuation provide all the information necessary to determine the truth value of any formula in $\mathcal{L}$.

Definition 6.5 *An **interpretation** I of $\mathcal{L}$ is a pair $(\mathcal{S}, \sigma)$, where $\mathcal{S}$ is a structure of $\mathcal{L}$ and σ is a valuation of $\mathcal{S}$.*

Let $\mathcal{I} = (\mathcal{S}, \sigma)$ be an interpretation and t be a term (i.e., a variable or constant symbol) of $\mathcal{L}$. Then $t^{\mathcal{I}}$ denotes the element in the domain of $\mathcal{S}$ to which the term t refers. More precisely,

$$t^{\mathcal{I}} = \begin{cases} \sigma(x) & \text{if } t \text{ is the variable } x \\ \boldsymbol{c}^{\mathcal{S}} & \text{if } t \text{ is the constant symbol } \boldsymbol{c} \end{cases}$$

6.3.3 Truth value of a formula

We are finally in a position to formally define the truth value of formulas in a first-order language $\mathcal{L}$. The definition is by structural induction on the formulas.

Definition 6.6 *Let $\mathcal{L}$ be a first-order language and $\mathcal{S}$ be a structure for $\mathcal{L}$. The truth value of a formula F in $\mathcal{L}$ in interpretation $\mathcal{I} = (\mathcal{S}, \sigma)$, for any valuation σ of $\mathcal{S}$, is defined as follows:*

Basis: *F is an atomic formula, say $F = A(t_1, \ldots, t_n)$, where A is an n-ary predicate symbol of $\mathcal{L}$ and each t_i is a term of $\mathcal{L}$. In this case, F is true in (S, σ) if $(t_1^{\mathcal{I}}, \ldots, t_n^{\mathcal{I}}) \in A^{\mathcal{S}}$, and is false otherwise.*

Induction Step: *F is not an atomic formula. Then F is constructed from one or two formulas F_1, F_2, using a propositional connective or a quantifier. Assume, by induction, that we have determined the truth value of F_1 and F_2 in interpretation $(\mathcal{S}, \sigma)$, for each valuation σ of $\mathcal{S}$. The truth value of F in $(\mathcal{S}, \sigma)$, for any valuation σ of $\mathcal{S}$, is now defined as follows, depending on how F is obtained from F_1 and F_2.*

- *$F = \neg F_1$. In this case, F is true in $(\mathcal{S}, \sigma)$ if F_1 is false in $(\mathcal{S}, \sigma)$, and is false otherwise.*
- *$F = (F_1 \wedge F_2)$. In this case F is true in $(\mathcal{S}, \sigma)$ if F_1 and F_2 are both true in $(\mathcal{S}, \sigma)$, and is false otherwise.*
- *$F = (F_1 \vee F_2)$. In this case F is true in $(\mathcal{S}, \sigma)$ if at least one of F_1 and F_2 is true in $(\mathcal{S}, \sigma)$, and is false otherwise.*
- *$F = (F_1 \rightarrow F_2)$. In this case F is false in $(\mathcal{S}, \sigma)$ if F_1 is true and F_2 is false in $(\mathcal{S}, \sigma)$, and is true otherwise.*
- *$F = (F_1 \leftrightarrow F_2)$. In this case F is true in $(\mathcal{S}, \sigma)$ if F_1 and F_2 have the same truth value in $(\mathcal{S}, \sigma)$, and is false otherwise.*
- *$F = \forall x\, F_1$. In this case F is true in $(\mathcal{S}, \sigma)$ if F_1 is true in $(\mathcal{S}, \sigma|^x_v)$ for every v in the domain of $\mathcal{S}$, and is false otherwise.*
- *$F = \exists x\, F_1$. In this case F is true in $(\mathcal{S}, \sigma)$ if F_1 is true in $(\mathcal{S}, \sigma|^x_v)$ for some v in the domain of $\mathcal{S}$, and is false otherwise.*

If a formula F is true (respectively, false) in interpretation $\mathcal{I}$, we will say that $\mathcal{I}$ ***satisfies*** (respectively, ***falsifies***) F.

Example 6.1 Let $\mathcal{N}$ be the following structure for the language of arithmetic $\mathcal{LA}$:

- The domain is the set of natural numbers, $\mathbb{N}$.
- The predicate symbol $S(x, y, z)$ is interpreted as the predicate $x + y = z$; $P(x, y, z)$ is interpreted as the predicate $x \cdot y = z$; and $L(x, y)$ is interpreted as the predicate $x < y$. (As is always the case, $\approx$ is interpreted as the equality predicate.)
- The constant symbols ***0*** and ***1*** are interpreted by the natural numbers 0 and 1, respectively.

Let $\mathcal{Z}$ be another structure for $\mathcal{LA}$ that is identical to $\mathcal{N}$ except that the domain is the entire set of integers $\mathbb{Z}$, not only the natural numbers. Consider the following formula F_1 of $\mathcal{LA}$

$$F_1 : \qquad P(x, x, \boldsymbol{1})$$

Informally, in both $\mathcal{N}$ and $\mathcal{Z}$, F_1 says that $x^2 = 1$. In the context of natural numbers, there is only one number whose square is equal to 1, namely 1 itself. Thus,

$$\text{an interpretation } (\mathcal{N}, \sigma) \text{ satisfies } F_1 \text{ if and only if } \sigma(x) = 1. \tag{6.1}$$

In the context of the integers, there are exactly two integers whose square is 1, namely -1 and 1. Thus,

$$\text{an interpretation } (\mathcal{Z}, \sigma) \text{ satisfies } F_1 \text{ if and only if } \sigma(x) = 1 \text{ or } \sigma(x) = -1. \tag{6.2}$$

Let F_2 be the following formula of $\mathcal{LA}$

$$F_2 : \qquad \exists y \left(L(y, x) \wedge P(x, y, \boldsymbol{0}) \right)$$

Informally, in both $\mathcal{N}$ and $\mathcal{Z}$ this formula says that

$$\text{there is a number } y < x \text{ such that } x \cdot y = 0. \tag{6.3}$$

In the context of the natural numbers, (6.3) is true if $x > 0$: In this case, we can choose y to be 0 and this satisfies both the requirement that $y < x$ and the requirement that $x \cdot y = 0$. On the other hand, (6.3) is false if $x = 0$: In this case, there is no natural number less than x, and so the first of the two requirements that y must satisfy fails. Therefore,

$$\text{an interpretation } (\mathcal{N}, \sigma) \text{ satisfies } F_2 \text{ if and only if } \sigma(x) > 0. \tag{6.4}$$

In the context of the integers, (6.3) is true if $x \geq 0$. If $x > 0$, this is so for the same reason as we saw above; if $x = 0$ this is so because we can choose y to be any negative integer and with this choice, both the requirement that $y < x$ and the requirement that $x \cdot y = 0$ are met. On the other hand, (6.3) is false if $x < 0$. This is because then x is negative; so any integer $y < x$ is also negative, and the product of two negative integers cannot be 0. Thus, the two requirements that y must satisfy ($y < x$ and $x \cdot y = 0$) cannot both be met. Therefore,

$$\text{an interpretation } (\mathcal{Z}, \sigma) \text{ satisfies } F_2 \text{ if and only if } \sigma(x) \geq 0. \tag{6.5}$$

Let F be the formula $\forall x\,(F_1 \to F_2)$; i.e.,

$$F: \qquad \forall x \Big(P(x, x, \boldsymbol{1}) \to \exists y \left(L(y, x) \wedge P(x, y, \boldsymbol{0})\right)\Big) \tag{6.6}$$

First we consider the truth value of F in structure $\mathcal{N}$. Let σ be an arbitrary valuation of $\mathcal{N}$. Consider the valuations that agree with σ in all variables other than x — that is, valuations $\sigma|^x_v$, for all possible values of $v \in \mathbb{N}$. If $v \neq 1$ then, by (6.1), $(\mathcal{N}, \sigma|^x_v)$ falsifies F_1 and so it satisfies $F_1 \to F_2$. If $v = 1$ then, by (6.4), $(\mathcal{N}, \sigma|^x_v)$ satisfies F_2 and so it also satisfies $F_1 \to F_2$. In other words, for all $v \in \mathbb{N}$, $(\mathcal{N}, \sigma|^x_v)$ satisfies $F_1 \to F_2$. This means that $(\mathcal{N}, \sigma)$ satisfies $\forall x\,(F_1 \to F_2)$, i.e., F. Since σ is an arbitrary valuation of $\mathcal{N}$, it follows that

$$\text{every interpetation } (\mathcal{N}, \sigma) \text{ satisfies } F. \tag{6.7}$$

Next we consider the truth value of F in structure $\mathcal{Z}$. Let σ be an arbitrary valuation of $\mathcal{Z}$. Consider the valuation $\sigma|^x_{-1}$, which agrees with σ on all variables other than x and assigns to x the value -1. By (6.2), $(\mathcal{Z}, \sigma|^x_{-1})$ satisfies F_1 and by (6.5) it falsifies F_2. Therefore, $(\mathcal{Z}, \sigma|^x_{-1})$ falsifies $F_1 \to F_2$. In other words, it is not the case that for all $v \in \mathbb{Z}$, $(\mathcal{Z}, \sigma|^x_v)$ satisfies $F_1 \to F_2$. This means that $(\mathcal{Z}, \sigma)$ falsifies $\forall x\,(F_1 \to F_2)$, i.e., F. Since σ is an arbitrary valuation of $\mathcal{Z}$, it follows that

$$\text{every interpetation } (\mathcal{Z}, \sigma) \text{ falsifies } F. \tag{6.8}$$

End of Example 6.1

The formulas F_1 and F_2 in Example 6.1 have only one free variable, namely x. Looking back at (6.1), (6.2), (6.4) and (6.5) we notice that the truth value of F_1 and F_2 in interpretations of $\mathcal{N}$ and $\mathcal{Z}$ depends only on what the valuation does to variable x, which happens to be the free variable of these formulas. This reflects a general fact, expressed as the following theorem.

Theorem 6.7 *Let F be a formula of a first-order language $\mathcal{L}$, $\mathcal{S}$ be a structure of $\mathcal{L}$, and σ, σ' be valuations of $\mathcal{S}$ so that, for any x that is free in F, $\sigma(x) = \sigma'(x)$. Then F is true in $(\mathcal{S}, \sigma)$ if and only if F is true in $(\mathcal{S}, \sigma')$.*

Informally, this theorem says that the values of variables that are not free in a formula do not affect its truth value. The theorem can be proved by a straightforward (though tedious) structural induction on F.

As a consequence of this theorem, we are now in a position to make more precise our earlier remark that a first-order formula is "about" its *free* variables, or that it "represents" a predicate of those variables — and not of variables that appear in the formula but which name *quantified* objects. Given a structure of a first-order language, we can view a first-order formula as a predicate of its free variables as follows: Let $\mathcal{S}$ be a structure of a first-order language $\mathcal{L}$, and D be the domain of $\mathcal{S}$. A formula F of $\mathcal{L}$ with free variables $x_1, x_2, \ldots, x_k$ defines the k-ary predicate on D which is satisfied by a k-tuple $(a_1, a_2, \ldots, a_k) \in D^k$ if and only if F is satisfied by the interpretation $(\mathcal{S}, \sigma)$ for any valuation σ of $\mathcal{S}$ where $\sigma(x_i) = a_i$, for all i such that $1 \leq i \leq k$. (This predicate is well-defined because, by Theorem 6.7, F is satisfied by all such interpretations or by none of them.) Therefore, we can view the formula F as a function that maps structures of $\mathcal{L}$ into predicates. As we will see in Section 6.7, this view of formulas is crucial in the important application of predicate logic to databases.

Example 6.2 As we saw in Example 6.1, in structure $\mathcal{N}$, F_1 represents the predicate "$x = 1$", while in $\mathcal{Z}$, it represents the predicate "$x = -1$ or $x = 1$" — cf. (6.1) and (6.2). Also, in $\mathcal{N}$, F_2 represents the predicate "$x > 0$", while in $\mathcal{Z}$, it represents the predicate "$x \geq 0$" — cf. (6.4) and (6.5).

As another example, consider the formula $\exists z\, P(x, z, y)$ of $\mathcal{LA}$. In both $\mathcal{N}$ and $\mathcal{Z}$, this expressed the predicate "there is some number which multiplied by x is equal to y", in other words, the predicate "y is a multiple of x". **End of Example 6.2**

Another consequence of Theorem 6.7 is that the truth value of a sentence (i.e., a formula with no free variables) depends *only* on the structure, but not on the valuation: If an interpretation $(\mathcal{S}, \sigma)$ satisfies a sentence F, then the interpretation $(\mathcal{S}, \sigma')$, for *any* valuation σ' of $\mathcal{S}$, also satisfies F. For this reason when F is a sentence, we sometimes speak of the truth value of F in a *structure*, rather than in an interpretation.

Example 6.3 Note that the formula (6.6) in Example 6.1 is a sentence. As we saw in (6.7), this sentence is true in every interpretation whose structure is $\mathcal{N}$, regardless of the valuation; so we say that the sentence is true in $\mathcal{N}$. We also saw in (6.8) that this sentence is false in every interpretation whose structure is $\mathcal{Z}$, regardless of the valuation; so we say that the sentence is false in $\mathcal{Z}$. **End of Example 6.3**

6.4 Validity and satisfiability

In Definition 5.5 we defined the notions of valid and satisfiable formulas in the context of propositional logic. Analogous terms can be defined for first-order formulas.

Definition 6.8 *Let F be a formula F of the first-order language $\mathcal{L}$. F is*

- ***valid*** *if and only if it is satisfied by every interpretation of $\mathcal{L}$;*
- ***satisfiable*** *if and only if it is satisfied by some interpretation of $\mathcal{L}$; and*
- ***unsatisfiable*** *if and only if it is not satisfied by any interpretation of $\mathcal{L}$.*

This definition is very similar to the corresponding definition for propositional formulas. The only difference is that we substituted the word "interpretation" for "truth assignment". This is because an interpretation is what makes a first-order formula true or false, in just the way a truth assignment is what makes a propositional formula true or false.

It follows immediately from Definition 6.8 that a first-order formula is valid if and only if its negation is unsatisfiable.

Example 6.4 Consider a first-order language $\mathcal{L}$ with unary predicate symbols A and B, and a constant symbol $\boldsymbol{c}$. The following formula is valid:

$$\forall x\, A(x) \to A(\boldsymbol{c}). \tag{6.9}$$

To see why, let $\mathcal{S}$ be any structure for $\mathcal{L}$, and σ be any valuation of $\mathcal{S}$. Suppose that the antecedent of (6.9), $\forall x\, A(x)$, is true in interpretation $(\mathcal{S}, \sigma)$. This means that every element of the domain of $\mathcal{S}$ belongs to $A^{\mathcal{S}}$ (i.e., the relation to which $\mathcal{S}$ associates the predicate symbol A). In particular, $\boldsymbol{c}^{\mathcal{S}}$ (i.e., the element of the domain to which $\mathcal{S}$ associates the constant symbol $\boldsymbol{c}$) belongs to $A^{\mathcal{S}}$. Thus, $(\mathcal{S}, \sigma)$ satisfies $A(\boldsymbol{c})$, the consequent of (6.9). We proved that if $(\mathcal{S}, \sigma)$ satisfies $\forall x\, A(x)$ then it also satisfies $A(\boldsymbol{c})$. This means that $(\mathcal{S}, \sigma)$ satisfies $\forall x\, A(x) \to A(\boldsymbol{c})$. Since $(\mathcal{S}, \sigma)$ is an arbitrary interpretation, (6.9) is valid.

The following formula is satisfiable, but it is not valid.

$$\exists x\, A(x) \wedge B(y) \tag{6.10}$$

To see why it is satisfiable, suppose $\mathcal{S}$ is a structure of $\mathcal{L}$ that has domain $\mathbb{N}$, associates the predicate symbol $A(x)$ with the predicate "x is an even number" and the predicate symbol $B(x)$ with the predicate "x is an odd number". Let σ be a valuation of $\mathcal{S}$ such that $\sigma(y) = 1$. Under $(\mathcal{S}, \sigma)$, (6.10) is interpreted as "there is an even number and 1 is odd", which is certainly true. Since there is an interpretation that satisfies it, the formula is satisfiable.

To see why (6.10) is not valid, suppose $\mathcal{S}$ is as above, and let σ' be a valuation of $\mathcal{S}$ such that $\sigma'(y) = 4$. Under $(\mathcal{S}, \sigma')$ the formula is interpreted as "there is an even number and 4 is odd", which is false. Since there is an interpretation that falsifies it, the formula is not valid.

The formula

$$\forall x \big(A(x) \to B(x)\big) \wedge A(\boldsymbol{c}) \wedge \neg B(\boldsymbol{c}) \tag{6.11}$$

is unsatisfiable. To see why, let $\mathcal{S}$ be any structure of $\mathcal{L}$ and σ be any valuation of $\mathcal{S}$. For interpretation $(\mathcal{S}, \sigma)$ to satisfy (6.11), it must satisfy each of its three conjuncts. If it satisfies the first conjunct, $\forall x\,\big(A(x) \to B(x)\big)$, we have that every element of the domain that belongs to $A^{\mathcal{S}}$ also belongs to $B^{\mathcal{S}}$. If, in addition, the interpretation satisfies the second conjunct, $A(\boldsymbol{c})$, we have that $\boldsymbol{c}^{\mathcal{S}}$ belongs to $A^{\mathcal{S}}$ — and therefore it also belongs to $B^{\mathcal{S}}$. But this contradicts the third conjunct, $\neg B(\boldsymbol{c})$, which requires that $\boldsymbol{c}^{\mathcal{S}}$ *not* belong to $B^{\mathcal{S}}$. This means that no interpretation can satisfy this formula, and therefore (6.11) is unsatisfiable. **End of Example 6.4**

6.5 Logical implication and logical equivalence

We can now define the concepts of logical implication and logical equivalence in the context of first-order logic (see Definition 5.6). The definitions are identical to those for propositional logic, except that we use "interpretation" in the place of "truth assignment".

Definition 6.9 *A formula F* ***logically implies*** *formula F' if and only if every interpretation that satisfies F also satisfies F'.*

Example 6.5 We assume we are working in a first-order language $\mathcal{L}$ with unary predicate symbols A and B, and constant symbol $\boldsymbol{c}$. Following are two logical equivalences of formulas in $\mathcal{L}$.

(1) $\forall x\,\big(A(x)\to B(x)\big)$ logically implies $\exists x\, A(x)\to\exists x\, B(x)$. To see why, let $\mathcal{S}$ be any structure for $\mathcal{L}$ and assume that $\mathcal{S}$ satisfies the first formula, $\forall x\,\big(A(x)\to B(x)\big)$.[4] Thus, any element of the domain that belongs to $A^{\mathcal{S}}$ also belongs to $B^{\mathcal{S}}$. So, if some element of the domain belongs to $A^{\mathcal{S}}$, then certainly some element (the same one, if no other!) belongs to $B^{\mathcal{S}}$. In other words the second formula, $\exists x\, A(x)\to\exists x\, B(x)$, is also satisfied by $\mathcal{S}$. Since $\mathcal{S}$ is an arbitrary structure for $\mathcal{L}$, this means that the first formula logically implies the second.

(2) $\forall x\,\big(A(x)\to B(x)\big) \wedge A(\boldsymbol{c})$ logically implies $B(\boldsymbol{c})$. To see why, let $\mathcal{S}$ be any structure for $\mathcal{L}$ and suppose that $\mathcal{S}$ satisfies the first formula, $\forall x\,\big(A(x)\to B(x)\big) \wedge A(\boldsymbol{c})$. Then (i) each element of the domain that belongs to $A^{\mathcal{S}}$ also belongs to $B^{\mathcal{S}}$, and (ii) $\boldsymbol{c}^{\mathcal{S}}$ belongs to $A^{\mathcal{S}}$. Since (i) is true for each element of the domain, it is true, in particular, for $\boldsymbol{c}^{\mathcal{S}}$. That is, if $\boldsymbol{c}^{\mathcal{S}}$ belongs to $A^{\mathcal{S}}$ then $\boldsymbol{c}^{\mathcal{S}}$ also belongs to $B^{\mathcal{S}}$. By (b), $\boldsymbol{c}^{\mathcal{S}}$ does belong to $A^{\mathcal{S}}$. Therefore, $\boldsymbol{c}^{\mathcal{S}}$ belongs to $B^{\mathcal{S}}$. This means that the second formula, $B(\boldsymbol{c})$, is also satisfied by $\mathcal{S}$. Since $\mathcal{S}$ is an arbitrary structure for $\mathcal{L}$, this means that the first formula logically implies the second.

End of Example 6.5

Note that the two logical implications of the preceding example hold *regardless of the specific meaning of the predicate and constant symbols*. In each case, the truth of the first formula implies the truth of the second *by virtue of the structure of the formulas and the*

[4] Since both formulas are sentences, as we saw in Section 6.3.3, we needn't worry about the valuations; the truth value of the formulas depends only on the structure.

meanings of the logical symbols — not because of any particular meaning attached to the predicate and constant symbols. This is what the definition of logical implication requires: the truth of one formula must imply the truth of the other in *all* interpretations — i.e., regardless of the meaning assigned to the predicate and constant symbols.

We can also define logical equivalence between first-order formulas in a manner that is analogous to the corresponding definition for propositional formulas (see Definition 5.8)

Definition 6.10 *A formula F is **logically equivalent** to formula F' if and only if each interpretation either satisfies both or falsifies both.*

From this definition we get that F is logically equivalent to F' if and only if each of F, F' logically implies the other. Also it is straightforward to verify that the relation "is logically equivalent to" (which we abbreviate as LEQV) is reflexive (F LEQV F), symmetric (if F LEQV F' then F' LEQV F), and transitive (if F LEQV F' and F' LEQV F'', then F LEQV F'').

To prove that F and F' are logically equivalent we must show that the two formulas have the same truth value in *every* interpretation. To prove that F and F' are *not* logically equivalent, it is enough to produce one interpretation that satisfies one and falsifies the other.

At this point it is important to recall the remark we made in Section 6.3.2 regarding the interpretation of the equality predicate symbol: A structure may interpret predicate symbols in any manner, with the exception of the symbol $\approx$ which *must* be interpreted as the equality predicate. This constraint is relevant when we try to establish that a formula logically implies (or is logically equivalent to) another. To do this we must show something about *all* possible interpretations, subject to the requirement that $\approx$ has only one possible meaning.

A result analogous to Theorems 5.7 and 5.9 holds for first-order formulas.

Theorem 6.11 *Let F and F' be formulas of a first-order language. Then,*

(a) F logically implies F' if and only if $F \rightarrow F'$ is a valid formula.

(b) F is logically equivalent to F' if and only if $F \leftrightarrow F'$ is a valid formula.

The proof is very similar to the proof of Theorem 5.7, and is left as an exercise.

6.6 Some important logical equivalences

In this section we list some important logical equivalences. These can be used and combined to prove additional logical equivalences.

Duality of quantifiers

Ia. $\neg \forall x\, F$ LEQV $\exists x\, \neg F$, for any formula F and any variable x.

This simply states that "not every x has property F" amounts to the same thing as "there is some x that does not have property F". It is possible to verify this logical equivalence by resorting to Definition 6.6, but as you will discover when you do this, essentially it comes down to recognising that the above two expressions mean exactly the same thing. Similarly,

Ib. $\neg\exists x\, F$ LEQV $\forall x\, \neg F$, for any formula F and any variable x.

This says that "no x has property F" amounts to the same things as "every x fails to have property F".

These two equivalences bear a resemblance to DeMorgan's laws. They reveal a duality between $\forall$ and $\exists$ that is analogous to the duality between $\wedge$ and $\vee$. This is not terribly surprising because a universal quantifier is a form of conjunction: it asserts that a certain fact is true for this value of x *and* that value of x *and* that value of x — for each possible value in the domain; while an existential quantifier is a form of disjunction: it asserts that a fact is true for this value of x *or* that value of x *or* that value of x — again, for each possible value in the domain.

From now on we will use the symbol $\mathbf{Q}$ to indicate a generic quantifier symbol that is either $\forall$ or $\exists$. We will also use $\overline{\mathbf{Q}}$ to indicate the dual quantifier of $\mathbf{Q}$; i.e., if $\mathbf{Q}$ is $\forall$ then $\overline{\mathbf{Q}}$ is $\exists$, and if $\mathbf{Q}$ is $\exists$ then $\overline{\mathbf{Q}}$ is $\forall$. With this notation, the two logical equivalences (Ia) and (Ib) may be written as

I. $\neg\mathbf{Q}x\, F$ LEQV $\overline{\mathbf{Q}}x\, \neg F$, for any formula F and any variable x.

Factoring quantifiers

IIa. $E \wedge \mathbf{Q}x\, F$ LEQV $\mathbf{Q}x\, (E \wedge F)$, for any formulas E, F and any variable x *that is not free in E*.

The requirement that x is not free in E is very important. Without it this logical equivalence would not hold. To see this note that, if x is free in E, then x is free in $E \wedge \mathbf{Q}x\, F$ while it is not free in $\mathbf{Q}x(E \wedge F)$. Since the two formulas do not even have the same free variables they cannot, in general, be logically equivalent: they represent predicates of different things.

A similar logical equivalence holds for disjunctions of formulas:

IIb. $E \vee \mathbf{Q}x\, F$ LEQV $\mathbf{Q}x\, (E \vee F)$, for any formulas E, F and any variable x *that is not free in E*.

Renaming of quantified variables

As we discussed earlier, quantified variables are "dummy" variables. They are used to allow us to name objects and assert that something holds for all such objects or at least one such object. It is not important which particular name we use. In other words, if we rename a quantified variable in a formula, then the resulting formula is logically equivalent to the original one. Actually, there is a catch: the new name of the variable should not be a name that is already used for some other purpose in the formula! This observation can be formalised as a logical equivalence. First we need some notation: If F is a formula and x, y are variables, F^x_y denotes the formula obtained by replacing *every free* occurrence of x in F by y.

III. $\mathbf{Q}x\, F$ LEQV $\mathbf{Q}y\, F^x_y$, for any formula F and any variables x, y so that y *does not occur* in F.

As an example of this, we have:

$$\exists x\,(\underbrace{P(u,v,w)\wedge S(x,w)\rightarrow P(u,v,x)}_{F})\quad \textsc{leqv}\quad \exists y\,(\underbrace{P(u,v,w)\wedge S(y,w)\rightarrow P(u,v,y)}_{F^x_y})$$

A slightly more complicated example is this:

$$\exists x\,(\underbrace{P(u,v,w)\wedge S(x,w)\rightarrow \exists x\,P(u,v,x)}_{F})\quad \textsc{leqv}\quad \exists y\,(\underbrace{P(u,v,w)\wedge S(y,w)\rightarrow \exists x\,P(u,v,x)}_{F^x_y})$$

Notice that, in the second example, when we renamed x to y in F to obtain F^x_y, we did not rename the occurrence of x in the consequent of the implication (i.e., in the subformula $\exists x\,P(u,v,x)$); this is because that occurrence of x is *not* free.

Also notice the caveat that the new name must not occur anywhere in F. If it does, then the resulting formula would not, in general, be logically equivalent to the original formula. For instance, suppose $F = P(u,v,w)\wedge S(x,w)\rightarrow \exists x\,P(u,v,x)$. If we rename every free occurrence of x to u in this example we get the formula $F^x_u = P(u,v,w)\wedge S(u,w)\rightarrow \exists x\,P(u,v,x)$. Now, however, the formulas

$$\exists x\,(\underbrace{P(u,v,w)\wedge S(x,w)\rightarrow \exists x\,P(u,v,x)}_{F})\qquad \text{and}\qquad \exists u\,(\underbrace{P(u,v,w)\wedge S(u,w)\rightarrow \exists x\,P(u,v,x)}_{F^x_u})$$

are *not* logically equivalent: variable u is free in the former, while it is not in the latter. Since the two formulas do not have the same free variables they are not "about" the same things and thus they cannot be logically equivalent.

Finally, notice that this rule pertains specifically to renaming of *quantified* variables. In general, if we rename a free variable of a formula, the resulting formula is not logically equivalent to the original one. That is, in general, E is *not* logically equivalent to E^x_y, *even if* y is a variable that does not occur in E. For example, take the formula $E = M(x)$, so that $E^x_y = M(y)$. These two formulas are not logically equivalent. For instance, consider the "standard" structure $\mathcal{S}$ of $\mathcal{LF}$, where the domain is the set of all people and the predicate symbol M stands for the predicate "is-male". Let σ be a valuation so that $\sigma(x)$ = Groucho and $\sigma(y)$ = Cleopatra. The interpretation $(\mathcal{S},\sigma)$ satisfies E but falsifies E^x_y; hence the two formulas are not logically equivalent.

Substitution instances of propositional equivalences

IV. Let P and Q be *propositional* formulas with propositional variables $x_1, x_2, \ldots, x_n$ such that P LEQV Q. Let $F_1, F_2, \ldots, F_n$ be arbitrary first-order formulas. Let P' and Q' be the (first-order) formulas obtained from P and Q, respectively, by substituting x_i by F_i, for each i, $1 \le i \le n$. Then P' LEQV Q'.

As an example of this, we have:

$$\underbrace{\forall x\,S(x,y)}_{F_1}\rightarrow\underbrace{\exists z\,M(z)}_{F_2}\quad \textsc{leqv}\quad \neg\underbrace{\forall x\,S(x,y)}_{F_1}\vee\underbrace{\exists z\,M(z)}_{F_2}$$

This follows from the propositional equivalence $x_1 \rightarrow x_2$ $\textsc{leqv}$ $\neg x_1 \vee x_2$, by substituting F_1 and F_2 for x_1 and x_2, respectively.

Subformula substitution

As in propositional logic, we can replace a piece of a formula by a logically equivalent expression without affecting the meaning of the original formula. More precisely, the following rule holds:

V. Let F be any first-order formula, E be any subformula of F, E' be any formula that is logically equivalent to E, and F' be the formula that results from F by replacing E with E'. Then F' is logically equivalent to F.

As an example of this, we have:

$$\underbrace{\overbrace{\neg\forall x\, S(x,y)}^{E} \rightarrow \exists z\, M(z)}_{F} \quad \textsc{leqv} \quad \underbrace{\overbrace{\exists x\, \neg S(x,y)}^{E'} \rightarrow \exists z\, M(z)}_{F'}$$

This is because, by (I), $\neg\forall x\, S(x,y)$ $\textsc{leqv}$ $\exists x\, \neg S(x,y)$ and so we can substitute the former by the latter in any formula, resulting in an equivalent formula.

Additional logical equivalences

By combining the five rules described so far in this section, we can derive additional logical equivalences. In particular we can derive some "factoring" logical equivalences similar to rules (IIa) and (IIb). These will be especially useful in the next section.

IIc. $\mathbf{Q}x\, E \wedge F$ $\textsc{leqv}$ $\mathbf{Q}x\,(E \wedge F)$, for any formulas E, F and any variable x *that is not free in* F.

IId. $\mathbf{Q}x\, E \vee F$ $\textsc{leqv}$ $\mathbf{Q}x\,(E \vee F)$, for any formulas E, F and any variable x *that is not free in* F.

Here is how we can derive the first of these:

$$\begin{array}{lll}
 & \mathbf{Q}x\, E \wedge F & \\
\textsc{leqv} & F \wedge \mathbf{Q}x\, E & \text{[by (IV) and commutativity of } \wedge] \\
\textsc{leqv} & \mathbf{Q}x\,(F \wedge E) & \text{[by (IIa)]} \\
\textsc{leqv} & \mathbf{Q}x\,(E \wedge F) & \text{[by (IV) and (V)]}
\end{array}$$

There are also quantifier factoring rules involving implication:

IIe. $\mathbf{Q}x\, E \rightarrow F$ $\textsc{leqv}$ $\overline{\mathbf{Q}}x\,(E \rightarrow F)$, for any formulas E, F and any variable x *that is not free in* F.

Notice that when a quantifier that appears on the left hand side of an implication is factored, it is changed to its dual. The following derivation of this rule shows why:

$$\begin{array}{lll}
 & \mathbf{Q}x\, E \rightarrow F & \\
\textsc{leqv} & \neg\mathbf{Q}x\, E \vee F & \text{[by (IV) and the} \rightarrow \text{law]} \\
\textsc{leqv} & \overline{\mathbf{Q}}x\, \neg E \vee F & \text{[by (I)]} \\
\textsc{leqv} & \overline{\mathbf{Q}}x\, (\neg E \vee F) & \text{[by (IId)]} \\
\textsc{leqv} & \overline{\mathbf{Q}}x\, (E \rightarrow F) & \text{[by (IV) and (V)]}
\end{array}$$

In a similar way, we can prove that

IIf. $E \rightarrow \mathbf{Q}x\, F$ LEQV $\mathbf{Q}x\, (E \rightarrow F)$, for any formulas E, F and any variable x *that is not free in* E.

Here notice that when a quantifier is factored from the *right* side of an implication, it is *not* changed to its dual! By working out the derivation of this rule, you will see exactly why that is so.

6.7 Predicate logic and relational databases

A database system is a software system that allows users to store and manipulate the information that describes some enterprise in a convenient and flexible manner. Predicate logic provides the mathematical basis and the conceptual framework for the most popular type of database systems in use, known as *relational database systems*. In this section we explain the basic ideas behind this important application of predicate logic to computer science.

It is perhaps easiest to explain the connection between relational databases and predicate logic by examining a simple example. Suppose we want to create a database about a library. In this database we need to record information about the *books* that the library owns, the *subscribers* who have the right to borrow books from the library, and about which books have been *borrowed* by subscribers. In relational databases, all information is stored as relations or, equivalently, as predicates. (Recall, from Section 6.1.1, that we can view a relation as a predicate and vice versa. Consequently, we will sometimes use these two terms interchangeably even though, technically speaking, they refer to different kinds of mathematical objects: a relation is a set of tuples while a predicate is a function from tuples to the set $\{\text{true}, \text{false}\}$.)

We can think of a book as an entity relating a *book id* (a string that uniquely identifies the book), a *title*, an *author's name*, and perhaps other pieces of information (publisher, date of publication, and so on) that we will ignore in this example. In relational database terms, these "pieces of information" are called the *attributes* of the relation that describes the books owned by the library. Therefore, the set of books currently owned by the library can be described mathematically as a relation

$$\mathit{Book} \subseteq B \times T \times N$$

where B is the set of book ids, T is the set of titles and N is the set of author names. Equivalently, this set of books can be described as a predicate

$$\mathit{Book}(b, t, n)$$

where b is the argument that corresponds to the book's unique id, t is the argument that corresponds to the book's title and n is the argument that corresponds to the name of the book's author. A triple (β, τ, ν) satisfies this predicate (is a member of the relation) if and only if the library owns a book whose id is β, has title τ and is written by author ν.

Similarly, the library's subscribers can be described as a relation with attributes *SIN* (the subscriber's Social Insurance Number which we assume uniquely identifies him or her), *name* and *address*. In other words, the library's set of subscribers will be viewed as a predicate

$$\mathit{Subscriber}(s, n, a)$$

where s is the argument that corresponds to a subscriber's SIN, n is the argument that corresponds to his or her name and a is the argument that corresponds to his or her address. A triple (σ, ν, α) satisfies this predicate if and only if a subscriber with SIN σ has name ν and lives at address α.

Finally, the information about which books have been borrowed might be described as a relation with attributes *SIN*, *book id* and *due date*. *SIN* refers to a subscriber, *book id* refers to a book that this subscriber has borrowed and *due date* refers to the date by which the book must be returned. In other words, the information about borrowed books will be described as the predicate

$$\mathit{Borrowed}(s, b, d)$$

which is satisfied by a triple (σ, β, δ) if and only if the subscriber whose SIN is σ has borrowed the book whose book id is β and the due date for the book's return to the library is δ.

Each relation is stored in the database as a two-dimensional array. The columns of the array correspond to the relation's attributes and its rows contain the elements of the relation, which are called *tuples*. For example, the *Book* relation might be the array shown in Figure 6.2. Note that the library owns two copies of *One hundred years of solitude* by Gabriel Garcia Marquez; each has its own distinct book id.

Book ID	Title	Author's Name
POET017	Iliad	Homer
MATH092	Metamathematics	Kleene
CSCI001	The art of computer programming	Knuth
FICT171	One hundred years of solitude	Marquez
FICT576	One hundred years of solitude	Marquez
FICT923	Love in the time of cholera	Marquez

Figure 6.2: An example for *Book* relation

6.7.1 Queries

The main reason for creating a database is so that users can then *query* the database to retrieve information they are interested in. The creator of the database usually does not know *a priori* what queries the users will want to pose. Thus, the database system must provide a language that is precise so that it can be processed by a computer, as well as flexible and general so that a user can formulate a wide range of queries. The genius of the relational database model is that, because the information in the database is represented in the form of predicates, we can use first-order formulas as our query language. A query, in the relational database model, is simply a predicate that describes the information that the user wishes to retrieve. As we saw in Section 6.1.2, given certain "base" predicates, we can use first-order formulas to express other predicates. For example, in that section we saw how to combine the predicates "x is male", "x is y's sibling" and "x and y are the parents of z" in order to express the predicate "x is y's uncle".

Now the "base" predicates are the relations that make up the database — *Book*, *Subscriber* and *Borrowed*, in our example. Suppose a user wants to find out the titles of all the books written by Marquez that the library owns. This query can be expressed by the formula

$$\exists b\, \mathit{Book}(b, t, \text{“Marquez”}) \tag{6.12}$$

Notice that the information we want, namely titles of books, corresponds to a *free* variable in the formula. The variable b is existentially quantified, and "Marquez" is a constant string. The way to interpret the above predicate as a query is: Find the set of all values τ that can be substituted for the free variable t, so that for some book id, say β, the triple (β, τ, "Marquez") is in the relation (satisfies the predicate) *Book*. If the *Book* relation is the one shown in Figure 6.2, then this query would return two titles as its answer:

One hundred years of solitude
Love in the time of cholera

Note that the title "One hundred years of solitude" is returned only once, even though there are two copies of this book. This is because the query returns the *set* of values for the free variables that satisfy the predicate; since the result is a set, it cannot contain the same element twice.

Let us contrast (6.12) with the query below:

$$\mathit{Book}(b, t, \text{“Marquez”}) \tag{6.13}$$

Syntactically, the only difference is that in this query the existential quantifier of (6.12) was eliminated. This predicate has two free variables, b and t. This predicate can be interpreted as the following query: Find the set of all pairs of values β and τ that can be substituted (simultaneously) for the free variables b and t respectively, so that the triple (β, τ, "Marquez") satisfies the predicate *Book*. If the state of the relation *Book* is as shown in Figure 6.2, Query (6.13) returns as its answer the following three tuples:

FICT171	One hundred years of solitude
FICT576	One hundred years of solitude
FICT923	Love in the time of cholera

Note that the first two tuples are different because they differ in the book id attribute.

Consider now a different query. Suppose a user is interested in retrieving the names and addresses of all subscribers who have borrowed books that are due on January 1, 2000 — perhaps to notify them that the library will be closed on that day due to the millennium celebrations. This query can be expressed by the following first-order formula:

$$\exists s \left(\mathit{Subscriber}(s, n, a) \wedge \exists b \, \mathit{Borrowed}(s, b, 2000/01/01) \right) \tag{6.14}$$

The two attributes that we are interested in retrieving, the name and address of subscribers, are the free variables of this formula, n and a respectively. The expression 2000/01/01 that appears as the third argument of predicate *Borrowed* is a constant (representing a date in the obvious way). This formula is interpreted as the following query: Find the set of all pairs of values ν and α that can be simultaneously substituted for the variables n and a, respectively, so that there is a subscriber with some SIN, say σ, name ν and address α (i.e., the triple (σ, ν, α) satisfies the predicate *Subscriber*), who has borrowed a book with some id, say β, that is due on January 1, 2000 (i.e., the triple $(\sigma, \beta, 2000/01/01)$ satisfies the predicate *Borrowed*).

Note that both occurrences of variable s in (6.14), as an argument of *Subscriber* and of *Borrowed*, are bound to the same quantifier, the leading $\exists s$. This is because we want both of these variables to refer to the same individual: the subscriber whose name and address we want to retrieve is the same as the subscriber who has borrowed a book that's due on January 1, 2000. The query expressed by the formula below is quite different from that expressed by (6.14):

$$\exists s \, \mathit{Subscriber}(s, n, a) \wedge \exists s \, \exists b \, \mathit{Borrowed}(s, b, 2000/01/01)$$

Here the two occurrences of the variable s are bound to different quantifiers. Despite the fact that the same symbol is used, the fact that the two occurrences are bound to different quantifiers means that they can refer to different individuals. Some thought will show that the above formula expresses the query which returns the set of *names and addresses of all subscribers* if there is a subscriber who has borrowed a book that is due on January 1, 2000; and returns nothing (the empty set) if no subscriber has borrowed a book that is due on that date! The two formulas bear a superficial syntactic resemblance to each other but they mean quite different things. In Section 6.11 we will discuss in greater detail the binding of variables to quantifiers and how this affects the meaning of a formula.

The query expressed by (6.14) can also be expressed by other formulas. In fact, any formula that is logically equivalent to (6.14) expresses the same query. For example, from the equivalences of Section 6.6, it is easy to verify that the formula

$$\exists s \, \exists b \left(\mathit{Subscriber}(s, n, a) \wedge \mathit{Borrowed}(s, b, 2000/01/01) \right)$$

also expresses the same query. Another formula that expresses the same query is

$$\neg \forall s \, \forall b \left(\mathit{Subscriber}(s, n, a) \rightarrow \neg \mathit{Borrowed}(s, b, 2000/01/01) \right)$$

If we try to grasp the meaning of this particular formula, it is perhaps not readily apparent that it is just a different way of expressing the same query as the previous two formulas.

Nevertheless, by applying the techniques of Section 6.6 we can check that the formula is logically equivalent to the previous two and, as such, expresses exactly the same query.

Note that (6.14) involves two relations, *Subscriber* and *Borrowed*, while (6.12) and (6.13) involve one relation, *Book*. Our next example involves all three relations of the database. Suppose we wish to find the names of all subscribers who have borrowed a copy of *Metamathematics* by Kleene, and the due date by which the borrowed copy is (or was) to be returned. The following first-order formula expresses this query:

$$\exists s \Big(\exists a\, \mathit{Subscriber}(s,n,a) \wedge \exists b \big(\mathit{Book}(b, \text{“Metamathematics”}, \text{“Kleene”}) \wedge \\ \mathit{Borrowed}(s,b,d)\big)\Big) \tag{6.15}$$

This formula has two free variables, n and d. It can be interpreted as the following query: Find all pairs of values ν and δ that can be substituted for the free variables n and d, respectively, so that for some SIN, say σ, there is a subscriber with SIN σ whose name is ν (and who lives at some address), and for some book id, say β, the book with id β is *Metamathematics* by Kleene and subscriber σ has borrowed β and must return it by date δ.

Let us consider now another example query. Suppose we want to retrieve the names of all subscribers who have borrowed all books written by Marquez that the library owns. First, we note that this description of the query is actually ambiguous. It could mean that we want to retrieve the names of all subscribers who have borrowed every single copy of every book written by Marquez that the library owns. In this case, if the state of the relation *Book* is that shown in Figure 6.2, a subscriber would have to have borrowed both copies of *One hundred years of solitude* as well as the (single) copy of *Love in the time of cholera* to be included in the answer. Alternatively (and perhaps more plausibly) we may want to retrieve the names of all subscribers who have borrowed at least one copy of every book written by Marquez that the library owns; in this case, a subscriber who has borrowed only one copy of *One hundred years of solitude* and the (single) copy of *Love in the time of cholera* will be included in the answer.[5]

The two different meanings of this query are captured by different (and, of course, nonequivalent) first-order formulas. The first meaning can be expressed by

$$\exists s \exists a \Big(\mathit{Subscriber}(s,n,a) \wedge \forall b \big(\exists t\, \mathit{Book}(b,t,\text{“Marquez”}) \rightarrow \exists d\, \mathit{Borrowed}(s,b,d)\big)\Big) \tag{6.16}$$

We can paraphrase this as: Find all the names ν that can be substituted for the variable n so that there is a subscriber called ν who has some SIN σ (and lives at some address) and, furthermore, for every book id β, if β is written by Marquez (and has some title), then subscriber σ has borrowed β (for some return date).

[5] It is perhaps appropriate for me to confess that I was not conscious of the two possible meanings of this query until I started thinking about how to express it as a first-order formula. A useful byproduct of the process of expressing an informally specified query as a first-order formula is that, in doing so, one often uncovers ambiguities that are not so easy to spot when the query is expressed in natural language.

```
CREATE SCHEMA Library
    CREATE TABLE Book
    (
        BookID character (7),
        Title character varying (100),
        AuthorName character varying (40)
    )
    CREATE TABLE Subscriber
    (
        SIN numeric (9),
        Name character varying (40),
        Address character varying (60)
    )
    CREATE TABLE Borrowed
    (
        SIN numeric (9),
        BookID character (7),
        ReturnDate date
    )
```

Figure 6.3: The specification of the library database in SQL

The second meaning can be expressed by the following formula:

$$\exists s\, \exists a \left(\mathit{Subscriber}(s, n, a) \wedge \forall t \left(\exists b\, \mathit{Book}(b, t, \text{“Marquez”}) \rightarrow \right.\right.$$
$$\left.\left. \exists b' \left(\mathit{Book}(b', t, \text{“Marquez”}) \wedge \exists d\, \mathit{Borrowed}(s, b', d) \right) \right) \right)$$

We can paraphrase this as: Find all the names ν that can be substituted for the variable n so that there is a subscriber called ν who has some SIN σ (and lives at some address) and, furthermore, for every title τ, if there is a book (with some id) with title τ written by Marquez, then there is a book also with title τ and written by Marquez but with a possibly different id β', which the subscriber σ has borrowed (and must return by some due date).

6.7.2 Data definition and data manipulation languages

A relational database system provides two languages to its users; a *data definition language* (or *DDL*) and a *data manipulation language* (or *DML*). In this subsection we explain the purpose of these languages, and we provide some examples. Our examples are written in SQL, the most common language for relational databases which is supported by all the major commercial database products. The goal of these examples is merely to give a taste of these languages, not to provide anything approaching a thorough description.

The DDL allows a user to specify the *database schema*: the names of the relations that will exist in the database, the names of each relation's attributes, and the data type from which each attribute will derive its values. Figure 6.3 shows the definition of our example library database in SQL. In SQL the term "table" is synonymous to what we have been calling a relation or a predicate. For each relation, the definition contains a comma-separated list of attribute names (e.g., *BookID*, *Title* and *AuthorName* for the relation *Book*) and the data type from which each attribute derives its values. The parenthesised number refers to the length of a character string, or the maximum number of digits in numeric types. If the character string has variable length (denoted by the keyword "varying") the parenthesised number refers to the maximum length of the string.

Database systems support the basic data types of programming languages such as integers, reals, strings and Booleans, and perhaps other specialised data types such as dates. For each data type there is particular syntax for representing its elements. For instance, integers might be represented in the usual decimal notation, strings might be represented as alphanumeric sequences delimited by quotation marks, and dates might be represented in the format YYYY/MM/DD.

In terms of predicate logic, the representations of data types are *constant symbols*. Note that the entire representation of an element of a data type is viewed as a *single* constant symbol, even if it is written using several characters. For example, 1998/10/22 is a constant symbol denoting the date October 22, 1998, and 17 is a constant symbol denoting the integer seventeen. The symbol denoting the integer seventeen would be different if, for example, we were representing integers in octal rather than decimal. Similarly, a different symbol would denote the date October 22, 1998 if we were using a different format for dates.

The second language provided by the database system, the DML, allows users to query and modify the database. For example, Figure 6.4 shows one way of expressing query (6.14) in SQL. Roughly speaking this SQL query says: Retrieve the *Name* and *Address* attributes from each tuple t of the *Subscriber* relation for which there exists some tuple in the *Borrowed* relation whose *SIN* attribute matches that of t and whose *ReturnDate* attribute is January 1, 2000. The requirement that the tuples from *Subscriber* and *Borrowed* relations have the same value in the *SIN* attribute reflects the fact that in (6.14) the two occurrences of variable s (that corresponds to the *SIN* attribute) are bound to the same quantifier. Also note that the attributes *Name* and *Address* that appear in the first SELECT command correspond to the two free variables n and a of (6.14). This example should convince you that queries expressed in "real-world" database query languages are very close in spirit (though not in syntax) to first-order formulas.

6.7.3 Relational databases as structures of a first-order language

In this subsection we explain exactly what it means to say that a first-order formula "expresses" a query. To do so we must first gain a more detailed understanding of the relationship between relational databases and predicate logic. As we will see, by creating a relational database we are effectively defining a first-order language $\mathcal{L}$ (Section 6.2.1) and a structure $\mathcal{S}$ for $\mathcal{L}$ (Section 6.3.2).

```
SELECT Name, Address
FROM Subscriber
WHERE EXISTS
(
    SELECT ∗
    FROM Borrowed
    WHERE Subscriber.SIN = Borrowed.SIN AND Borrowed.ReturnDate = 2000/01/01
)
```

Figure 6.4: Query (6.14) expressed in SQL

In terms of predicate logic, a DDL specification of a database schema corresponds to choosing a first-order language. Recall, from Section 6.2.1, that to choose a first-order language we must fix (a) a set of predicate symbols and their associated arities, and (b) a set of constant symbols.[6]

- The predicate symbols of the language are the relation names given in the DDL specification. The arity of each predicate symbol is the number of attributes of the corresponding relation name.
- The constant symbols of the language are the representations of the data types of the relations' attributes.

In our example, the first-order language defined by the DDL specification has three predicate symbols: *Book*, *Subscriber* and *Borrowed*, each of arity 3. It has constant symbols consisting of the representations of integers, strings, and dates.

As we mentioned in Section 6.7.2, the DML allows users to query and modify the database. To modify the database, the DML provides commands to insert, delete and update tuples in the relations. Using these commands, the users can keep the database current by reflecting changes to the state of the enterprise that is modeled by the database. For example, when the library acquires a new book a user should insert a tuple describing that book into the *Book* relation; when a subscriber's address changes, a user should modify the address attribute of the tuple that corresponds to that subscriber. We refer to the particular relations that make up the database at some point in time as the *database state* (at that time).

We now explain how a database state determines a *structure* for the first-order language that was defined by the DDL specification of the database schema. Recall, from Section 6.3.2, that a structure specifies three things: the domain of discourse, the relation with which each predicate symbol is associated, and the element in the domain of discourse with which each constant symbol is associated.

- The domain of discourse is already defined by database schema: it is the union of the data types of the relations' attributes.

[6] Technically, we also need to fix the set of variables. As we have been doing all along, we will use lower case letters of the Latin alphabet, sometimes primed or subscripted, as variables. Thus, we will not bother to explicitly identify the set of variables.

- For each relation name R defined in the schema, the database state has an actual relation (i.e., set of tuples) of the appropriate arity. This is the relation associated with the predicate symbol that corresponds to R.
- The association of constant symbols to elements of the domain of discourse is the natural one. Recall that each constant symbol is a representation of some element of a data type. To each element x of a data type we associate the constant symbol that represents x. For example, we associate the constant symbol 17 to the integer seventeen, and the constant symbol 1998/10/22 to the date October 22, 1998.

We are now ready to address the main point of this subsection; namely, what exactly it means for a first-order formula to "express" a query. First of all, what is a query? Intuitively, a query describes what information to retrieve from the current database state. In a relational database, the database state is a collection of relations (one for each relation name defined in the schema), and the retrieved information is a relation. In mathematical terms, then, we can think of a query as a function which maps any database state S to a relation. A first-order formula is a representation of such a function. Here is how.

As we have seen, we can think of the database schema as a first-order language $\mathcal{L}$ and we can think of the database state as a structure $\mathcal{S}$ of $\mathcal{L}$. Let F be a first-order formula of $\mathcal{L}$ and $free(F)$ be the set of free variables of F. If σ is a valuation of structure $\mathcal{S}$, let σ_F denote the function $\sigma \upharpoonright free(F)$. (Recall that a valuation is a function mapping variable names to elements of the domain of discourse; see Section 0.6 for the definition of the restriction of a function.) Since $free(F)$ is a finite set of variables, say $x_1, x_2, \ldots, x_k$, we can view σ_F as the k-tuple $(\sigma_F(x_1), \sigma_F(x_2), \ldots, \sigma_F(x_k))$.[7] Formula F represents the query which maps the database state $\mathcal{S}$ to the relation consisting of the following set of tuples:

$$\{\sigma_F : \ \sigma \text{ is a valuation such that } (\mathcal{S}, \sigma) \text{ satisfies } F\} \tag{6.17}$$

In other words, the query expressed by F returns the valuations that satisfy F in the structure defined by the database state, restricted to the free variables of F. Note that, by Theorem 6.7, once a structure $\mathcal{S}$ is fixed, whether an interpretation $(\mathcal{S}, \sigma)$ satisfies F depends *only* on the values that σ assigns to the free variables of F. Thus, in restricting the valuations only to those variables in (6.17) we are not "hiding" any information that is relevant to whether the formula F is satisfied.

Now that we have explained how a first-order formula expresses a query, it is instructive to revisit a point that we made informally (and without justification) in the previous subsection. Namely, that if a formula F expresses a query, then any formula that is logically equivalent to F expresses the same query. To see why this is the case note that if F and F' are logically equivalent then, by definition, for any structure $\mathcal{S}$ and any valuation σ, $(\mathcal{S}, \sigma)$ satisfies F if and only if it satisfies F'. This means that the two formulas F and F' return the same set of tuples *for any database state*. In other words, F and F' express the same query.

[7] In order to view σ_F as a tuple, we must order the set of variables $free(F)$ in some fashion — because the elements of a tuple have an order. This is not a problem because any such order of the elements of $free(F)$ will do.

It is perhaps interesting to draw the reader's attention to an analogy that exists between propositional and first-order formulas as *representations of functions*. In Section 5.10 we saw how propositional formulas can be viewed as representations of Boolean functions. This fact has important practical implications for the design of digital hardware. In this section we saw how first-order formulas can be viewed as representations of functions from database states to relations. This fact has important practical implications for the design of database systems.

6.7.4 Integrity constraints

In addition to making possible the definition of the database schema, the DDL also allows a user to specify *integrity constraints*. These are facts that must hold in every database state. The reason for imposing such constraints is to prevent other users from applying to the database updates that would result in a meaningless or illegal database state. Interestingly, first-order formulas provide a convenient language for specifying such constraints.

For example, we mentioned that the attribute book id is supposed to be a unique identifier for each copy of a book in the library. We can express this requirement by the following formula:

$$\forall b\, \forall t\, \forall n\, \forall b'\, \forall t'\, \forall n' \big((Book(b,t,n) \land Book(b',t',n') \land \approx(b,b')) \to (\approx(t,t') \land \approx(n,n')) \big)$$

Informally, this says that if two tuples of the *Book* relation agree on the book id attribute, then they must also agree on the remaining attributes — i.e., they are the same tuple! In other words, book id is a unique identifier for the *Book* relation. (In database terminology, such an attribute is called a *key* of the relation.)

Here is another example of an integrity constraint. It is reasonable for the library to require that only subscribers may borrow books. This requirement can be expressed by the following formula:

$$\forall s \big(\exists b\, \exists d\, Borrowed(s,b,d) \to \exists n\, \exists a\, Subscriber(s,n,a) \big)$$

That is, if a person with SIN s has borrowed a book then that person must be subscriber.

As can be seen from these examples, formulas that express integrity constraints are sentences, i.e., formulas that do not have any free variables. In contrast, formulas that express queries have free variables, namely the attributes of the tuples that we want to retrieve.

6.7.5 Limitations on the expressive power of predicate logic

We remarked earlier on the analogy between the use of propositional formulas as representations of Boolean functions, and the use of first-order formulas as representations of queries. From Theorem 5.20 we know that we can represent *every* Boolean function by a propositional formula. So, it is natural to ask whether we can represent every query — i.e., every function from database states to relations — as a first-order formula. The answer, in this case, is negative: There are natural queries that one might want to pose which are not expressible by first-order formulas.

Such a query in our example database is

$$\text{Find the subscribers who have borrowed a majority (i.e., more than half)} \\ \text{of the books owned by the library.} \tag{6.18}$$

It is interesting to note that for each positive integer k we can write a (different) first-order formula that represents the query

Find the subscribers who have borrowed at least k books owned by the library.

(Hint: We can say this by saying that there are books $b_1, b_2, \ldots, b_k$, distinct from each other, that the subscriber has borrowed.) Nevertheless, no formula can represent (6.18).

Another interesting and natural example of a query that cannot be expressed by a first-order formula is the so-called *transitive closure* query. To explain this type of query, consider the following example. Some software engineering tools use a database to store information about the various components of a software package and about how these components are related. For example, such a database might contain a relation *Procedure* that contains information about the procedures of the software package. This relation might have attributes *ProcedureName*, *NoOfArguments* and *ProcedureAuthor*, specifying, respectively, the name of the procedure, the number of arguments the procedure has and the name of its author. In addition, we might have a relation *Calls* with attributes *Caller* and *Callee*, each of which is a procedure name. A tuple (p, q) is in this relation if and only if the procedure whose name is p calls the procedure whose name is q. If, during the development of the software package, a change is made to the code of some procedure p it may be necessary to update all other procedures that call p directly or indirectly (i.e., via a chain of intermediate calls). Therefore, the user of the software engineering database might wish to express the following query:

$$\text{Find the names of all procedures that directly or indirectly call } p. \tag{6.19}$$

It turns out that this query cannot be expressed by a first-order formula. On the other hand, it is not hard to see that for each integer k we can write a (different) first-order formula that expresses the following query:

Find the names of all procedures that call p via a chain of at most k intermediate calls.

The fact that Queries (6.18) and (6.19) cannot be expressed by first-order formulas is an interesting and nontrivial result in mathematical logic, with practical implications. Because of it, the query languages of real database systems include some special facilities that enable users to ask some of the queries that are not expressible by first-order formulas. The proof of this result, however, is beyond the scope of our brief survey of the subject in these notes. In general, the ability of a formal system (such as predicate logic) to represent various mathematical objects (such as functions) is referred to as the *expressive power* of the system. The study of the expressive power of different formal systems is an important subject in mathematical logic and theoretical computer science.[8]

[8] Our investigation of the completeness and incompleteness of various sets of connectives, in Section 5.11, is

6.8 Logical implication revisited

Definition 6.9 states what it means for a formula to be logically implied by another. It is sometimes useful to speak of a formula being logically implied by *a set* of formulas. The meaning of this is given by the following definition.

Definition 6.12 *A set of formulas Φ logically implies a formula F if and only if every interpretation that satisfies* all *formulas in Φ also satisfies F.*

Example 6.6 Let $\mathcal{L}$ be a first-order language with one binary predicate symbol L, and consider the following formulas of $\mathcal{L}$:

$$\begin{aligned}
F_1 :&\quad \forall x\, \neg L(x,x)\\
F_2 :&\quad \forall x\, \forall y\, \forall z\, \big(L(x,y) \wedge L(y,z) \to L(x,z)\big)\\
F_3 :&\quad \forall x\, \forall y\, \big(L(x,y) \to \neg L(y,x)\big)\\
F_4 :&\quad \forall x\, \forall y\, \big(L(x,y) \vee L(y,x)\big)
\end{aligned}$$

Intuitively, F_1 says that the binary relation represented by L is irreflexive, F_2 says that the relation is transitive, F_3 says that the relation is antisymmetric, and F_4 says that the relation is a total order.

Let $\Phi = \{F_1, F_2\}$. A structure that satisfies both formulas in Φ must therefore interpret L as an irreflexive and transitive relation. There are many such relations: For example, it is easy to verify that the $<$ relation over the natural numbers has these properties, as does the relation $\prec$ over $\mathbb{N} \times \mathbb{N}$, defined as follows:

$$(a,b) \prec (a',b') \quad \text{if and only if } a < a' \text{ and } b < b'$$

For example, $(2,3) \prec (3,5)$ but $(0,3) \not\prec (1,2)$ and $(3,2) \not\prec (1,1)$.

We claim that Φ logically implies F_3. Suppose, for contradiction, that it does not. This means that there is a structure $\mathcal{S}$ that satisfies F_1 and F_2 but falsifies F_3. Since F_3 is false in $\mathcal{S}$, it means that there are elements a and b in the domain of $\mathcal{S}$ such that $(a,b) \in L^{\mathcal{S}}$ and $(b,a) \in L^{\mathcal{S}}$. But then, since $\mathcal{S}$ satisfies F_2, it follows that $(a,a) \in L^{\mathcal{S}}$, which contradicts the assumption that $\mathcal{S}$ satisfies F_1. We have therefore shown that Φ logically implies F_3. Intuitively this means that *every relation that is irreflexive and transitive is necessarily antisymmetric.* Hence, $<$ and $\prec$ are both antisymmetric.

On the other hand, Φ does *not* logically imply F_4. To see this, consider the structure $\mathcal{S}$ that has domain $\mathbb{N} \times \mathbb{N}$ and interprets the predicate symbol L as the relation $\prec$. As we mentioned before, this structure satisfies both formulas in Φ. Clearly, it does not satisfy F_4 since $(0,3) \not\prec (1,2)$ and $(1,2) \not\prec (0,3)$ — i.e., there are elements of the domain neither of which

an elementary example of the study of the expressive power of formal systems. There, we saw that in some cases — e.g., if we are allowed to use only $\wedge$ and $\vee$ — by restricting the connectives that can be used we reduce the expressive power of formulas; while, in other cases — e.g., if we are allowed to use only $\neg$ and $\vee$ — restricting the connectives that can be used does not affect the expressive power of formulas.

is related to the other by $\prec$. Since there exists a structure that satisfies all formulas in Φ and falsifies F_4, Φ does not logically imply F_4.

It is also easy to check that Φ does *not* logically imply $\neg F_4$. We leave the argument proving this as an exercise. Intuitively, the fact that Φ does not logically imply either F_4 or its negation means that there are irreflexive partial orders that are not total orders as well as ones that are. **End of Example 6.6**

The notion of a set of formulas logically implying another is important. Among other things, we can use to describe formally how mathematics works. Each mathematical theory (number theory, group theory, geometry, etc) starts with a set of *axioms* which can be expressed as first-order sentences in some suitable language. Starting with these axioms, we can prove the theorems of the mathematical theory, which can also be expressed as first-order sentences. The set of sentences that represent the axioms of the theory logically imply each sentence that represents a theorem of the theory.

It is very interesting to note that the process of proof can also be formalised in logic, as a set of well-defined mechanical rules which allow us to combine sentences that represent axioms and already proven theorems to yield sentences that represent new theorems. This important branch of logic, called proof theory, has many important applications to computer science but lies outside the scope of our brief survey of logic in these notes.

6.9 Prenex normal form

It is sometimes convenient to deal with formulas in which all the quantifiers appear at the beginning. It turns out, that it is always possible to do this: any formula can be transformed into a logically equivalent one in which all quantifiers appear in the front.

Definition 6.13 *A first-order formula is in* ***Prenex Normal Form (PNF)*** *if and only if it is of the form*

$$\mathbf{Q}_1 x_1 \mathbf{Q}_2 x_2 \ldots \mathbf{Q}_k x_k \, E$$

where $k \geq 0$, each $\mathbf{Q}_i$ is a quantifier ($\forall$ or $\exists$) and E is a quantifier-free *first-order formula.*

For example, $\forall y\,(S(x,y) \to M(x))$ and $\forall x \exists y\,(P(x,y,z) \wedge \neg M(x) \to S(x,z))$ are formulas in PNF. On the other hand, $\forall x \exists y\,(P(x,y,z) \wedge \neg M(x) \to \exists z\, S(x,z))$ and $\exists y\, S(x,y) \to F(x)$ are not in PNF. The second of these may at first appear to be in PNF since it does consist of a quantifier, $\exists y$, followed by a quantifier-free formula, $S(x,y) \to F(x)$. However, this formula is really an abbreviation for $(\exists y\, S(x,y) \to F(x))$, with the outermost parentheses removed, and as can be seen this formula in *not* in PNF. In contrast, the formula $\exists y\,(S(x,y) \to F(x))$ *is* in PNF. As we have discussed, however, this is different from (and, in fact, not even logically equivalent to) $\exists y\, S(x,y) \to F(x)$.

Theorem 6.14 *For any first-order formula F there is a PNF formula $\hat{F}$ that is logically equivalent to F.*

The basic idea is to transform F into $\hat{F}$ by moving the quantifiers in F outwards one-at-a-time, using the "factoring" rules (IIa)–(IIf), and the subformula substitution rule (V). Sometimes we can't directly apply the factoring rules because we need to factor a quantifier $\mathbf{Q}x$ in a subformula such as $E \wedge \mathbf{Q}x\, F$, and x *does* occur free in E. In this (and similar) cases, we first apply the renaming rule (III) and thereby eliminate this difficulty so that we can then apply the relevant factoring rule. The formal proof of Theorem 6.14 is by structural induction on F, and is left as an exercise. Instead we illustrate the process described above by an example. In justifying each step we suppress any reference to rule (V) (subformula substitution):

$$\begin{array}{lll}
 & \forall x \Big(\forall z \left(P(x,y,z) \vee \exists u\, S(z,u)\right) \rightarrow M(z)\Big) & \\
\text{LEQV} & \forall x \Big(\forall v \left(P(x,y,v) \vee \exists u\, S(v,u)\right) \rightarrow M(z)\Big) & \text{[by (III)]} \\
\text{LEQV} & \forall x \exists v \Big(\left(P(x,y,v) \vee \exists u\, S(v,u)\right) \rightarrow M(z)\Big) & \text{[by (IIe)]} \\
\text{LEQV} & \forall x \exists v \Big(\exists u \left(P(x,y,v) \vee S(v,u)\right) \rightarrow M(z)\Big) & \text{[by (IIb)]} \\
\text{LEQV} & \forall x \exists v \forall u \Big(\left(P(x,y,v) \vee S(v,u)\right) \rightarrow M(z)\Big) & \text{[by (IIe)]}
\end{array}$$

The formula on the last line is in PNF and is logically equivalent to the formula we started with.

As this examples shows, to put a formula in Prenex Normal Form, we sometimes need to rename variables. This is one of the technical reasons for requiring the first-order language to have an unlimited supply of variables, to which we alluded in Footnote 1.

6.10 Order of quantifiers

The order of quantifiers of *the same type* does not matter. In general, for any formula E, $\mathbf{Q}x\mathbf{Q}y\, E$ LEQV $\mathbf{Q}y\mathbf{Q}x\, E$. On the other hand, however, the order of quantifiers of *different* types can be very important for the meaning of the formula. As an example, suppose the domain is the set of people, and $L(x,y)$ stands for the predicate "x loves y". Compare the meanings of the formulas

$$\forall x \exists y\, L(x,y) \qquad \text{and} \qquad \exists y \forall x\, L(x,y)$$

The formula on the left states that everybody loves someone; while the one on the right states that someone is loved by everyone! Similarly, if the domain is the set of natural numbers, and $L(x,y)$ stands for the predicate $x < y$, the formula on the left states that for each number there is a larger one (which is true), while the one on the right states that there is a number that is larger than all numbers (which is false).

Let us look more closely into the difference between the formulas $\forall x \exists y\, E$ and $\exists y \forall x\, E$. The first formula, $\forall x \exists y\, E$ is true (in some interpretation) if, for every element x of the domain, we can find some element y so that E holds. Note that the y we choose may depend on the x;

in particular, for different x's we can choose different y's. In contrast, the formula $\exists y \forall x\, E$ is true if there exists an element of the domain so that for every element of the domain E holds. Now we must choose a single y that "works" for all possible values of x; we don't get to choose different y's for different x's!

In general, $\exists y \forall x\, E$ logically implies $\forall x \exists y\, E$ for any formula E. The converse is not necessarily true. In some *special* cases $\exists y \forall x\, E$ and $\forall x \exists y\, E$ can be equivalent. As an exercise show that $\forall x \exists y\, (M(x) \wedge F(y))$ is logically equivalent to $\exists y \forall x\, (M(x) \wedge F(y))$ (hint: transform one to the other using the rules of Section 6.6).

6.11 Reference and its effect on meaning

In this section we explore, in greater detail, the subject of Section 6.2.3 — namely, free and nonfree occurrences of variables — and we discuss the effect of variable names on the meaning of a formula. All the examples in this section involve formulas in the language of familial relationships $\mathcal{LF}$ (see page 146) and can be interpreted by thinking of predicate symbols P, S, M and F as expressing the predicates parents-of, sibling-of, is-male and is-female, respectively.

6.11.1 Scope of quantifiers, and binding of variables

Consider a first-order formula E that contains a subformula of the form $\mathbf{Q}x\, E'$. We refer to the subformula E' as the ***scope*** of the quantifier $\mathbf{Q}x$ in E. Informally, the scope of a quantifier in E is the subformula of E to which the quantifier "applies". In the example below we use braces to show the scope of the various quantifiers that appear in a formula:

$$\exists x\ \underbrace{(\forall y\ \overbrace{(S(x,y) \to F(y))}^{\text{scope of } \forall y} \wedge \exists u\ \overbrace{S(u,x)}^{\text{scope of } \exists u})}_{\text{scope of } \exists x}$$

(This formula says that there is someone all of whose siblings are female and who has a sibling.) In terms of the tree representation of formulas, the scope of a quantifier is the subtree whose root is the node that contains that quantifier.

Using this idea of the scope of a quantifier, we can rephrase Definition 6.2 as follows: An occurrence of variable x is free in a formula if it does *not* occur within the scope of a $\mathbf{Q}x$ quantifier. Now consider an occurrence of x that is not free, i.e., that occurs within the scope of a $\mathbf{Q}x$ quantifier. We want to determine the particular quantifier to which that occurrence of x refers — or, to use different terminology, the quantifier to which the occurrence of x ***is bound***. As we will see, this is critical for understanding the meaning of a formula.

We can't simply say that an occurrence of x is bound to the $\mathbf{Q}x$ quantifier within whose scope it appears, because there may be several such quantifiers. For instance, consider the formula

$$\forall x \underset{\blacktriangle}{\forall y}\, (F(y) \wedge \underset{\blacklozenge}{\forall y}\ \underbrace{(S(x,y) \to M(y))}_{E} \to \neg S(y,x))$$

(This formula says that every female person is not the sibling of anyone all of whose siblings are male.) The two occurrences of y in the subformula E are in the scope of *two* $\forall y$ quantifiers, marked with ▲ and ♦. So, the question is: are the two occurrences of y in E bound to the ▲ or to the ♦ quantifier in the above formula?

Some thought will convince you that the truth value of formulas (see Definition 6.6 in Section 6.3.3) is defined in such a manner that an occurrence of a variable x that is within the scope of multiple x-quantifiers (as is the case for y in our previous example) is bound to the *closest* such quantifier. Thus, in our example, the two occurrences of y in E are bound to the $\forall y$ quantifier marked with ♦.

Notice that, by the definition of first-order formulas, if an occurrence of a variable appears within two subformulas, say E_1 and E_2, then one of E_1, E_2 must be a subformula of the other. (This is because formulas are built up in a tree-like fashion from smaller ones.) As a result, when an occurrence of x is within the scope of multiple x-quantifiers, there is a unique such quantifier that is "closest" to the occurrence of x. In terms of the tree representation of formulas, an occurrence of a variable x is bound to the first quantifier in the path from the node that contains the occurrence to the root of the tree. So, our rule about which quantifier an occurrence of x is bound to is well-defined.

To summarise, each occurrence of a variable x in a formula is either free or bound; in the latter case, it is bound to a particular $\mathbf{Q}x$ quantifier: the closest one within whose scope the occurrence of x lies. Note that, within a single formula, different occurrences of the same variable may be free or bound; and that different nonfree occurrences of a variable may be bound to the same or to different quantifiers. In the example below we have used asterisks to mark the free occurrences of variables, and arrows to indicate the quantifier to which each nonfree occurrence of a variable is bound.

$$\forall \underleftarrow{\underleftarrow{x\,(S(x}, \underset{*}{y})\to\exists \overleftarrow{\overleftarrow{u\,(\exists \underleftarrow{v P(v}, u}, \underset{*}{y}) \wedge \exists \underleftarrow{v\, P(v}, u}, x})))$$

(This formula says that every sibling of y has at least one parent in common with y.)

6.11.2 Binding of variables and meaning

The binding of variables to quantifiers crucially affects a formula's meaning. For example, compare the following two formulas:

$$\forall x\,\big(M(x) \vee F(x)\big) \qquad \text{and} \qquad \forall x\, M(x) \vee \forall x\, F(x)$$

The first formula asserts that each person is either male or female (which is at least plausible), while the second formula asserts that either each person is male or each person is female (which is surely false). Note the difference in the binding of variables to quantifiers in these two formulas: in the first, the two occurrences of x are bound to the same universal quantifier; in the second, each occurrence of x is bound to a different universal quantifier. It is precisely this difference that results in the two different meanings.

The binding of variables to quantifiers in first-order formulas plays a role analogous to that often played by pronouns (and related constructs) in English. For instance one (slightly awkward) way of expressing the formula $\forall x\,(M(x) \vee F(x))$ is to say: For each person, either *it* is male or *it* is female — where the pronoun "it" refers (twice) to "the person in question". In this case the binding is relatively simple because we have two references to the same person, so we can use the pronoun "it" without creating any ambiguity as to which person it refers.

In contrast, consider the formula $\forall x \exists y\, S(x, y)$ (everybody has a sibling). If we tried to express this using pronouns to bind the persons involved to the quantifiers to which they refer we would get:

For each person there is a person so that *it* is *its* sibling

which is hopelessly ambiguous. One way to make clear, in English, which of the two instances of "it" refers to which of the two persons in question is by using the words "former" and "latter". So, we might express this statement as:

For each person there is a person so that *the latter* is a sibling of *the former*.

Things get much more complicated if there are more quantifiers involved. These examples should make clear the great advantage of first-order logic over natural languages in expressing complex statements in a relatively succinct *and completely unambiguous manner*. The mechanism by which variables are bound to quantifiers is key in securing these beneficial properties of first-order logic as a formal notation.

The idea that the complex predicates that arise in mathematical statements can be conveniently and precisely expressed using a fairly simple vocabulary, involving quantifiers, was fruitfully pursued by the great German logician and philosopher Gottlob Frege towards the end of the 19th century. He devised a precise notation for expressing predicates in this manner which included, crucially, a mechanism for binding variables to quantifiers. Although his notation was much more complicated and awkward compared to that we use in logic today, he is widely considered as the founder of modern logic.

6.11.3 Variables in formulas and variables in programs

The rule for binding occurrences of variables to quantifiers is very similar to the scoping rules of most programming languages. Consider, for instance, the following C program:

```
main()
{
    int x; x=1; printf("%d",x);
    if (1) {int x; x=2; printf("%d",x);}
    if (1) {printf("%d",x); x=3;}
    if (1) {int x; x=4; printf("%d",x);}
    printf("%d",x);
}
```

The sequence of values printed by this program is: $1, 2, 1, 4, 3$. To see this you should match (or bind) each occurrence of variable x to the declaration (i.e., the statement `int x`) to which it refers; occurrences bound to the same declaration refer to the same object x, while occurrences bound to different declarations refer to different objects named x. The scoping rules used to determine which variable occurrence binds to which declaration in a program are the same as those used to determine which (nonfree) occurrence of a variable binds to which quantifier in a first-order formula.

There is another interesting analogy between variables in a formula and variables in a program: A *free* variable in a formula is similar to a *global* variable in a program; in contrast, a *bound* variable in a formula is similar to a *local* variable in a program. If we take two procedures P_1 and P_2 that use x as a global variable, and put them together in the context of a larger program P, the references to x in P_1 and P_2 are references to the same object x. Similarly, if we take two formulas E_1 and E_2 that use x as a free variable (e.g., E_1 might be $M(x)$ and E_2 might be $F(x)$) and we put them together to form a larger formula (e.g., the formula $M(x) \vee F(x)$), the occurrences of x in this larger formula refer to the same object x.

In contrast, if we take two procedures P_1 and P_2 each of which use x as a local variable, and put them together in the context of a larger program P, the references to x in P_1 and P_2 refer to different objects named x. Similarly, if we take two formulas E_1 and E_2 in which x is not free (e.g., E_1 might be $\forall x\, M(x)$ and E_2 might be $\forall x\, F(x)$) and we put them together to form a larger formula (e.g., the formula $\forall x\, M(x) \vee \forall x\, F(x)$), the occurrences of x in this larger formula refer to different objects x.

As we remarked in our discussion of the Renaming rule in Section 6.6, renaming *free* variables does not preserve logical equivalence while renaming *bound* variables does. This makes perfect sense in terms of our analogy between variables in formulas and variables in programs. If we change the name of a *free* variable we change the meaning of a formula — just as if we change the name of a *global* variable used in a program P we change what the program does: Another program that uses P may no longer work correctly because it expects a value in one global variable and P (after the renaming) puts that value in another. In contrast, if we change the name of a *bound* variable in a formula we don't change the meaning of the formula — just as if we change the name of a *local* variable used in a program P we don't change what the program does: Any other program that uses P will work just as well after the renaming as it did before.

6.11.4 Renaming revisited

Recall the renaming logical equivalence, i.e., rule (III) in Section 6.6. This rule says that if we take a formula $\mathbf{Q}x\, E$ and we rename the quantified variable, x, and all the occurrences of x that are bound to the leading quantifier, $\mathbf{Q}x$, by a new variable, y, *that does not occur in* E, then the resulting formula, $\mathbf{Q}y\, E^x_y$, is logically equivalent to the original one, $\mathbf{Q}x\, E$.

The essence of this rule is that the renaming must occur in such a manner as to preserve the binding of variables to quantifiers. Although the particular names of the quantified variables are not important to a formula's meaning, the pattern of variable-to-quantifier binding is critical. Change this binding and, in general, you change the meaning of the formula.

The requirement that the new name y that we use to replace x should not occur in E is stronger than necessary. In some cases, we can preserve the variable-to-quantifier bindings while using a variable name that does occur in E. A more general (but more complex) rule for when x can be renamed y in $\mathbf{Q}x\,E$, while preserving the variable-to-quantifier binding, is the following: (a) there is no free occurrence of y in E, and (b) there is no free occurrence of x in E that is within the scope of a y-quantifier. We now discuss the necessity of (a) and (b), starting with the former.

If y is a free variable in E, then $\mathbf{Q}x\,E$ and $\mathbf{Q}y\,E^x_y$ have different sets of free variables: y is free in the former, but not in the latter. Thus, the two formulas are, in general, not logically equivalent.

Regarding (b), if x occurs free within the scope of a y-quantifier in E, then any such occurrence is bound to a different quantifier in $\mathbf{Q}x\,E$ than in $\mathbf{Q}y\,E^x_y$: In the former it is bound to the leading $\mathbf{Q}x$ quantifier (since it is free in E), while in the latter it is bound to some y quantifier within E^x_y, and not to the leading $\mathbf{Q}y$ quantifier. Since the renaming of x to y in this case changes the binding of variables to quantifiers, the meaning of the resulting formula, in general, is different and the two formulas are not logically equivalent.

For a concrete example, consider the formula

$$\exists x\ \underbrace{\Big(M(x) \land \forall y\,\big(S(x,y)\rightarrow F(y)\big)\Big)}_{E}$$

(This says that there is someone who is male and all of whose siblings are female.) Variable x occurs free within the scope of a $\forall y$ quantifier in E. If we rename each free occurrence of x to y in E we get

$$\exists y\ \underbrace{\Big(M(y) \land \forall y\,\big(S(y,y)\rightarrow F(y)\big)\Big)}_{E^x_y}$$

(This says that there is someone who is male and everyone who is oneself's sibling is female — quite different from what $\exists x\,E$ says!) In contrast, if we rename each free occurrence of x to z in E we get

$$\exists z\ \underbrace{\Big(M(z) \land \forall y\,\big(S(z,y)\rightarrow F(y)\big)\Big)}_{E^x_z}$$

(This, just like $\exists x\,E$, says that there is someone who is male and all of whose siblings are female.)

In summary, the revised and more general renaming rule is:

III′. $\mathbf{Q}x\,E$ LEQV $\mathbf{Q}y\,E^x_y$, for any formula E and any variables x, y such that y does not occur free in E and x does not occur free within the scope of a y-quantifier in E.

Exercises

1. Consider the first-order language of arithmetic described on page 146 of the notes. Let $\mathcal{N}$ and $\mathcal{Z}$ be structures for this language, with domains $\mathbb{N}$ and $\mathbb{Z}$, respectively, and the standard meaning for the predicate symbols. More formally:

$$\begin{array}{ll}
S^{\mathcal{N}} = \{(a,b,c) \in \mathbb{N}^3 : a+b=c\} & S^{\mathcal{Z}} = \{(a,b,c) \in \mathbb{Z}^3 : a+b=c\} \\
P^{\mathcal{N}} = \{(a,b,c) \in \mathbb{N}^3 : a \cdot b=c\} & P^{\mathcal{Z}} = \{(a,b,c) \in \mathbb{Z}^3 : a \cdot b=c\} \\
L^{\mathcal{N}} = \{(a,b) \in \mathbb{N}^2 : a<b\} & L^{\mathcal{Z}} = \{(a,b) \in \mathbb{Z}^2 : a<b\} \\
\approx^{\mathcal{N}} = \{(a,b) \in \mathbb{N}^2 : a=b\} & \approx^{\mathcal{Z}} = \{(a,b) \in \mathbb{Z}^2 : a=b\} \\
\boldsymbol{0}^{\mathcal{N}} = 0 & \boldsymbol{0}^{\mathcal{Z}} = 0 \\
\boldsymbol{1}^{\mathcal{N}} = 1 & \boldsymbol{1}^{\mathcal{Z}} = 1
\end{array}$$

For each of the sentences below, state whether it is true or false in each of $\mathcal{N}$ and $\mathcal{Z}$. Justify your answer by translating the formula into a statement (in precise English) about numbers, and then explain why that statement is true or false for natural numbers and for integers.

(a) $\exists x \, \forall y \, \big(L(x,y) \vee \approx(x,y)\big)$

(b) $\exists x \, \forall y \, \big(L(y,x) \vee \approx(x,y)\big)$

(c) $\forall x \, \exists y \, L(x,y)$

(d) $\forall x \, \exists y \, L(y,x)$

(e) $\forall x \, \exists y \, S(x,y,\boldsymbol{0})$

(f) $\exists x \, \forall y \, S(x,y,\boldsymbol{0})$

(g) $\exists x \, \forall y \, S(x,y,y)$

(h) $\forall x \, \forall y \, \Big(L(x,y) \rightarrow \forall u \, \forall v \, \big(P(x,x,u) \wedge P(y,y,v) \rightarrow L(u,v)\big)\Big)$

2. Consider the same first-order language of arithmetic as in the previous exercise, enriched with a new constant symbol $\boldsymbol{2}$ which is intended to represent the integer 2. In this question we will be working with this language in the structure $\mathcal{N}$ also defined in the previous exercise. The following formula *Prime*(x) expresses the predicate "x is prime" in $\mathcal{N}$:

$$\mathit{Prime}(x): \qquad L(\boldsymbol{1},x) \wedge \forall y \, \forall z \, \Big(P(y,z,x) \rightarrow \big(\approx(y,\boldsymbol{1}) \vee \approx(z,\boldsymbol{1})\big)\Big)$$

(a) ***Goldbach's conjecture*** asserts that every even integer greater than 2 can be expressed as the sum of two prime numbers. Nobody knows whether this is true or false. Write a formula to express Goldbach's conjecture. In your answer you may use the predicate *Prime*(x) for which a formula was given above.

(b) ***Twin primes*** are prime numbers that are two apart; e.g., 3 and 5 are twin primes, as are 17 and 19. The ***twin-prime conjecture*** asserts that there are infinitely many twin primes. Nobody knows whether this is true or false. Write a formula to express the twin-prime conjecture. (**Hint:** You can say that there are infinitely many numbers with property P by saying that for each number there is a larger number with property P.)

3. In the previous exercise we asserted that the formula $\mathit{Prime}(x)$ expresses the predicate "x is prime" in $\mathcal{N}$: More precisely, this means that an interpretation $(\mathcal{N}, \sigma)$ satisfies $\mathit{Prime}(x)$ if any only if $\sigma(x)$ is a prime number.

(a) Explain why the formula $\mathit{Prime}(x)$ given in the previous exercise does ***not*** express "x is prime" in $\mathcal{Z}$ (where the domain is ***all*** integers, including negative ones).

(b) Give a formula that expresses "x is prime" in $\mathcal{Z}$.

4. Write a formula to express Proposition 1.7 (page 27) in the notes.

5. Professors John Friedlander of the University of Toronto and Henryk Iwaniec of Rutgers University recently proved that there are infinitely many primes of the form $y^2 + z^4$, where y and z are integers. Write a first-order formula in the language of arithmetic that expresses this fact. Your formula may contain the predicate $\mathit{Prime}(x)$, as a shorthand for the formula given in Exercise 2.

6. For each of the following assertions, state whether it is true or false, and justify your answer. In (a)–(d) A and B are unary predicates in the first-order language.

(a) $\forall x \, \big(A(x) \rightarrow B(x)\big)$ logically implies $\exists x \, \big(A(x) \wedge B(x)\big)$.

(b) $\exists x \, \big(A(x) \rightarrow \neg B(x)\big)$ is logically equivalent to $\neg \forall x \, \big(A(x) \wedge B(x)\big)$.

(c) $\exists x \, A(x) \wedge \exists x \, \neg A(x)$ is logically equivalent to $\exists x \, \big(A(x) \wedge \neg A(x)\big)$.

(d) $\exists x \, A(x) \vee \exists x \, \neg A(x)$ is logically equivalent to $\exists x \, \big(A(x) \vee \neg A(x)\big)$.

(e) For any first-order formulas E and F such that x does not appear free in F, $\forall x \, E \leftrightarrow F$ is logically equivalent to $\forall x \, (E \leftrightarrow F)$.

7. Consider a relational database that describes an aspect of the activities in a university. Specifically, the database schema consists of the following relations:

- $\mathit{Student}(s, n, a)$ — a tuple (s, n, a) belongs to this relation if the student whose SIN is s has name n and address a. We assume that the SIN uniquely identifies a student; however, there could be different students with the same name (or address).
- $\mathit{Course}(c, n)$ — a tuple (c, n) belongs to this relation if there is a course whose code is c (e.g., CSCB38) and whose name is n (e.g., Discrete Mathematics).
- $\mathit{Takes}(s, c, y, m)$ — a tuple (s, c, y, m) belongs to this relation if the student whose SIN is s took (or is now taking) course c in year y and received a mark of m. We assume that if a student is presently taking the course but has not completed it yet, the mark m has value I (for "incomplete").
- $\mathit{Teaches}(n, c, y)$ — a tuple (n, c, y) belongs to this relation if the professor whose name is n taught (or is now teaching) course c in year y. We assume that each professor is uniquely identified by her/his name, and that each course is offered only once in each year.

Write formulas to express the following queries:

(a) Find the name and address of every student who took (or is taking) CSCB38 in 1999.

(b) Find the names of all courses ever taught by MacLean.

(c) Find the names of all students who received an A in CSCB70 when Panario taught the course.

(d) Find the names of all courses that were taught *only* by Rackoff.

(e) Find the names of all courses taught in 1999 by any professor who has never failed a student in any course that she/he ever taught. (A student fails a course, if her/his mark is F.)

(f) Find the name of every student who has received an A in every course that she/he has completed (i.e., every course that the student has taken except those that she/he is presently taking).

(g) Find the name of every course that has not been taken by any student who has taken (or is presently taking) CSCB38.

8. Let F and F' be first-order formulas that express queries Q and Q', respectively, and suppose that F logically implies F'. Which of the following is true?

(a) Query Q always returns at least as many tuples as query Q' does, regardless of the database state.

(b) Query Q always returns no more tuples than query Q' does, regardless of the database state.

(c) Query Q may return more, fewer, or equally many tuples as query Q' does, depending on the database state.

9. Suppose we wish to retrieve the titles of the books that have been borrowed by all subscribers. Note that this description is ambiguous because it is not clear whether we mean to retrieve the books every copy of which was borrowed by all subscribers, or the books at least one copy of which was borrowed by all subscribers. Give two first-order formulas, each expressing one of the two meanings of this query.

10. Consider the query expressed by Formula (6.16). If we ask this query in a database state in which the library owns no book by Marquez (i.e., there is no tuple of the form $(*, *,$ "Marquez"$)$ in relation *Book*), what set of tuples will be returned?

11. For each of the statements below state whether it is true or false and justify your answer. For any formula F and any set of formulas Φ:

(a) If Φ logically implies F then Φ does not logically imply $\neg F$.

(b) If Φ does not logically imply F then Φ logically implies $\neg F$.

Chapter 7

FINITE STATE AUTOMATA AND REGULAR EXPRESSIONS

7.1 Introduction

In this chapter we look at a small part of a great mathematical theory, known as the theory of formal languages and automata, which was largely motivated by applications in computer science. We will give some sense of the wide applicability of this theory in Section 7.1.3, but first we need to establish some basic definitions.

7.1.1 Strings and languages

Definition 7.1 *An* ***alphabet*** *is a set Σ whose elements are called* ***symbols****. A* ***string*** *(over a specified alphabet Σ) is a finite sequence of symbols from Σ. The empty sequence is a string and is denoted ϵ. The set of all strings over alphabet Σ is denoted Σ^*.*

When discussing strings there are some notational conventions to which we will adhere. When we write out a string as a sequence we generally do not separate its elements by commas and we do not enclose the entire sequence within angled brackets $\langle \ldots \rangle$. Thus, if the alphabet is $\{0, 1\}$, we write 0100 instead of $\langle 0, 1, 0, 0 \rangle$; if the alphabet is $\{a, b, \ldots z\}$, we write *turn* instead of $\langle t, u, r, n \rangle$.[1] We use lower-case letters near the beginning of the (English) alphabet, such as a, b, c, to denote generic symbols in Σ. We use lower-case letters near the end of the (English) alphabet, such as x, y, z, to denote generic strings in Σ^*.

Strings, being (finite) sequences, inherit definitions, operations and relationships that apply to sequences in general (see Chapter 0, page 11). We repeat some of these here. The ***length***

[1] This means that the symbols of the alphabet must be regarded as "atomic" — i.e., not decomposable into smaller units. For example, we can't have symbols a and aa in the alphabet because then our convention of leaving out the commas would make the string aaa ambiguous: is it the sequence whose first element is a and whose second element is aa (i.e., $\langle a, aa \rangle$), the sequence whose first element is aa and whose second element is a (i.e., $\langle aa, a \rangle$), or the sequence that has three elements, all of them a (i.e., $\langle a, a, a \rangle$)? The assumption of symbol "atomicity" can be made with no loss of generality, since we can think of the k elements of alphabet Σ as the atomic symbols $a_1, a_2, \ldots, a_k$.

of a string x, denoted $|x|$, is the number of elements in the sequence x; e.g., $|mississippi| = 11$; $|\epsilon| = 0$. The ***concatenation*** of strings x and y, denoted $x \circ y$ or xy, is the sequence obtained by juxtaposing y after x; e.g., $abaa \circ babb$ is the string $abaababb$. The ***reversal*** of string x, denoted $(x)^R$ is the string obtained by listing the elements of x in reverse order; e.g. $(abab)^R$ is the string $baba$. For any $k \in \mathbb{N}$, we define the ***k-th power*** of a string x, denoted x^k by induction on k:

$$x^k = \begin{cases} \epsilon, & \text{if } k = 0 \\ x^{k-1} \circ x, & \text{if } k > 0 \end{cases} \tag{7.1}$$

Thus, exponentiation of strings is repeated concatenation.

Two strings x, y are ***equal***, denoted $x = y$, if $|x| = |y|$ and the ith symbol of x is the same as the ith symbol of y, for every i such that $1 \leq i \leq |x|$. A string x is a ***substring*** of string y if there are strings x', x'' (either or both of which may be ϵ) such that $x'xx'' = y$; if $x' \neq \epsilon$ or $x'' \neq \epsilon$ then x is a ***proper substring*** of y.[2] A string x is a ***prefix*** of string y if there is a string x' (possibly $x' = \epsilon$) s.t. $xx' = y$; if $x' \neq \epsilon$ then x is a ***proper prefix*** of y. We say that x is a ***suffix*** of y if there is an x' (possibly $x' = \epsilon$) s.t. $x'x = y$; if $x' \neq \epsilon$ then x is a ***proper suffix*** of y. Thus, *turn* is a (proper) substring of *nocturnal*, a (proper) prefix of *turnip*, and a (proper) suffix of *saturn*.

We have defined the set of strings over Σ as the set of all finite sequences whose elements belong to Σ. We can also define the same set recursively.

Definition 7.2 *Let $\mathcal{S}_\Sigma$ be the smallest set such that:*

BASIS: $\epsilon \in \mathcal{S}_\Sigma$.

INDUCTION STEP: *If $x \in \mathcal{S}_\Sigma$ and $a \in \Sigma$ then $xa \in \mathcal{S}_\Sigma$.*

Using induction it is easy to prove that the set defined above is the set of all strings over Σ. That is,

Theorem 7.3 *The set $\mathcal{S}_\Sigma$ defined in Definition 7.2 is equal to Σ^*.*

In other words, this theorem assures us that the set we defined recursively is truly the set of strings over Σ.

This alternative definition of the set of strings over Σ is quite useful because it allows us to define various operations on strings and to prove various properties of strings, using structural induction. We illustrate this with some examples.

Example 7.1 The reversal of a string x, $(x)^R$, can be defined recursively, by structural induction on x.

BASIS: $x = \epsilon$. In this case, $(x)^R = \epsilon$.

INDUCTION STEP: $x = ya$, for some $y \in \Sigma^*$ and $a \in \Sigma$, where we assume, inductively, that $(y)^R$ has been defined. In this case, $(x)^R = a(y)^R$.

[2] Note that the notion of substring corresponds to the notion of *contiguous* subsequence, rather than to the notion of subsequence.

We can now use structural induction to prove an interesting property of the reversal operation.

Theorem 7.4 *For all strings x and y, $(xy)^R = (y)^R(x)^R$.*

Proof. Let $P(y)$ be the predicate on strings

$$P(y): \qquad \text{for any string } x,\ (xy)^R = (y)^R(x)^R$$

Using structural induction we will prove that $P(y)$ holds for all y.
Basis: $y = \epsilon$. Then by the basis of the recursive definition of reversal, $y = (y)^R = \epsilon$. Thus,

$$\begin{aligned}
(xy)^R &= (x\epsilon)^R && \text{[since } y = \epsilon \text{, in this case]}\\
&= (x)^R && \text{[since } w\epsilon = w \text{, for any string } w\text{]}\\
&= \epsilon(x)^R && \text{[since } \epsilon w = w \text{, for any string } w\text{]}\\
&= (y)^R(x)^R && \text{[since } (y)^R = \epsilon \text{, in this case]}
\end{aligned}$$

so $P(y)$ holds.
Induction Step: Let y' be an arbitrary string such that $P(y')$ holds, i.e., for any x, $(xy')^R = (y')^R(x)^R$. We must prove that $P(y)$ also holds for any string y that can be constructed from y' by the inductive step of the recursive definition of strings. That is, we must prove that, for any $a \in \Sigma$, if $y = y'a$ then $P(y)$ holds. We have,

$$\begin{aligned}
(xy)^R &= (xy'a)^R && \text{[since } y = y'a \text{, in this case]}\\
&= a(xy')^R && \text{[by the recursive definition of reversal]}\\
&= a((y')^R(x)^R) && \text{[by the induction hypothesis]}\\
&= (a(y')^R)(x)^R && \text{[by associativity of string concatenation]}\\
&= (y'a)^R(x)^R && \text{[by the recursive definition of reversal]}\\
&= (y)^R(x)^R && \text{[since } y = y'a \text{, in this case]}
\end{aligned}$$

so $P(y)$ holds. □

We will be using proofs by structural induction on strings extensively in this chapter.

End of Example 7.1

Definition 7.5 *A **language** (over alphabet Σ) is a subset of Σ^*.*

Note that a language may be an infinite set; each string in the language, however, is finite. Also note that $\emptyset$ and $\{\epsilon\}$ are different languages. When we use the word "language" in the sense of Definition 7.5, we sometimes qualify it with the adjective "formal" to distinguish it from other uses of that word.

7.1.2 Operations on languages

Let L, L' be languages over Σ. It is natural to combine these in order to obtain new languages, using various set-theoretic or string-oriented operations.

Complementation: $\overline{L} = \Sigma^* - L$.

Union: $L \cup L' = \{x : x \in L \text{ or } x \in L'\}$.

Intersection: $L \cap L' = \{x : x \in L \text{ and } x \in L'\}$.

Concatenation: $L \circ L'$. This is the language consisting of concatenations of strings from L and L'. That is, $L \circ L' = \{xy : x \in L \text{ and } y \in L'\}$. Note that, by the definition of concatenation, it follows that $\emptyset \circ L = L \circ \emptyset = \emptyset$, for every language L.

Kleene star: $L^{\circledast}$. Informally, the Kleene star (or Kleene closure) of language L, denoted $L^{\circledast}$, is the set of all possible concatenations of 0 or more strings in L.[3] Formally, $L^{\circledast}$ is defined (by induction) as the smallest language such that:

Basis: $\epsilon \in L^{\circledast}$.

Induction Step: If $x \in L^{\circledast}$ and $y \in L$ then $xy \in L^{\circledast}$.

It is easy to see that for any language L, a string x belongs to $L^{\circledast}$ if and only if either $x = \epsilon$ or there exists some integer $k \geq 1$ and strings $x_1, \ldots, x_k \in L$ such that $x = x_1 x_2 \ldots x_k$. This alternative characterisation of the Kleene star operation is sometimes useful.

Language exponentiation: Informally, for any $k \in \mathbb{N}$, L^k is the set of strings obtained by concatenating k strings of L. More precisely, the definition is by induction on k. For any $k \in \mathbb{N}$,

$$L^k = \begin{cases} \{\epsilon\}, & \text{if } k = 0 \\ L^{k-1} \circ L, & \text{if } k > 0 \end{cases}$$

Note that, by this definition, $L^1 = L$. See Exercise 2 for an interesting connection between language exponentiation and the Kleene star operation.

Reversal: The reversal of L, $\mathbf{Rev}(L)$ is the set of reversals of the strings in L. That is,

$$\mathbf{Rev}(L) = \{(x)^R : x \in L\}$$

[3] This operation is named after Stephen Kleene, a mathematician who made many important contributions to areas of logic that are closely related to computer science. It is traditional to use L^* (rather than $L^{\circledast}$) to denote the Kleene star of language L. However, it is also traditional to use the same symbol, '$*$', as the regular expression operator that corresponds to this operation. Regular expressions, which will be introduced in the next section, are formal expressions that denote languages. At this point, it is important to keep firmly in mind the distinction between the operation itself and the formal symbol used in regular expressions to denote it. To emphasise this distinction we use the symbol $\circledast$ for the former and $*$ for the latter. This is very similar to the reason why in Chapter 6 we use different symbols for the predicate symbol denoting equality, $\approx$, and the actual predicate of equality, $=$ (see Footnote 2 on page 146).

7.1.3 Examples

The definition of formal language given earlier is certainly related to real languages. For example, if the alphabet consists of the letters of the Latin alphabet (in lower and upper case), certain well-defined punctuation marks (like period, comma, semicolon, etc), and certain other well-defined special symbols such as "blank", then indeed some strings over this alphabet belong to the English language and others do not. For instance, the string

This is a legitimate string of the English language.

belongs to this language, while the string

hlgkqk jlkjhllla kjafdgoeirtmbd

does not. Therefore, we can think of the English language as a set of strings.[4] Formal languages can be used to model and fruitfully study a wide variety of phenomena and artifacts. We now present some examples to illustrate this fact.

Example 7.2 ***Propositional logic and first-order logic formulas.*** The set of propositional formulas defined in Chapter 5 (see Definition 5.1, page 112) is a language over the alphabet $PV \cup \{(,), \neg, \wedge, \vee, \rightarrow, \leftrightarrow\}$, where PV is the set of propositional variables. Indeed, some strings over this alphabet, like

$$((x \wedge \neg y) \rightarrow z)$$

belong to this language and others, like

$$(x \neg y) \wedge \vee z($$

do not. Definition 5.1 determines precisely which strings belong to that language. Actually, strictly speaking, what we defined is not a *single* language but rather an entire *class* of languages — one for each choice of the set of propositional variables. If the set of propositional variables is infinite (as is sometimes desirable) this example illustrates the possibility of languages over infinite alphabets.

Similar comments apply regarding the language of first-order formulas defined in Chapter 6 (see, in particular, Sections 6.2.1 and 6.2.2). **End of Example 7.2**

Example 7.3 ***Formal languages and compilation.*** Many aspects of programming languages can be conveniently described as languages in the sense of the mathematical definition given above. Consider, for example, an assignment statement, available in all procedural programming languages, that might have the form

$$\textit{variable-name} := \textit{numeral}$$

[4] Of course, the view of natural language as merely a set of strings is a very narrow one. There are aspects of natural language (such as syntax and even semantics) for which this point of view can be, to some extent, useful — and many other aspects (such as literature and poetry) for which this view is totally useless.

where *variable-name* is an identifier such as x or *temp*; and *numeral* is a numeric constant, such as 17 or -2.17.

The set of ***identifiers*** can be viewed as the language L_{id} consisting of all nonempty alphanumeric strings whose first symbol is alphabetic. Similarly, we can view numerals (in decimal notation) as the language L_{num} consisting of all nonempty strings over the alphabet $\{0, 1, 2, \ldots, 9\}$. If we want to allow the representation of negative numbers and of fractional numbers, we can define a more complicated language of numerals L_{num} which includes strings such as -17, $+3.17$ and $-.17$. This is the language over the alphabet

$$\{0, 1, 2, \ldots, 9\} \cup \{+, -, .\}$$

that consists of all nonempty strings that are the concatenation of four strings x_1, x_2, x_3 and x_4, where

- x_1 is the empty string, or the string consisting of just the symbol $+$, or the string consisting of just the symbol $-$;
- x_2 is a string over the alphabet $\{0, 1, 2, \ldots, 9\}$;
- x_3 is the empty string, or the string consisting of just the symbol .; and
- x_4 is a string over the alphabet $\{0, 1, 2, \ldots, 9\}$.

Not only various pieces of a programming language (like the set of identifiers, or numerals), but the entire programming language itself can be viewed as a formal language in the sense of Definition 7.5. The symbols of the alphabet of this language are so-called ***lexical tokens***. These are things like identifiers (elements of a language such as L_{id} defined above), numeric constants (elements of a language such as L_{num} defined above), reserved keywords of the language (strings such as *if*, *then*, *else*, *for*, *while*, etc), certain symbols (such as semicolon, parentheses etc), and so on. Then the programming language consists of the set of strings over the alphabet of lexical tokens that conform to certain syntactic rules whose precise nature depends on the particular programming language at hand.

The process of compiling a program in a given programming language consists of three phases. The first task, called ***lexical analysis***, is to break up the program into its lexical tokens. The second task, called ***parsing***, is to determine whether the given program is syntactically correct — i.e., whether the lexical tokens have been strung together in accordance with the rules of the programming language. If the program is found to be syntactically correct, the compiler proceeds to the third phase, called ***code generation***. In this phase the given program is translated into an “equivalent” program in a different language that can be executed — typically, the machine language of the processor on which the program will be run.

Each of the three phases of compilation is a computational task involving formal languages. Lexical analysis and parsing require algorithms for ***language recognition***. This is the task of determining whether a given string belongs to some specified language. Code generation requires algorithms for ***language translation***. This is the task of mapping strings from a given “source language” (e.g., the language of C programs) into “equivalent” strings in another “target language” (e.g., the language of programs consisting of Intel Pentium instructions).

The details of the various facets of compilation are fascinating and the subject of separate courses. The point we wish to make here is that the abstract mathematical theory of formal languages (glimpses of which we will get in this chapter) has important practical applications. It is fair to say that, without advances in this theory, we would not have high-level programming languages. **End of Example 7.3**

Example 7.4 ***Describing the behaviour of systems.*** Our final example illustrates the use of languages to formally model the desired behaviour of a simple distributed system. Suppose we have a small network with two workstations belonging to two users sharing a common printer. A user who wishes to print a document does so by invoking the "print" command on his/her workstation. Of course, the printer should not be printing documents of both users at the same time — otherwise, the pages of the two users' documents will come out all mixed together. For this reason, the two users must coordinate their access to the printer to ensure the property of ***mutual exclusion***: only one user's job can be printing at any one time. Each user can issue a request to use the printer, the printer can grant access to a requesting user, and a user who has been granted access can release the printer.

We want to give a formal description of the correct behaviour of such a system. There are at least two reasons for doing so:

Specification: The designer of the system must *specify* the desired behaviour of the system, so that the implementor knows what kind of system to build.

Documentation: The implementor of the system must *document* the behaviour of the system he/she built, so that future implementors can modify the system.

When we say that we want to describe the "correct behaviour of the system" we mean, for example, that we must preclude a behaviour in which a user relinquishes the printer without having been previously granted access to it, or that the printer grants access to a user who has not previously requested it. We also want to preclude other undesirable behaviours, such as the printer granting access to both users at the same time.

We can use formal languages to provide a precise description of the correct behaviour of such a system, as we now proceed to show. There are three kinds of events of interest in this system: The "request" event, indicating that a user requests access to the printer; the "grant" event, indicating that the printer has granted a user's request; and the "done" event, indicating that a user relinquishes its access. Since there are two users, we will need to distinguish between events of the two users. Thus we use the symbols $\mathtt{R}$, $\mathtt{G}$ and $\mathtt{D}$, respectively, to denote the request, grant and done events for the first user; similarly, $\mathtt{r}$, $\mathtt{g}$ and $\mathtt{d}$ denote these three events for the second user.

We can now view a behaviour of the system (not necessarily a correct one), as a string over the alphabet consisting of these six events: $\Sigma = \{\mathtt{R}, \mathtt{G}, \mathtt{D}, \mathtt{r}, \mathtt{g}, \mathtt{d}\}$. The relative position of two events in the string indicates their temporal ordering: if the symbol corresponding to an event e precedes (in the string) the symbol corresponding to event e', then the string models a behaviour of the system in which e occurred before e'. Here are some example strings in Σ^*:

$$\begin{aligned} x &= \texttt{RrgdGrDRGDRGDgd} \\ y &= \texttt{RGrDdGD} \\ z &= \texttt{rgRGDd} \end{aligned}$$

Let us trace the first string in detail to understand what system behaviour it models: The first event that occurs in this behaviour is that the first user requests access to the printer (the initial `R` symbol). Next, the second user requests access to the printer, the request is granted, and the user then releases the printer (this corresponds to the substring `rgd`). Then the first user's pending request is granted, the second user requests access to the printer again, and the first user (who presently has access to the printer) relinquishes the printer (this corresponds to the substring `GrD`). At this point there is one pending request by the second user. Before this request is honoured, however, the first user requests, is granted and relinquishes access to the printer twice (this corresponds to the substring `RGDRGD`). After that, the pending request of the second user is granted and, finally, the printer is released (this corresponds to the final substring `gd`).

The second string, y, corresponds to an incorrect behaviour of the system: the second user relinquishes the printer without having previously gained access to it (there is a `d` symbol but no preceding `g` symbol). The third string, z, corresponds to an incorrect behaviour in which the mutual exclusion property is violated: after the prefix `rgRG` both users have access to the printer at the same time.

We can describe the correct behaviour of the system, as a language L_{sys} over the alphabet Σ consisting of those strings in which the sequence of events does not violate the desired behaviour. Specifically, a string x in Σ^* belongs to L_{sys} if and only if all of the following hold.

(a) if x contains a symbol in $\{\texttt{R}, \texttt{G}, \texttt{D}\}$ (respectively, $\{\texttt{r}, \texttt{g}, \texttt{d}\}$), then the first such symbol is `R` (respectively, `r`);

(b) for any $y \in \Sigma^*$, if $\texttt{R}y$ (respectively, $\texttt{r}y$) is a suffix of x, then y contains some symbol in $\{\texttt{R}, \texttt{G}, \texttt{D}\}$ (respectively, $\{\texttt{r}, \texttt{g}, \texttt{d}\}$) and the first such symbol is `G` (respectively, `g`);

(c) for any $y \in \Sigma^*$, if $\texttt{G}y$ (respectively, $\texttt{g}y$) is a suffix of x, then y contains some symbol in $\{\texttt{R}, \texttt{G}, \texttt{D}\}$ (respectively, $\{\texttt{r}, \texttt{g}, \texttt{d}\}$) and the first such symbol is `D` (respectively, `d`);

(d) for any $y \in \Sigma^*$, if $\texttt{D}y$ (respectively, $\texttt{d}y$) is a suffix of x, and y contains a symbol in $\{\texttt{R}, \texttt{G}, \texttt{D}\}$ (respectively, $\{\texttt{r}, \texttt{g}, \texttt{d}\}$), then the first such symbol is `R` (respectively, `r`);

(e) for any $y \in \Sigma^*$, if $\texttt{G}y\texttt{g}$ (respectively, $\texttt{g}y\texttt{G}$) is a substring of x, then y must contain `D` (respectively `d`).

The first four properties ensure that each user goes through (zero or more) cycles of requesting the printer, being granted access to it, and then relinquishing it. In particular, a user can't just get to use the printer and then relinquish it without being granted access first; and the printer can't grant access to a user who has not requested it. Furthermore, if a user does request access, then the system is obligated to eventually grant it; and if a user is granted access, he/she is obligated to relinquish it. The last property ensures mutual exclusive access to the printer: If at some point access to the printer is granted to one user, access to the printer cannot be subsequently granted to the other user until the first user is done.

Note that the above description of the system behaviour permits a situation where a user requests access to the printer, but the other user requests, is granted access and relinquishes the printer a thousand times before the first user's original request is honoured. If such "unfair" behaviour is undesirable, we may wish to rule out such strings from our language, by imposing additional restrictions on the acceptable strings. For example, we might require that, in addition to properties (a)–(e), a string x must satisfy the following property in order to be in the language of acceptable behaviours:

(f) For any $y_1, y_2 \in \Sigma^*$, if $\mathtt{R}y_1\mathtt{r}y_2$ (respectively, $\mathtt{r}y_1\mathtt{R}y_2$) is a substring of x, then the first occurrence of `G` (respectively, `g`) precedes the first occurrence of `g` (respectively, `G`) in y_1y_2.

This property states that access to the printer is on a first-come-first-served basis: A user who requests access to the printer before the other will be granted access first. Less stringent fairness requirements (for example, a user's request may be overtaken by the other at most twice) can be specified as well.

This example describes a somewhat simplistic system, but it illustrates important ideas. Formal language theory and its tools are used extensively in practice to describe the legal behaviour of systems, both for specification and for documentation purposes.

End of Example 7.4

7.1.4 Overview

The mathematical theory of formal languages is a mature field of computer science with important and diverse applications that include programming languages, artificial intelligence (especially natural language processing), specifications of hardware and software, and even bioinformatics (an emerging field at the intersection of computer science and biology). In this course we will look at some interesting and important aspects of this theory, but in truth we will just barely scratch the surface.

We will start our exploration with regular expressions — a simple but useful notation for describing a certain class of languages. Among other applications, this notation forms the basis of the search capabilities of powerful editors such as *emacs* and *vi*, of pattern-matching utilities such as *grep* and *awk*, and of scripting languages such as *perl*. We then investigate a mathematical model of an abstract machine, called finite state automaton. Such an automaton takes a string as input; after processing it symbol-by-symbol, the automaton either *accepts* or *rejects* the string. We can regard such a machine as a description of a language — namely, the language consisting of the set of the strings that it accepts. We will prove that finite state automata and regular expressions are, in a precise sense, equivalent formalisms: they both describe exactly the same set of languages. Each of these two equivalent methods of describing languages has its own advantages, and so both are important to know. We also discuss the limitations of these two methods of describing languages: Despite their wide applicability, it turns out that there are important classes of languages that cannot be expressed using these formalisms. There are other, more powerful, mathematical formalisms with which to express such languages; we will see one such formalism in Chapter 8.

7.2 Regular expressions

7.2.1 Definitions

Regular expressions are a precise and succinct notation for describing languages. An example of a regular expression is

$$(0+1)((01)^*0)$$

This expression describes the set of strings that:

- *start with* 0 or 1 (indicated by the subexpression $(0+1)$);
- *are then followed by zero or more repetitions of* 01 (indicated by the subexpression $(01)^*$); and
- *end with* 0 (indicated by the final subexpression 0).

For example, 001010 and 10 are strings in the language described by this regular expression, while 0110 is not.

Regular expressions contain three operators: alternation $+$, informally meaning "or"; concatenation, informally meaning "followed by"; and repetition $*$, informally meaning "zero or more repetitions of". As usual, parentheses are used to indicate the order in which the operators are applied. The precise definition of regular expressions is given below by induction.

Definition 7.6 *Let Σ be a finite alphabet. The set of* ***regular expressions*** *$\mathcal{RE}$ (over Σ) is the smallest set such that:*

regexp describes is a set of strings — aka language

BASIS: *$\emptyset$, ϵ, and a (for each $a \in \Sigma$) belong to $\mathcal{RE}$.*

INDUCTION STEP: *If R and S belong to $\mathcal{RE}$, then $(R+S)$, (RS) and R^* also belong to $\mathcal{RE}$.*

Following are some examples of regular expressions (over the alphabet $\{0,1\}$):

$$0 \qquad (01) \qquad (0+1) \qquad ((01)+(10)) \qquad (11)^* \qquad (((01)+(10))(11)^*)^*$$

Each regular expression denotes a language (over Σ). In the examples above, the first regular expression denotes the language $\{0\}$; the second example denotes $\{01\}$; the third example denotes $\{0,1\}$; the fourth example denotes $\{01,10\}$; the fifth example denotes the language consisting of strings that are zero or more repetitions of the string 11 — i.e. $\{\epsilon, 11, 1111, 111111, \ldots\}$; the last example denotes the language consisting of strings that are the concatenation of zero or more strings, each of which starts with either 01 or 10, and is followed by zero or more repetitions of the string 11.

In general, the language denoted by a regular expression is defined inductively, as follows:

Definition 7.7 *$\mathcal{L}(R)$, the* ***language denoted by a regular expression*** *R, is defined by structural induction on R:*

Basis: *If R is a regular expression by the basis of Definition 7.6, then either $R = \emptyset$, or $R = \epsilon$, or $R = a$, for some $a \in \Sigma$. For each of these cases we define $\mathcal{L}(R)$:*

- *$\mathcal{L}(\emptyset) = \emptyset$ (the empty language, consisting of no strings);*
- *$\mathcal{L}(\epsilon) = \{\epsilon\}$ (the language consisting of just the empty string); and*
- *for any $a \in \Sigma$, $\mathcal{L}(a) = \{a\}$ (the language consisting of just the one-symbol string a).*

Induction Step: *If R is a regular expression by the induction step of Definition 7.6, then either $R = (S + T)$, or $R = (ST)$ or $R = S^*$, for some regular expressions S and (for the first two cases) T, where we can assume that $\mathcal{L}(S)$ and $\mathcal{L}(T)$ have been defined, inductively. For each of the three cases we define $\mathcal{L}(R)$:*

- *$\mathcal{L}\big((S+T)\big) = \mathcal{L}(S) \cup \mathcal{L}(T)$;*
- *$\mathcal{L}\big((ST)\big) = \mathcal{L}(S) \circ \mathcal{L}(T)$; and*
- *$\mathcal{L}(S^*) = \big(\mathcal{L}(S)\big)^{\circledast}$.*

The table below gives some additional examples of regular expressions (over $\Sigma = \{0, 1\}$), along with the language that each denotes:

Expression	Language
$(0+1)^*$	all strings of 0s and 1s (i.e., Σ^*)
$(0(0+1)^*)$	all nonempty strings of 0s and 1s that begin with 0
$((0+1)(0+1)^*)$	all nonempty strings of 0s and 1s
$(0(0+1))^*$	all strings of 0s and 1s that can be broken into ≥ 0 two-symbol blocks where the first symbol of each block is 0

doesnt have to be only 1s or 0s in a roll

7.2.2 Precedence rules

As is usually the case with formal expressions, if we insist in writing out all the parentheses required by the formal definition, we get formidable-looking expressions even for simple things. We can simplify the notation considerably, without introducing ambiguities, by agreeing on certain conventions that reduce the number of parentheses used. We will use the following conventions:

(a) We leave out the outermost pair of parentheses. Thus, $(0+1)(11)^*$ is an abbreviation of $((0+1)(11)^*)$.

(b) The precedence order of the binary operators is: concatenation before union. For example, $RS^* + T$ is an abbreviation of $((RS^*) + T)$; in general, this does *not* denote the same language as $(R(S^* + T))$.

(c) When the same binary operator is applied several times in a row, we can leave out the parentheses and assume that grouping is to the right. Thus, 1011 is an abbreviation of $(1(0(11)))$, and $11 + 01 + 10 + 11$ is an abbreviation of $(11 + (01 + (10 + 11)))$.

7.2.3 Examples of regular expressions

We will now consider some examples where we are given a description of a language and we are asked to design a regular expression that denotes it. In all our examples we will assume that the alphabet is $\Sigma = \{0, 1\}$.

Example 7.5 Let L_1 be the set of strings in Σ^* that contain *at most two* 0s. A regular expression for L_1 is

$$R_1 = 1^* + 1^*01^* + 1^*01^*01^*$$

Informally, our regular expression is correct (that is, it denotes L_1), because the first term, 1^*, denotes the set of strings that contain no 0s at all; the second term, 1^*01^*, denotes the set of strings that contain exactly one 0; and the third term, $1^*01^*01^*$, denotes the set of strings that contain exactly two 0s. This example is so simple that a more detailed proof is not really needed; this informal explanation is convincing enough to most people. If challenged, however, we must be able to produce a rigorous proof — as we will illustrate in subsequent, less obvious, examples.

Another regular expression that denotes the same set is

$$R_1' = 1^*(0 + \epsilon)1^*(0 + \epsilon)1^*$$

Informally explain why. **End of Example 7.5**

Example 7.6 Let L_2 be the set of strings that contain an *even number* of 0s. A regular expression for L_2 is

$$R_2 = 1^*(01^*01^*)^*$$

Perhaps now it is not quite so obvious that this regular expression is, in fact, correct. How can we convince ourselves that it is? We need to prove that:

$$\text{for every string } x,\ x \in L_2 \text{ if and only if } x \in \mathcal{L}(R_2) \qquad (7.2)$$

(Note carefully that proving *just* the "if" direction or *just* the "only if" direction in (7.2), does not prove the correctness of R_2!)

[IF] Let x be an arbitrary string in $\mathcal{L}(1^*(01^*01^*)^*)$. Thus, there is some $k \in \mathbb{N}$, and $y_0, y_1, \ldots, y_k \in \Sigma^*$ such that

$$y_0 \in \mathcal{L}(1^*) \quad \text{and} \quad y_i \in \mathcal{L}(01^*01^*), \text{ for all } i \text{ such that } 1 \le i \le k$$

Therefore, there are $\ell_0, \ell_1, \ldots, \ell_k \in \mathbb{N}$ and $m_1, m_2, \ldots, m_k \in \mathbb{N}$ such that

$$y_0 = 1^{\ell_0} \quad \text{and} \quad y_i = 01^{\ell_i}01^{m_i}, \text{ for all } i \text{ such that } 1 \le i \le k$$

note how to express nested kleene star here with i the trick is to express kleene star in language as expotentiation of string

So, x has exactly $2k$ 0s (none in y_0 and two in each y_i, $1 \leq i \leq k$). Since $2k$ is even, $x \in L_2$, as wanted.

[ONLY IF] Let x be an arbitrary string in L_2. Thus, x contains an even number of 0s, say $2k$ for some $k \in \mathbb{N}$. Now we define $k+1$ consecutive substrings of x, $y_0, y_1, \ldots, y_k$, delimited by the 0s as suggested below (where each boxed substring consists entirely of 1s, but may be empty):

$$x = \underbrace{\boxed{1\cdots1}}_{y_0}\ \underbrace{0\boxed{1\cdots1}0\boxed{1\cdots1}}_{y_1}\ \underbrace{0\boxed{1\cdots1}0\boxed{1\cdots1}}_{y_2}\ \cdots\ \cdots\ \cdots\ \underbrace{0\boxed{1\cdots1}0\boxed{1\cdots1}}_{y_k}$$

More precisely, if $k = 0$ (i.e., if x contains no 0s) then let $y_0 = x$. Otherwise, let $y_0, y_1, \ldots, y_k$ be defined as follows:

- y_0 is the prefix of x up to (but not including) the first 0; thus $y_0 = 1^{\ell_0}$, for some $\ell_0 \in \mathbb{N}$, and so $y_0 \in \mathcal{L}(1^*)$.

- For each i such that $1 \leq i < k$, y_i is the substring of x from (and including) the $(2i-1)$-st 0 up to (but excluding) the $(2i+1)$-st 0; thus $y_i = 01^{\ell_i}01^{m_i}$, for some $\ell_i, m_i \in \mathbb{N}$, and so $y_i \in \mathcal{L}(01^*01^*)$.

- y_k is the suffix of x from (and including) the $(2k-1)$-st 0 to the end; thus $y_k = 01^{\ell_k}01^{m_k}$, for some $\ell_k, m_k \in \mathbb{N}$, and so $y_k \in \mathcal{L}(01^*01^*)$.

Since $x = y_0 y_1 \ldots y_k$, for some $k \in \mathbb{N}$, where $y_0 \in \mathcal{L}(1^*)$ and $y_i \in \mathcal{L}(01^*01^*)$ for all i, $1 \leq i \leq k$, it follows that $x \in \mathcal{L}(1^*(01^*01^*)^*)$, as wanted.

Since every string in $\mathcal{L}(R_2)$ is in L_2 ("if" part) and, conversely every string in L_2 is in $\mathcal{L}(R_2)$ ("only if" part), R_2 denotes L_2. (See Exercise 5 in connection with this example.)

End of Example 7.6

Example 7.7 Let L_3 be the set of strings that contain the substring 01 exactly once. A regular expression that denotes L_3 is $R_3 = 1^*0^*011^*0^*$.

The key observation needed to prove that R_3 is correct is the following claim:

$$\text{for any string } x\text{, 01 is not a substring of } x \text{ if and only if } x \in \mathcal{L}(1^*0^*) \qquad (7.3)$$

We now prove this claim.

[IF] If $x \in \mathcal{L}(1^*0^*)$ then $x = 1^\ell 0^m$, for some $\ell, m \in \mathbb{N}$. Since in this string there is no 1 following a 0 (regardless of what ℓ and m are) 01 is not a substring of x.

[ONLY IF] Suppose x does not have 01 as a substring. This means that every 0 in x (if one exists) must be preceded by every 1 in x (if one exists). In other words, x must be of the form $1^\ell 0^m$, for some $\ell, m \in \mathbb{N}$. Hence, $x \in \mathcal{L}(1^*0^*)$.

Using (7.3) it is now easy to show that R_3 denotes L_3. We leave the detailed argument as an exercise. **End of Example 7.7**

These examples illustrate several points. First, there is no "cookbook" approach to designing regular expressions that denotes a specified language.[5] In each case, we have to think (sometimes hard) for a way of expressing what the given language says using the limited "vocabulary" of regular expressions. Experience, acquired by working out many examples, helps one get better at this. In many ways, this is a task much like programming. In fact, some kinds of programming (e.g., with the popular scripting language *perl*, or with the pattern-matching language *awk*) require a great deal of just this activity: designing regular expressions that match various patterns.

When in doubt about the correctness of a regular expression (and sometimes even when not in doubt!) it is useful to *prove* that the regular expression "works". To do this we have to show that (a) every string in the language of interest is denoted by the regular expression, and (b) every string denoted by the regular expression is in the language of interest. In carrying out such proofs, we sometimes catch errors in the regular expression; indeed, sometimes a glitch in the proof suggests how to fix the problem. The amount of detail with which we may feel compelled to prove (a) and (b) may vary depending on the complexity of the regular expression (and the language) that we are dealing with.

7.2.4 Equivalence of regular expressions

Two regular expressions R and S are ***equivalent***, written $R \equiv S$, if they denote the same same language — i.e., if $\mathcal{L}(R) = \mathcal{L}(S)$. For example, $(0^*1^*)^* \equiv (0+1)^*$; both expressions denote the set of all strings (over $\{0,1\}$). For all regular expressions R, S and T, the following equivalences hold:

- **Commutativity of union:** $(R+S) \equiv (S+R)$.
- **Associativity of union:** $((R+S)+T) \equiv (R+(S+T))$.
- **Associativity of concatenation:** $((RS)T) \equiv (R(ST))$.
- **Left distributivity:** $(R(S+T)) \equiv ((RS)+(RT))$.
- **Right distributivity:** $((S+T)R) \equiv ((SR)+(TR))$.
- **Identity for union:** $(R+\emptyset) \equiv R$.
- **Identity for concatenation:** $(R\epsilon) \equiv R$ and $(\epsilon R) \equiv R$
- **Annihilator for concatenation:** $(\emptyset R) \equiv \emptyset$ and $(R\emptyset) \equiv \emptyset$.
- **Idempotence of Kleene star:** $R^{**} \equiv R^*$.

[5] Except, perhaps, if the language itself is described using some other formalism, such as finite state automata. See Section 7.6 for more on this.

Most of these rules follow directly from corresponding properties of set-theoretic operators (e.g., the commutativity and associativity of union) or of sequence-theoretic operators (e.g., the associativity of concatenation). In Section 7.2.5 we will present the proof of the (left) distributivity rule in some detail.

7.2.5 Proving the equivalence of regular expressions

We now present some examples of equivalence proofs for regular expressions. We begin with a proof of the (left) distributivity law.

Theorem 7.8 *For all regular expressions R, S and T, $R(S+T) \equiv RS+RT$.*

PROOF. We must prove that $\mathcal{L}(R(S+T)) = \mathcal{L}(RS+RT)$. To do so it is enough to prove that the left-hand-side is a subset of the right-hand-side, and vice-versa. Let Σ be the alphabet over which the regular expressions are defined.

(1) $\mathcal{L}(R(S+T)) \subseteq \mathcal{L}(RS+RT)$. Let x be an arbitrary string in $\mathcal{L}(R(S+T))$; we must prove that $x \in \mathcal{L}(RS+RT)$. Since $x \in \mathcal{L}(R(S+T))$, there are $y, z \in \Sigma^*$ such that $x = yz$, $y \in \mathcal{L}(R)$ and $z \in \mathcal{L}(S+T)$. Since $z \in \mathcal{L}(S+T)$, by definition of $\mathcal{L}$, either $z \in \mathcal{L}(S)$ or $z \in \mathcal{L}(T)$. We consider each case:

CASE 1. $z \in \mathcal{L}(S)$. Then $x = yz$, where $y \in \mathcal{L}(R)$ and $z \in \mathcal{L}(S)$. By definition of $\mathcal{L}$, $x \in \mathcal{L}(RS)$; therefore surely $x \in \mathcal{L}(RS) \cup \mathcal{L}(RT) = \mathcal{L}(RS+RT)$.

CASE 2. $z \in \mathcal{L}(T)$. Similarly, in this case we can prove that $x \in \mathcal{L}(RS+RT)$.

So, in either case $x \in \mathcal{L}(RS+RT)$, as wanted.

(2) $\mathcal{L}(RS+RT) \subseteq \mathcal{L}(R(S+T))$. Let x be an arbitrary string in $\mathcal{L}\big(RS+RT)$; we must prove that $x \in \mathcal{L}(R(S+T))$. Since $x \in \mathcal{L}(RS+RT)$, by definition of $\mathcal{L}$, either $x \in \mathcal{L}(RS)$ or $x \in \mathcal{L}(RT)$. We consider each case:

CASE 1. $x \in \mathcal{L}(RS)$. Thus, there are $y, z \in \Sigma^*$ so that $x = yz$, $y \in \mathcal{L}(R)$ and $z \in \mathcal{L}(S)$. Since $z \in \mathcal{L}(S)$, surely $z \in \mathcal{L}(S) \cup \mathcal{L}(T) = \mathcal{L}(S+T)$. Since $y \in \mathcal{L}(R)$ and $z \in \mathcal{L}(S+T)$, it follows that $yz \in \mathcal{L}(R(S+T))$, and since $x = yz$, we have that $x \in \mathcal{L}(R(S+T))$.

CASE 2. $x \in \mathcal{L}(RT)$. Similarly, in this case we can show that $x \in \mathcal{L}(R(S+T))$.

So, in either case $x \in \mathcal{L}(R(S+T))$, as wanted. □

Note that to prove the equivalence of two regular expressions we must prove that every string belonging to one also belongs to the other *and vice-versa*. Since regular expression equivalence amounts to equality between two sets (the languages denoted by the two expressions) we must effectively prove subset inclusion *in both directions.*

The following "substitution theorem" allows us to replace a subexpression of a regular expression R by an equivalent one without affecting the language denoted.

Theorem 7.9 *Let R be a regular expression, S be a regular expression that is a subexpression of R, S' be a regular expression that is equivalent to S, and R' be the regular expression that is obtained from R by substituting S' for S. Then R' is equivalent to R.*

You should compare this to Theorem 5.10 in Chapter 5 (cf. page 123), and to the subformula substitution logical equivalence rule in Chapter 6 (cf. page 160). The equivalence below illustrates its use.

$$\underbrace{\underbrace{(10+1001)}_{S}(01)^*}_{R} \equiv \underbrace{10\underbrace{(\epsilon+01)}_{S'}(01)^*}_{R'}$$

The proof of Theorem 7.9 is left as an exercise; it can be shown by using structural induction on R.

We can approach the task of proving that two regular expressions are equivalent from "first principles", as in the proof of Theorem 7.8, or by using previously established equivalences — or by some combination of these.

Example 7.8 It can be shown (from first principles) that, for any regular expression R,

$$((\epsilon+R)R^*) \equiv R^* \tag{7.4}$$

(Do it!) Now we can prove that $(0110+01)(10)^* \equiv 01(10)^*$, as follows:

$$\begin{aligned}
(0110+01)(10)^* &\equiv (01(10+\epsilon))(10)^* && \text{[by distributivity, and Theorem 7.9]}\\
&\equiv 01((10+\epsilon)(10)^*) && \text{[by associativity of concatenation]}\\
&\equiv 01((\epsilon+10)(10)^*) && \text{[by commutativity of union, and Theorem 7.9]}\\
&\equiv 01(10)^* && \text{[by equivalence (7.4), and Theorem 7.9]}
\end{aligned}$$

as wanted. Theorem 7.9 is so natural that we will typically omit reference to it when we use it. **End of Example 7.8**

To prove that two regular expressions are *not* equivalent, it is enough to exhibit a particular string which belongs to the language denoted by one expression but not to the language denoted by the other — and to prove that fact. For example, to prove that $(01)^*(11+0) \not\equiv 0(00)^*$ it is enough to point out that $11 \in \mathcal{L}\big((01)^*(11+0)\big)$ (because $\epsilon \in \mathcal{L}\big((01)^*\big)$, $11 \in \mathcal{L}(11+0)$, and $11 = \epsilon \circ 11$), but $11 \notin \mathcal{L}\big(0(00)^*\big)$ (because no string in $\mathcal{L}\big(0(00)^*\big)$ contains any 1s).

7.2.6 How expressive are regular expressions?

Regular expressions contain operators that correspond directly to some operations on languages (union, concatenation and Kleene star), but no operators that correspond to other operations of interest (for example, intersection or complementation). This means that some languages are fairly easy to describe using regular expressions, while for others it is not clear that we can describe them at all using this particular notation.

For example, as we have seen, the set of strings (over $\{0,1\}$) that contain an even number of 0s is denoted by $1^*(01^*01^*)^*$. The set of strings that contain at least two 1s is denoted by $0^*10^*1(0+1)^*$. Thus, the set of strings that contain *either* an even number of 0s *or* at least two 1s (i.e., the union of these two languages) is denoted $1^*(01^*01^*)^* + 0^*10^*1(0+1)^*$. But what about the set of strings that contain *both* an even number of 0s *and* at least two 1s (i.e., the intersection of these two languages)? There is no immediate way of combining the expressions denoting each of the two languages to obtain an expression that denotes their intersection. It is not even clear that there is a regular expression that denotes this set!

It turns out that there is. In fact, there is a general result stating that if each of two languages is denoted by a regular expression, then so is their intersection — no matter what these languages are. We do not yet have the tools with which to prove this, however. (See Exercise 7 for an interesting analogous result concerning the operation of ***reversal***, rather than intersection.)

As another example, consider the language of legal behaviours of the simple distributed system that we discussed in Example 7.4. Is there a regular expression that denotes this set? At this point, it is not at all clear what the answer is. It turns out that the answer is affirmative.

Finally, consider the set of strings over $\{0,1\}$ that have an equal number of 0s and 1s. This language certainly has a simple description in English. Is there a regular expression that denotes it? The answer turns out to be no.[6] Again, we don't yet have the tools with which to prove this fact.

In the remainder of this chapter we will study a quite different way of modeling languages. Instead of seeking a precise and convenient notation for a language, we will be looking at a mathematical "machine" or "automaton" whose job is to look at input strings and sort out which strings belong to the language and which ones do not. Along with other benefits, this alternative point of view on languages will provide us the necessary tools to answer the kinds of questions posed above.

7.3 Deterministic finite state automata

7.3.1 Definitions

A ***deterministic finite state automaton*** (DFSA) is a mathematical model of a machine which, given any input string x, ***accepts*** or ***rejects*** x. The automaton has a finite set of ***states***, including a designated ***initial state*** and a designated set of ***accepting*** states. The automaton is started in its initial state and reads the input string one symbol at a time. Upon reading a symbol, the automaton enters a new state (which depends on the present state of the automaton and the symbol it read) and proceeds to the next symbol of the input string. The automaton stops when it has processed the entire input string in this manner: if, when it stops, it is in an accepting state, it "accepts" the input; otherwise, it "rejects" the input.

It is customary to represent automata as directed graphs, with nodes (circles) corresponding

[6] It is important to understand precisely in what sense this is so. The claim is not merely that we've been insufficiently clever to think of a regular expression denoting this language. Rather, *we can prove* that nobody will ever be able to think of such an expression because none exists!

accepts odd number of
zeros and odd number of 1s

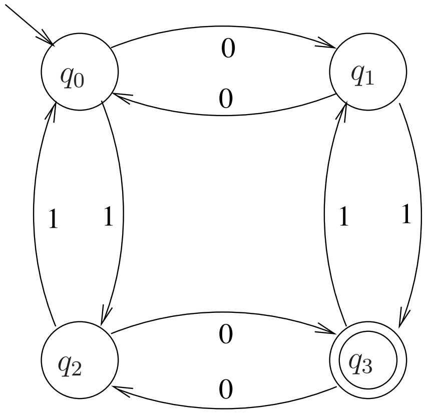
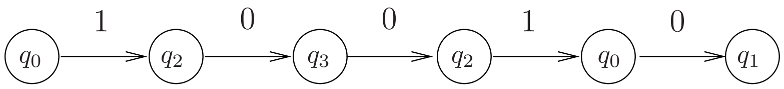


Figure 7.1: Diagrammatic representation of a DFSA

to states, and edges (arcs) labeled with the symbols of the alphabet, as shown in Figure 7.1. An edge from state q to state q', labeled with symbol a, indicates that if the current state is q and the current input symbol is a, the automaton will move to state q'. The initial state of the automaton is indicated by drawing an unlabeled edge into that state coming from no other state. The accepting states are indicated by double circles.

Example 7.9 Let us consider the operation of the DFSA depicted in Figure 7.1, on some sample input strings. Suppose the input string is $x = 10010$. The automaton is initially in state q_0. After reading the first symbol of x the automaton follows the transition indicated by the arc that emanates from state q_0 and is labeled with that symbol — in this case, 1. This transition leads to state q_2, so after reading the first symbol, the automaton is in state q_2. The automaton then reads the next symbol in the input string, namely 0. Since it is presently in state q_2, and the transition labeled with 0 leads to state q_3, the automaton moves to that state after reading the second symbol. The automaton then reads the next (third) symbol of the input string, which is 0. The transition labeled 0 emanating from the present state, q_3, leads back to state q_2, and that is the state to which the automaton moves after reading the third symbol of the input string. The next symbol is 1, so the automaton now moves to state q_0 (since the transition labeled 1 from the current state, q_2, leads to state q_0). Finally, after reading the last symbol, 0, the automaton moves to state q_1. Since this is not an accepting state, the automaton rejects the input string 10010. The sequence of states visited by the DFSA and the transitions followed in processing the input string 10010 are shown below.



Now, suppose the input string is $y = 100101 = x1$. After reading x, which comprises the first 5 symbols of y, the automaton is in state q_1 (as explained in detail in the previous paragraph). After reading the last symbol of y, namely 1, the automaton moves to state q_3, because the transition labeled 1 that emanates from q_1 leads to q_3. Since the automaton is in an accepting state after reading all the symbols of y, it accepts 100101. **End of Example 7.9**

As the preceding example illustrates, the processing of a string by a DFSA corresponds in a natural way to a path (starting in the initial state) in the graph that represents the DFSA. This natural correspondence between strings and paths is the source of some important intuitions for results we will encounter later (especially in Section 7.6.2), so it is important to keep it in mind. We now present the formal definition of a DFSA, and of what it means for it to accept or reject a string.

Definition 7.10 *A DFSA M is a quintuple $M = (Q, \Sigma, \delta, s, F)$, where*

- *Q is a* finite *set of* ***states****.*
- *Σ is a* finite *alphabet.*
- *$\delta : Q \times \Sigma \to Q$ is the* ***transition function****. In terms of the diagrammatic representation of the FSA, $\delta(q, a) = q'$ means that there is an edge labeled a from state q to state q'.*
- *$s \in Q$ is the* ***start*** *or* ***initial*** *state.*
- *$F \subseteq Q$ is the set of* ***accepting*** *states.*

Given a DFSA M as above, we can define the ***extended transition function*** $\delta^* : Q \times \Sigma^* \to Q$. Intuitively, if $q \in Q$ and $x \in \Sigma^*$, $\delta^*(q, x)$ denotes the state in which the automaton will move after it processes input x starting in state q. For instance, if δ is the transition function of the DFSA shown in Figure 7.1, then $\delta^*(q_0, 10010) = q_1$, and $\delta^*(q_0, 100101) = q_3$ (see Example 7.9). More precisely,

Definition 7.11 *Let $\delta : Q \times \Sigma \to Q$ be the transition function of a DFSA. The* ***extended transition function*** *of the DFSA is the function $\delta^* : Q \times \Sigma^* \to Q$ defined by structural induction on x (recall the recursive definition of strings, Definition 7.2):*

BASIS: *$x = \epsilon$. In this case, $\delta^*(q, x) = q$.*

INDUCTION STEP: *$x = ya$, for some $y \in \Sigma^*$ and $a \in \Sigma$, where we assume, by induction, that $\delta^*(q, y)$ has been defined. In this case, $\delta^*(q, x) = \delta(\delta^*(q, y), a)$.*

If $\delta^*(q, x) = q'$ we say that x ***takes the automaton*** M ***from*** q ***to*** q'. Using the definition of δ^* we can now formally define what it means for a DFSA to accept (or reject) a string, and a host of other important concepts.

Definition 7.12 *A string $x \in \Sigma^*$ is* ***accepted*** *(or* ***recognised****) by M, if and only if $\delta^*(s, x) \in F$ — i.e., if and only if x takes the automaton from the initial state to an accepting state. The language* ***accepted*** *(or* ***recognised****) by a DFSA M, denoted $\mathcal{L}(M)$, is the set of all strings accepted by M.*

7.3.2 Conventions for DFSA diagrams

The diagrammatic description of DFSA is extremely convenient and widely used. To make this description even more helpful, we adopt some conventions that make the diagrams clearer and easier to comprehend. We now describe and illustrate the main two conventions that we will be using.

Combining transitions: Suppose that there are several symbols in the alphabet, say $a_1, a_2, \ldots, a_k$ (for some $k \geq 2$), all of which cause a transition from a given state q to the *same* state q'. In other words, we have that $\delta(q, a_i) = q'$ for all i such that $1 \leq i \leq k$. Instead of drawing k individual transitions from q to q', each labeled with a_i, for some i, we draw a *single* transition from q to q' and label it with "$a_1, a_2, \ldots, a_k$".

Example 7.10 Suppose that we want to draw a diagram for a DFSA that accepts the set of strings in $\{0,1\}^*$ containing at least one symbol. The diagram shown in Figure 7.2(a) accepts this language. In Figure 7.2(b) we show the same diagram "abbreviated" by using the convention of combining common transitions. **End of Example 7.10**

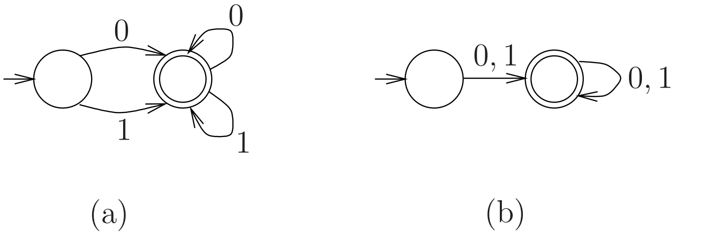


Figure 7.2: Diagram of a DFSA accepting $(0+1)(0+1)^*$.

Eliminating dead states: Given a DFSA, we say that a state q' ***is reachable from*** state q, if there is a path in the graph representation of the DFSA that starts at q and ends at q'. A state q is ***dead*** if no accepting state is reachable from q. Suppose that, at some point during the processing of a string x by the DFSA, we enter a dead state. This means that the path corresponding to x (in the graph that represents the DFSA) goes through a dead state. By definition, once we are in a dead state there is no hope of ever reaching an accepting state. This means that x is already doomed; although we have not yet seen all its symbols, we have seen enough of it to know that it will not be accepted by the DFSA. To make the graph easier to read we usually leave out of the graph the nodes corresponding to the dead states of a DFSA, as well as the edges corresponding to transitions into (or out of) such states.

Example 7.11 Suppose that we want to draw the diagram of a DFSA that accepts the set of strings denoted by the regular expression 101^*0 — i.e., the strings that start with 10, followed by zero or more 1s, and end with a 0. This language is accepted by the DFSA shown in Figure 7.3(a).

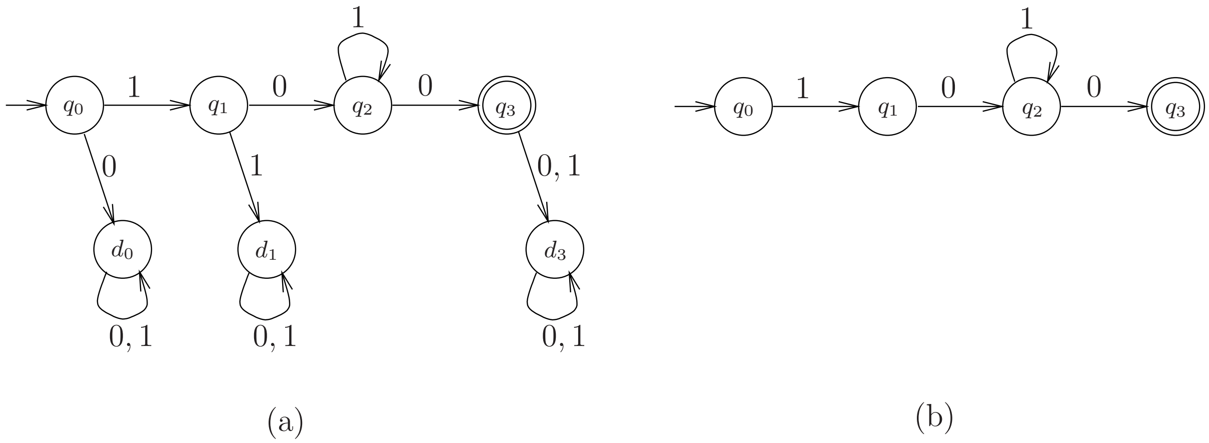


Figure 7.3: DFSA accepting 101*0.

States d_0, d_1 and d_3 are dead states: the only accepting state, q_3, is not reachable from any of them. Intuitively, d_0 corresponds to the "bad" strings (i.e., strings not in the language) whose first symbol is 0 instead of the required 1; d_1 corresponds to the bad strings whose first symbol is the required 1, but whose second symbol is 1 instead of the required 0, and d_3 corresponds to the bad strings that start with 10, continue on with zero or more 1s, followed by a 0, but then have other symbols as well. Note that there are also bad strings which correspond to paths in the graph that start at q_0 and end in q_2, which is a nonaccepting state, but it is not a dead one. These are strings that are not in the language, but are prefixes of strings that are in the language. In contrast, a string x that corresponds to a path ending in a dead state is not only itself bad, but so are all strings of which it is a prefix. Put in a different way, x cannot be completed to a string that is in the language.

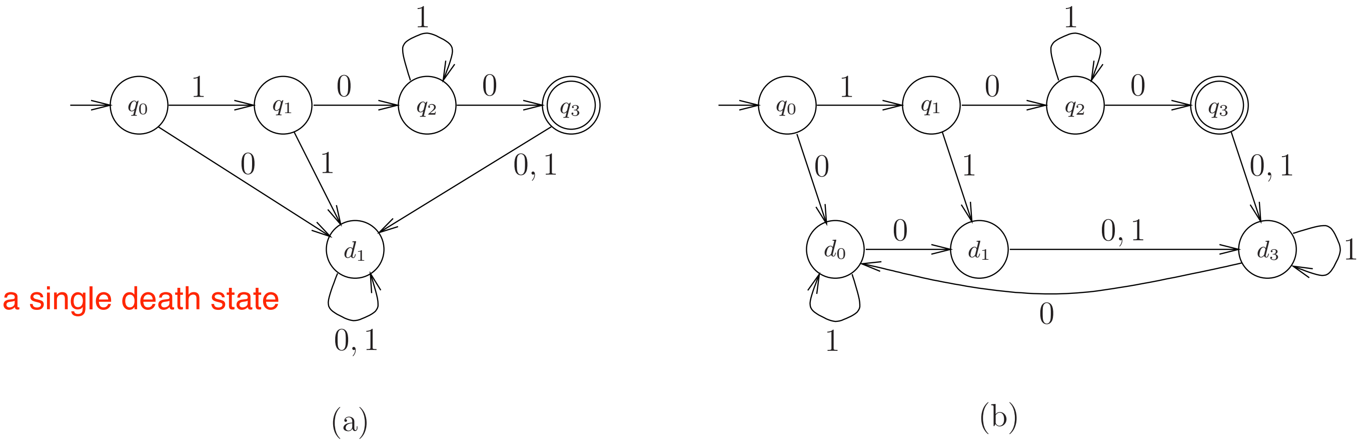
In Figure 7.3(b) we show the same DFSA with all dead states (and associated transitions) eliminated. This diagram is a lot clearer than the one in Figure 7.3(a), but it raises some questions. Specifically, suppose we want to determine what this automaton does when the input string is 001. We start at state q_0. To determine the next state we must follow the transition from q_0 that is labeled 0; but there is no such transition in the diagram! The reason is that this transition was removed from the diagram because it was leading to a dead state. Since it was leading to a dead state, the prefix of the input seen so far is such that it can never be completed to a string that belongs in the language accepted by the DFSA, so we may as well reject the input right away!

This justifies the following rule regarding DFSA diagrams with missing transitions: *Any input string requiring a transition that is missing from the diagram is rejected.*

so if there is no path leading to next state, assume to be in dead state

End of Example 7.11

We say that a DFSA diagram is ***complete***, if there is a transition on every symbol from each state. For example, the diagram in Figure 7.3(a) is complete, while the diagram in Figure 7.3(b) is not. Given the DFSA diagram that has resulted from a complete DFSA



a single death state

Figure 7.4: Two other DFSA accepting 101*0.

diagram by eliminating dead states, it may be impossible to recover the original diagram. To see why, notice that if we eliminate the dead states from any one of the DFSA diagrams shown in Figures 7.3(a), 7.4(a) or 7.4(b), we get the DFSA diagram shown in Figure 7.3(b). This "irreversibility" of the elimination of dead states should cause no alarm: all of these automata accept the same language, namely 101*0. It is, of course, possible to reconstruct *some* complete DFSA diagram that accepts the same language, by adding a single dead state to which all the missing transitions go — as in Figure 7.4(a).

7.3.3 Designing and proving the correctness of DFSA

There are two tasks commonly associated with DFSA: One is to *design* a DFSA that accepts a particular language. The second task is to *prove* that a DFSA accepts a language. The two tasks are obviously related: If we have designed a DFSA that we *believe* accepts a language, we want to be able to *show* that it does. If our design is incorrect, we will likely catch the mistake in trying to carry out the proof — because we will be unable to do so! A glitch in the proof may also suggest ways of fixing the mistake in the design. In this subsection we illustrate these two tasks by means of some examples.

Example 7.12 Consider the four-state DFSA shown in Figure 7.1. It turns out that this DFSA accepts the language

$$L_1 = \{x \in \{0,1\}^* : x \text{ has an odd number of 0s and an odd number of 1s}\}$$

To prove this, it suffices to show that a string x takes the DFSA from q_0 (the initial state) to q_3 (the unique accepting state) if and only if x has an odd number of 0s and an odd number of 1s. This statement exactly characterises the set of strings that take the DFSA from the initial state to q_3; we refer to such a statement as an ***invariant*** of the state q_3: it describes what is true about that state, just like a loop invariant describes what is true about that loop.

something that stays true before and after state transition

It is natural to try proving this statement using induction (either structural induction on x or simple induction on the length of x). If you try to do so, however, you will run into the following difficulty: In the induction step of the proof, in order to infer the desired property of strings that take the DFSA from the initial state to q_3, it will be necessary to use state invariants for q_1 and q_2 (the states from which q_3 is reachable in one step). In turn, for this, we will need invariants for the states from which q_1 and q_2 are reachable in one step — namely, q_0 and q_3. This suggests that we must strengthen our goal: instead of proving the invariant for just the accepting state, we need to prove invariants for all states.[7] Some further thought leads us to the following predicate $P(x)$ that states invariants for the four states of the DFSA:

predicate includes every state of DFSA!

$$P(x): \quad \delta^*(q_0, x) = \begin{cases} q_0, & \text{if } x \text{ has an even \# of 0s and an even \# of 1s} \\ q_1, & \text{if } x \text{ has an odd \# of 0s and an even \# of 1s} \\ q_2, & \text{if } x \text{ has an even \# of 0s and an odd \# of 1s} \\ q_3, & \text{if } x \text{ has an odd \# of 0s and an odd \# of 1s} \end{cases} \tag{7.5}$$

A somewhat delicate point requires elaboration. A state invariant is an "if and only if" statement: it provides both *necessary* and *sufficient* conditions for a string x to take the automaton from the initial state to some particular state. The above predicate $P(x)$, however, appears to provide only sufficient conditions (it contains "if" statements, not "if and only if" ones). The reason that this is not a problem is that if $P(x)$ holds, then the converses of the four conditional statements in (7.5) will also hold. Intuitively, the reason is that these four cases actually cover all possibilities. Let's consider one case in particular. Suppose that $P(x)$ holds; we want to show that $\delta^*(q_0, x) = q_0$ *only if* x has an even number of 0s and an even number of 1s. Suppose, for contradiction, that this is not the case. That is, $\delta^*(q_0, x) = q_0$, yet x does not have an even number of 0s or an even number of 1s. But then, one of the other three cases would apply and, depending on which one of them does, $P(x)$ would imply that $\delta^*(q_0, x)$ is equal to q_1, q_2 or q_3 — in any case, *not* equal to q_0 which contradicts our assumption. (You will see this argument reflected in the proof of Theorem 7.14 below.) Therefore, the "only-if" directions hold as well. We formulate the invariant predicate $P(x)$ as "if" statements (as opposed to "if and only if" statements) because this facilitates the proof of the following key Lemma.

Lemma 7.13 *The predicate $P(x)$ defined in (7.5) is true for all strings x.*

PROOF. We use structural induction on x.

BASIS: $x = \epsilon$. In this case x has zero 0s and zero 1s. Since zero is an even number, and $\delta^*(q_0, \epsilon) = q_0$ (by definition of δ^* — see Definition 7.11), $P(x)$ holds, in this case.

INDUCTION STEP: $x = ya$ for some $y \in \Sigma^*$ and $a \in \Sigma$; we assume, by induction, that $P(y)$ holds. There are two cases.

[7] This is yet another example of a phenomenon we have encountered repeatedly: the need to strengthen the statement we wish to prove by induction, in order to make the induction step go through.

use induction on length of string x to prove state invariant

Case 1. $a = 0$. We have:

$$\begin{aligned} \delta^*(q_0, x) &= \delta^*(q_0, y0) && \text{[since } x = ya\text{]} \\ &= \delta(\delta^*(q_0, y), 0) && \text{[by definition of } \delta^*\text{]} \end{aligned}$$

By the induction hypothesis:

$$= \begin{cases} \delta(q_0, 0), & \text{if } y \text{ has an even number of 0s and an even number of 1s} \\ \delta(q_1, 0), & \text{if } y \text{ has an odd number of 0s and an even number of 1s} \\ \delta(q_2, 0), & \text{if } y \text{ has an even number of 0s and an odd number of 1s} \\ \delta(q_3, 0), & \text{if } y \text{ has an odd number of 0s and an odd number of 1s} \end{cases}$$

Because x has one more 0 and the same number of 1s as y, we have:

$$= \begin{cases} \delta(q_0, 0), & \text{if } x \text{ has an odd number of 0s and an even number of 1s} \\ \delta(q_1, 0), & \text{if } x \text{ has an even number of 0s and an even number of 1s} \\ \delta(q_2, 0), & \text{if } x \text{ has an odd number of 0s and an odd number of 1s} \\ \delta(q_3, 0), & \text{if } x \text{ has an even number of 0s and an odd number of 1s} \end{cases}$$

By inspecting the transition function δ of M_1,

$$= \begin{cases} q_1, & \text{if } x \text{ has an odd number of 0s and an even number of 1s} \\ q_0, & \text{if } x \text{ has an even number of 0s and an even number of 1s} \\ q_3, & \text{if } x \text{ has an odd number of 0s and an odd number of 1s} \\ q_2, & \text{if } x \text{ has an even number of 0s and an odd number of 1s} \end{cases}$$

as wanted.

Case 2. $a = 1$. Similar to Case 1. □

To prove that a DFSA accepts a language
1. find state invariant over all states
2. use state invariant to prove that L(M) = L. Or that Language described by DFSA is equivalent to language given. We prove subset of in both directions to show this

Theorem 7.14 *The DFSA M_1 in Figure 7.1 accepts L_1.*

Proof. Let x be an arbitrary string in L_1; i.e., a string with an odd number of 0s and an odd number of 1s. By Lemma 7.13, $\delta^*(q_0, x) = q_3$, and since q_3 is an accepting state, $x \in \mathcal{L}(M_1)$. This proves that

$$L_1 \subseteq \mathcal{L}(M_1) \tag{7.6}$$

Conversely, let x be an arbitrary string in $\mathcal{L}(M_1)$. Since q_3 is the only accepting state of M_1, $\delta^*(q_0, x) = q_3$. If x did not have an odd number of 0s and an odd number of 1s, by Lemma 7.13, $\delta^*(q_0, x)$ would be equal to q_0 or q_1 or q_2 — in any event, not equal to q_3 as we know is the case. Thus, x must have an odd number of 0s and an odd number of 1s, i.e., $x \in L_1$. This proves that

$$\mathcal{L}(M_1) \subseteq L_1 \tag{7.7}$$

(7.6) and (7.7) imply that $L_1 = \mathcal{L}(M_1)$, as wanted. □

This proof is typical of rigorous arguments that show that a DFSA accepts a certain language. The key idea is to find, for each state of the DFSA, an invariant that characterises the strings that take the DFSA from the initial state to that state. We prove that each invariant characterises the corresponding state by *simultaneous* induction (all invariants are proved at once). Proving all invariants simultaneously is important, because the induction step in the proof of one state's invariant may well make use of the induction hypothesis concerning a *different* state's invariant. (For instance, in our example, the proof of the invariant of each state makes use of the induction hypothesis for the invariants of the states from which it is reachable in one step.) The induction itself can be formulated as a structural induction on the definition of strings, or as a simple induction on the length of strings.

The practice of writing down (and carefully inspecting) state invariants for a DFSA is a useful one, even if we have no intention of carrying out a thorough proof of its correctness. It sometimes helps catch mistakes; it can also help simplify the designed DFSA, by combining states that have similar state invariants.

You may be wondering how one comes up with state invariants. Being able to determine the state invariants of a DFSA is tantamount to understanding what the automaton does. In the process of trying to comprehend how the automaton works, we are able to formulate such invariants. In practice, this process is often one of trial and error: as our understanding of the automaton becomes more precise, so does our ability to formulate the state invariants. It is actually possible to determine state invariants of a DFSA algorithmically. As we will see in Section 7.6.2, there is a particular computation which permits us to determine, for any state q, a regular expression R_q denoting the set of strings that take the DFSA from the initial state to q. This computation is rather laborious to carry out by hand, although quite easy to do by computer. Also, the regular expressions it produces tend to be rather complicated. In principle, these can be simplified — e.g., by using algebraic rules such as those we saw in Section 7.2.4. Unfortunately, however, the task of simplifying regular expressions is, in a precise sense, computationally infeasible. **End of Example 7.12**

Example 7.13 Let L_2 be the language denoted by $(0+1)^*011(0+1)^*$. In other words,

$$L_2 = \{x \in \{0,1\}^* : x \text{ contains 011 as a substring}\}$$

Our task is to design a DFSA that accepts this language. The diagram of a DFSA M_2 that does so is shown in Figure 7.5. Note that if we ever enter state q_3, we will never "escape" from it since, no matter what symbol we read next, we will remain in that state. A state with this property is called a ***trap*** state. In this case, the trap state q_3 is an accepting state.

It corresponds to the detection of substring 011. Once this substring is detected, the entire input string must be accepted, no matter what follows that substring. The accepting trap state captures precisely this fact.

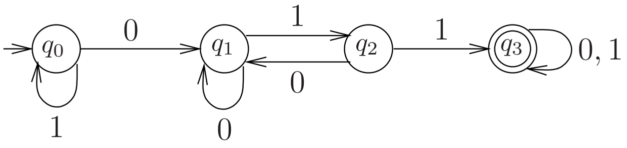


Figure 7.5: A DFSA M_2 that accepts $(0+1)^*011(0+1)^*$.

We can prove rigorously that this DFSA accepts the specified language, by using a similar approach as in the previous example. With each state of the DFSA we associate an invariant which characterises the strings that take the automaton from the initial state to that state. Specifically, we can prove that x takes the automaton from the initial state to

- q_0 if and only if x is of the form 1^*;
- q_1 if and only if x does not contain 011 as a substring and ends in 0;
- q_2 if and only if x does not contain 011 as a substring and ends in 01; and
- q_3 if and only if x contains 011 as a substring.

These four facts can be proved simultaneously by induction on the length of x. (We leave this as an exercise.) The correctness of our automaton follows immediately from the last fact, since q_3 is its only accepting state.

We also present an alternative, and perhaps simpler, proof that this DFSA is correct. This alternative proof does not require explicit state invariants. By inspection of the diagram of M_2 it is easy to show the following three facts:

(a) For *every* state q of the DFSA, 011 takes M_2 from q to q_3. (More formally, $\delta^*(q, 011) = q_3$.)

(b) For any string x, the shortest prefix x' of x that takes M_2 from the initial state to q_3 (if such a prefix exists), ends in 011. (More formally, if x' is a prefix of x such that $\delta^*(q_0, x') = q_3$ and for every proper prefix x'' of x', $\delta^*(q_0, x'') \neq q_3$, then there is some string y such that $x' = y011$.)

(c) Every string z takes M_2 from q_3 to q_3. (More formally, $\delta^*(q_3, z) = q_3$, for any z.)

Using these easy-to-verify observations we can now prove that $L_2 = \mathcal{L}(M_2)$.

First, we prove that $L_2 \subseteq \mathcal{L}(M_2)$. Let x be an arbitrary string in L_2. Thus, x contains 011 as a substring; i.e., there are strings y, z such that $x = y011z$. By (a), $\delta^*(\delta^*(q_0, y), 011) = q_3$. Thus, $\delta^*(q_0, y011) = q_3$. (Here we have used the following fact: for any state q and any strings

u, v, $\delta^*(q, uv) = \delta^*(\delta^*(q, u), v)$; this follows immediately from the definition of δ^*.) By (c), $\delta^*(q_0, y011z) = q_3$, i.e., $\delta^*(q_0, x) = q_3$. Since q_3 is an accepting state, $x \in \mathcal{L}(M_2)$, as wanted.

Next, we prove that $\mathcal{L}(M_2) \subseteq L_2$. Let x be an arbitrary string in $\mathcal{L}(M_2)$. Since q_3 is the only accepting state of M_2, $\delta^*(q_0, x) = q_3$. Let x' be the shortest prefix of x such that $\delta^*(q_0, x') = q_3$. By (b), there is some string y such that $x' = y011$. Since x' is a prefix of x, there is some string z such that $x = x'z$. Therefore, $x = y011z$; this implies that 011 is a substring of x, i.e., $x \in L_2$, as wanted. **End of Example 7.13**

Example 7.14 In this example, we want to design a DFSA that recognises the following language:

$$L_3 = \{x \in \{0,1\}^* : x \text{ contains neither 00 nor 11 as a substring}\}$$

The diagram of such a DFSA M_3 is shown in Figure 7.6. Note that this is not a complete DFSA; we have eliminated dead states according to the convention discussed earlier. Thus, although all the states shown in the diagram are accepting states, this DFSA does not accept all strings; there is an implicit dead state (a nonaccepting trap state) to which all missing transitions are supposed to lead.

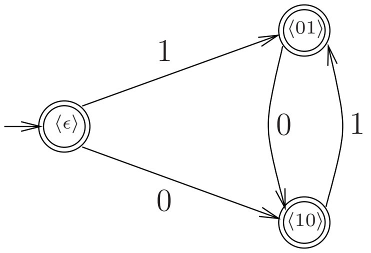


Figure 7.6: A DFSA M_3 for the set of strings that do not contain 00 or 11 as a substring

The states of this DFSA are named with strings (which we have surrounded with angled brackets to help remind us that they refer to states): $\langle\epsilon\rangle$, $\langle 01\rangle$ and $\langle 10\rangle$. These names reflect the intuition behind these states: the initial state, $\langle\epsilon\rangle$ is reachable only by the empty string; $\langle 01\rangle$ is reachable only by strings that do not contain the "forbidden patterns" (00 and 11) and in which the last two symbols are 01 (or the string has only one symbol, 1); the state $\langle 10\rangle$ has a similar interpretation. The idea is that the automaton "remembers" the last two symbols it has seen, so that its state depends on what those two symbols were. If it should ever come to find that the last two symbols seen are the same (i.e., 00 or 11), then the automaton enters a dead state and rejects the input. With these remarks in mind, you should be able to figure out the state invariants for the DFSA M_3, allowing you to prove its correctness. **End of Example 7.14**

Example 7.15 Our last example is a DFSA that accepts the language that describes the correct behaviour of a simple distributed system, discussed in Example 7.4. Such an automaton is shown in Figure 7.7.

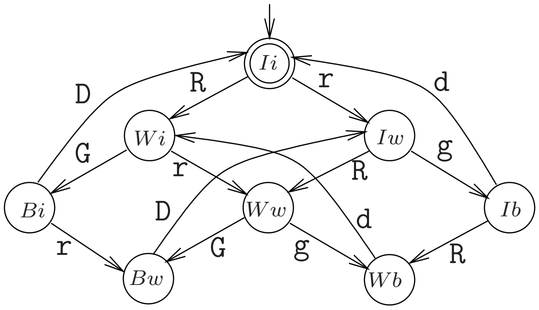


Figure 7.7: A DFSA for the simple distributed system of Example 7.4.

To understand how this works, imagine the two users of the distributed system. Each of them is in one of three states, as regards the use of the printer:

- idle: not interested in using the printer;
- waiting: having requested, but not yet been granted, access to the printer; and
- busy: having been granted access to the printer.

We use I, W and B to denote the idle, waiting and busy states of the first user; and similarly, i, w and b to denote these three states of the second user. Each state of the DFSA corresponds to some combination of states for the two users. For example, Ii indicates that both users are idle; Bw indicates that the first user is busy, while the second user is waiting. The mutual exclusion requirement forbids the state where both users are busy, so the state Bb does not appear in the diagram. Note that we have been using the term "state" to refer to two related, but distinct, things. There is the state of *a user* (which may be I, W or B, for the first user, and i, w and b, for the second user), and there is the state of *the entire system* (which may be one of the legal combinations of states of the two users). We use the terms "user state" and "system state" to distinguish between these. The states of the DFSA shown above are system states.

Each transition corresponds to the occurrence of one event, i.e., one of `R`, `G`, `D` (the events of the first user) or `r`, `g`, `d` (the events of the second user). An event that occurs at a user causes that user's state to change. For example, the occurrence of event `R` (a request by the first user) causes a change from user state I to user state W. The effect of the transition that corresponds to the occurrence of an event by one of the two users is to transform the system state so as to reflect the change in user state. For example, since an `R` event transforms the state of the first user from I to W, we have the following transitions labeled `R` between system states: from Ii to Wi, from Iw to Ww, and from Ib to Wb. Similar remarks apply to the other events.

With this explanation in mind, you should be able to see (and prove!) that the DFSA shown in Figure 7.7 accepts the language described in Example 7.4. **End of Example 7.15**

7.4 Nondeterministic finite state automata

7.4.1 Motivation and definitions

In a DFSA, a given state and current input symbol uniquely determine the next state of the automaton. It is for this reason that such automata are called deterministic. There is a variant of finite state automata, called ***nondeterministic*** finite state automata, abbreviated NFSA, where this is not the case: From a given state, when the automaton reads an input symbol a, there may be several states to which it may go next. Furthermore, the automaton may "spontaneously" move from one state to another without reading any input symbol; such state transitions are called ϵ-transitions. The choice of next state, and the possibility of a "spontaneous" state transition is not unrestricted, however: only *some* choices and *some* ϵ-transitions are possible.

Whenever there are multiple transitions that a NFSA can follow from its present state upon reading an input symbol (including, possibly, ϵ-transitions), the next state that the automaton occupies is not uniquely determined: there may be several states to which the automaton *could* move. From each of these states, the NFSA may have several choices for the next transition — and so on, at each step along the way. Thus, the computation of a NFSA on input x corresponds not to a single path in the state diagram, as in the case of DFSA, but to a *set* of paths. This means that after processing input x, the NFSA could be in any one of several states. The question now is: when should we say that the automaton accepts x?

If all the states in which the NFSA could be after processing x are accepting, it is certainly natural to say that the NFSA accepts x; and if none of them is accepting, it is natural to say that the NFSA rejects x. But what if some of these states are accepting and some are not? It turns out that a natural and useful convention is to consider the NFSA as accepting the string in this case. In other words, the NFSA accepts x if *there is (at least) one* computation path that it could follow (starting from the initial state) on input x that ends in an accepting state. A string is rejected only if *every* computation path the NFSA could have followed (starting from the initial state) ends in a nonaccepting state.

We will illustrate these somewhat informal ideas with two examples of NFSA. We will then give the formal definition of NFSA and of what it means for it to accept a string. We represent NFSA as graphs, similar to those of DFSA. In NFSA diagrams we may have multiple edges from a state q to states $q_1, q_2, \ldots, q_k$ with the same label a. This indicates that if the automaton is in state q and the next symbol it reads from the input is a, it may move to any one of the states $q_1, q_2, \ldots, q_k$, but to no other state. We may also have one or more transitions from state q to states $q'_1, q'_2, \ldots, q'_\ell$ labeled with the empty string, ϵ. This indicates that if the automaton is in state q it may spontaneously, without reading any input symbol, move to any one of the states $q'_1, q'_2, \ldots, q'_\ell$, but to no other state (other than itself, of course). We do not explicitly draw ϵ-transitions from a state to itself. We use the same conventions as in DFSA diagrams regarding missing transitions: these are assumed to lead to an implicit (not drawn) dead state.

Example 7.16 Consider the NFSA shown in Figure 7.8.

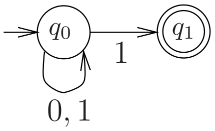


Figure 7.8: A NFSA that accepts $(0+1)^*1$.

This automaton is nondeterministic because there are two transitions labeled 1 from state q_0: one leading back to q_0 and the other leading to the accepting state q_1.

Let us consider the operation of this automaton on input string $x = 011$. The automaton starts in state q_0 and reads the first input symbol, 0. There is only one transition labeled 0 out of q_0, leading back to q_0; thus, the automaton enters (actually, remains in) state q_0. The next symbol of the input string is 1. There are two transitions labeled 1 out of state q_0, so now we have a choice. The automaton might remain in state q_0 or it might enter state q_1. Suppose that the automaton chooses to remain in state q_0. The next symbol of the input is 1, so again the automaton has a choice: it can remain in q_0 or go to state q_1. In this case, suppose that the automaton makes the latter choice. Since this is the last symbol of the input string, and the automaton is in an accepting state, the string 011 is accepted by the automaton.

Now let's back up one step and suppose that, when reading the third (and last) symbol the automaton had chosen to remain in state q_0 instead of choosing to move to state q_1. In that case, the automaton would be in a nonaccepting state at the end of processing 011. This does not mean that 011 is rejected, because (as shown in the previous paragraph) there is another choice that the automaton could have made, that would have caused it to be in an accepting state at the end of processing 011.

Let us also back up two steps and consider what would have happened if the automaton had chosen to move to state q_1 after reading the second symbol in 011. (Recall that, before reading that symbol, the automaton was in state q_0.) Then the automaton would read the third (and last) symbol of the string, 1, while in state q_1. There is no transition labeled 1 from that state, so according to our convention this particular sequence of choices leads to a dead (and therefore nonaccepting) state. Again, however, since another sequence of choices leads to an accepting state, the string 011 *is* accepted by this NFSA.

This discussion illustrates concretely an important point we made earlier: the processing of an input string by a NFSA does not necessarily result in a unique computation. It results in a *set* of possible computations, each corresponding to a particular sequence of choices made by the automaton in processing the input. In our example, there are three such computations, shown in Figure 7.9 in terms of the corresponding paths through the diagram. Note that the third path (i.e., computation) ends in a state labeled "dead" which is not explicitly drawn in the diagram, in accordance with our convention of eliminating states from which no accepting state is reachable. Since there is one path (computation) of 011 that ends in an accepting state, the NFSA accepts this string.

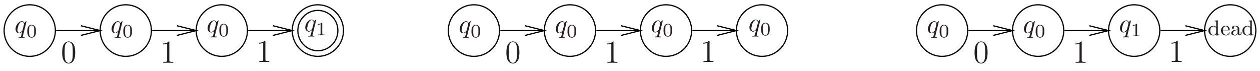


Figure 7.9: The three computations of the NFSA of Figure 7.8 on input 011.

Now consider what the automaton does on input string 010. It starts in state q_0 and remains in it after reading the first symbol, 0. After reading the second symbol 1, the automaton may remain in state q_0 or move to state q_1. After reading the third symbol, q_0 the automaton remains in state q_0 (if it had chosen to remain in state q_0 after reading the second symbol), or it enters a dead state (if it had chosen to move to state q_1 after reading the second symbol — note that there is no transition labeled 0 from state q_1). The two computations of the NFSA on input 010 are shown in Figure 7.10. Since *all* possible paths (computations) that the automaton could follow on input 010 end in nonaccepting states, the NFSA rejects this input.

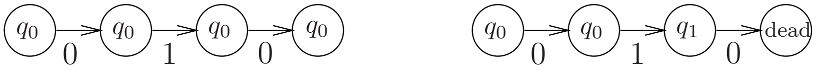


Figure 7.10: The two computations of the NFSA of Figure 7.8 on input 010.

We claim that the NFSA shown in Figure 7.8 accepts the set of strings in $\{0, 1\}^*$ that end with 1 — i.e., the strings denoted by the regular expression $(0 + 1)^*1$. It is obvious that any string that is accepted by the DFSA ends in 1. This is because there is only one accepting state, q_1, and any path from the initial state to q_1 must end by traversing the edge from q_0 to q_1 that is labeled 1. This means that every string accepted by this NFSA ends in 1.

Conversely, if a string ends in 1, we claim that it is accepted by this NFSA. To establish this claim we must show that after processing a string that ends in 1, the NFSA *could* be in the accepting state q_1. We use the following rule for deciding which of the two transitions labeled 1 to take, when we are in state q_0 and we read symbol 1 in the input: If the symbol 1 being read is the last one in the string, then take the transition to q_1; otherwise, take the transition leading back to q_0. It is obvious that if we process any string that ends in 1 by following this rule, the NFSA will be in state q_1 after it has processed the entire input string. This means that any string that ends in 1 is indeed accepted by the NFSA.

It is possible to construct a DFSA that accepts the same language. As we will see later (see Section 7.4.2), this is no accident: any language that is accepted by a NFSA, can also be accepted by a DFSA. **End of Example 7.16**

Example 7.17 Consider the NFSA shown in Figure 7.11(a). This automaton is nondeterministic because it contains an ϵ-transition.

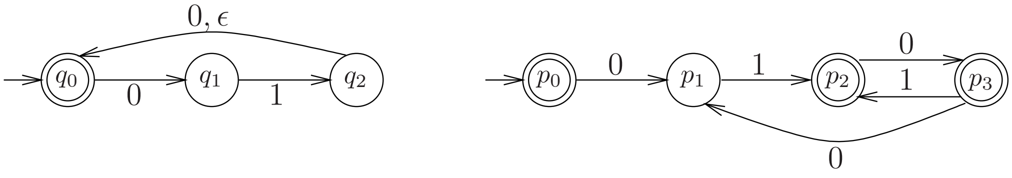


Figure 7.11: A NFSA and a DFSA that accept $(010 + 01)^*$.

Consider now what this automaton does on input 01. It starts in state q_0; upon reading the first symbol, 0, it must move to state q_1 (since there is only one transition labeled 0 from q_0, and it leads to state q_1). After reading the second (and last) symbol, 1, the automaton must move to state q_2. This is not an accepting state. Does the automaton then reject 01? No! There is actually another computation, one in which after reading the input 01 and reaching state q_2, as explained above, the automaton moves spontaneously, without reading any input symbol, to state q_0, which *is* an accepting state. This is possible because the NFSA has an ϵ-transition from q_2 to q_0. Figure 7.12 below shows the two possible paths (computations) of the NFSA on input 01. Since one of them ends in an accepting state, the NFSA accepts this string.

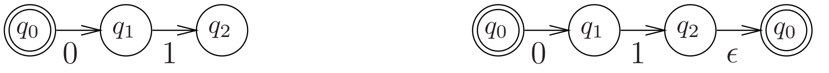


Figure 7.12: The two computations of the NFSA of Figure 7.11(a) on input 01.

Figure 7.13 below shows the two possible paths (computations) of the NFSA on input 0100. Since both of them end in nonaccepting states, the NFSA rejects this string.

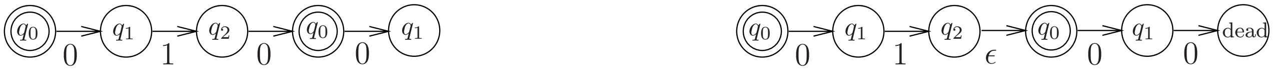


Figure 7.13: The two computations of the NFSA of Figure 7.11(a) on input 0100.

We claim that the NFSA shown in Figure 7.11(a) accepts the set of strings denoted by $(010{+}01)^*$. First, we show that the language accepted by the NFSA is a subset of $\mathcal{L}((010{+}01)^*)$. Let x be any string accepted by this automaton; an accepting computation of this automaton corresponds to a path that starts and ends in q_0, since this is the only accepting state. Such a path must be a repetition (zero or more times) of the cycle $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{0,\epsilon} q_0$, and therefore the string x must be a repetition of either 010 or $01\epsilon = 01$ (the sequence of labels along the edges of the loop). Thus, x is in $\mathcal{L}((010+01)^*)$, as wanted.

Conversely, we show that $\mathcal{L}((010+01)^*)$ is a subset of the language accepted by the NFSA. Let x be an arbitrary string in $\mathcal{L}((010+01)^*)$. Thus, x is a concatenation of some number of strings, each of which is either 010 or 01. Notice that 010 and 01 can take the NFSA from q_0 to q_0 via the cycles $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_0$ and $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{\epsilon} q_0$, respectively. Thus, any concatenation of such strings can take the NFSA from q_0 to q_0. Since q_0 is an accepting state, x is accepted by the NFSA, as wanted.

A *deterministic* FSA that accepts the same language (i.e., the strings denoted by $(010+01)^*$) is shown in Figure 7.11(b). In fact, it is possible to show that this DFSA has the smallest number of states amongst all DFSA that accept this language. The NFSA in Figure 7.11(a) has one state less than the *smallest* DFSA that accepts the same language. One state more or less is not such a big deal but, in general, there are languages that can be accepted by NFSA that are *much* smaller than the smallest DFSA for the same language (see Exercise 12). In our simple example, more important that the savings in the number of states is the gain in conceptual simplicity of the automaton. Most people would find it much easier to see (and prove) that the NFSA in Figure 7.11(a) accepts the language denoted by $(010+01)^*$, than to see (and prove) that the DFSA in Figure 7.11(b) accepts that language. **End of Example 7.17**

Now that we have developed some intuition about what NFSA look like and how they work, we are ready to give the mathematical definition.

Definition 7.15 *A* ***nondeterministic finite state automaton*** *(**NFSA**) is a quintuple $M = (Q, \Sigma, \delta, s, F)$, where*

- *Q is a* finite *set of* ***states****.*
- *Σ is a* finite *alphabet.*
- *$\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$ is the* ***transition function****.*
- *$s \in Q$ is the* ***start*** *or* ***initial*** *state.*
- *$F \subseteq Q$ is the set of* ***accepting*** *states.*

The only element of this definition that requires some clarification is the transition function δ. Intuitively, if the automaton is presently in state q then it may, without reading any input symbol, change its state to *any one* of the states in $\delta(q, \epsilon)$ (but to no other state). Furthermore, if the automaton is presently in state q and the present input symbol is a then it may change its state to *any one* of the states in $\delta(q, a)$ (but to no other state).

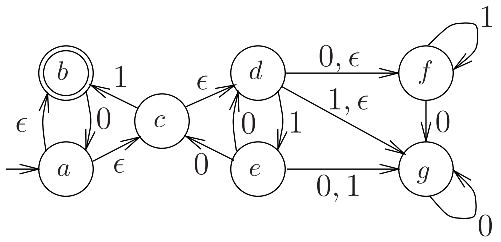


Figure 7.14: Illustrating the $\mathcal{E}$ notation

Given a NFSA M, we can define the ***extended transition function*** $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$. Intuitively, if $x \in \Sigma^*$ and $q \in Q$, $\delta^*(q, x)$ denotes the set of states in which the automaton *could* be if it starts in state q and processes all the symbols of string x. To define this, we first need the following notation: For any state q, $\mathcal{E}(q)$ denotes the set of states that are reachable from q by following (any number of) ϵ-transitions. Here, "any number" includes zero, so that for any state q, $q \in \mathcal{E}(q)$.

Example 7.18 Consider the NFSA in Figure 7.14. For this automaton, we have

$$\begin{aligned}
\mathcal{E}(a) &= \{a, b, c, d, f, g\}\\
\mathcal{E}(b) &= \{b\}\\
\mathcal{E}(c) &= \{c, d, f, g\}\\
\mathcal{E}(d) &= \{d, f, g\}\\
\mathcal{E}(e) &= \{e\}\\
\mathcal{E}(f) &= \{f\}\\
\mathcal{E}(g) &= \{g\}
\end{aligned}$$

End of Example 7.18

Definition 7.16 *Let $\delta : Q \times \Sigma \cup \{\epsilon\} \to \mathcal{P}(Q)$ be the transition function of a NFSA. The **extended transition function** of the NFSA is the function $\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q)$ defined by structural induction on x:*

Basis: *$x = \epsilon$. In this case, $\delta^*(q, x) = \mathcal{E}(q)$.*

Induction Step: *$x = ya$, for some $y \in \Sigma^*$ and $a \in \Sigma$; we assume, by induction, that $\delta^*(q, y)$ has been defined. In this case,*

$$\delta^*(q, x) = \bigcup_{q' \in \delta^*(q,y)} \left(\bigcup_{q'' \in \delta(q',a)} \mathcal{E}(q'') \right)$$

The induction step of this definition is formidable-looking, but it is possible to grasp it by analysing it piece-by-piece. Here is what it says, intuitively: To find the set of states in which the NFSA, started in state q, could be after reading input $x = ya$, we proceed as follows:

- First, we get to each state q' that can be reached by the NFSA, started in state q, after reading the prefix y of the input up to (but not including) the last symbol. The set of such states is $\delta^*(q, y)$, which we inductively assume is known.

- Next, we get to each state q'' that can be reached from q' by reading the last symbol, a, of the input. The set of such states is $\delta(q', a)$.

- Finally, we get to each state that can be reached from q'' by following any number of ϵ-transitions. The set of such states is $\mathcal{E}(q'')$.

Definition 7.17 *A NFSA $M = (Q, \Sigma, \delta, s, F)$* ***accepts*** *(or* ***recognises****) a string $x \in \Sigma^*$ if and only if $\delta^*(s, x) \cap F \neq \emptyset$. (In other words, M accepts x if and only if* at least one *of the possible states in which the automaton could be after processing input x is an accepting state.) The* ***language accepted by*** *M, $\mathcal{L}(M)$, is the set of strings accepted by M.*

Recall that $\delta(q, a)$ can be *any* subset of Q, including the *empty* set. This possibility, along with the definition of δ^*, raises the possibility that for some string $x \in \Sigma^*$, $\delta^*(s, x) = \emptyset$. If this is the case then surely $\delta^*(s, x) \cap F = \emptyset$ and, by the above definition, M does not accept x.

7.4.2 Equivalence of DFSA and NFSA

It is clear from the definition that a DFSA is a special case of a NFSA, where we restrict the transition function in certain ways. Specifically, for each state q we disallow ϵ-transitions except back to q, and we require that for any $a \in \Sigma$, $|\delta(q, a)| = 1$. Thus, nondeterministic FSA are at least as powerful as deterministic ones.

We have seen that the use of nondeterminism allows us to reduce the complexity of the automaton (in some quantifiable measure, such as the number of states). But does nondeterminism increase the *expressive power* of finite state automata? More precisely, is there a language that is accepted by a NFSA, but which is not accepted by *any* DFSA? We will prove that the answer to this question is negative: Both kinds of finite state automata accept precisely the same class of languages. This means that when we deal with finite state automata, we may assume the extra freedom of nondeterminism or require the greater discipline of determinism, as fits our purposes.

Given a NFSA $M = (Q, \Sigma, \delta, s, F)$ we show how to construct a DFSA $\widehat{M} = (\widehat{Q}, \Sigma, \widehat{\delta}, \widehat{s}, \widehat{F})$ that accepts the same language as M. This construction is called the ***subset construction***, because each state of $\widehat{M}$ is a *set of states* of M. Intuitively, input x takes $\widehat{M}$ from its initial state $\widehat{s}$ to state $\widehat{q}$ if and only if $\widehat{q}$ is the set of all states to which x *could* take M from its own initial state s (see Lemma 7.18, below). Formally, the components of $\widehat{M}$ are defined as follows:

- $\widehat{Q} = \mathcal{P}(Q)$.

- $\widehat{s} = \mathcal{E}(s)$ (i.e., the set of all states reachable from the initial state of the given NFSA via ϵ-transitions only).

- $\widehat{F} = \{\widehat{q} \in \widehat{Q} : \widehat{q} \cap F \neq \emptyset\}$ (i.e., all states that contain an accepting state of the given NFSA).

- For any $\widehat{q} \in \widehat{Q}$, and $a \in \Sigma$, $\widehat{\delta}(\widehat{q}, a) = \bigcup_{q' \in \widehat{q}} \left(\bigcup_{q'' \in \delta(q',a)} \mathcal{E}(q'') \right)$.

Here is an intuitive explanation of the transition function of $\widehat{M}$. To determine $\widehat{\delta}(\widehat{q}, a)$, the unique state that $\widehat{M}$ must enter when it is in state $\widehat{q}$ and reads input symbol a, we proceed as follows:

- For each of the states of M that make up $\widehat{q}$, say q', we determine (using M's nondeterministic transition function) the set of states to which M can move when it is in state q' and it reads input symbol a. The set of such states is $\delta(q', a)$.

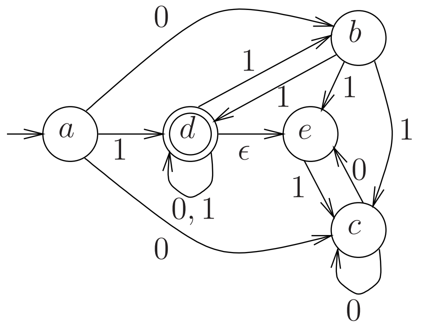
- For each of the states reached in the previous step, say q'', we determine (again, using M's transition function) the states that can be reached from q'' by following any number of ϵ-transitions. The set of such states is $\mathcal{E}(q'')$.

The set of states of M reachable by means of this two-step process is the desired state of $\widehat{M}$, $\widehat{\delta}(\widehat{q}, a)$.

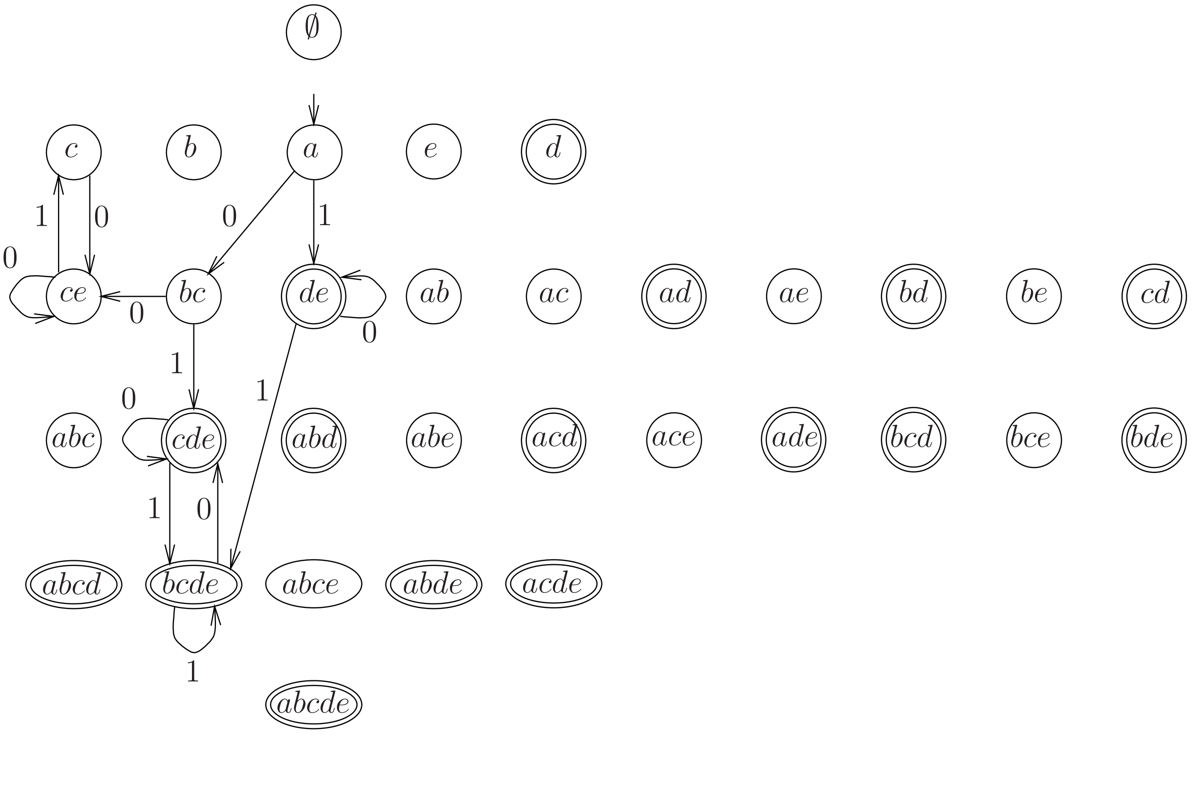
Note that the FSA $\widehat{M}$ defined by the subset construction is deterministic: From each of its states (a subset of M's states), a given input symbol can cause it to move to exactly one of its states.

Example 7.19 In Figure 7.15 we give an example of the subset construction. The given NFSA is shown at the top of the figure. It has five states: a, b, c, d and e. The initial state is a, and the only accepting state is d. The resulting DFSA is shown at the bottom of the figure. It has 32 states — one for each subset of $\{a, b, c, d, e\}$. To simplify the figure, each subset is indicated by listing its elements. For example, we write cde, instead of $\{c, d, e\}$. There are many transitions that are not shown in the figure, but these concern only states that are not reachable from the initial state of the DFSA.

The transitions shown were determined by using the rule specified in the subset construction for deriving $\widehat{\delta}$ from δ. Let us examine a few such transitions. We start from the initial state of the DFSA which, by the construction, is $\{a\}$ (written simply as a). Let us determine the state of the DFSA to which we go from $\{a\}$, when the input symbol in 0. We look at the NFSA, and notice that the states to which we can go from a on 0 are b and c. Thus, the state of the DFSA to which we can go from $\{a\}$ on 0 is the state $\{b, c\}$. Let us also determine the state of the DFSA to which we go from $\{a\}$ when the input symbol is 1. We consult at the NFSA and notice that the only transition from a labeled 1 is to state d. However, from d there is an ϵ-transition to e. Thus, in the NFSA, if we are in state a and the input symbol is 1, the next



A nondeterministic FSA



The deterministic FSA resulting from the subset construction
applied to the NFSA above

Figure 7.15: The subset construction

states to which we can go are d and e. Accordingly, in the DFSA, from $\{a\}$ we go to state $\{d,e\}$, when the input symbol is 1.

Now let's consider the transitions from state $\{b,c\}$. This state intuitively corresponds to being either in state b or in state c in the NFSA. Consider first the transitions on input 0. If we are in state b of the NFSA, then on input 0 we cannot go anywhere: this is a missing transition in the diagram, so formally we go to a dead state, not explicitly shown. From state c in the NFSA, on input 0 we can go either to c or to e. Thus, from state $\{b,c\}$ of the DFSA, on input 0 we go to state $\{c,e\}$. By a similar reasoning, we can determine that the transition from state $\{b,c\}$ in the DFSA on input 1 goes to state $\{c,d,e\}$.

Let us also consider the transition from state $\{c,e\}$ on input 1. From state c in the NFSA, on input 1 we cannot go anywhere; from state e, on input 1 we can only go to c. Thus, from state $\{c,e\}$ in the DFSA, on input 1 we go to state $\{c\}$.

Continuing in this way, we can determine the state transitions of the DFSA for all states that are reachable from the initial state. We can also determine the state transitions from states that are not reachable from the initial state, but these are useless so we don't bother. You should check the rest of the transitions, to make sure that you understand the subset construction. **End of Example 7.19**

The key to proving that the constructed DFSA $\widehat{M}$ accepts the same language as the given NFSA M is:

Lemma 7.18 *Let $M = (Q, \Sigma, \delta, s, F)$ be a NFSA, and $\widehat{M} = (\widehat{Q}, \Sigma, \widehat{\delta}, \widehat{s}, \widehat{F})$ be the DFSA obtained by applying the subset construction to M. For every $x \in \Sigma^*$, $\delta^*(s,x) = \widehat{\delta}^*(\widehat{s}, x)$.*

PROOF. Let $P(x)$ be the following predicate on strings:

$$P(x): \qquad \delta^*(s,x) = \widehat{\delta}^*(\widehat{s}, x)$$

We use structural induction to prove that $P(x)$ holds, for all strings x.

BASIS: $x = \epsilon$. By the definition of extended transition function for nondeterministic FSA, we have $\delta^*(s,x) = \mathcal{E}(s)$. By the definition of extended transition function for deterministic FSA and the definition of $\widehat{s}$ in the subset construction, we have $\widehat{\delta}^*(\widehat{s}, x) = \widehat{s} = \mathcal{E}(s)$. Thus, $\delta^*(s,x) = \widehat{\delta}^*(\widehat{s}, x)$.

INDUCTION STEP: $x = ya$, for some $y \in \Sigma^*$ and $a \in \Sigma$; we assume, by induction, that $P(y)$ holds, i.e., that $\delta^*(s,y) = \widehat{\delta}^*(\widehat{s}, y)$. We have

$$\begin{aligned}
\delta^*(s,x) &= \delta^*(s,ya) && \text{[by definition of } x\text{]} \\
&= \bigcup_{q' \in \delta^*(s,y)} \Bigg(\bigcup_{q'' \in \delta(q',a)} \mathcal{E}(q'') \Bigg) && \begin{array}{l}\text{[by definition of extended transition function} \\ \text{for nondeterministic FSA; cf. Definition 7.16]}\end{array} \\
&= \bigcup_{q' \in \widehat{\delta}^*(\widehat{s},y)} \Bigg(\bigcup_{q'' \in \delta(q',a)} \mathcal{E}(q'') \Bigg) && \text{[by the induction hypothesis]}
\end{aligned}$$

$$\begin{aligned}
&= \widehat{\delta}(\widehat{\delta}^*(\widehat{s}, y), a) && \text{[by definition of } \widehat{\delta} \text{ in subset construction]} \\
&= \widehat{\delta}^*(\widehat{s}, ya) && \text{[by definition of extended transition function} \\
& && \text{for deterministic FSA; cf. Definition 7.11]} \\
&= \widehat{\delta}^*(\widehat{s}, x) && \text{[since } x = ya\text{, in this case]}
\end{aligned}$$

as wanted. □

From this Lemma we immediately get:

Theorem 7.19 *Let M be a NFSA and $\widehat{M}$ be the DFSA obtained by applying the subset construction to M. Then, $\mathcal{L}(M) = \mathcal{L}(\widehat{M})$.*

Proof. For any string $x \in \Sigma^*$, we have:

$$\begin{aligned}
x \in \mathcal{L}(M) &\Leftrightarrow \delta^*(s, x) \cap F \neq \emptyset && \text{[by definition of acceptance by nondeterministic FSA;} \\
& && \text{cf. Definition 7.17]} \\
&\Leftrightarrow \widehat{\delta}^*(\widehat{s}, x) \cap F \neq \emptyset && \text{[by Lemma 7.18]} \\
&\Leftrightarrow \widehat{\delta}^*(\widehat{s}, x) \in \widehat{F} && \text{[by definition of } \widehat{F} \text{ in subset construction]} \\
&\Leftrightarrow x \in \mathcal{L}(\widehat{M}) && \text{[by definition of acceptance by deterministic FSA;} \\
& && \text{cf. Definition 7.12]}
\end{aligned}$$

Thus, $\mathcal{L}(M) = \mathcal{L}(\widehat{M})$, as wanted. □

Corollary 7.20 *The class of languages accepted by NFSA is the same as the class of languages accepted by DFSA.*

We will use the term "finite state automaton" (abbreviated FSA), without specifying whether it is deterministic or not, if this is not germane to the point being made. In particular, if all we care about is whether a language can be accepted by a finite state automaton, Theorem 7.19 and Corollary 7.19 assure us that we can assume that the automaton is deterministic or nondeterministic — whichever is more convenient. We will make use of this flexibility subsequently. Of course, if we are interested in other aspects of the automaton than just the language it accepts (e.g. the number of states it has), it can be quite relevant to know whether an automaton must be deterministic or not.

The subset construction results in an exponential blow-up of the number of states: If the given NFSA has n states, the resulting DFSA has 2^n states. The question therefore arises: Is such a huge increase in the number of states necessary? Could we have a more efficient construction, which would result in smaller DFSA? The answer turns out to be negative. The reason is that there exist languages which are accepted by "small" NFSA but are accepted only by “large” DFSA. Specifically, for each $n \in \mathbb{N}$, there is a language that is accepted by a NFSA with $n + 1$ states, but is not accepted by *any* DFSA with fewer than 2^n states. That is, the exponential explosion in the number of states inherent in the subset construction is, in general, necessary (at least within a factor of 2). This means that, in general, NFSA are much more efficient representations of languages than DFSA. This issue is explored in Exercise 12.

7.5 Closure properties of FSA-accepted languages

There are many natural operations by which we can combine languages to obtain other languages. It turns out that many of these operations, when applied to languages that are accepted by FSA result in a language that is also accepted by some FSA. If an operation has this property, we say that the class of languages accepted by FSA is closed under this operation. In this section we establish some closure properties of this type. Both the results themselves, and the techniques by which they can be proved, are quite important.

Theorem 7.21 *The class of languages that are accepted by FSA is closed under complementation, union, intersection, concatenation and the Kleene star operation. In other words, if L and L' are languages that are accepted by FSA, then so are all of the following: $\overline{L}$, $L \cup L'$, $L \cap L'$, $L \circ L'$ and $L^{\circledast}$.*

Proof. Let $M = (Q, \Sigma, \delta, s, F)$ and $M' = (Q', \Sigma, \delta', s', F')$ be FSA that accept L and L', respectively; i.e., $\mathcal{L}(M) = L$ and $\mathcal{L}(M') = L'$. For each of the above five operations, we will show how to construct finite state automata that accept the language defined by that operation. For most operations, we will describe the construction graphically, by showing how the given automata M and M' (which are depicted Figure 7.16) can be modified to construct the new automaton that accepts the desired language.

Figure 7.16: FSA M and M', accepting languages L and L'.

Complementation: By Theorem 7.19, we may assume, without loss of generality, that M is deterministic. Consider the FSA $\overline{M} = (Q, \Sigma, \delta, s, Q - F)$. Intuitively, in $\overline{M}$ we keep the same set of states and transition function as in M, but we make every accepting state of M nonaccepting, and we make every nonaccepting state of M accepting. We claim that $\overline{M}$ accepts

$\overline{L}$. Here is why. Let x be an arbitrary string in Σ^*.

$$\begin{aligned} x \in \mathcal{L}(\overline{M}) &\Leftrightarrow \delta^*(s,x) \in Q - F && \text{[by definition of acceptance by DFSA; cf. Definition 7.12]} \\ &\Leftrightarrow \delta^*(s,x) \notin F && \text{[by definition of } \overline{M}\text{]} \\ &\Leftrightarrow x \notin \mathcal{L}(M) && \text{[by definition of acceptance by DFSA; cf. Definition 7.12]} \end{aligned}$$

Thus $\mathcal{L}(\overline{M}) = \overline{\mathcal{L}(M)} = \overline{L}$, as wanted. (The assumption that M is deterministic is important in this case. If you are not sure why, you should carefully review the definition of acceptance for both deterministic and nondeterministic automata.)

Union: The NFSA $M_\cup$ shown diagrammatically in Figure 7.17 accepts $L \cup L'$. The states of $M_\cup$ are the states of M and M', and a new state denoted $s_\cup$, which is the initial state of $M_\cup$. Its transition function is as depicted in Figure 7.17.

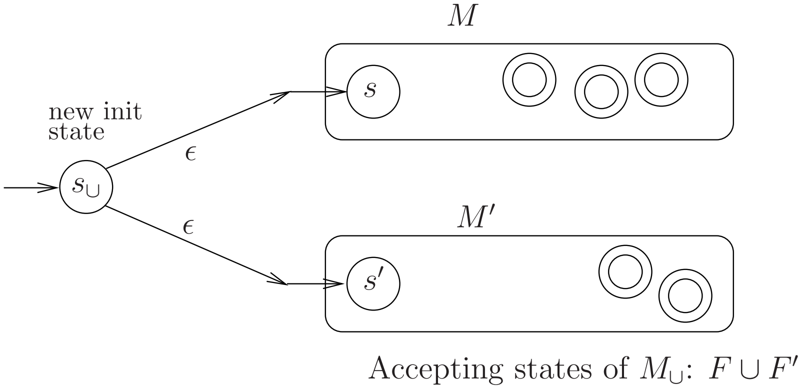


Figure 7.17: FSA $M_\cup$ that accepts $L \cup L'$.

To prove that $M_\cup$ accepts $L \cup L'$ we must show that, for any string x, x is accepted by $M_\cup$ if and only if $x \in L \cup L'$.

[IF] Let x be an arbitrary string in $L \cup L'$. Then either $x \in L$ or $x \in L'$. Without loss of generality, assume that $x \in L$ (the case $x \in L'$ is handled similarly). Then one possible computation of $M_\cup$ on input x is to first take the ϵ-transition from $s_\cup$ (the initial state of $M_\cup$) to s (the initial state of M), and then process the symbols of x as M would. Since $x \in L$, it is possible that M can be in one of its accepting states after processing x; by following the same sequence of moves after the initial ϵ-transition, $M_\cup$ can also be in the same state after processing x. Since every accepting state of M is also an accepting state of $M_\cup$, this is an accepting computation for x in $M_\cup$, so $M_\cup$ accepts x, as wanted.

[ONLY IF] Let x be an arbitrary string accepted by $M_\cup$. Consider any accepting computation of $M_\cup$ on x. The first move of such a computation is an ϵ-transition from $s_\cup$ to s or s'.

CASE 1. The first move of the accepting computation of $M_\cup$ on input x was an ϵ-transition from $s_\cup$ to s. The remaining moves of the computation must be legal moves of M, starting in state s and ending in an accepting state of M. (This is because, by the construction of $M_\cup$, once $M_\cup$ enters a state of M it can only follow transitions of that automaton, and the only accepting states of $M_\cup$ that are also states of M, are the accepting states of M.) Thus, the accepting computation of $M_\cup$ on x is also an accepting computation of M on input x. Hence, $x \in \mathcal{L}(M)$, i.e., $x \in L$.

Case 2. The first move of the accepting computation of $M_\cup$ on input x was an ϵ-transition from $s_\cup$ to s'. An argument similar to that in the previous case shows that, in this case, $x \in L'$.

Since either $x \in L$ or $x \in L'$, we have that $x \in L \cup L'$, as wanted.

The construction of $M_\cup$, shown diagrammatically in Figure 7.17, can also be described formally as follows. Assume, without loss of generality, that M and M' are nondeterministic. Let $s_\cup$ be a new state, i.e., a state that does not belong to Q or Q'. Then $M_\cup = (Q_\cup, \Sigma, \delta_\cup, s_\cup, F_\cup)$, where $Q_\cup = Q \cup Q' \cup \{s_\cup\}$, $F_\cup = F \cup F'$, and for any $q \in Q_\cup$ and $a \in \Sigma \cup \{\epsilon\}$,

$$\delta_\cup(q,a) = \begin{cases} \delta(q,a), & \text{if } q \in Q \\ \delta'(q,a), & \text{if } q \in Q' \\ \{s_\cup, s, s'\}, & \text{if } q = s_\cup \text{ and } a = \epsilon \end{cases}$$

Intersection: Since $L \cap L' = \overline{\overline{L} \cup \overline{L'}}$, the result follows from the previous two. Alternatively, we can give an explicit construction for an FSA $M_\cap = (Q_\cap, \Sigma, \delta_\cap, s_\cap, F_\cap)$ that accepts $L \cap L'$.

- $Q_\cap = Q \times Q'$
- $s_\cap = (s, s')$
- $F_\cap = F \times F'$
- $\delta_\cap\big((q,q'),a\big) = \big(\delta(q,a), \delta'(q',a)\big)$

This is known as the ***Cartesian product construction***. Intuitively, the machine $M_\cap$ runs both automata M_1 and M_2 concurrently. It starts both automata in their respective initial states. Upon reading an input symbol it makes a move on each of M and M'. It accepts if and only if, at the end, both automata are in an accepting state. We leave as an exercise the proof that the automaton resulting from this construction accepts $L \cap L'$ (see Exercise 8).

Concatenation: The NFSA $M_\circ$ shown diagrammatically in Figure 7.18 accepts $L \circ L'$. We leave as an exercise the proof that this construction is correct (i.e., that $M_\circ$ accepts $L \circ L'$) and a mathematical (as opposed to graphical) description of $M_\circ$.

Kleene star: The NFSA $M_\circledast$ shown diagrammatically in Figure 7.19 accepts $L^\circledast$. Again, we leave as an exercise the proof of correctness and a mathematical (as opposed to graphical) description of $M_\circledast$. □

Notice how, in the preceding proof, we have taken advantage of nondeterminism, and specifically of the use of ϵ-transitions, in the constructions of $M_\cup$, $M_\circ$ and $M_\circledast$. We have also taken advantage of determinism in the construction of $\overline{M}$.

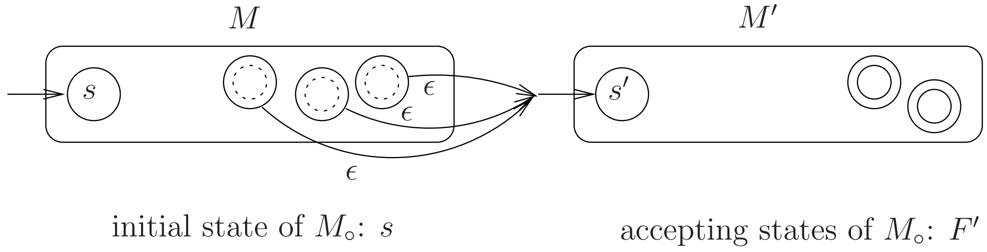

M
M'
s
s'
ϵ
ϵ
ϵ
initial state of $M_\circ$: s
accepting states of $M_\circ$: F'

Figure 7.18: FSA $M_\circ$ that accepts $L \circ L'$.

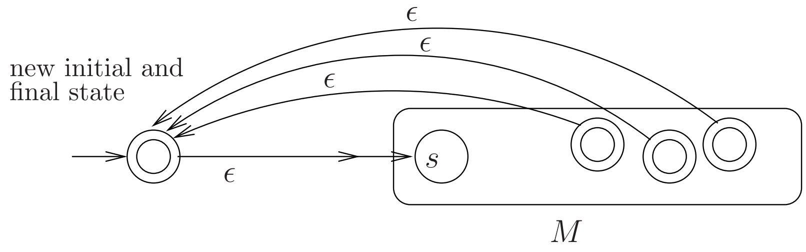


Figure 7.19: FSA $M_\circledast$ that accepts $L^\circledast$.

Many other closure properties of FSA-accepted languages can be proved. In some cases we can easily prove such properties by applying previously established closure properties.

Example 7.20 For any languages L, L' over alphabet Σ, define the operation $\Cap$ as follows:

$$L \Cap L' = \{x \in \Sigma^* : \ x \in L \cap L' \text{ or } x \notin L \cup L'\}$$

That is, $L \Cap L'$ is the set of strings that are in both languages or in neither language. From its definition, it is immediate that $L \Cap L' = (L \cap L') \cup \overline{(L \cup L')}$. In view of this, and the fact that the class of FSA-accepted languages is closed under union, intersection and complementation, it follows that this class is also closed under the operation $\Cap$. In other words, if there are FSA that accept L and L', then there is a FSA that accepts $L \Cap L'$. **End of Example 7.20**

7.6 Equivalence of regular expressions and FSA

We now prove that regular expressions and FSA are equivalent in their power to describe languages. That is, any language denoted by a regular expression is accepted by some FSA (this shows that FSA are at least as powerful as regular expressions); and, conversely, any language accepted by a FSA can be denoted by a regular expression (this shows that regular expressions are at least as powerful as FSA).

This is a result that is both important in practice, and mathematically interesting. We will note the practical relevance of this result as we go along. Its mathematical interest arises from the fact that it establishes the equivalence of two very different, though natural, ways to describe languages. This increases our confidence that the underlying concept is important. If many different roads lead to Rome, we gather that Rome is an important city!

7.6.1 *From regular expressions to FSA*

We first prove that given a regular expression R, we can construct a FSA M that accepts precisely the language denoted by R. The basic idea is to recursively construct FSA that accept the languages denoted by the subexpressions of R, and combine these automata (much in the way we saw in the proof of closure properties in the previous section) to accept the language denoted by the entire expression R. Naturally, then, the proof is by structural induction on the construction of R.

Theorem 7.22 *For every regular expression R there is a FSA M such that $\mathcal{L}(M) = \mathcal{L}(R)$.*

Proof. By structural induction on the definition of R.

Basis: Figure 7.20 shows FSA that accept the languages denoted by the expressions in the basis of the definition of regular expressions: $\emptyset$, ϵ, and a for each $a \in \Sigma$.

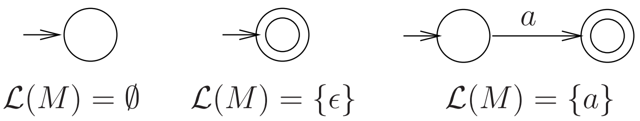


Figure 7.20: FSA for the regular expressions $\emptyset$, ϵ and $a \in \Sigma$.

Induction Step: Suppose $R = (S + S')$, where S and S' are regular expressions. By the induction hypothesis, we assume that there are FSA that accept $\mathcal{L}(S)$ and $\mathcal{L}(S')$. By Theorem 7.21, the class of languages accepted by FSA is closed under union. Thus, $\mathcal{L}(S) \cup \mathcal{L}(S')$, which is $\mathcal{L}(R)$, is also accepted by some FSA. In fact, the proof of Theorem 7.21 and, in particular, Figure 7.17 shows how to construct such a FSA.

The two remaining cases, $R = (SS')$ and $R = S^*$, are shown in the same way since, by Theorem 7.21, the class of languages accepted by FSA is also closed under concatenation and Kleene star. □

Example 7.21 In Figure 7.21 we show how to construct a FSA that accepts the language denoted by $(0 + 1)(11)^*$. We show the construction step-by-step, bottom-up: We start with FSA that accept the simplest subexpressions of $(0 + 1)(11)^*$ (as in the basis of the proof of Theorem 7.22). We then show how to combine these to obtain FSA that accept larger and larger subexpressions (as in the induction step of the proof of Theorem 7.22), until we have constructed a FSA that accepts the entire expression. Note that the induction step of the proof of Theorem 7.22 uses the constructions for union, concatenation and the Kleene star shown in the proof of Theorem 7.21. **End of Example 7.21**

The construction of the FSA from the regular expression illustrated in the previous example is completely algorithmic. It decomposes the regular expression, recursively constructing FSA for the larger subexpressions out of the FSA it previously constructed for the simpler subexpressions. Of course, the final (or intermediate) FSA produced can be simplified by eliminating obviously unnecessary ϵ-transitions. A somewhat simplified FSA that accepts $(0+1)(11)^*$ is shown at the bottom of Figure 7.21.

This construction is used by pattern-matching utilities such as the Unix program *grep*, as we explain below. Roughly speaking, the user of *grep* specifies two parameters: a regular expression and a file. The job of *grep* is to print each line of the file that contains (as a substring) any string denoted by the regular expression. For instance, the Unix command

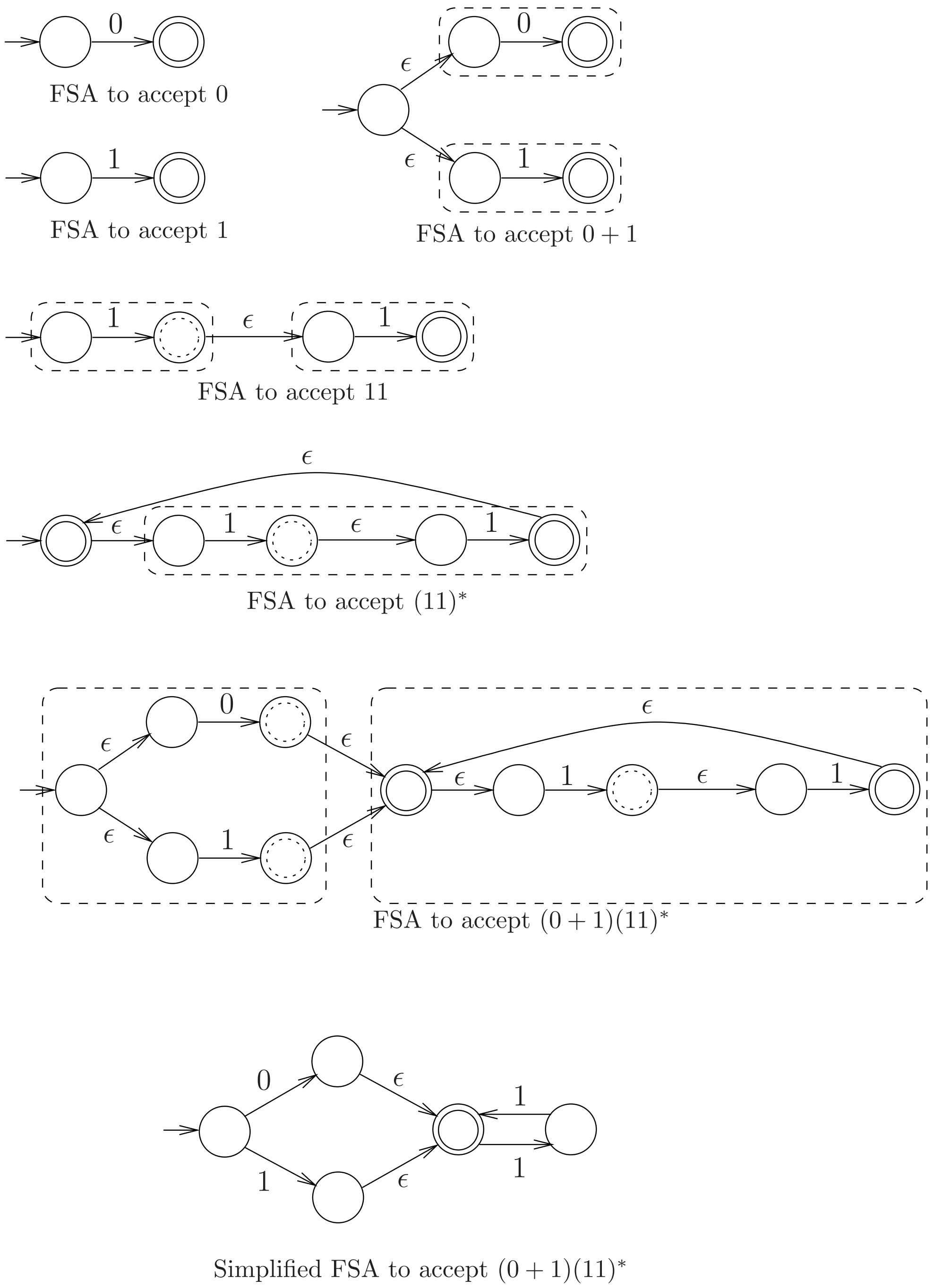
```
grep 'g[ae]m.*t' /usr/dict/words
```

results in the following output

```
amalgamate
bergamot
gambit
gamut
geminate
gemstone
largemouth
ligament
```

The first argument of this command, `'g[ae]m.*t'`, is a regular expression written in the syntax required by *grep*. The second argument is a file name. The regular expression has the following meaning (ignoring the surrounding quotation marks, which are needed for reasons we will not explain here): The notation `[ae]` stands for "either `a` or `e`" (i.e., $(\mathtt{a}+\mathtt{e})$, in our notation). The dot (.) is shorthand for "any character whatsoever". The star ($*$) has the normal meaning: any number of repetitions. Thus, this regular expression denotes the set of strings that start with `g`, followed by either `a` or `e`, followed by `m`, followed by anything than ends in `t`. In standard Unix systems, the file `/usr/dict/words` contains a list of English words. Therefore this command says: find all the English words (in the Unix dictionary) that contain, as a substring, the pattern described above. The answer is the list shown above. (This particular use of *grep* is handy when playing hangman or doing crossword puzzles!)

The *grep* utility makes use of the construction implicit in the proof of Theorem 7.22 and illustrated in Example 7.21. Suppose $'\mathtt{E}'$ is the regular expression specified by the user in the command line. *grep* first modifies this to $'.*\mathtt{E}.*'$. This denotes the set of strings that contain a string in `E` as a substring. It then transforms this regular expression to a NFSA as explained



FSA to accept 0

FSA to accept 1

FSA to accept $0 + 1$

FSA to accept 11

FSA to accept $(11)^*$

FSA to accept $(0 + 1)(11)^*$

Simplified FSA to accept $(0 + 1)(11)^*$

Figure 7.21: Constructing a FSA that accepts $(0 + 1)(11)^*$.

above. For each line x of the specified file, *grep* "runs" the constructed NFSA using x as input; if the NFSA accepts x, *grep* prints x, otherwise it does not.[8]

7.6.2 From FSA to regular expressions

Next we will prove that for any FSA M we can construct a regular expression that denotes precisely the language accepted by M. This construction is much less obvious than the reverse direction that we saw in the previous section. Unlike regular expressions, finite state automata do not have a recursive definition: We do not define each FSA in terms of "smaller" FSA. As a result there is no obvious peg on which to hang an inductive argument. The proof is, as we will see, an inductive one, but it takes considerable ingenuity to see how to frame it. This theorem is due to Stephen Kleene, who proved it in 1956.

Theorem 7.23 *For every FSA M there is a regular expression R such that $\mathcal{L}(R) = \mathcal{L}(M)$.*

PROOF. Let $M = (Q, \Sigma, \delta, s, F)$. Without loss of generality assume that M is deterministic, and that the set of states is $Q = \{1, 2, \ldots, n\}$, for some $n \in \mathbb{N}$. That is, the states are numbered from 1 to n.

Let $x \in \Sigma^*$ and $i \in Q$. Suppose we start M in state i and run it with string x as input, As we have seen, the computation of M on input x corresponds to a path in the transition diagram of the FSA. The labels on this path are the symbols of x. For each $i, j \in Q$ (not necessarily distinct), and each k such that $0 \leq k \leq n$, we will now define a set of strings, denoted L^k_{ij}. Intuitively, this is the set of strings that take M from state i to state j through a computation path in which all *intermediate* states (i.e., all states on the path except the first and last) are no greater than k. More precisely, $x \in L^k_{ij}$ if and only if for some $\ell \geq 0$, there is a sequence of (not necessarily distinct) symbols in Σ, $a_1, a_2, \ldots, a_\ell$, and a sequence of (not necessarily distinct) states $s_0, s_1, \ldots, s_\ell$, so that:

- $x = a_1 a_2 \cdots a_\ell$ (x is the concatenation of the sequence of symbols).
- $s_0 = i$ and $s_\ell = j$ (the sequence of states starts in i and ends in j).
- For each m such that $0 < m < \ell$, $s_m \leq k$ (every state in the sequence, except the first and last, is at most k).
- For each m such that $1 \leq m \leq \ell$, $s_m = \delta(s_{m-1}, a_m)$ (every state in the sequence, except the first, is reached from the previous one by a transition of M upon seeing the corresponding symbol).

We claim that

$$\text{For all } k,\ 0 \leq k \leq n, \text{ for all } i, j \in Q, \text{ there is a regular expression } R^k_{ij} \text{ that denotes the set } L^k_{ij} \tag{7.8}$$

[8] *grep* has many other useful features. For instance, we can find all lines that contain strings denoted by the regular expression as a prefix or suffix (instead of containing the expression as a substring). It also contains a number of handy abbreviations for regular expressions. For full details you should consult the Unix manual.

Let us, for the moment, assume that we have proved (7.8). We can complete the proof of the theorem as follows: Let s be the initial state, and $f_1, f_2, \ldots, f_t$ be the accepting states of M, for some $t \in \mathbb{N}$. The language accepted by M is the set of all strings that can take M from the initial state to any accepting state. That is,

$$\mathcal{L}(M) = \begin{cases} \emptyset, & \text{if } t = 0 \\ L^n_{sf_1} \cup \cdots \cup L^n_{sf_t}, & \text{if } t \geq 1 \end{cases}$$

Accordingly, the regular expression

$$R = \begin{cases} \emptyset, & \text{if } t = 0 \\ R^n_{sf_1} + \cdots + R^n_{sf_t}, & \text{if } t \geq 1 \end{cases}$$

denotes $\mathcal{L}(M)$, where $R^n_{sf_1}, \ldots, R^n_{sf_t}$ are regular expressions which, by (7.8), denote the languages $L^n_{sf_1}, \ldots, L^n_{sf_t}$, respectively.

We now return to the proof of (7.8). For any $i, j \in Q$,

$$L^0_{ij} = \begin{cases} \{a \in \Sigma : \delta(i,a) = j\}, & \text{if } i \neq j \\ \{\epsilon\} \cup \{a \in \Sigma : \delta(i,a) = j\}, & \text{if } i = j \end{cases} \tag{7.9}$$

$$L^{k+1}_{ij} = L^k_{ij} \cup (L^k_{i,k+1} \circ (L^k_{k+1,k+1})^{\circledast} \circ L^k_{k+1,j}) \qquad \text{for } 0 \leq k < n \tag{7.10}$$

Let us now explain why these equations hold. By definition, L^0_{ij} is the set of strings x that take M from i to j without going through any intermediate states. Therefore, Equation (7.9) states the following: The only strings that take M from state i to state $j \neq i$ without going through any intermediate states are strings consisting of a single symbol, say a, such that $\delta(i,a) = j$. Similarly, the only strings that take M from state i to state i without going through any intermediate states are the empty string and strings consisting of a single symbol, say a, such that $\delta(i,a) = i$.

Equation (7.10) states the following: A string x that takes M from i to j without going through a state higher than $k+1$, either

(a) takes M from i to j without even going through a state higher than k (in which case x is in L^k_{ij}), or

(b) it consists of a prefix that takes M from i to $k+1$ without going through a state higher than k (this prefix is in $L^k_{i,k+1}$), followed by zero or more substrings each of which takes M from $k+1$ back to $k+1$ without going through a state higher than k (this substring is in $(L^k_{k+1,k+1})^{\circledast}$), finally followed by a suffix that takes M from $k+1$ to j without going through a state higher than k (this suffix is in $L^k_{k+1,j}$).

Given this characterisation of L^k_{ij} for each k such that $0 \leq k \leq n$, we can now prove (7.8) by induction on k. More precisely, let $P(k)$ be the predicate:

$P(k)$: For all $i, j \in Q$, there is a regular expression R^k_{ij} that denotes L^k_{ij}

We will use induction to prove that $P(k)$ holds for every integer k such that $0 \leq k \leq n$.

Basis: $k = 0$. By (7.9), L^0_{ij} is a finite subset of $\Sigma \cup \{\epsilon\}$. In other words, one of the following two cases holds: (a) $L^0_{ij} = \emptyset$, or (b) for some $t \geq 1$, $L^0_{ij} = \{a_1, \ldots, a_t\}$, where $a_m \in \Sigma \cup \{\epsilon\}$, for each m such that $1 \leq m \leq t$. Accordingly, the regular expression $\emptyset$ or $a_1 + \cdots + a_t$ denotes L^0_{ij}. Therefore, $P(0)$ holds letting R^0_{ij} be $\emptyset$ or $a_1 + \cdots + a_t$, depending on which of the two cases applies.

Induction Step: Let m be an arbitrary integer such that $0 \leq m < n$, and suppose that $P(m)$ holds; i.e., for all $i, j \in Q$, there is a regular expression R^m_{ij} that denotes L^m_{ij}. We must prove that $P(m+1)$ holds as well. Let i, j be arbitrary states in Q. We must prove that there is a regular expression R^{m+1}_{ij} that denotes L^{m+1}_{ij}.

Since $0 \leq m < n$, we have by (7.10):

$$L^{m+1}_{ij} = L^m_{ij} \cup (L^m_{i,m+1} \circ (L^m_{m+1,m+1})^{\circledast} \circ L^m_{m+1,j}) \tag{7.11}$$

Furthermore, by the induction hypothesis, there are regular expressions R^m_{ij}, $R^m_{i,m+1}$, $R^m_{m+1,m+1}$ and $R^m_{m+1,j}$ that denote, respectively, the sets L^m_{ij}, $L^m_{i,m+1}$, $L^m_{m+1,m+1}$ and $L^m_{m+1,j}$. Then, by (7.11), the regular expression $R^m_{ij} + R^m_{i,m+1}(R^m_{m+1,m+1})^* R^m_{m+1,j}$ denotes L^{m+1}_{ij}. Thus, $P(m+1)$ holds, as wanted.

This completes the proof of (7.8), and thereby of the theorem. □

Example 7.22 Consider the 3-state DFSA shown in Figure 7.22. We will determine a regular expression that denotes the language accepted by that automaton, by following the inductive computation described in the preceding proof.

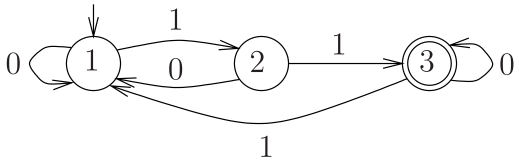


Figure 7.22: A finite state automaton

The table below shows regular expressions R^k_{ij} for $k \in \{0, 1, 2\}$ for this automaton. These expressions were computed by using Equations (7.9)–(7.10) and applying some equivalence-preserving simplifications.

	$k = 0$	$k = 1$	$k = 2$
R_{11}^k	$\epsilon + 0$	0^*	$0^*(\epsilon + 1(00^*1)^*00^*)$
R_{12}^k	1	0^*1	$0^*1(00^*1)^*$
R_{13}^k	$\emptyset$	$\emptyset$	$0^*1(00^*1)^*1$
R_{21}^k	0	00^*	$(00^*1)^*00^*$
R_{22}^k	ϵ	$\epsilon + 00^*1$	$(00^*1)^*$
R_{23}^k	1	1	$(00^*1)^*1$
R_{31}^k	1	10^*	$1(\epsilon + 0^*1(00^*1)^*0)0^*$
R_{32}^k	$\emptyset$	10^*1	$10^*1(00^*1)^*$
R_{33}^k	$\epsilon + 0$	$\epsilon + 0$	$\epsilon + 0 + 10^*1(00^*1)^*1$

The entries of this table were computed inductively, column-by-column. The entries of the first column (labeled "$k = 0$") are easily derived from the FSA, using Equation (7.9). Then we calculated the entries of the second column ($k = 1$), using Equation (7.10) and the expressions already derived in the first column. Similarly for the third column ($k = 2$).

With these expressions at hand, we can now derive a regular expression that denotes the language accepted by the FSA. Since there is a unique accepting state, 3, and the start state is 1, the language accepted by the FSA is L_{13}^3. Using Equation (7.10) and the expressions in the last column we obtain the following regular expression for the language accepted by the FSA:

$$0^*1(00^*1)^*1 + 0^*1(00^*1)^*1(\epsilon + 0 + 10^*1(00^*1)^*1)^*(\epsilon + 0 + 10^*1(00^*1)^*1)$$

which can be simplified to:

$$0^*1(00^*1)^*1(0 + 10^*1(00^*1)^*1)^*$$

End of Example 7.22

The proof of Theorem 7.23 is constructive. It doesn't merely prove the existence of a regular expression that denotes the language accepted by a FSA; it actually gives an algorithm to determine this regular expression. The algorithm follows a particular method called *dynamic programming*. Like divide-and-conquer, dynamic programming is a recursive problem-solving technique that finds the solution to a "large" instance of a problem by first determining solutions to "smaller" instances of the same problem, and then combining these to obtain the solution to the "large" instance. Many important problems can be solved efficiently by algorithms based on this method.

One of the many applications of the procedure described in the proof of Theorem 7.23 and illustrated in the previous example is the automatic derivation of state invariants for FSA, to which we alluded in Section 7.3.3 (cf. page 207). Recall that the invariant of a state q is a description of the set of strings that take the automaton from the initial state to q — that is, the set L_{sq}^n, where s is the initial state and n is the number of states. The method described above shows how to compute a regular expression that describes that set.

7.6.3 Regular languages

Theorems 7.22 and 7.23, and Corollary 7.20 immediately imply:

Corollary 7.24 *Let L be a language. The following three statements are equivalent:*

(a) L is denoted by a regular expression.

(b) L is accepted by a deterministic FSA.

(c) L is accepted by a nondeterministic FSA.

Thus the same class of languages is described by any one of these three formalisms: regular expressions, DFSA and NFSA. This class of languages is important enough to have a special name!

Definition 7.25 *A language is called* ***regular*** *if and only if it is denoted by some regular expression; or, equivalently, if and only if it is accepted by a (deterministic or nondeterministic) FSA.*

Now we are in a position to answer some of the questions we had posed in Section 7.2.6. Suppose we have two regular expressions R and R'. Is there a regular expression that denotes $\mathcal{L}(R) \cap \mathcal{L}(R')$? The answer to this question was not clear before now: regular expressions do not have an operator to directly represent intersection, nor is it obvious how to express intersection in terms of the available operators. At this point however, we have enough technical machinery to be able to answer this question. Yes, such a regular expression exists: The languages denoted by regular expressions are (by definition) regular. Regular languages are closed under intersection (by Corollary 7.24 and Theorem 7.21). Thus, $\mathcal{L}(R) \cap \mathcal{L}(R')$ can be denoted by a regular expression.

Actually, the results we have derived allow us to do more. Not only can we say that a regular expression, $R_\cap$, that denotes $\mathcal{L}(R) \cap \mathcal{L}(R')$ exists; we can actually algorithmically construct such an expression, given R and R'. Here is how:

(1) From R and R' we construct two FSA M and M' that accept $\mathcal{L}(R)$ and $\mathcal{L}(R')$, respectively. (The way to do this was described in the proof of Theorem 7.22 and illustrated in Example 7.21.)

(2) From M and M' we construct a FSA $M_\cap$ that accepts the intersection of the languages accepted by M and M', i.e., $\mathcal{L}(R) \cap \mathcal{L}(R')$. (The way to do this was described in the proof of Theorem 7.21.)

(3) From the FSA $M_\cap$ we can construct a regular expression $R_\cap$ that denotes the language accepted by $M_\cap$, i.e., $\mathcal{L}(R) \cap \mathcal{L}(R')$. (The way to do this was described in the proof of Theorem 7.23 and illustrated in Example 7.22.)

We can use similar techniques to answer other questions raised in Section 7.2.6. For example, we can prove that, for any regular expression R, there is a regular expression $\overline{R}$ that denotes the complement of the language denoted by R. In fact, we can show how to construct $\overline{R}$ from R, algorithmically.

One question raised in Section 7.2.6 that we have not yet addressed, however, is proving that there are certain languages that cannot be denoted by regular expressions. This question is taken up next.

7.7 Proving nonregularity: the Pumping Lemma

Although regular languages arise naturally in many applications, it is also true that many languages of interest are not regular: they are not denoted by any regular expression or, equivalently, they are not accepted by any FSA. The fact that a FSA has only a fixed number of states means that it can only "remember" a bounded amount of things. Thus, if we have a language such that there is no *a priori* bound on the amount of information that the automaton needs to remember to decide whether a string is in the language or not, then such a language is not regular.

An example of a language that has this property is $\{x \in \{0,1\}^* : x \text{ has as many 0s as 1s}\}$. It appears that an automaton that accepts this language must "remember" the difference in the number of 0s and 1s it has seen so far. Since there are infinitely many differences of 0s and 1s that strings can have, it would appear that we need an automaton with an infinite number of states to accept this language.

Another example of a language with a similar property is $\{0^n1^n : n \geq 0\}$. In this case, it appears that the automaton would have to remember how many 0s it has seen when it gets to the boundary between 0s and 1s, so as to make sure that the string has the same number of 1s. Since there is no bound on the number of 0s in the strings that are in the language, it would again appear that we need an infinite state automaton to accept this language.

In fact, these arguments can be made precise, and rigorous proofs can be given that the two languages mentioned above are not regular. There is a particularly useful tool for proving that languages are not regular, called the "Pumping Lemma". Intuitively, it states that any sufficiently long string of a regular language L has a nonempty substring which can be repeated ("pumped") an arbitrary number of times, with the resulting string still being in L.

Recall that if x is a string and $k \geq 0$ an integer, x^k is the string obtained by repeating x, k times. (See Equation (7.1) on page 184 for the formal, inductive, definition.)

Theorem 7.26 ***(Pumping Lemma)*** *Let $L \subseteq \Sigma^*$ be a regular language. Then there is some $n \in \mathbb{N}$ (that depends on L) so that every $x \in L$ that has length n or more satisfies the following property:*

> *There are $u, v, w \in \Sigma^*$ such that $x = uvw$, $v \neq \epsilon$, $|uv| \leq n$, and $uv^kw \in L$, for all $k \in \mathbb{N}$*

Proof. Let $M = (Q, \Sigma, \delta, s, F)$ be a FSA that accepts L. Without loss of generality we assume that M is deterministic. Let n be the number of states of this automaton.[9] Consider any string $x \in L$ such that $|x| = \ell \geq n$. Let $x = a_1a_2 \dots a_\ell$, where $a_m \in \Sigma$ for all m such that $1 \leq m \leq \ell$. Figure 7.23 shows the sequence of states $q_0, q_1, \dots, q_\ell$ through which M goes as it reads each symbol a_t of the input string x, starting in the initial state $q_0 = s$.

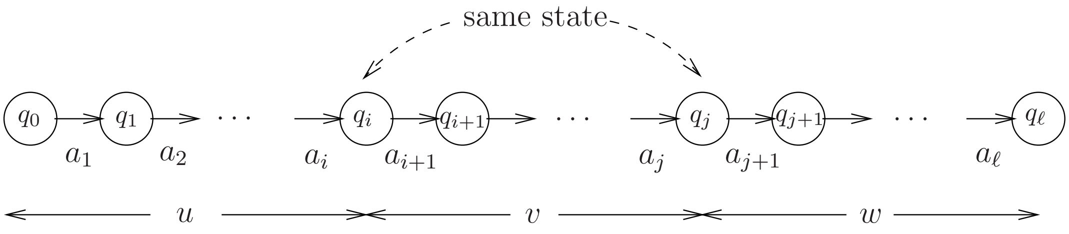


Figure 7.23: Illustration of Pumping Lemma

This sequence of states is formally defined by induction:

$$q_m = \begin{cases} s, & \text{if } m = 0 \\ \delta(q_{m-1}, a_m), & \text{if } 0 < m \leq \ell \end{cases}$$

Consider the prefix $q_0, q_1, \dots, q_n$ of this sequence. This sequence has $n+1$ states; since M has only n distinct states, some state is repeated (at least) twice in $q_0, q_1, \dots, q_n$. That is, $q_i = q_j$ for some i, j such that $0 \leq i < j \leq n$. Let $u = a_1 \dots a_i$ ($u = \epsilon$, if $i = 0$), $v = a_{i+1} \dots a_j$, and $w = a_{j+1} \dots a_\ell$ ($w = \epsilon$, if $j = \ell$). By definition, $x = uvw$. Furthermore, $v \neq \epsilon$, since $i < j$; and $|uv| \leq n$, since $j \leq n$. It remains to prove that $uv^kw \in L$, for all $k \in \mathbb{N}$.

We have $\delta^*(q_i, v) = q_j = q_i$. That is, v takes M from state q_i back to state q_i. Hence, any number of repetitions of v (including zero) will have the same effect. In other words, $\delta^*(q_i, v^k) = q_i = q_j$, for any $k \in \mathbb{N}$. In addition, from the definition of u and w we have that $\delta^*(q_0, u) = q_i$, and $\delta^*(q_j, w) = q_\ell$. Hence, $\delta^*(q_0, uv^kw) = q_\ell$, for all $k \geq 0$. Since $x \in L$ and $\delta^*(q_0, x) = q_\ell$, it follows that $q_\ell \in F$. Therefore $uv^kw \in L$, for all $k \in \mathbb{N}$, as wanted. □

Example 7.23 In this example we illustrate an application of the Pumping Lemma. Consider the language $L = \{x \in \{0,1\}^* : x \text{ has as many 0s as 1s}\}$. We want to show that L is not regular. Assume, for contradiction, that L is regular. Let $n \in \mathbb{N}$ be the natural number which the Pumping Lemma asserts that exists if L is regular.

Let x be any string in L that starts with n 0s. (For example, $x = 0^n1^n$ is such a string, among many others.) By the Pumping Lemma, there are strings $u, v, w \in \Sigma^*$ such that $x = uvw$, $v \neq \epsilon$ and $|uv| \leq n$, so that $uv^kw \in L$, for all $k \geq 0$. Since x begins with n 0s and $|uv| \leq n$, it follows that $v = 0^m$, for some $1 \leq m \leq n$ ($m \neq 0$, since $v \neq \epsilon$).

[9] Recall that, in the statement of the theorem, we say that n depends on L. This is clear now, since n is defined as the number of states of some DFSA that accepts L.

But if x has equally many 0s and 1s, then for all $k \neq 1$, uv^kw has different number of 0s and 1s: fewer 0s than 1s if $k = 0$, and more 0s than 1s if $k > 1$. This means that $uv^kw \notin L$ for all $k \neq 1$, contrary to the Pumping Lemma. Therefore L cannot be regular.

End of Example 7.23

Exercises

1. Give an example of a language L such that $L^{\circledast} = L$. Can a language with this property be finite? Justify your answer.

2. The exponentiation operation on languages is defined (inductively) as follows. For any language L and $k \in \mathbb{N}$,

$$L^k = \begin{cases} \{\epsilon\}, & \text{if } k = 0 \\ L^{k-1} \circ L, & \text{if } k > 0 \end{cases}$$

(a) Prove that $L^{\circledast} = \cup_{k \in \mathbb{N}} L^k$. (This provides an alternative definition of the Kleene star operation.)

(b) Prove that a language $L \neq \emptyset$ has the property that $L^{\circledast} = L$ if and only if $L = L \circ L$.

3. State whether the string 010 belongs to the language denoted by each of the following regular expressions. Briefly justify your answer.

(a) $(\epsilon + 1)0^*1^*0$

(b) $0(11)^*0$

(c) $(\epsilon + 0)(1^*1^*)^*(0 + 1)$

(d) $(0 + 1)(0 + 1)^*$

(e) $(0 + 1)(0^* + 1^*)$

4. Let R, S and T be arbitrary regular expressions. For each of the following assertions, state whether it is true or false, and justify your answer.

(a) If $RS \equiv SR$ then $R \equiv S$.

(b) If $RS \equiv RT$ and $R \not\equiv \emptyset$ then $S \equiv T$.

(c) $(RS + R)^*R \equiv R(SR + R)^*$.

5. Consider the language L_2 (over $\{0, 1\}$) consisting of the strings that have an even number of 0s. Does the regular expression $1^*((01^*0)1^*)^*$ denote this set? How about each of the following expressions: $(1 + (01^*0))^*$, $(1^*(01^*0))^*$, and $(1^*(01^*0)^*)^*$? In each case carefully justify your answer.

6. A regular expression is called $\emptyset$-*free* if it does not contain the regular expression $\emptyset$ as a subexpression. More precisely, the set of $\emptyset$-free regular expressions is defined inductively just as the set of regular expressions (see Definition 7.6, page 192 in the notes) except that in the basis of the definition we omit the expression $\emptyset$. Prove that any regular expression is equivalent either to an $\emptyset$-free regular expression, or to the regular expression $\emptyset$.

(Hint: Use structural induction on regular expressions.)

7. Prove that for any regular expression S, there is a regular expression $\hat{S}$ such that $\mathcal{L}(\hat{S}) = \mathbf{Rev}(\mathcal{L}(S))$. In other words, if a language can be denoted by a regular expression, then so can its reversal. (You should prove this in two ways: One that does not involve FSA, and one that does.)

8. Prove that the FSA $M_\cap$ obtained via the Cartesian product construction from FSA M_1 and M_2 (see page 224) accepts the intersection of the languages accepted by M_1 and M_2.

Does this construction work regardless of whether M_1 and M_2 are deterministic or nondeterministic? If M_1 and M_2 are deterministic, is $M_\cap$ necessarily deterministic? Justify your answers.

9. Prove that for every language L, if L is accepted by a NFSA then it is accepted by a NFSA that has exactly one accepting state. Does the same result hold for *deterministic* FSA? Justify your answer.

10. Consider the following languages over alphabet $\Sigma = \{0, 1\}$:

$$\{x \in \{0,1\}^* : x \neq \epsilon \text{ and the first and last symbols of } x \text{ are different}\}$$
$$\{0^n 1^m : n, m \geq 0 \text{ and } n + m \text{ is odd}\}$$
$$\{x \in \{0,1\}^* : \text{every 0 in } x \text{ is immediately preceded and followed by 1}\}$$

For each of these languages, construct a DFSA that accepts it, and a regular expression that denotes it. Prove that your automata and regular expressions are correct.

11. Let L be a language over Σ that is accepted by some FSA. Prove that each of the following languages is also accepted by some FSA.

(a) The set of prefixes of L, $\text{Prefix}(L) = \{x \in \Sigma^* : xy \in L, \text{for some } y \in \Sigma^*\}$.

(b) The set of suffixes of L, $\text{Suffix}(L) = \{x \in \Sigma^* : yx \in L, \text{for some } y \in \Sigma^*\}$.

(c) The set of maximal strings of L, $\text{Max}(L) = \{x \in L : \text{for any } y \in \Sigma^*, \text{if } y \neq \epsilon \text{ then } xy \notin L\}$.

12. Let $L_4 = \{x \in \{0,1\}^* : x = y1z, \text{for some } y, z \in \{0,1\}^* \text{ s.t. } |z| = 3\}$. That is, L_4 consists of all strings with at least 4 symbols, where the 4th symbol from the end is 1.

(a) Give a regular expression for L_4.

(b) Give a nondeterministic FSA that accepts L_4.

(c) Apply the subset construction on the NFSA from part (b) to get a deterministic FSA that accepts L_4.

(d) Prove that *every* deterministic FSA that accepts L_4 must have at least 16 states.

(Hint: Prove that in any deterministic FSA that accepts L_4, distinct input strings of length 4 must lead (from the initial state) to *distinct* states.)

(e) Generalise from this example to show that, for every positive integer n, there is a language L_n that is accepted by some NFSA with $n + 1$ states, but is not accepted by any DFSA with fewer than 2^n states.

13. Consider the ***converses*** of the closure properties of regular languages discussed in Section 7.5. Since the complement of a set's complement is the set itself, it is clear that the converse of the closure under complementation holds. In other words, if $\overline{L}$ is regular then so is L. But how about the converse of the other closure properties?

(a) For every language L, is it the case that if $L^{\circledast}$ is regular then L is also regular? Justify your answer.

(b) For every languages L and L' is it the case that if $L \cup L'$ is regular then L and L' are also regular? Justify your answer. If the answer is negative, is it the case that if $L \cup L'$ is regular then *at least one* of L and L' is regular?

(c) For every languages L and L' is it the case that if $L \circ L'$ is regular then L and L' are also regular? Justify your answer. If the answer is negative, is it the case that if $L \circ L'$ is regular then *at least one* of L and L' is regular?

Chapter 8

CONTEXT-FREE GRAMMARS AND PUSHDOWN AUTOMATA

8.1 Introduction

In this chapter we turn our attention to a class of languages, called ***context-free languages*** (***CFL*** for short). All modern programming languages belong to this class. (Note that programming languages are not regular as they contain nonregular constructs, such as arbitrarily nested but properly balanced parentheses.) Consequently, CFL are central in the design of programming languages and compilers. Here we take a very brief glimpse of this important subject. We will introduce two different characterisations of these languages. The first characterisation is based on the formalism of ***context-free grammars***, and the second characterisation is based on ***push-down automata***, an extension of finite-state automata. We will then prove that these two characterisations are equivalent — i.e., they capture the same class of languages.

8.2 Context-free grammars

8.2.1 Informal description

As speakers of a "natural" language we act in two capacities:

- We *recognise* syntactically correct utterances. In this capacity we act analogously to finite state automata, which "read" input strings and classify them as "good" or "bad".

- We *generate* syntactically correct utterances. In this capacity we act analogously to regular expressions, which can be viewed as rules for generating "good" strings.

For example, the regular expression $01(00+11)^*10$ can be viewed as the rule to generate strings that begin with 01, continue with the concatenation of any number of 00s or 11s, and end with 10.

There is a more general mechanism for describing how to generate strings using a formalism known as a context-free grammar (CFG for short). Shown below is a CFG that generates the language denoted by the above regular expression.

$$\begin{aligned}
&S \to BME\\
&B \to 01\\
&E \to 10\\
&M \to 00M\\
&M \to 11M\\
&M \to \epsilon
\end{aligned}$$

The CFG is specified as a list of so-called ***productions***. Each production is shown as a string that contains the symbol "→". To the left of "→" is a single symbol — S, B, M or E in our case; these symbols are called ***variables***. To the right of "→" is a string consisting of variables and ***terminals***, the symbols of the strings generated by the grammar — 0 and 1 in our case. In our example, the right-hand side of the first production consists only of variables, that of the second and third production consists only of terminals, and that of the fourth and fifth production consists of both variables and terminals. The right-hand side of the last production is the empty string. Finally, there is a special ***start*** variable, usually denoted by S.

The intuitive interpretation of the above grammar is as follows. The first production says that any string in the language generated by the grammar consists of three parts: a "beginning", represented by the symbol B, a "middle", represented by the symbol M, and an "end", represented by the symbol E. The second production says that the beginning of the string is 01. The third production says that the end of the string is 10. The last three productions say that the middle part of the string is one of three things: the string 00 followed by anything that could also be in the middle part; the string 11 followed by anything that could also be in the middle part; or the empty string.

This grammar generates strings according to the following process, called a ***derivation***.

(a) We start by writing down the start symbol of the grammar, S.

(b) At each step of the derivation, we may replace any *variable* symbol of the string generated so far by the right-hand side of any production that has that symbol on the left.

(c) The process ends when it is impossible to apply a step of the type described in (b).

If the string at the end of this process consists entirely of terminals, it is a string in the language generated by the grammar.

For example, below we show the derivation of the string 0100111110 from our grammar.

$$\begin{aligned}
S &\Rightarrow BME && [\text{use } S \to BME] \\
&\Rightarrow B00ME && [\text{use } M \to 00M] \\
&\Rightarrow B0011ME && [\text{use } M \to 11M] \\
&\Rightarrow B001111ME && [\text{use } M \to 11M] \\
&\Rightarrow B001111E && [\text{use } M \to \epsilon] \\
&\Rightarrow B00111110 && [\text{use } E \to 10] \\
&\Rightarrow 0100111110 && [\text{use } B \to 01]
\end{aligned}$$

The symbol $\Rightarrow$ means that the string after the symbol can be derived from the string before it by applying a single production of the grammar. The comments in square brackets indicate the production used in each step.

8.2.2 Formal definitions

With the preceding example in mind we now formally define context-free grammars and derivations. A ***context-free grammar*** (or ***CFG***, for short) is a tuple $G = (V, \Sigma, P, S)$, where V is a set of ***variables***, Σ is a set of ***terminals***, P is a set of ***productions***, and S is a particular element of V, called the ***start symbol***. We require that the set of variables and set of terminals be disjoint, i.e., $V \cap \Sigma = \emptyset$. Each production in P has the form $A \to \alpha$, where $A \in V$ and $\alpha \in (V \cup \Sigma)^*$.

To formally define derivations in grammar G, we first define the relation $\Rightarrow_G$ between strings in $(V \cup \Sigma)^*$: For any $\alpha, \beta \in (V \cup \Sigma)^*$, $\alpha \Rightarrow_G \beta$ if and only if $\alpha = \alpha_1 A \alpha_2$, $\beta = \alpha_1 \gamma \alpha_2$, and $A \to \gamma$ is a production in P, where $\alpha_1, \alpha_2 \in (V \cup \Sigma)^*$. Intuitively, β is obtained from α by a single application of the production rule $A \to \gamma$.

Let $\alpha, \beta \in (V \cup \Sigma)^*$. A ***derivation of*** β ***from*** α in grammar G is a sequence $\alpha_1, \alpha_2, \ldots, \alpha_k$ of strings in $(V \cup \Sigma)^*$, such that $\alpha_1 = \alpha$, $\alpha_k = \beta$, and for each i such that $1 \leq i < k$, $\alpha_i \Rightarrow_G \alpha_{i+1}$. The ***number of steps*** in this derivation is defined to be $k - 1$. If there is a derivation of β from α, we write $\alpha \Rightarrow^* \beta$, read "$\alpha$ yields β", When the grammar G is clear from the context, we may drop the subscript from $\Rightarrow_G$ and $\Rightarrow_G^*$.

Given a CFG $G = (V, \Sigma, P, S)$, we define the ***language generated by*** G, denoted $\mathcal{L}(G)$, as the set of all strings of *terminals* that can be derived from the start symbol of G. That is,

$$\mathcal{L}(G) = \{x \in \Sigma^* : S \Rightarrow_G^* x\}.$$

A language L is a ***context-free language*** (abbreviated ***CFL***) if it is the language generated by some context-free grammar.

8.2.3 Examples of CFG

We usually describe a CFG simply by listing its productions. Unless otherwise stated, the set of variables V is assumed to be the set of symbols on the left-hand sides of productions; the set of terminals Σ is assumed to be the set of symbols that appear on the right-hand side of some production but not on the left-hand side of any production; and the start symbol S is assumed to be the left-hand side of the first production.

Example 8.1 Following is a CFG that generates the language $\{0^n 1^n : n \in \mathbb{N}\}$. Recall that this language is not regular.

$$\begin{aligned} &S \to 0S1 \\ &S \to \epsilon \end{aligned}$$

We leave a detailed proof that this grammar generates $\{0^n 1^n : n \in \mathbb{N}\}$ as an exercise. We note that such a proof must show that (a) for each $n \in \mathbb{N}$, $S \Rightarrow^* 0^n 1^n$; and, conversely, (b) for each $x \in \{0,1\}^*$, if $S \Rightarrow^* x$ then $x = 0^n 1^n$ for some $n \in \mathbb{N}$. **End of Example 8.1**

Example 8.2 Let $L_b \subseteq \{(,)\}^*$ be the set of balanced strings of parentheses. Formally, this is defined as the set of strings of parentheses x so that (a) x has the same number of left and right parentheses; and (b) for every prefix y of x, y has at least as many left parentheses as right parentheses. (It is easy to show that this language is not regular.) Following is a CFG that generates L_b.

$$\begin{aligned} &S \to SS \\ &S \to (S) \\ &S \to \epsilon \end{aligned}$$

We leave it as an exercise to prove that this grammar generates L_b. **End of Example 8.2**

Example 8.3 Let $L_e = \{x \in \{0,1\}^* : x \text{ has the same number of 0s and 1s}\}$. Following is a CFG that generates this language

$$\begin{array}{lll} S \to 0B & A \to 0S & B \to 1S \\ S \to 1A & A \to 1AA & B \to 0BB \\ S \to \epsilon & & \end{array}$$

Proving that this grammar generates L_e is interesting and considerably more complex than the proofs of correctness for the previous two examples, so we give the more delicate parts of the proof in some detail. The intuition for this grammar is that strings in $\{0,1\}^*$ derived from S have the same number of 0s and 1s, those derived from A have one more 0 than 1, and those derived from B have one more 1 than 0.

The set of variables of this grammar is $V = \{S, A, B\}$, and the set of terminals is $\Sigma = \{0, 1\}$. For any $\alpha \in (V \cup \Sigma)^*$, define

$$\begin{aligned} \mathbf{zero}(\alpha) &= \text{number of occurrences of } A \text{ and } 0 \text{ in } \alpha \\ \mathbf{one}(\alpha) &= \text{number of occurrences of } B \text{ and } 1 \text{ in } \alpha \end{aligned}$$

Using simple induction on n it is easy to prove that for any $\alpha \in (V \cup \Sigma)^*$, if $S \Rightarrow^* \alpha$ is an n-step derivation, then $\mathbf{zero}(\alpha) = \mathbf{one}(\alpha)$. From this it follows that for all $x \in \{0, 1\}^*$, if $S \Rightarrow^* x$ then x has equally many 0s and 1s, i.e. $x \in L_e$.

We must now prove the converse, i.e., that for all $x \in \{0, 1\}^*$, if $x \in L_e$ then $S \Rightarrow^* x$. To do so we will need to prove something stronger. Consider the predicate

$$\begin{aligned} P(n): \qquad & \text{if } x \in \{0,1\}^* \text{ and } |x| = n \text{ then} \\ & \text{(a) if } \mathbf{zero}(x) = \mathbf{one}(x) \text{ then } S \Rightarrow^* x \\ & \text{(b) if } \mathbf{zero}(x) = \mathbf{one}(x) + 1 \text{ then } A \Rightarrow^* x \\ & \text{(c) if } \mathbf{zero}(x) + 1 = \mathbf{one}(x) \text{ then } B \Rightarrow^* x \end{aligned}$$

We use complete induction to prove that $P(n)$ holds for every $n \in \mathbb{N}$. (Note that part (a) immediately implies that for all $x \in \{0, 1\}^*$, if $x \in L_e$ then $S \Rightarrow^* x$. Parts (b) and (c) are needed for the induction.)

Let k be an arbitrary natural number. Assume that, for all j such that $0 \leq j < k$, $P(j)$ holds. We must prove that $P(k)$ also holds.

Case 1. $k = 0$. Then $x = \epsilon$. Part (a) is true because $S \Rightarrow^* \epsilon$ in one step using the production $S \to \epsilon$. Parts (b) and (c) are trivially true because $\mathbf{zero}(\epsilon) = \mathbf{one}(\epsilon) = 0$. Thus $P(0)$ is true.

Case 2. $k > 0$. Thus, $x = 0y$ or $x = 1y$ for some $y \in \{0, 1\}^*$ such that $0 \leq |y| < k$. There are three cases to consider, depending on the difference between the number of 0s and 1s in x; for each case we will consider two subcases, depending on whether x starts with 0 or 1.

(a) $\mathbf{zero}(x) = \mathbf{one}(x)$. Assume that x starts with 0, i.e., $x = 0y$ for some $y \in \{0, 1\}^*$. (The subcase where $x = 1y$ is similar.) Then $\mathbf{zero}(y) + 1 = \mathbf{one}(y)$ and by the induction hypothesis $B \Rightarrow^* y$. Thus, $S \Rightarrow 0B \Rightarrow^* 0y = x$, so $S \Rightarrow^* x$, as wanted.

(b) $\mathbf{zero}(x) = \mathbf{one}(x) + 1$. First consider the subcase in which x starts with 0, i.e., $x = 0y$ for some $y \in \{0, 1\}^*$. Then $\mathbf{zero}(y) = \mathbf{one}(y)$ and by the induction hypothesis $S \Rightarrow^* y$. Thus, $A \Rightarrow 0S \Rightarrow^* 0y = x$, i.e., $A \Rightarrow^* x$, as wanted.

Next consider the case in which x starts with 1, i.e., $x = 1y$ for some $y \in \{0, 1\}^*$. Then $\mathbf{zero}(y) = \mathbf{one}(y) + 2$. Thus, there are strings $y_1, y_2 \in \{0, 1\}^*$ so that $y = y_1y_2$, $\mathbf{zero}(y_1) = \mathbf{one}(y_1) + 1$ and $\mathbf{zero}(y_2) = \mathbf{one}(y_2) + 1$. (To see this, consider every prefix y' of y, starting with $y' = \epsilon$, all the way until $y' = y$. For each such prefix y', consider the difference $\mathbf{zero}(y') - \mathbf{one}(y')$. Note that this can be negative. However, initially it is 0 and at the end it is 2. Thus, for some prefix y_1 of y, $\mathbf{zero}(y_1) - \mathbf{one}(y_1) = 1$. Now let y_2 be the suffix of y after y_1. Since $\mathbf{zero}(y) - \mathbf{one}(y) = 2$ and $\mathbf{zero}(y_1) - \mathbf{one}(y_1) = 1$, it follows that $\mathbf{zero}(y_2) - \mathbf{one}(y_2) = 1$. These are the strings y_1 and y_2 with the desired properties.) By the induction hypothesis, $A \Rightarrow^* y_1$ and $A \Rightarrow^* y_2$. So, $A \Rightarrow 1AA \Rightarrow^* 1y_1y_2 = 1y = x$. That is, $A \Rightarrow^* x$, as wanted.

(c) $\mathbf{zero}(x) + 1 = \mathbf{one}(x)$. This is similar to case (b).

End of Example 8.3

8.3 CFG are more powerful than regular expressions

As the examples in the previous section show, there are context-free grammars that generate non-regular languages. It is not hard to prove that every regular language can be generated by a CFG. Perhaps the simplest proof of this fact is to show that for any regular expression R there is a CFG that generates the language denoted by R. The proof is by structural induction on the regular expression: We show (a) how to generate the languages denoted by the simplest regular expressions, and (b) how to generate the language denoted by a "complex" regular expression R, given grammars that generate the regular expressions out of which R is constructed. The details are given below.

Theorem 8.1 *For every regular expression R there is a CFG G such that $\mathcal{L}(G) = \mathcal{L}(R)$.*

PROOF. By structural induction on the definition of R.

BASIS: The language denoted by the regular expression $\emptyset$ is generated by a CFG with no productions. The language denoted by the regular expression ϵ is generated by the CFG with the single production $S \to \epsilon$. The language denoted by the regular expression a, for any $a \in \Sigma$, is generated by the CFG with the single production $S \to a$.

INDUCTION STEP: Let R_1 and R_2 be regular expressions. Assume that there CFG G_1 and G_2 that generate $\mathcal{L}(R_1)$ and $\mathcal{L}(R_2)$ respectively. For every way of constructing a regular expression R out of R_1 (and possibly R_2), we show a grammar G that generates the language denoted by R. Without loss of generality, we can assume that the variables of G_1 and G_2 are disjoint (we can always rename the variables of a grammar to ensure this). Let S_1 and S_2 be the start variables of G_1 and G_2, and S be a new variable (i.e., not one of the variables of G_1 and G_2). S will be the start symbol of the grammar that generates $\mathcal{L}(R)$.

There are three cases, depending on how R is constructed from its subexpressions.

CASE 1. $R = (R_1 + R_2)$. Let G be the grammar whose productions are the productions of G_1 and G_2, and the following two additional productions: $S \to S_1$, $S \to S_2$. It is obvious that $\mathcal{L}(G) = \mathcal{L}(G_1) \cup \mathcal{L}(G_2)$, and so $\mathcal{L}(G) = \mathcal{L}(R)$.

CASE 2. $R = (R_1R_2)$. Let G be the grammar whose productions are the productions of G_1 and G_2, and the following additional production: $S \to S_1S_2$. It is obvious that $\mathcal{L}(G) = \mathcal{L}(G_1) \circ \mathcal{L}(G_2)$, and so $\mathcal{L}(G) = \mathcal{L}(R)$.

Case 3. $R = R_1^*$. Let G be the grammar whose productions are the productions of G_1, and the following two additional productions: $S \to S_1 S$, $S \to \epsilon$. It is obvious that $\mathcal{L}(G) = \mathcal{L}(G_1)^{\circledast}$, and so $\mathcal{L}(G) = \mathcal{L}(R)$. □

An immediate consequence of this theorem and the fact that there are nonregular languages that can be generated by CFG is the following:

Corollary 8.2 *The class of context-free languages is a proper superset of the class of regular languages.*

8.4 Right-linear grammars and regular languages

In fact, there is a restricted kind of context-free grammars, called right-linear grammars, which generate *exactly* the class of regular languages.

A CFG $G = (V, \Sigma, P, S)$ is ***right-linear*** if every production in P is of the form $A \to \epsilon$ or $A \to xB$, where $A, B \in V$ and $x \in \Sigma^*$. That is, the right-hand side of every production is either the empty string, or a (possibly empty) string of terminals followed by a *single* variable. We can further restrict the second type of productions by requiring that $|x| \leq 1$ — i.e., the single variable on the right-hand side is preceded by either the empty string or a single terminal. Such grammars are called ***strict right-linear***.

It is easy to see that strict right-linear grammars generate exactly the class of regular languages. This is because strict right-linear grammars correspond in a natural way to FSA: We can view the variables of the grammar as the states of the automaton, and a production of the form $A \to aB$, where $a \in \Sigma \cup \{\epsilon\}$, as the transition from state A to state B on input a. The start state of the automaton is the start variable of the grammar, and its accepting states are all the variables A for which the grammar has productions of the form $A \to \epsilon$. Conversely, we can view a DFSA as a strict right-linear grammar in a similar way. It is not hard to see that an accepting computation of the FSA on input x corresponds to the derivation of x in the grammar, and vice-versa. Thus, any regular language is generated by a strict right-linear grammar; and, conversely, the language generated by any strict right-linear grammar is regular.

More formally, given a strict right-linear grammar $G = (V, \Sigma, P, S)$, we construct the following NFSA $M = (Q, \Sigma, \delta, s, F)$:

- $Q = V$.
- For every $A, B \in V$ and $a \in \Sigma \cup \{\epsilon\}$, $\delta(A, a) = \{B : A \to aB$ is a production in $P\}$.
- $s = S$.
- $F = \{A : A \to \epsilon$ is a production in $P\}$.

We can now prove that, for every $x \in \Sigma^*$, $B \in \delta^*(A, x)$ if and only if $A \Rightarrow^* x$. The forward direction can be shown by a simple induction on the length of the computation path that takes M from A to B on input x, and the backward direction can be shown by simple induction on the length of the derivation of x from A. (These proofs are left as exercises.) From this, it follows immediately that:

Theorem 8.3 *The languages generated by strict right-linear grammars are regular.*

Conversely, given a DFSA $M = (Q, \Sigma, \delta, s, F)$ we define a strict right-linear grammar $G = (V, \Sigma, P, S)$ as follows:

- $V = Q$.
- $P = \{A \to aB : \ A, B \in V, a \in \Sigma \text{ and } \delta(A, a) = B\} \cup \{A \to \epsilon : A \in F\}$,
- $S = s$.

Again, it is easy to prove (by induction as indicated above) that $B = \delta^*(A, x)$ if and only if $A \Rightarrow^* x$. From this it follows that:

Theorem 8.4 *Every regular language can be generated by a strict right-linear grammar.*

Theorems 8.3 and 8.4 imply

Corollary 8.5 *The class of languages generated by strict right-linear languages is the class of regular languages.*

It is also easy to see that general right-linear grammars are equivalent in power to strict right-linear ones. One direction of this equivalence is trivial (since a strict right-linear grammar is a special case of right-linear grammar). For the other direction it suffices to show that for every right-linear grammar there is a strict right-linear grammar that generates the same language. This is done by replacing every production of a right-linear grammar that does not conform to the requirements of strict right-linear grammars by a finite set of productions that do, in a way that does not change the language generated by the grammar. Specifically, a production of the form

$$A \to a_1 a_2 \dots a_k B$$

where $k \geq 2$ and $a_i \in \Sigma$ for each i, $1 \leq i \leq k$, is replaced by

$$\begin{aligned}
&A \to a_1 B_1 \\
&B_1 \to a_2 B_2 \\
&\vdots \\
&B_{k-2} \to a_{k-1} B_{k-1} \\
&B_{k-1} \to a_k B
\end{aligned}$$

where $B_1, \dots, B_{k-1}$ are new variables. Therefore, Corollary 8.5 can also be stated as:

Corollary 8.6 *The class of languages generated by right-linear languages is the class of regular languages.*

This gives us yet another characterisation of regular languages: They are the languages denoted by regular expressions; or, equivalently, the languages accepted by (deterministic of nondeterministic) FSA; or, equivalently, the languages generated by right-linear grammars.

8.5 Pushdown automata

8.5.1 Motivation and informal overview

In Chapter 7 we saw two equivalent chararisations of regular languages: One, taking the point of view of a language generator, defines as regular those languages that can be described by regular expressions. The other, taking the point of view of a language acceptor, defines as regular those languages that can be accepted by FSA.

In Section 8.2 we defined CFL, the class of languages that can be generated by CFG. The question then arises whether there is a natural mathematical model of automata that accept the class of context-free languages. The answer is affirmative, and the automata in question are called ***pushdown automata***, abbreviated ***PDA***.

As we saw in our discussion of the pumping lemma for regular languages, the main limitation of FSA arises from the fact that all the information a FSA can "remember" about its input string must be encapsulated in its state. Since the FSA has finitely many states, it can only "remember" a bounded amount of information about its input, no matter how long that input is. One way to circumvent this limitation is to endow the automaton with auxiliary storage, into which it can record some information about the input it reads. The automaton can analyse the information it has recorded on the auxiliary storage to determine whether to accept or reject the input. A PDA is a FSA augmented with such an auxiliary storage which, however, can be accessed in a restricted fashion. Specifically, the auxiliary storage is a *stack*. The automaton can only *push* a symbol on top of the stack, or it can *pop* (remove) the symbol presently at the top of the stack. So, the automaton can remember an unbounded amount of information about the input (since the size of the stack is unlimited), but it can access that information in a restricted way.

On the basis of this informal (and still quite vague) description of a PDA, one can imagine how such an automaton could recognise the nonregular language $\{0^n1^n : n \in \mathbb{N}\}$: We start reading the input, and as long as we read 0s we push onto the stack a corresponding number of instances of some symbol, say X. Upon reading the first 1 in the input string the automaton switches to popping Xs from the stack — one for every 1 encountered. If the stack is empty when the entire input string has been read then the string was of the form 0^n1^n and it is accepted; otherwise, it is not of that form and it is rejected.

8.5.2 Formal definitions

Formally, a ***pushdown automaton (PDA)*** is a tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$, where

- Q is the finite nonempty set of ***states***;
- Σ is the finite set of ***input symbols***;
- Γ is the finite set of ***stack symbols***;
- $q_0 \in Q$ is the ***start*** state;
- $F \subseteq Q$ is the set of ***accepting*** states; and

- δ is the ***transition function***, and is explained below.

The transition function's signature is:

$$\delta : Q \times (\Sigma \cup \{\epsilon\}) \times (\Gamma \cup \{\epsilon\}) \to \mathcal{P}\big(Q \times (\Gamma \cup \{\epsilon\})\big)$$

In other words, δ maps a state, an input symbol (or the empty string), and a stack symbol (or the empty string) to a *set* of pairs, each consisting of a state and a stack symbol (or the empty string). The interpretation of the transition function is as follows: Suppose that $(q, Y) \in \delta(p, a, X)$. This means that when in state p, upon reading a from the input string, and with X at the top of the stack, the automaton can enter state q, replacing X with Y. If $a = \epsilon$ this transition takes place without the automaton actually reading anything from the input string. There are four possibilities regarding the symbol X read and the symbol Y written at the top of the stack:

- $X = \epsilon$ and $Y \in \Gamma$. In this type of transition, the automaton pushes Y onto the stack;

- $X \in \Gamma$ and $Y = \epsilon$. In this type of transition, the automaton pops X from the stack;

- $X, Y \in \Gamma$. In this type of transition, the automaton replaces X by Y at the top of the stack.

- $X = Y = \epsilon$. In this type of transition, the automaton does not consult the stack at all: it moves from its present state to the next just on the basis of the input symbol a (or spontaneously if $a = \epsilon$).

We will now define computations of a PDA M. First we introduce the notion of configuration of M, which is a complete description of the state of the computation of M at some instant in time. It comprises the present state of the automaton, the portion of the input that the automaton has not yet read, and the contents of the stack. Formally, a ***configuration*** of M is a triple (p, x, α) consisting of a state $p \in Q$ (the present state of the automaton), a string $x \in \Sigma^*$ (the portion of the input not yet read), and a string $\alpha \in \Gamma^*$ (the present contents of the stack). In writing the content of the stack as a string $X_1 X_2 \ldots X_k \in \Gamma^*$, X_1 is the symbol at the top of the stack, X_2 is the symbol below X_1 and so on; X_k is the symbol at the bottom of the stack. By convention, we use lower-case letters near the start of the Latin alphabet, such as a, b, c, to denote individual input symbols in Σ (or, sometimes, ϵ); lower-case letters near the end of the Latin alphabet, such as w, x, y, z, to denote input strings in Σ^*; upper-case letters near the end of the Latin alphabet, such as X, Y, Z, to denote individual stack symbols in Γ (or, sometimes, ϵ); and lower-case letters near the start of the Greek alphabet, such as α, β, γ to denote stack strings in Γ^*.

A move of M describes how the configuration of M changes as a result of a single state transition of M. Suppose that the present configuration of M is $(p, ax, X\alpha)$. This means that M is in state p, the unread portion of its input is ax (where a is the leftmost unread symbol of the input or ϵ), and the stack contains $X\alpha$ (where X is the symbol at the top of the stack or ϵ). Then, if $\delta(p, a, X) = (q, Y)$, in one move M can read the leftmost input symbol a, replace the symbol X at the top of the stack by Y, and change its state to q. As a result of this move the configuration becomes $(q, x, Y\alpha)$. More formally, we say that M ***moves in one step from configuration*** C ***to configuration*** C', written $C \vdash_M C'$, if and only if $C = (p, ax, X\alpha)$, $C' = (q, x, Y\alpha)$ and $(q, Y) \in \delta(p, a, X)$, for some $p, q \in Q$, $a \in \Sigma \cup \{\epsilon\}$, $x \in \Sigma^*$, $X, Y \in \Gamma \cup \{\epsilon\}$ and $\alpha \in \Gamma^*$.

Let C and C' be configurations of M. A ***computation*** of M from C to C' is a sequence of configurations $C_1, \ldots, C_k$, where k is a positive integer, so that $C_1 = C$, $C_k = C'$, and for all i such that $1 \leq i < k$, $C_i \vdash_M C_{i+1}$. The ***number of steps*** in this computation is defined to be $k - 1$. If there is a computation of M from C to C', we write $C \vdash_M^* C'$. Note that for every configuration C of M, $C \vdash_M^* C$, through a computation of zero steps. When the automaton M is clear from the context we may drop the subscript M from $\vdash_M$ and $\vdash_M^*$.

We now have all the equipment we need to define formally the set of strings accepted by the PDA M. Informally, M starts in an initial configuration, with M in its start state s, the entire input string x being unread, and the stack being empty. We will say that x is accepted by M if there is a computation at the end of which M is in a configuration where M is left in an accepting state, all of x has been read, and the stack is left empty. More precisely, the PDA $M = (Q, \Sigma, \Gamma, \delta, s, F)$ ***accepts*** $x \in \Sigma^*$ if and only if there is some $q \in F$ so that $(s, x, \epsilon) \vdash_M^* (q, \epsilon, \epsilon)$. The language accepted by M, denoted $\mathcal{L}(M)$, is the set of all strings accepted by M:

$$\mathcal{L}(M) = \{x \in \Sigma^* : M \text{ accepts } x\}.$$

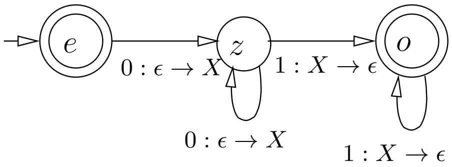
Note that our definition of PDA allows for non-deterministic behaviour: In general, in a single step the automaton may move from its present configuration to one of a number of possible configurations. Thus, from a given initial configuration with input string x there may be several different computations leading to configurations in which the entire input has been read. If there is at least one computation leading to such a configuration where the automaton is in an accepting state and the stack is empty, then x is accepted according to our definition. This is similar to our definition of acceptance by nondeterministic FSA.

It is possible to define a deterministic version of PDA by suitably restricting the transition function so as to ensure that from each configuration the automaton can move in one step to at most one configuration. We note in passing that in the case of PDA, this restriction limits the class of languages that are accepted. That is, there exist languages that can be accepted by nondeterministic PDA which, however, cannot be accepted by any deterministic PDA. This is in sharp contrast to the corresponding situation with FSA: as we have seen (via the subset construction) the deterministic and nondeterministic versions of FSA accept the same class of languages.

8.5.3 Examples of PDA

We now demonstrate some concrete PDA. Similar to FSA, PDA can be conveniently described diagrammatically. The states of the PDA are the nodes of a directed graph, with the initial state and accepting states designated exactly as in the case of FSA. A directed edge from node q to node q' labeled with "$a : X \to Y$", where $a \in \Sigma \cup \{\epsilon\}$ and $X, Y \in \Gamma \cup \{\epsilon\}$, indicates that the PDA can move from state q to state q' if it reads a from its input string and replaces the X at the top of the stack by Y — in other words, it indicates that in the transition function δ of the PDA, $(q', Y) \in \delta(q, a, X)$.

Example 8.4 The diagram of a PDA that accepts $\{0^n1^n : n \in \mathbb{N}\}$ is shown below.



Formally, this is the PDA $M = (Q, \Sigma, \Gamma, \delta, s, F)$ where

- $Q = \{e, z, o\}$
- $\Sigma = \{0, 1\}$
- $\Gamma = \{X\}$
- $s = e$
- $F = \{e, o\}$

and the transition function δ is defined as follows. For all $q \in Q$, $a \in \Sigma \cup \{\epsilon\}$ and $Y \in \Gamma \cup \{\epsilon\}$,

$$\delta(q, a, Y) = \begin{cases} \{(z, X)\}, & \text{if } q \in \{e, z\},\ a = 0 \text{ and } Y = \epsilon \\ \{(o, \epsilon)\}, & \text{if } q \in \{z, o\},\ a = 1 \text{ and } Y = X \\ \emptyset, & \text{otherwise} \end{cases}$$

Intuitively, this PDA works as follows. It starts in state e. If the input string is ϵ, it stays in that state and accepts (since the stack is empty). If the input string starts with 1, the automaton rejects: there is no state to which it can go on input 1 from state e. If the input string starts with 0, the automaton enters state z, indicating that it is now processing the prefix of consecutive 0s, and pushes X onto the stack. As long as the automaton reads 0s, it remains in state z and pushes one X on the stack for each 0 it reads. When the automaton first sees a 1, it moves to state o, indicating that it is now processing the suffix of consecutive 1s, and pops X from the top of the stack. As long as the automaton reads 1s, it remains in that state and pops an X from the stack for each 1 it reads. If the automaton reads 0 while in state o, it rejects: there is no state to which it can go on input 0 from state o. The automaton accepts a nonempty input string x if, when it is done reading x, it is in state o and the stack is empty. This indicates that the input string consisted of a some number of 0s followed by the same number of 1s.

To see, in more rigorous terms, that this PDA accepts $\{0^n1^n : n \in \mathbb{N}\}$, we note the following "state invariants": The PDA is in

- state e if and only if the prefix of the input string that it has processed so far is ϵ and the stack is empty.

- state z if and only if the prefix of the input string that it has processed so far is 0^n and the stack is X^n, where n is a positive integer.

- state o if and only if the prefix of the input string that it has processed so far is 0^n1^m and the stack is X^{n-m}, where m and n are positive integers such that $m \leq n$.

More precisely,

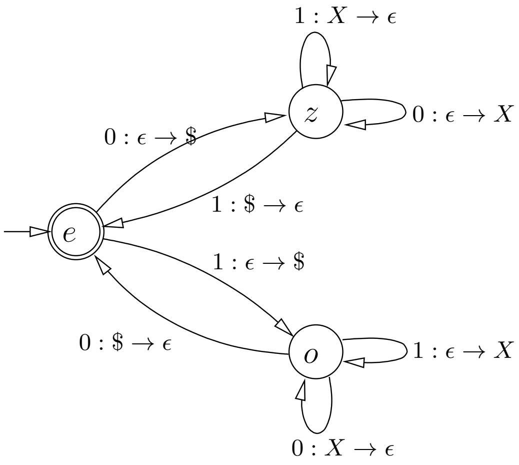
$$\begin{aligned}
(e, x, \epsilon) \vdash^* (e, y, \alpha) \quad &\Leftrightarrow \quad x = y \text{ and } \alpha = \epsilon \\
(e, x, \epsilon) \vdash^* (z, y, \alpha) \quad &\Leftrightarrow \quad x = 0^n y \text{ and } \alpha = X^n, \text{ for some positive integer } n \\
(e, x, \epsilon) \vdash^* (o, y, \alpha) \quad &\Leftrightarrow \quad x = 0^n 1^m y \text{ and } \alpha = X^{n-m}, \text{ for some positive integers } m \leq n.
\end{aligned}$$

These "state invariants" can be easily proved by induction on the length of the input string prefix that has been processed so far.

The correctness of the PDA, i.e., the fact that it accepts $\{0^n1^n : n \in \mathbb{N}\}$, follows immediately from this: Since e and o are all the accepting states of the PDA, a string x is accepted by the PDA if and only if $(e, x, \epsilon) \vdash^* (e, \epsilon, \epsilon)$ or $(e, x, \epsilon) \vdash^* (o, \epsilon, \epsilon)$. By the state invariant, this implies that M accepts x if and only if $x = \epsilon$ or $x = 0^n1^n$ for some positive integer n, i.e., if and only if $x \in \{0^n1^n : n \in \mathbb{N}\}$. **End of Example 8.4**

Example 8.5 Below is the diagram of a PDA that accepts the language

$$L_e = \{x \in \{0,1\}^* : x \text{ has the same number of 0s and 1s}\}.$$



Informally, the PDA works as follows: It is in state e as long as the prefix of the input string that it has processed so far has the same number of 0s and 1s and the stack is empty. From this state, upon seeing a 0, the PDA enters state z and pushes the symbol \$ on the stack. This indicates that this is the last (bottom) symbol in the stack. While in state z, as long as the PDA sees a 0, it pushes an X on the stack and remains in state z. Being in this state indicates that the prefix of the input string processes so far has an excess of 0s. The stack contains $\$X^{n-1}$, where n is the number of excess 0s in the prefix. While in state z, if the PDA sees a 1 and the top symbol of the stack is an X, it removes that symbol (reducing the number of excess 0s by one) and remains in state z. If in state z and the top symbol on the stack is \$, the PDA removes that symbol (making the stack empty) and returns to state e. This indicates that the prefix seen so far has an equal number of 0s and 1s. A symmetric situation arises when the PDA is in state e and sees a 1: It then enters state o, which indicates that the prefix seen so far has an excess of 1s, and keeps track of the number of excess 1s in the stack as before.

The correctness of this PDA is based on the following state invariants, which can be proved easily by induction on the length of the input string prefix processed so far. The PDA is in

- state e if and only if the prefix of the input string that it has processed so far has an equal number of 0s and 1s and the stack is empty.
- state z if and only if the prefix of the input string that it has processed so far has n more 0s than 1s and the stack is $\$X^{n-1}$, where n is a positive integer.
- state o if and only if the prefix of the input string that it has processed so far has n more 1s than 0s and the stack is $\$X^{n-1}$, where n is a positive integer.

End of Example 8.5

8.6 Equivalence of CFG and PDA

We now prove that CFG and PDA are equivalent, in that they describe the same class of languages. More precisely, for every CFG there is a PDA that accepts the language generated by that grammar. Conversely, for every PDA there is a CGF that generates the language accepted by that automaton.

8.6.1 From a CFG to a PDA

First we prove that, given any CFG G there is a PDA that accepts the language generated by G. The proof explicitly constructs the PDA.

Leftmost derivations

To understand this construction we need to focus on a special class of derivations. We say that a derivation is ***leftmost*** iff in every step the variable that gets replaced by the right-hand side of some production is the leftmost one. More precisely, the derivation $\alpha_1 \Rightarrow \alpha_2 \Rightarrow \cdots \Rightarrow \alpha_k$ is leftmost if, for each i such that $1 \leq i < k$, $\alpha_i = xA\gamma$, for some $x \in \Sigma^*$, $A \in V$ and $\gamma \in (\Sigma \cup V)^*$; and $\alpha_{i+1} = x\beta\gamma$, for some production $A \to \beta$.

Example 8.6 Let G be the grammar with start symbol S and the following productions:

$$\begin{array}{ll} S \to ABS & A \to 001 \\ S \to B & B \to 01 \\ & B \to 11 \end{array}$$

Shown below is a derivation of 001110010111 in this grammar.

$$\begin{aligned} S &\Rightarrow ABS \\ &\Rightarrow ABABS \\ &\Rightarrow ABABB \\ &\Rightarrow 001BABB \\ &\Rightarrow 001BAB11 \\ &\Rightarrow 001B001B11 \\ &\Rightarrow 001110010111 \end{aligned}$$

This is not a leftmost derivation. A leftmost derivation of the same string (in the same grammar) is

$$\begin{aligned} S &\Rightarrow ABS \\ &\Rightarrow 001BS \\ &\Rightarrow 00111S \\ &\Rightarrow 00111ABS \\ &\Rightarrow 00111001BS \\ &\Rightarrow 0011100101S \\ &\Rightarrow 0011100101B \\ &\Rightarrow 001110010111 \end{aligned}$$

End of Example 8.6

The phenomenon observed in the preceding example is not an accident: If a string has a derivation from a context-free grammar, it must also have a leftmost derivation from that grammar. Intuitively this is because every variable that appears in a derivation must sooner or later be replaced by the right-hand side of some production, and we can always choose to replace the leftmost variable in each step. A rigorous argument proving this is given below.

Lemma 8.7 *Let $G = (V, \Sigma, P, S)$ be a CFG, and $x \in \Sigma^*$ be such that $S \Rightarrow^* x$. Then there is a leftmost derivation of x in G.*

Proof. Let $x \in \Sigma^*$ be such that $S \Rightarrow^* x$. We must prove that there is a leftmost derivation of x from S. We call a step of a derivation ***leftmost*** if the production used in that step is applied to the leftmost variable. Thus, a derivation is leftmost if and only if every step is a leftmost step.

Let D be a derivation of x from S, and let n be the number of steps in D. So, D is $\alpha_0 \Rightarrow \alpha_1 \Rightarrow \cdots \Rightarrow \alpha_n$, where $\alpha_0 = S$ and $\alpha_n = x$. Without loss of generality we can assume that the first non-leftmost step of D, if any, is no earlier than the first non-leftmost step of any other n-step derivation of x from S.

We claim that D is a leftmost derivation. To prove this, we will assume the contrary and derive a contradiction. So, assume D is not a leftmost derivation, and let k be the first non-leftmost step of D. Thus, for some $A, B \in V$, $y \in \Sigma^*$ and $\alpha, \beta, \beta' \in (V \cup \Sigma)^*$, $\alpha_{k-1} = yA\alpha B\beta$, $\alpha_k = yA\alpha\beta'\beta$ and $B \to \beta'$ is a production. Since $x \in \Sigma^*$, there is a later step of D, say step $\ell > k$, where the leftmost A in α_k is eliminated by the application of some production. That is, for some $\gamma \in (V \cup \Sigma)^*$, $\alpha_{\ell-1} = yA\gamma$, $\alpha_\ell = y\alpha'\gamma$, and $A \to \alpha'$ is a production. Thus, the derivation D is:

$$\underbrace{S}_{\alpha_0} \Rightarrow \cdots \Rightarrow \underbrace{yA\alpha B\beta}_{\alpha_{k-1}} \Rightarrow \underbrace{yA\alpha\beta'\beta}_{\alpha_k} \Rightarrow \cdots \Rightarrow \underbrace{yA\gamma}_{\alpha_{\ell-1}} \Rightarrow \underbrace{y\alpha'\gamma}_{\alpha_\ell} \Rightarrow \cdots \Rightarrow \underbrace{x}_{\alpha_n}$$

From this it follows that $\alpha\beta'\beta \Rightarrow^* \gamma$ in $(\ell - 1) - k = \ell - k - 1$ steps, and that $y\alpha'\gamma \Rightarrow^* x$ in $n - \ell$ steps. This implies that the following is a derivation:

$$\begin{aligned} S &\Rightarrow^* yA\alpha B\beta && \text{[in } k-1 \text{ steps, as in } D\text{]} \\ &\Rightarrow y\alpha'\alpha B\beta && \text{[in 1 step, applying } A \to \alpha'\text{]} \\ &\Rightarrow y\alpha'\alpha\beta'\beta && \text{[in 1 step, applying } B \to \beta'\text{]} \\ &\Rightarrow^* y\alpha'\gamma && \text{[in } \ell-k-1 \text{ steps]} \\ &\Rightarrow^* x && \text{[in } n-\ell \text{ steps]} \end{aligned}$$

Thus, this is an n-step derivation of x from S in which the first non-leftmost step occurs later than in step k. This contradicts the definition of D, as the n-step derivation of x from S in which the first non-leftmost step (if any) occurs as late as possible. We conclude that D is a leftmost derivation, which proves the lemma. □

Informal description of the PDA construction

Suppose now we are given a CFG $G = (V, \Sigma, P, S)$. We show how to construct a PDA $M = (Q, \Sigma, \Gamma, \delta, s, F)$ that accepts the language generated by G. Intuitively, M works as follows. To accept a string x generated by G, M "simulates" a leftmost derivation of x, by keeping the string derived so far on the stack. It starts by placing S on the stack, and then simulates the steps of x's derivation one a time, by replacing the leftmost variable in the string generated so far by the right-hand side of the production used in that step. M uses nondeterminism to "guess" which of the potentially several productions with a given variable on the left-hand side to apply. The only difficulty with implementing this idea is that M can only look at the top symbol on the stack, and the leftmost *variable* of the string generated so far, i.e., the symbol it needs to replace, might not be at the top of the stack. This problem can be easily solved as follows. Notice that any symbols to the left of the leftmost variable in the string generated so far must match a prefix of the input string x. Thus, M removes from the stack any prefix of the input string, thereby "consuming" the corresponding portion of the input.

Example 8.7 Consider the grammar in Example 8.6. We will show how the PDA constructed to simulate this grammar accepts the string 001110010111. The PDA starts by pushing the start symbol of the grammar S on the stack. It then pops the top symbol of the stack and, without "consuming" any input string symbols, it pushes the symbols ABS on the stack. (The PDA can push only one symbol at a time so, technically, it needs three steps to do so: First a step to push S, then one to push B and finally one to push A.) After these actions are taken, the PDA is in a configuration where it has consumed none of the input and the stack contains ABS.

Note that the PDA had a choice when it popped the top symbol S from the stack. It could have replaced it by ABS or by B, since our grammar has productions $S \to ABS$ and $S \to B$. The PDA will nondeterministically choose the first of these because that's the one used in the leftmost derivation it is simulating. Because PDA can be nondeterministic, "guessing" which production to apply is not a problem.

In its next step, the PDA pops the top symbol on the stack, A, and pushes 001 on the stack, thereby simulating the second step in the leftmost derivation, which applies the production $A \to 001$. Again this is done without the PDA "consuming" any symbols from the input. Thus, the PDA is now in a configuration where the input is 001110010111 and the stack contains $001BS$.

Note that the top symbol on the stack is a terminal, so the PDA cannot simulate a production to replace it. Instead, in its next step, the PDA "matches" the 001 on top of the stack to the prefix 001 of the input string. In other words, it removes 001 from the stack and "consumes" 001 from the input string (in three steps). The PDA is now left with the task of simulating the derivation of the remainder of the input, namely 110010111, from the contents of the stack, namely BS.

To do so, it repeats the process described above: Without consuming any symbols from the remainder of the input, it pops the top symbol of the stack, B, and replaces it by the right-hand side of the production applied in the next step of the leftmost derivation it simulates, namely, $B \to 11$. Now the PDA is in a configuration where the remainder of the input is 110010111 and the stack contains $11S$.

The PDA "matches" the 11 at the top of the stack with the prefix 11 of the remainder of the input, by popping that string from the stack and "consuming" it from the input. This causes the PDA to enter the configuration with 0010111 as the remainder of the input and S on the stack.

The rest of the computation proceeds as follows:

- The PDA simulates the next production in the derivation, $S \to ABS$, resulting in configuration where the remainder of the input is 0010111 and the stack contains ABS.

- The PDA simulates the next production in the derivation, $A \to 001$, resulting in configuration where the remainder of the input is 0010111 and the stack contains $001BS$.

- The PDA consumes 001 from the input and removes it from the stack, resulting in configuration where the remainder of the input is 0111 and the stack contains BS.

- The PDA simulates the next production in the derivation, $B \to 01$, resulting in configuration where the remainder of the input is 0111 and the stack contains $01S$.

- The PDA consumes 01 from the input and removes it from the stack, resulting in configuration where the remainder of the input is 11 and the stack contains S.

- The PDA simulates the next production in the derivation, $S \to B$, resulting in configuration where the remainder of the input is 11 and the stack contains B.

- The PDA simulates the next production in the derivation, $B \to 11$, resulting in configuration where the remainder of the input is 11 and the stack contains 11.

- The PDA consumes 11 from the input and removes it from the stack, resulting in configuration where the remainder of the input is ϵ and the stack also contains ϵ.

At this point the simulation ends, and since the entire input has been consumed and the stack is empty, the input string is accepted. **End of Example 8.7**

The PDA construction

As gleaned from this example, the PDA simulating grammar G performs two types of actions:

(I) simulating the application of a production $A \to \alpha$ of G, by replacing the variable A at the top of the stack by the string α; and

(II) consuming a portion w of the input string and popping w from the stack.

We will refer to these as type (I) and type (II) actions, respectively.

We now present the formal details of this construction. Let $G = (V, \Sigma, P, S)$ be a CFG. Define the PDA $M = (Q, \Sigma, \Gamma, \delta, s, F)$ as follows; this will be the PDA that simulates G.

- Q consists of the start state s, the "main" state of the automaton q, and a set of "extra" states E needed for the moves that implement type (I) actions of M. Each production $\pi \in P$ will require us to add some states in E, as we will explain below. So, $Q = \{s, q\} \cup E$.

- $\Gamma = V \cup \Sigma$. The stack will contain a string of variables and terminals of G.
- The start state of M is s.
- $F = \{q\}$.

It remains to we define the transition function δ of M. To get the simulation going, the PDA first pushes S onto the stack, and enters its "main" state, q. Thus, we have $\delta(s, \epsilon, \epsilon) = \{(q, S)\}$.

To implement actions of type (II), we add the following transitions: for each $a \in \Sigma$, $\delta(q, a, a) = \{(q, \epsilon)\}$. That is, while in its "main" state q, M can consume a terminal from its input string and pop the same terminal from the top of the stack.

Finally, we implement actions of type (I) as follows. For each production $\pi \in P$, where π is $A \to a_1 a_2 \ldots a_k$ for some $a_1, a_2, \ldots, a_k \in V \cup \Sigma$, the set of "extra" states E contains states $q_\pi^1, q_\pi^2, \ldots, q_\pi^{k-1}$ and the transition function δ has the following transitions:

$$\begin{aligned} \delta(q, \epsilon, A) &= \{(q_\pi^{k-1}, a_k)\} \\ \delta(q_\pi^i, \epsilon, \epsilon) &= \{(q_\pi^{i-1}, a_i)\}, \qquad \text{for all } i,\ 1 < i < k \\ \delta(q_\pi^1, \epsilon, \epsilon) &= \{(q, a_1)\} \end{aligned}$$

Informally, these transitions replace A from the top of the stack by $a_1 a_2 \ldots a_k$, enabling M to simulate the production $A \to a_1 a_2 \ldots a_k$.

Proof that the constructed PDA accepts $\mathcal{L}(G)$

We can now prove that $\mathcal{L}(M) = \mathcal{L}(G)$. That is, for every $x \in \Sigma^*$,

$$x \in \mathcal{L}(G) \text{ if and only if } x \in \mathcal{L}(M). \tag{8.1}$$

The forward direction (only if) of (8.1) follows immediately from the following:

Lemma 8.8 *For every $x \in \Sigma^*$ and $\alpha \in (V \cup \Sigma)^*$ so that α is either empty or starts with a variable, if there is a leftmost derivation $S \Rightarrow^* x\alpha$ then $(q, x, S) \vdash^* (q, \epsilon, \alpha)$.*

The desired result follows from this lemma by taking $\alpha = \epsilon$: If $x \in \mathcal{L}(G)$ then there is a derivation $S \Rightarrow^* x$. By Lemma 8.7, there is a leftmost derivation $S \Rightarrow^* x$. By Lemma 8.8, $(q, x, S) \vdash^* (q, \epsilon, \epsilon)$. By construction of the PDA, $(s, x, \epsilon) \vdash (q, x, S)$. Combining with the preceding computation, $(s, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$. Since q is an accepting state, $x \in \mathcal{L}(M)$. We now give the proof of Lemma 8.8.

PROOF. The proof is by induction on the number of steps in the leftmost derivation $S \Rightarrow^* x\alpha$. To be more precise, consider the predicate

$P(n)$: for every $x \in \Sigma^*$ and $\alpha \in (V \cup \Sigma)^*$ s.t. α is either empty or starts with a variable, if there is an n-step leftmost derivation $S \Rightarrow^* x\alpha$, then $(q, x, S) \vdash^* (q, \epsilon, \alpha)$.

We use induction to prove that $P(n)$ is true for all $n \in \mathbb{N}$.

Basis: $n = 0$. If the derivation has zero steps, then $x = \epsilon$ and $\alpha = S$, and so $(q, x, S) \vdash^* (q, \epsilon, \alpha)$ is a zero-step computation. So, $P(0)$ holds.

Induction Step: Let k be an arbitrary natural number and suppose that $P(k)$ holds. We will prove that $P(k+1)$ holds as well. Let $x \in \Sigma^*$ and $\alpha \in (V \cup \Sigma)^*$, where α is either empty or starts with a variable. Suppose that there is a $(k+1)$-step leftmost derivation $S \Rightarrow^* x\alpha$. We must prove that $(q, x, \epsilon) \vdash^* (q, \epsilon, \alpha)$.

Consider the last step of the leftmost derivation $S \Rightarrow^* x\alpha$. This step must be $yA\beta \Rightarrow y\gamma\beta$, for some $y \in \Sigma^*$, $A \in V$ and $\beta, \gamma \in (V \cup \Sigma)^*$, where $A \to \gamma$ is a production of G, and $y\gamma\beta = x\alpha$. Recall that $x \in \Sigma^*$ and α is either empty or starts with a variable. So,

$$\gamma\beta = y'\alpha, \quad \text{for some } y' \in \Sigma^* \text{ such that } yy' = x. \tag{8.2}$$

Since $yA\beta \Rightarrow y\gamma\beta$ is the last step of the $(k+1)$-step leftmost derivation $S \Rightarrow^* x\alpha$, there is a k-step leftmost derivation $S \Rightarrow^* yA\beta$. By induction hypothesis,

$$(q, y, \epsilon) \vdash^* (q, \epsilon, A\beta). \tag{8.3}$$

Since $A \to \gamma$ is a production of G, using transitions that implement type (I) actions it follows that $(q, \epsilon, A\beta) \vdash^* (q, \epsilon, \gamma\beta)$, and since $\gamma\beta = y'\alpha$ (see (8.2))

$$(q, \epsilon, A\beta) \vdash^* (q, \epsilon, y'\alpha). \tag{8.4}$$

Combining (8.3) and (8.4), $(q, y, \epsilon) \vdash^* (q, \epsilon, y'\alpha)$. From this it follows that

$$(q, yy', \epsilon) \vdash^* (q, y', y'\alpha). \tag{8.5}$$

By using transitions that implement type (II) actions, $(q, y', y'\alpha) \vdash^* (q, \epsilon, \alpha)$. Combining this with (8.5) and recalling that $yy' = x$ (see (8.2)), we get that $(q, x, \epsilon) \vdash^* (q, \epsilon, \alpha)$, as wanted. □

The backward direction (if) of (8.1) follows from the following:

Lemma 8.9 *For every $x \in \Sigma^*$ and $\alpha \in (V \cup \Sigma)^*$, if $(q, x, S) \vdash^* (q, \epsilon, \alpha)$ then there is a leftmost derivation $S \Rightarrow^* x\alpha$.*

The desired result follows from this by taking $\alpha = \epsilon$: If $x \in \mathcal{L}(M)$ then there must be a computation $(s, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$. Since the only transition from s pushes S onto the stack without reading anything from the input, if we eliminate the first move from this computation it follows that $(q, x, S) \vdash^* (q, \epsilon, \epsilon)$. By the lemma, $S \Rightarrow^* x$, i.e., $s \in \mathcal{L}(G)$. We now give the proof of Lemma 8.9.

Proof. Recall that computations of M involve "actions" of two types: Type (I) actions replace the top variable on the stack by the right-hand side of a production π without reading any symbols from the input string. Type (II) actions read one symbol from the input string and pop that same symbol from the top of the stack. The proof of this lemma is by induction on the number of *actions* in the computation $(q, x, S) \vdash^* (q, \epsilon, \alpha)$ (Recall that each type (II) action is implemented by a sequence of steps: one to pop the variable from the top of the stack, and one for each symbol on the right-hand side of the production that has to be pushed onto the stack. It is for this reason that the induction is on the number of actions, rather than on the number of steps, in the computation.)

Consider the predicate

$P(n):$ for every $x \in \Sigma^*$ and $\alpha \in (V \cup \Sigma)^*$, if there is a computation $(q, x, S) \vdash^* (q, \epsilon, \alpha)$ with n actions, then $S \Rightarrow^* x\alpha$.

We use induction to prove that $P(n)$ is true for all $n \in \mathbb{N}$.

Basis: $n = 0$. If the computation has zero actions, then $x = \epsilon$ and $\alpha = S$, and so $S \Rightarrow^* x\alpha$ is a zero-step derivation. So, $P(0)$ holds.

Induction Step: Let k be an arbitrary natural number and suppose that $P(k)$ holds. We will prove that $P(k+1)$ holds as well. Let $x \in \Sigma^*$ and $\alpha \in (V \cup \Sigma)^*$. Suppose that there is a computation $(q, x, S) \vdash^* (q, \epsilon, \alpha)$ with $k+1$ actions. We must prove that $S \Rightarrow^* x\alpha$. There are two cases, depending on the type of the last action of the computation $(q, x, S) \vdash^* (q, \epsilon, \alpha)$.

Case 1. The last action is of type (I). In this case we have

$$(q, x, S) \vdash^* (q, \epsilon, A\beta) \vdash^* (q, \epsilon, \gamma\beta)$$

where $(q, x, S) \vdash^* (q, \epsilon, A\beta)$ is a computation with k actions, $A \in V$ and $\beta, \gamma \in (V \cup \Sigma)^*$ such that $A \to \gamma$ is a production of G and $\gamma\beta = \alpha$. By the induction hypothesis, $S \Rightarrow^* xA\beta$. Since $A \to \gamma$ is a production, $S \Rightarrow^* x\gamma\beta$. And since $\gamma\beta = \alpha$, $S \Rightarrow^* x\alpha$, as wanted.

Case 2. The last action is of type (II). In this case we have

$$(q, x, S) \vdash^* (q, a, a\alpha) \vdash (q, \epsilon, \alpha)$$

where $(q, x, S) \vdash^* (q, a, a\alpha)$ is a computation with k actions and $a \in \Sigma$. Then $x = ya$, for some $y \in \Sigma^*$, and $(q, y, S) \vdash^* (q, \epsilon, a\alpha)$ is a computation with k actions. By the induction hypothesis, $S \Rightarrow^* ya\alpha$, and since $ya = x$, $S \Rightarrow^* x\alpha$, as wanted. □

As we have seen, Lemmata 8.8 and 8.9 imply (8.1) — i.e., that the language accepted by the constructed PDA M is the same as the language generated by the given grammar G. Therefore,

Corollary 8.10 *Every context-free language is accepted by a PDA.*

8.6.2 From a PDA to a CFG

Now we prove the other half of the equivalence of CFG and PDA: Given a PDA M we show how to construct a CFG G that generates the language accepted by M. Thus, the grammar G should be constructed in such a manner that for any $x \in \Sigma^*$, there is a derivation $S \Rightarrow^* x$ if and only if x is accepted by M.

Informal description of the CFG construction

To develop some intuition about how this construction works, let's recall the construction of a regular expression that denotes the language accepted by a given DFSA (cf. Theorem 7.23). In that case we wanted to show how to construct a regular expression that denotes a string x if and only if x takes the automaton from the start state to an accepting state. To do this, however, we had to do more: We showed how to construct regular expressions that denote the strings that take the automaton from any state p (not necessarily the start state) to any state q (not necessarily an accepting state).

In a similar spirit, we will construct a grammar with variables A_{pq}, for every pair of states p and q of the given PDA. The grammar is constructed so that for every $x \in \Sigma^*$, there is a derivation $A_{pq} \Rightarrow^* x$ if and only if the PDA has a computation $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$. In other words, if and only if the PDA, started in state p with empty stack and input string x, can "consume" x and end up in state q with empty stack. If we can do this, we can then introduce a new variable S and productions $S \to A_{sp}$, for s the start state of M and each accepting state p of M. This grammar will generate precisely the set of strings accepted by M.

To accomplish this, we need to make a simplifying assumption about the given PDA. Namely, in each move it either pushes or pops a symbol on top of the stack. Thus, our PDA does not have any moves in which it does not change the stack, or any moves in which it directly replaces the symbol at the top of the stack by another symbol. We can make this assumption without loss of generality because we can convert *any* given PDA to one that accepts the same language and satisfies this assumption. We can do this as follows:

- We replace every transition from state p to state q in which the given PDA reads $a \in \Sigma \cup \{\epsilon\}$ from the input string and does not change the stack, by a pair of consecutive transitions: In the first of these the PDA, upon reading a, moves from state p to an intermediate state labeled (p, q), and pushes a dummy symbol X onto the stack. In the second, the PDA moves from the intermediate state (p, q) to state q and pops the just-pushed dummy symbol X from the stack, without reading anything from the input string.

- We replace every transition from state p to state q in which the given PDA reads $a \in \Sigma \cup \{\epsilon\}$ from the input string and changes the top of the stack from X to Y, by a pair of consecutive transitions: In the first of these the PDA, upon reading a, moves from state p to an intermediate state labeled (p, Y, q), and pops X from the stack. In the second, the PDA moves from the intermediate state (p, Y, q) to state q and pushes Y onto the stack, without reading anything from the input string.

To understand how G is constructed, consider a computation of the automaton M that takes it from state p with empty stack to state q with empty stack, consuming input string x. If this computation is nontrivial (i.e., contains at least one step) then there are two possibilities:

(a) The stack becomes empty at some intermediate stage of this computation. This means that the automaton goes from state p with empty stack to some intermediate state r with empty stack, having consumed some prefix y of x, and then goes from state r with empty stack to state q with empty stack, having consumed the remainder of x. We capture this case by introducing the production $A_{pq} \to A_{pr}A_{rq}$ in our grammar. Intuitively this says that if a string y takes M from state p (with empty stack) to state r (with empty stack), and a string y' takes M from state r (with empty stack) to state q (with empty stack), then the string yy' takes M from state p (with empty stack) to state q (with empty stack).

(b) The stack does not become empty at any intermediate stage of this computation. Recall that every step of our automaton must either push or pop a symbol on the stack. When the computation starts, the stack is empty; thus, its first step must push some symbol on the stack. Similarly, when the computation ends, the stack is empty; thus, its last step of must pop that symbol from the stack. In more detail, in this case, in the first step of such a computation, the automaton starts in state p with empty stack, reads some symbol $a \in \Sigma \cup \{\epsilon\}$ from the input, pushes some symbol X onto the stack and enters some state p'. Then, it follows a computation that takes it from state p' with just X on the stack to some state q' again with just X on the stack. And, finally, in the last step of the computation, it reads some symbol $b \in \Sigma \cup \{\epsilon\}$ from the input, pops the X from the stack and moves from state q' to q with empty stack. We capture this case by introducing the production $A_{pq} \to aA_{p'q'}b$ for our grammar, whenever the automaton has a transition that takes it from state p to state p' and pushes X onto the stack upon reading $a \in \Sigma \cup \{\epsilon\}$, and a transition that takes it from state q' to state q and pops X from the stack upon reading $b \in \Sigma \cup \{\epsilon\}$. Intuitively this production says that, when such transitions exist, if a string y takes M from state p' (with empty stack) to state q' (with empty stack), then the string ayb takes M from state p (with empty stack) to state q (with empty stack).

Finally, the grammar "simulates" trivial zero-step computations by introducing productions $A_{pp} \to \epsilon$, for every state p of the PDA M. Intuitively, such transitions say that the input string ϵ takes M from p (with empty stack) to p (with empty stack).

The CFG construction

With this informal description and partial justification in mind, we give the details of the construction and a rigorous proof that it works. Given a PDA $M = (Q, \Sigma, \Gamma, \delta, s, F)$ we construct the following CFG $G = (V, \Sigma, P, S)$. As explained earlier, we can assume without loss of generality that δ is such that in each move of M the automaton either pushes a new symbol on top of the stack, or pops the symbol presently on top of the stack.

The set of variables of G is $V = \{A_{pq} : p, q \in Q\} \cup \{S\}$. The set of productions P of G consists of the following four types of productions:

- $A_{pq} \to A_{pr}A_{qr}$, for all $p, q, r \in Q$.
- $A_{pq} \to aA_{p'q'}b$, for all $p, q, p', q' \in Q$ and $a, b \in \Sigma \cup \{\epsilon\}$ so that for some $X \in \Gamma$, $(p', X) \in \delta(p, a, \epsilon)$ and $(q, \epsilon) \in \delta(q', b, X)$.
- $A_{pp} \to \epsilon$, for all $p \in Q$.

- $S \to A_{sp}$, for all $p \in F$ (recall that s is the start state of M).

Proof that the constructed CFG generates $\mathcal{L}(M)$

We will now prove that the constucted CFG G generates the language accepted by M. This follows directly from the following:

Lemma 8.11 *For all $x \in \Sigma^*$ and all $p, q \in Q$, $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$ if and only if $A_{pq} \Rightarrow^* x$.*

Proof. First we prove the forward direction by induction on the number of steps in the computation $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$. Consider the predicate

$P(n)$: For all $x \in \Sigma^*$ and all $p, q \in Q$,
if there is an n-step computation $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$ then $A_{pq} \Rightarrow^* x$.

We will prove that $P(n)$ holds for all $n \in \mathbb{N}$ using complete induction. Let k be an arbitrary natural number. Assume that $P(j)$ holds for all integers j such that $0 \leq j < k$. We must prove that $P(k)$ holds as well. Let $x \in \Sigma^*$ and $p, q \in Q$ be such that there is a k-step computation $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$. We must prove that $A_{pq} \Rightarrow^* x$.

Case 1. $k = 0$. If $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$ is a zero-step computation, it must be that $p = q$ and $x = \epsilon$. Then $A_{pp} \to \epsilon$ is a production and so $A_{pp} \Rightarrow^* \epsilon$, as wanted.

Case 2. There are two subcases, depending on whether the computation $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$ includes an intermediate configuration with empty stack.

Subcase 2(a). There is some $r \in Q$ and $x_1, x_2 \in \Sigma^*$ so that $x_1 x_2 = x$,

$$(p, x_1 x_2, \epsilon) \vdash^* (r, x_2, \epsilon) \tag{8.6}$$

is a k_1-move computation, and

$$(r, x_2, \epsilon) \vdash^* (q, \epsilon, \epsilon) \tag{8.7}$$

is a k_2-move computation, where k_1 and k_2 are positive integers such that $k_1 + k_2 = k$. Thus, $0 < k_1, k_2 < k$, and the induction hypothesis applies to computations with k_1 and k_2 steps. By (8.6) there is a k_1-step computation $(p, x_1, \epsilon) \vdash^* (r, \epsilon, \epsilon)$, and so by the induction hypothesis, $A_{pr} \Rightarrow^* x_1$. By (8.7) and the induction hypothesis, $A_{rq} \Rightarrow^* x_2$. Since $A_{pq} \to A_{pr} A_{rq}$ is a production of G, we have $A_{pq} \Rightarrow A_{pr} A_{rq} \Rightarrow^* x_1 x_2$, and since $x_1 x_2 = x$, $A_{pq} \Rightarrow^* x$, as wanted.

Subcase 2(b). There is no intermediate configuration with empty stack in the computation $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$. Recall that, by assumption, *every* step of M either pushes a symbol onto the stack or pops one from the stack. Since at the start of this computation the stack is empty, the first step must push some symbol X onto the empty stack, read some $a \in \Sigma \cup \{\epsilon\}$ from the input string, and change the state from p to some p'. Similarly, since at the end of the computation the stack is empty, the last step of the computation must pop from the stack the X pushed by the first step, read some $b \in \Sigma \cup \{\epsilon\}$ from the input string, and change the state from some q' to q. In other words, the computation is

$$(p, ayb, \epsilon) \vdash (p', yb, X) \vdash^* (q', b, X) \vdash (q, \epsilon, \epsilon)$$

where $x = ayb$ for some $a, b \in \Sigma \cup \{\epsilon\}$ and $y \in \Sigma^*$. The step $(p, ayb, \epsilon) \vdash (p', yb, X)$ implies that $(p', X) \in \delta(p, a, \epsilon)$, and the step $(q', b, X) \vdash (q, \epsilon, \epsilon)$ implies that $(q, \epsilon) \in \delta(q', b, X)$. Then, by construction,

$$A_{pq} \to aA_{p'q'}b \quad \text{is a production of } G. \tag{8.8}$$

The computation $(p', yb, X) \vdash^* (q', b, X)$ has $k - 2$ steps. Thus, there is a $(k - 2)$-step computation $(p', y, \epsilon) \vdash^* (q', \epsilon, \epsilon)$. By the induction hypothesis,

$$A_{p'q'} \Rightarrow^* y. \tag{8.9}$$

By (8.8) and (8.9), $A_{pq} \Rightarrow aA_{p'q'}b \Rightarrow^* ayb$. Since $x = ayb$, we have $A_{pq} \Rightarrow^* x$, as wanted.

We now prove the backward direction of the lemma by induction on the number of steps in the derivation $A_{pq} \Rightarrow^* x$. Consider the predicate

$Q(n)$: For all $x \in \Sigma^*$ and all $p, q \in Q$,
if there is an n-step derivation $A_{pq} \Rightarrow^* x$ then $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$.

We will use complete induction to prove that $Q(n)$ holds for all positive integers n. Let k be an arbitrary positive integer. Assume that $Q(j)$ holds for all integers j such that $1 \leq j < k$. We must prove that $Q(k)$ holds as well. Let $x \in \Sigma^*$ and $p, q \in Q$ be such that there is a k-step derivation $A_{pq} \Rightarrow^* x$. We must prove that $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$.

Case 1. $k = 1$. There is only one type of production in G that has no variables on the right-hand side, namely $A_{pp} \to \epsilon$, where $p \in Q$. So, if $A_{pq} \Rightarrow^* x$ is a one-step derivation, it must be that $q = p$ and $x = \epsilon$. In that case, $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$ via a zero-step computation.

Case 2. There are two types of productions with A_{pq} on the left-hand side. Accordingly, there are two subcases, depending on the first step of the derivation $A_{pq} \Rightarrow^* x$.

Subcase 2(a). The derivation $A_{pq} \Rightarrow^* x$ starts with $A_{pq} \Rightarrow A_{pr}A_{rq}$, for some $r \in Q$. Then there are $x_1, x_2 \in \Sigma^*$ such that $x = x_1x_2$, and $A_{pr} \Rightarrow^* x_1$ and $A_{rq} \Rightarrow^* x_2$. The number of steps in each of the derivations $A_{pr} \Rightarrow^* x_1$ and $A_{rq} \Rightarrow^* x_2$ is strictly less than k and so, by the induction hypothesis, there exist computations $(p, x_1, \epsilon) \vdash^* (r, \epsilon, \epsilon)$ and $(r, x_2, \epsilon) \vdash^* (q, \epsilon, \epsilon)$. Thus, there exists a computation

$$(p, x_1x_2, \epsilon) \vdash^* (r, x_2, \epsilon) \vdash^* (q, \epsilon, \epsilon)$$

and since $x_1x_2 = x$, $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$, as wanted.

SUBCASE 2(b). The derivation $A_{pq} \Rightarrow^* x$ starts with $A_{pq} \Rightarrow aA_{p'q'}b$, for some $p', q' \in Q$ and $a, b \in \Sigma \cup \{\epsilon\}$. Then the derivation $aA_{p'q'}b \Rightarrow^* x$ has fewer than k steps, and so there is a derivation $A_{p'q'} \Rightarrow^* y$ with fewer than k steps, where $y \in \Sigma^*$ is such that $ayb = x$. By the induction hypothesis,

$$(p', y, \epsilon) \vdash^* (q', \epsilon, \epsilon) \tag{8.10}$$

The fact that $A_{pq} \to aA_{p'q'}b$ is a production implies that, for some $X \in \Gamma$, $(p', X) \in \delta(q, a, \epsilon)$ and $(q, \epsilon) \in \delta(q', b, X)$. Thus,

$$(p, ayb, \epsilon) \vdash (p', yb, X) \tag{8.11}$$

and

$$(q', b, X) \vdash (q, \epsilon, \epsilon) \tag{8.12}$$

From (8.10), it follows that

$$(p', yb, X) \vdash^* (q', b, X) \tag{8.13}$$

By (8.11), (8.13) and (8.12), and recalling that $ayb = x$, we get that $(p, x, \epsilon) \vdash^* (q, \epsilon, \epsilon)$, as wanted. □

Using this lemma it is now easy to see that $\mathcal{L}(G) = \mathcal{L}(M)$. For any $x \in \Sigma^*$,

$$\begin{array}{llll}
x \in \mathcal{L}(G) & \text{iff} & S \Rightarrow^* x & \text{[by definition of } \mathcal{L}(G)] \\
 & \text{iff} & A_{sp} \Rightarrow^* x, \text{ for some } p \in F & \text{[because all productions with } S \text{ on the} \\
 & & & \text{left are of the form } S \to A_{sp},\ p \in F] \\
 & \text{iff} & (s, x, \epsilon) \vdash^* (p, \epsilon, \epsilon) & \text{[by Lemma 8.11]} \\
 & \text{iff} & x \in \mathcal{L}(M) & \text{[by definition of } \mathcal{L}(M)]
\end{array}$$

Therefore,

Corollary 8.12 *Every language accepted by a PDA is context-free.*

We can summarise Corollaries 8.10 and 8.12 as

Corollary 8.13 *The class of languages accepted by PDA is the class of context-free languages (i.e., the class of languages generated by CFG).*

Exercises

1. Give a CFG that generates and a PDA that accepts each of the following languages:

- $\{x\#x^R : x \in \{0,1\}^*\}$
- $\{xx^R : x \in \{0,1\}^*\}$
- $\{0^n 1^{2n} : n \in \mathbb{N}\}$
- $\{0^m 1^n 2^{m+n} : m, n \in \mathbb{N}\}$
- $\{0^m 1^n 0^k 1^\ell : m, n, k, \ell \in \mathbb{N} \text{ and } m + n = k + \ell\}$
- $\{0^m 1^n 2^k : m = n \text{ or } n = k\}$

2. Prove that the following CFL generates the set of strings over $\{0, 1\}$ that have equally many 0s as 1s:

$$\begin{aligned} S &\to 0S1S \\ S &\to 1S0S \\ S &\to \epsilon \end{aligned}$$

3. Let $\Sigma = \{\emptyset, \epsilon, 0, 1, +, *, (,)\}$. Give a CFG with set of terminals Σ that generates the set of regular expressions over $\{0, 1\}$.

4. Define the "shuffle" $L \bowtie L'$ of two languages L and L' as follows:

$$\begin{aligned} L \bowtie L' = \{x \in \Sigma^* : \quad & \text{either } x = \epsilon \text{, or} \\ & \text{there is an integer } k > 0 \text{ and strings } y_1, y_2, \ldots, y_k \in L \text{ and } y_1', y_2', \ldots, y_k' \in L' \\ & \text{so that } x = y_1 y_1' y_2 y_2' \cdots y_k y_k'\} \end{aligned}$$

Prove that, for all context-free languages L and L', $L \bowtie L'$ is also a context-free language.