# Chapter 1

# Automata: The Methods and the Madness

Automata theory is the study of abstract computing devices, or "machines." Before there were computers, in the 1930's, A. Turing studied an abstract machine that had all the capabilities of today's computers, at least as far as in what they could compute. Turing's goal was to describe precisely the boundary between what a computing machine could do and what it could not do; his conclusions apply not only to his abstract *Turing machines*, but to today's real machines.

In the 1940's and 1950's, simpler kinds of machines, which we today call "finite automata," were studied by a number of researchers. These automata, originally proposed to model brain function, turned out to be extremely useful for a variety of other purposes, which we shall mention in Section 1.1. Also in the late 1950's, the linguist N. Chomsky began the study of formal "grammars." While not strictly machines, these grammars have close relationships to abstract automata and serve today as the basis of some important software components, including parts of compilers.

In 1969, S. Cook extended Turing's study of what could and what could not be computed. Cook was able to separate those problems that can be solved efficiently by computer from those problems that can in principle be solved, but in practice take so much time that computers are useless for all but very small instances of the problem. The latter class of problems is called "intractable," or "NP-hard." It is highly unlikely that even the exponential improvement in computing speed that computer hardware has been following ("Moore's Law") will have significant impact on our ability to solve large instances of intractable problems.

All of these theoretical developments bear directly on what computer scientists do today. Some of the concepts, like finite automata and certain kinds of formal grammars, are used in the design and construction of important kinds of software. Other concepts, like the Turing machine, help us understand what

we can expect from our software. Especially, the theory of intractable problems lets us deduce whether we are likely to be able to meet a problem "head-on" and write a program to solve it (because it is not in the intractable class), or whether we have to find some way to work around the intractable problem: find an approximation, use a heuristic, or use some other method to limit the amount of time the program will spend solving the problem.

In this introductory chapter, we begin with a very high-level view of what automata theory is about, and what its uses are. Much of the chapter is devoted to a survey of proof techniques and tricks for discovering proofs. We cover deductive proofs, reformulating statements, proofs by contradiction, proofs by induction, and other important concepts. A final section introduces the concepts that pervade automata theory: alphabets, strings, and languages.

## 1.1 Why Study Automata Theory?

There are several reasons why the study of automata and complexity is an important part of the core of Computer Science. This section serves to introduce the reader to the principal motivation and also outlines the major topics covered in this book.

### 1.1.1 Introduction to Finite Automata

Finite automata are a useful model for many important kinds of hardware and software. We shall see, starting in Chapter 2, examples of how the concepts are used. For the moment, let us just list some of the most important kinds:

1. Software for designing and checking the behavior of digital circuits.
2. The "lexical analyzer" of a typical compiler, that is, the compiler component that breaks the input text into logical units, such as identifiers, keywords, and punctuation.
3. Software for scanning large bodies of text, such as collections of Web pages, to find occurrences of words, phrases, or other patterns.
4. Software for verifying systems of all types that have a finite number of distinct states, such as communications protocols or protocols for secure exchange of information.

While we shall soon meet a precise definition of automata of various types, let us begin our informal introduction with a sketch of what a finite automaton is and does. There are many systems or components, such as those enumerated above, that may be viewed as being at all times in one of a finite number of "states." The purpose of a state is to remember the relevant portion of the system's history. Since there are only a finite number of states, the entire history generally cannot be remembered, so the system must be designed carefully, to

remember what is important and forget what is not. The advantage of having only a finite number of states is that we can implement the system with a fixed set of resources. For example, we could implement it in hardware as a circuit, or as a simple form of program that can make decisions looking only at a limited amount of data or using the position in the code itself to make the decision.

**Example 1.1:** Perhaps the simplest nontrivial finite automaton is an on/off switch. The device remembers whether it is in the "on" state or the "off" state, and it allows the user to press a button whose effect is different, depending on the state of the switch. That is, if the switch is in the off state, then pressing the button changes it to the on state, and if the switch is in the on state, then pressing the same button turns it to the off state.

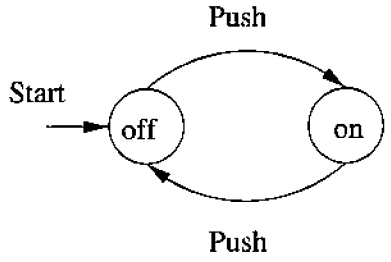


Figure 1.1: A finite automaton modeling an on/off switch

The finite-automaton model for the switch is shown in Fig. 1.1. As for all finite automata, the states are represented by circles; in this example, we have named the states *on* and *off*. Arcs between states are labeled by "inputs," which represent external influences on the system. Here, both arcs are labeled by the input *Push*, which represents a user pushing the button. The intent of the two arcs is that whichever state the system is in, when the *Push* input is received it goes to the other state.

One of the states is designated the "start state," the state in which the system is placed initially. In our example, the start state is *off*, and we conventionally indicate the start state by the word *Start* and an arrow leading to that state.

It is often necessary to indicate one or more states as "final" or "accepting" states. Entering one of these states after a sequence of inputs indicates that the input sequence is good in some way. For instance, we could have regarded the state *on* in Fig. 1.1 as accepting, because in that state, the device being controlled by the switch will operate. It is conventional to designate accepting states by a double circle, although we have not made any such designation in Fig. 1.1. □

**Example 1.2:** Sometimes, what is remembered by a state can be much more complex than an on/off choice. Figure 1.2 shows another finite automaton that could be part of a lexical analyzer. The job of this automaton is to recognize

the keyword `then`. It thus needs five states, each of which represents a different position in the word `then` that has been reached so far. These positions correspond to the prefixes of the word, ranging from the empty string (i.e., nothing of the word has been seen so far) to the complete word.

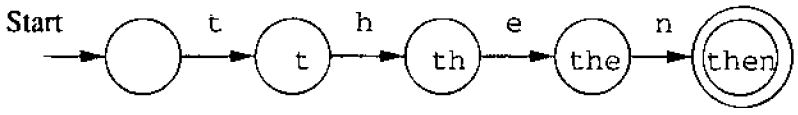


Figure 1.2: A finite automaton modeling recognition of `then`

In Fig. 1.2, the five states are named by the prefix of `then` seen so far. Inputs correspond to letters. We may imagine that the lexical analyzer examines one character of the program that it is compiling at a time, and the next character to be examined is the input to the automaton. The start state corresponds to the empty string, and each state has a transition on the next letter of `then` to the state that corresponds to the next-larger prefix. The state named `then` is entered when the input has spelled the word `then`. Since it is the job of this automaton to recognize when `then` has been seen, we could consider that state the lone accepting state. □

### 1.1.2 Structural Representations

There are two important notations that are not automaton-like, but play an important role in the study of automata and their applications.

1. *Grammars* are useful models when designing software that processes data with a recursive structure. The best-known example is a "parser," the component of a compiler that deals with the recursively nested features of the typical programming language, such as expressions — arithmetic, conditional, and so on. For instance, a grammatical rule like $E \Rightarrow E + E$ states that an expression can be formed by taking any two expressions and connecting them by a plus sign; this rule is typical of how expressions of real programming languages are formed. We introduce context-free grammars, as they are usually called, in Chapter 5.

2. *Regular Expressions* also denote the structure of data, especially text strings. As we shall see in Chapter 3, the patterns of strings they describe are exactly the same as what can be described by finite automata. The style of these expressions differs significantly from that of grammars, and we shall content ourselves with a simple example here. The UNIX-style regular expression `'[A-Z][a-z]*[ ][A-Z][A-Z]'` represents capitalized words followed by a space and two capital letters. This expression represents patterns in text that could be a city and state, e.g., `Ithaca NY`. It misses multiword city names, such as `Palo Alto CA`, which could be captured by the more complex expression

```
'([A-Z][a-z]*[ ])*[ ][A-Z][A-Z]'
```

When interpreting such expressions, we only need to know that `[A-Z]` represents a range of characters from capital "A" to capital "Z" (i.e., any capital letter), and `[ ]` is used to represent the blank character alone. Also, the symbol * represents "any number of" the preceding expression. Parentheses are used to group components of the expression; they do not represent characters of the text described.

### 1.1.3 Automata and Complexity

Automata are essential for the study of the limits of computation. As we mentioned in the introduction to the chapter, there are two important issues:

1. What can a computer do at all? This study is called "decidability," and the problems that can be solved by computer are called "decidable." This topic is addressed in Chapter 9.

2. What can a computer do efficiently? This study is called "intractability," and the problems that can be solved by a computer using no more time than some slowly growing function of the size of the input are called "tractable." Often, we take all polynomial functions to be "slowly growing," while functions that grow faster than any polynomial are deemed to grow too fast. The subject is studied in Chapter 10.

## 1.2 Introduction to Formal Proof

If you studied plane geometry in high school any time before the 1990's, you most likely had to do some detailed "deductive proofs," where you showed the truth of a statement by a detailed sequence of steps and reasons. While geometry has its practical side (e.g., you need to know the rule for computing the area of a rectangle if you need to buy the correct amount of carpet for a room), the study of formal proof methodologies was at least as important a reason for covering this branch of mathematics in high school.

In the USA of the 1990's it became popular to teach proof as a matter of personal feelings about the statement. While it is good to feel the truth of a statement you need to use, important techniques of proof are no longer mastered in high school. Yet proof is something that every computer scientist needs to understand. Some computer scientists take the extreme view that a formal proof of the correctness of a program should go hand-in-hand with the writing of the program itself. We doubt that doing so is productive. On the other hand, there are those who say that proof has no place in the discipline of programming. The slogan "if you are not sure your program is correct, run it and see" is commonly offered by this camp.

Our position is between these two extremes. Testing programs is surely essential. However, testing goes only so far, since you cannot try your program on every input. More importantly, if your program is complex -- say a tricky recursion or iteration — then if you don't understand what is going on as you go around a loop or call a function recursively, it is unlikely that you will write the code correctly. When your testing tells you the code is incorrect, you still need to get it right.

To make your iteration or recursion correct, you need to set up an inductive hypothesis, and it is helpful to reason, formally or informally, that the hypothesis is consistent with the iteration or recursion. This process of understanding the workings of a correct program is essentially the same as the process of proving theorems by induction. Thus, in addition to giving you models that are useful for certain types of software, it has become traditional for a course on automata theory to cover methodologies of formal proof. Perhaps more than other core subjects of computer science, automata theory lends itself to natural and interesting proofs, both of the *deductive* kind (a sequence of justified steps) and the *inductive* kind (recursive proofs of a parameterized statement that use the statement itself with "lower" values of the parameter).

### 1.2.1 Deductive Proofs

As mentioned above, a deductive proof consists of a sequence of statements whose truth leads us from some initial statement, called the *hypothesis* or the *given statement(s)*, to a *conclusion* statement. Each step in the proof must follow, by some accepted logical principle, from either the given facts, or some of the previous statements in the deductive proof, or a combination of these.

The hypothesis may be true or false, typically depending on values of its parameters. Often, the hypothesis consists of several independent statements connected by a logical AND. In those cases, we talk of each of these statements as a hypothesis, or as a given statement.

The theorem that is proved when we go from a hypothesis $H$ to a conclusion $C$ is the statement "if $H$ then $C$." We say that $C$ is *deduced* from $H$. An example theorem of the form "if $H$ then $C$" will illustrate these points.

**Theorem 1.3:** If $x \geq 4$, then $2^x \geq x^2$. □

It is not hard to convince ourselves informally that Theorem 1.3 is true, although a formal proof requires induction and will be left for Example 1.17. First, notice that the hypothesis $H$ is "$x \geq 4$." This hypothesis has a parameter, $x$, and thus is neither true nor false. Rather, its truth depends on the value of the parameter $x$; e.g., $H$ is true for $x = 6$ and false for $x = 2$.

Likewise, the conclusion $C$ is "$2^x \geq x^2$." This statement also uses parameter $x$ and is true for certain values of $x$ and not others. For example, $C$ is false for $x = 3$, since $2^3 = 8$, which is not as large as $3^2 = 9$. On the other hand, $C$ is true for $x = 4$, since $2^4 = 4^2 = 16$. For $x = 5$, the statement is also true, since $2^5 = 32$ is at least as large as $5^2 = 25$.

Perhaps you can see the intuitive argument that tells us the conclusion $2^x \geq x^2$ will be true whenever $x \geq 4$. We already saw that it is true for $x = 4$. As $x$ grows larger than 4, the left side, $2^x$ doubles each time $x$ increases by 1. However, the right side, $x^2$, grows by the ratio $\left(\frac{x+1}{x}\right)^2$. If $x \geq 4$, then $(x+1)/x$ cannot be greater than 1.25, and therefore $\left(\frac{x+1}{x}\right)^2$ cannot be bigger than 1.5625. Since $1.5625 < 2$, each time $x$ increases above 4 the left side $2^x$ grows more than the right side $x^2$. Thus, as long as we start from a value like $x = 4$ where the inequality $2^x \geq x^2$ is already satisfied, we can increase $x$ as much as we like, and the inequality will still be satisfied.

We have now completed an informal but accurate proof of Theorem 1.3. We shall return to the proof and make it more precise in Example 1.17, after we introduce "inductive" proofs.

Theorem 1.3, like all interesting theorems, involves an infinite number of related facts, in this case the statement "if $x \geq 4$ then $2^x \geq x^2$" for all integers $x$. In fact, we do not need to assume $x$ is an integer, but the proof talked about repeatedly increasing $x$ by 1, starting at $x = 4$, so we really addressed only the situation where $x$ is an integer.

Theorem 1.3 can be used to help deduce other theorems. In the next example, we consider a complete deductive proof of a simple theorem that uses Theorem 1.3.

**Theorem 1.4:** If $x$ is the sum of the squares of four positive integers, then $2^x \geq x^2$.

**PROOF:** The intuitive idea of the proof is that if the hypothesis is true for $x$, that is, $x$ is the sum of the squares of four positive integers, then $x$ must be at least 4. Therefore, the hypothesis of Theorem 1.3 holds, and since we believe that theorem, we may state that its conclusion is also true for $x$. The reasoning can be expressed as a sequence of steps. Each step is either the hypothesis of the theorem to be proved, part of that hypothesis, or a statement that follows from one or more previous statements.

By "follows" we mean that if the hypothesis of some theorem is a previous statement, then the conclusion of that theorem is true, and can be written down as a statement of our proof. This logical rule is often called *modus ponens*; i.e., if we know $H$ is true, and we know "if $H$ then $C$" is true, we may conclude that $C$ is true. We also allow certain other logical steps to be used in creating a statement that follows from one or more previous statements. For instance, if $A$ and $B$ are two previous statements, then we can deduce and write down the statement "$A$ and $B$."

Figure 1.3 shows the sequence of statements we need to prove Theorem 1.4. While we shall not generally prove theorems in such a stylized form, it helps to think of proofs as very explicit lists of statements, each with a precise justification. In step (1), we have repeated one of the given statements of the theorem: that $x$ is the sum of the squares of four integers. It often helps in proofs if we name quantities that are referred to but not named, and we have done so here, giving the four integers the names $a$, $b$, $c$, and $d$.

| | Statement | Justification |
|---|---|---|
| 1. | $x = a^2 + b^2 + c^2 + d^2$ | Given |
| 2. | $a \geq 1$; $b \geq 1$; $c \geq 1$; $d \geq 1$ | Given |
| 3. | $a^2 \geq 1$; $b^2 \geq 1$; $c^2 \geq 1$; $d^2 \geq 1$ | (2) and properties of arithmetic |
| 4. | $x \geq 4$ | (1), (3), and properties of arithmetic |
| 5. | $2^x \geq x^2$ | (4) and Theorem 1.3 |

Figure 1.3: A formal proof of Theorem 1.4

In step (2), we put down the other part of the hypothesis of the theorem: that the values being squared are each at least 1. Technically, this statement represents four distinct statements, one for each of the four integers involved. Then, in step (3) we observe that if a number is at least 1, then its square is also at least 1. We use as a justification the fact that statement (2) holds, and "properties of arithmetic." That is, we assume the reader knows, or can prove simple statements about how inequalities work, such as the statement "if $y \geq 1$, then $y^2 \geq 1$."

Step (4) uses statements (1) and (3). The first statement tells us that $x$ is the sum of the four squares in question, and statement (3) tells us that each of the squares is at least 1. Again using well-known properties of arithmetic, we conclude that $x$ is at least $1 + 1 + 1 + 1$, or 4.

At the final step (5), we use statement (4), which is the hypothesis of Theorem 1.3. The theorem itself is the justification for writing down its conclusion, since its hypothesis is a previous statement. Since the statement (5) that is the conclusion of Theorem 1.3 is also the conclusion of Theorem 1.4, we have now proved Theorem 1.4. That is, we have started with the hypothesis of that theorem, and have managed to deduce its conclusion. □

### 1.2.2 Reduction to Definitions

In the previous two theorems, the hypotheses used terms that should have been familiar: integers, addition, and multiplication, for instance. In many other theorems, including many from automata theory, the terms used in the statement may have implications that are less obvious. A useful way to proceed in many proofs is:

- If you are not sure how to start a proof, convert all terms in the hypothesis to their definitions.

Here is an example of a theorem that is simple to prove once we have expressed its statement in elementary terms. It uses the following two definitions:

1. A set $S$ is *finite* if there exists an integer $n$ such that $S$ has exactly $n$ elements. We write $\|S\| = n$, where $\|S\|$ is used to denote the number

of elements in a set $S$. If the set $S$ is not finite, we say $S$ is *infinite*. Intuitively, an infinite set is a set that contains more than any integer number of elements.

2. If $S$ and $T$ are both subsets of some set $U$, then $T$ is the *complement* of $S$ (with respect to $U$) if $S \cup T = U$ and $S \cap T = \emptyset$. That is, each element of $U$ is in exactly one of $S$ and $T$; put another way, $T$ consists of exactly those elements of $U$ that are not in $S$.

**Theorem 1.5:** Let $S$ be a finite subset of some infinite set $U$. Let $T$ be the complement of $S$ with respect to $U$. Then $T$ is infinite.

**PROOF:** Intuitively, this theorem says that if you have an infinite supply of something ($U$), and you take a finite amount away ($S$), then you still have an infinite amount left. Let us begin by restating the facts of the theorem as in Fig. 1.4.

| Original Statement | New Statement |
|---|---|
| $S$ is finite | There is a integer $n$ such that $\|S\| = n$ |
| $U$ is infinite | For no integer $p$ is $\|U\| = p$ |
| $T$ is the complement of $S$ | $S \cup T = U$ and $S \cap T = \emptyset$ |

Figure 1.4: Restating the givens of Theorem 1.5

We are still stuck, so we need to use a common proof technique called "proof by contradiction." In this proof method, to be discussed further in Section 1.3.3, we assume that the conclusion is false. We then use that assumption, together with parts of the hypothesis, to prove the opposite of one of the given statements of the hypothesis. We have then shown that it is impossible for all parts of the hypothesis to be true and for the conclusion to be false at the same time. The only possibility that remains is for the conclusion to be true whenever the hypothesis is true. That is, the theorem is true.

In the case of Theorem 1.5, the contradiction of the conclusion is "$T$ is finite." Let us assume $T$ is finite, along with the statement of the hypothesis that says $S$ is finite; i.e., $\|S\| = n$ for some integer $n$. Similarly, we can restate the assumption that $T$ is finite as $\|T\| = m$ for some integer $m$.

Now one of the given statements tells us that $S \cup T = U$, and $S \cap T = \emptyset$. That is, the elements of $U$ are exactly the elements of $S$ and $T$. Thus, there must be $n + m$ elements of $U$. Since $n + m$ is an integer, and we have shown $\|U\| = n + m$, it follows that $U$ is finite. More precisely, we showed the number of elements in $U$ is some integer, which is the definition of "finite." But the statement that $U$ is finite contradicts the given statement that $U$ is infinite. We have thus used the contradiction of our conclusion to prove the contradiction

of one of the given statements of the hypothesis, and by the principle of "proof by contradiction" we may conclude the theorem is true. □

Proofs do not have to be so wordy. Having seen the ideas behind the proof, let us reprove the theorem in a few lines.

**PROOF**: (of Theorem 1.5) We know that $S \cup T = U$ and $S$ and $T$ are disjoint, so $\|S\| + \|T\| = \|U\|$. Since $S$ is finite, $\|S\| = n$ for some integer $n$, and since $U$ is infinite, there is no integer $p$ such that $\|U\| = p$. So assume that $T$ is finite; that is, $\|T\| = m$ for some integer $m$. Then $\|U\| = \|S\| + \|T\| = n + m$, which contradicts the given statement that there is no integer $p$ equal to $\|U\|$. □

### 1.2.3 Other Theorem Forms

The "if-then" form of theorem is most common in typical areas of mathematics. However, we see other kinds of statements proved as theorems also. In this section, we shall examine the most common forms of statement and what we usually need to do to prove them.

#### Ways of Saying "If-Then"

First, there are a number of kinds of theorem statements that look different from a simple "if $H$ then $C$" form, but are in fact saying the same thing: if hypothesis $H$ is true for a given value of the parameter(s), then the conclusion $C$ is true for the same value. Here are some of the other ways in which "if $H$ then $C$" might appear.

1. $H$ implies $C$.
2. $H$ only if $C$.
3. $C$ if $H$.
4. Whenever $H$ holds, $C$ follows.

We also see many variants of form (4), such as "if $H$ holds, then $C$ follows," or "whenever $H$ holds, $C$ holds."

**Example 1.6:** The statement of Theorem 1.3 would appear in these four forms as:

1. $x \geq 4$ implies $2^x \geq x^2$.
2. $x \geq 4$ only if $2^x \geq x^2$.
3. $2^x \geq x^2$ if $x \geq 4$.
4. Whenever $x \geq 4$, $2^x \geq x^2$ follows.

□

### Statements With Quantifiers

Many theorems involve statements that use the *quantifiers* "for all" and "there exists," or similar variations, such as "for every" instead of "for all." The order in which these quantifiers appear affects what the statement means. It is often helpful to see statements with more than one quantifier as a "game" between two players — for-all and there-exists — who take turns specifying values for the parameters mentioned in the theorem. "For-all" must consider all possible choices, so for-all's choices are generally left as variables. However, "there-exists" only has to pick one value, which may depend on the values picked by the players previously. The order in which the quantifiers appear in the statement determines who goes first. If the last player to make a choice can always find some allowable value, then the statement is true.

For example, consider an alternative definition of "infinite set": set $S$ is *infinite* if and only if for all integers $n$, there exists a subset $T$ of $S$ with exactly $n$ members. Here, "for-all" precedes "there-exists," so we must consider an arbitrary integer $n$. Now, "there-exists" gets to pick a subset $T$, and may use the knowledge of $n$ to do so. For instance, if $S$ were the set of integers, "there-exists" could pick the subset $T = \{1, 2, \ldots, n\}$ and thereby succeed regardless of $n$. That is a proof that the set of integers is infinite.

The following statement looks like the definition of "infinite," but is *incorrect* because it reverses the order of the quantifiers: "there exists a subset $T$ of set $S$ such that for all $n$, set $T$ has exactly $n$ members." Now, given a set $S$ such as the integers, player "there-exists" can pick any set $T$; say $\{1, 2, 5\}$ is picked. For this choice, player "for-all" must show that $T$ has $n$ members for *every* possible $n$. However, "for-all" cannot do so. For instance, it is false for $n = 4$, or in fact for any $n \neq 3$.

In addition, in formal logic one often sees the operator $\rightarrow$ in place of "if-then." That is, the statement "if $H$ then $C$" could appear as $H \rightarrow C$ in some mathematical literature; we shall not use it here.

### If-And-Only-If Statements

Sometimes, we find a statement of the form "$A$ if and only if $B$." Other forms of this statement are "$A$ iff $B$,"[1] "$A$ is equivalent to $B$," or "$A$ exactly when $B$." This statement is actually two if-then statements: "if $A$ then $B$," and "if $B$ then $A$." We prove "$A$ if and only if $B$" by proving these two statements:

[1] Iff, short for "if and only if," is a non-word that is used in some mathematical treatises for succinctness.

### How Formal Do Proofs Have to Be?

The answer to this question is not easy. The bottom line regarding proofs is that their purpose is to convince someone, whether it is a grader of your classwork or yourself, about the correctness of a strategy you are using in your code. If it is convincing, then it is enough; if it fails to convince the "consumer" of the proof, then the proof has left out too much.

Part of the uncertainty regarding proofs comes from the different knowledge that the consumer may have. Thus, in Theorem 1.4, we assumed you knew all about arithmetic, and would believe a statement like "if $y \geq 1$ then $y^2 \geq 1$." If you were not familiar with arithmetic, we would have to prove that statement by some steps in our deductive proof.

However, there are certain things that are required in proofs, and omitting them surely makes the proof inadequate. For instance, any deductive proof that uses statements which are not justified by the given or previous statements, cannot be adequate. When doing a proof of an "if and only if" statement, we must surely have one proof for the "if" part and another proof for the "only-if" part. As an additional example, inductive proofs (discussed in Section 1.4) require proofs of the basis and induction parts.

1. The *if part*: "if $B$ then $A$," and
2. The *only-if part*: "if $A$ then $B$," which is often stated in the equivalent form "$A$ only if $B$."

The proofs can be presented in either order. In many theorems, one part is decidedly easier than the other, and it is customary to present the easy direction first and get it out of the way.

In formal logic, one may see the operator $\leftrightarrow$ or $\equiv$ to denote an "if-and-only-if" statement. That is, $A \equiv B$ and $A \leftrightarrow B$ mean the same as "$A$ if and only if $B$."

When proving an if-and-only-if statement, it is important to remember that you must prove both the "if" and "only-if" parts. Sometimes, you will find it helpful to break an if-and-only-if into a succession of several equivalences. That is, to prove "$A$ if and only if $B$," you might first prove "$A$ if and only if $C$," and then prove "$C$ if and only if $B$." That method works, as long as you remember that each if-and-only-if step must be proved in both directions. Proving any one step in only one of the directions invalidates the entire proof.

The following is an example of a simple if-and-only-if proof. It uses the notations:

1. $\lfloor x \rfloor$, the *floor* of real number $x$, is the greatest integer equal to or less than $x$.

2. $\lceil x \rceil$, the *ceiling* of real number $x$, is the least integer equal to or greater than $x$.

**Theorem 1.7:** Let $x$ be a real number. Then $\lfloor x \rfloor = \lceil x \rceil$ if and only if $x$ is an integer.

**PROOF:** (Only-if part) In this part, we assume $\lfloor x \rfloor = \lceil x \rceil$ and try to prove $x$ is an integer. Using the definitions of the floor and ceiling, we notice that $\lfloor x \rfloor \leq x$, and $\lceil x \rceil \geq x$. However, we are given that $\lfloor x \rfloor = \lceil x \rceil$. Thus, we may substitute the floor for the ceiling in the first inequality to conclude $\lceil x \rceil \leq x$. Since both $\lceil x \rceil \leq x$ and $\lceil x \rceil \geq x$ hold, we may conclude by properties of arithmetic inequalities that $\lceil x \rceil = x$. Since $\lceil x \rceil$ is always an integer, $x$ must also be an integer in this case.

(If part) Now, we assume $x$ is an integer and try to prove $\lfloor x \rfloor = \lceil x \rceil$. This part is easy. By the definitions of floor and ceiling, when $x$ is an integer, both $\lfloor x \rfloor$ and $\lceil x \rceil$ are equal to $x$, and therefore equal to each other. □

### 1.2.4 Theorems That Appear Not to Be If-Then Statements

Sometimes, we encounter a theorem that appears not to have a hypothesis. An example is the well-known fact from trigonometry:

**Theorem 1.8:** $\sin^2\theta + \cos^2\theta = 1$. □

Actually, this statement *does* have a hypothesis, and the hypothesis consists of all the statements you need to know to interpret the statement. In particular, the hidden hypothesis is that $\theta$ is an angle, and therefore the functions sine and cosine have their usual meaning for angles. From the definitions of these terms, and the Pythagorean Theorem (in a right triangle, the square of the hypotenuse equals the sum of the squares of the other two sides), you could prove the theorem. In essence, the if-then form of the theorem is really: "if $\theta$ is an angle, then $\sin^2\theta + \cos^2\theta = 1$."

## 1.3 Additional Forms of Proof

In this section, we take up several additional topics concerning how to construct proofs:

1. Proofs about sets.
2. Proofs by contradiction.
3. Proofs by counterexample.

### 1.3.1 Proving Equivalences About Sets

In automata theory, we are frequently asked to prove a theorem which says that the sets constructed in two different ways are the same sets. Often, these sets are sets of character strings, and the sets are called "languages," but in this section the nature of the sets is unimportant. If $E$ and $F$ are two expressions representing sets, the statement $E = F$ means that the two sets represented are the same. More precisely, every element in the set represented by $E$ is in the set represented by $F$, and every element in the set represented by $F$ is in the set represented by $E$.

**Example 1.9:** The *commutative law of union* says that we can take the union of two sets $R$ and $S$ in either order. That is, $R \cup S = S \cup R$. In this case, $E$ is the expression $R \cup S$ and $F$ is the expression $S \cup R$. The commutative law of union says that $E = F$. □

We can write a set-equality $E = F$ as an if-and-only-if statement: an element $x$ is in $E$ if and only if $x$ is in $F$. As a consequence, we see the outline of a proof of any statement that asserts the equality of two sets $E = F$; it follows the form of any if-and-only-if proof:

1. Proof that if $x$ is in $E$, then $x$ is in $F$.
2. Prove that if $x$ is in $F$, then $x$ is in $E$.

As an example of this proof process, let us prove the *distributive law of union over intersection*:

**Theorem 1.10:** $R \cup (S \cap T) = (R \cup S) \cap (R \cup T)$.

**PROOF:** The two set-expressions involved are $E = R \cup (S \cap T)$ and

$$F = (R \cup S) \cap (R \cup T)$$

We shall prove the two parts of the theorem in turn. In the "if" part we assume element $x$ is in $E$ and show it is in $F$. This part, summarized in Fig. 1.5, uses the definitions of union and intersection, with which we assume you are familiar.

Then, we must prove the "only-if" part of the theorem. Here, we assume $x$ is in $F$ and show it is in $E$. The steps are summarized in Fig. 1.6. Since we have now proved both parts of the if-and-only-if statement, the distributive law of union over intersection is proved. □

### 1.3.2 The Contrapositive

Every if-then statement has an equivalent form that in some circumstances is easier to prove. The *contrapositive* of the statement "if $H$ then $C$" is "if not $C$ then not $H$." A statement and its contrapositive are either both true or both false, so we can prove either to prove the other.

To see why "if $H$ then $C$" and "if not $C$ then not $H$" are logically equivalent, first observe that there are four cases to consider:

| | Statement | Justification |
|---|---|---|
| 1. | $x$ is in $R \cup (S \cap T)$ | Given |
| 2. | $x$ is in $R$ or $x$ is in $S \cap T$ | (1) and definition of union |
| 3. | $x$ is in $R$ or $x$ is in both $S$ and $T$ | (2) and definition of intersection |
| 4. | $x$ is in $R \cup S$ | (3) and definition of union |
| 5. | $x$ is in $R \cup T$ | (3) and definition of union |
| 6. | $x$ is in $(R \cup S) \cap (R \cup T)$ | (4), (5), and definition of intersection |

Figure 1.5: Steps in the "if" part of Theorem 1.10

| | Statement | Justification |
|---|---|---|
| 1. | $x$ is in $(R \cup S) \cap (R \cup T)$ | Given |
| 2. | $x$ is in $R \cup S$ | (1) and definition of intersection |
| 3. | $x$ is in $R \cup T$ | (1) and definition of intersection |
| 4. | $x$ is in $R$ or $x$ is in both $S$ and $T$ | (2), (3), and reasoning about unions |
| 5. | $x$ is in $R$ or $x$ is in $S \cap T$ | (4) and definition of intersection |
| 6. | $x$ is in $R \cup (S \cap T)$ | (5) and definition of union |

Figure 1.6: Steps in the "only-if" part of Theorem 1.10

1. $H$ and $C$ both true.

2. $H$ true and $C$ false.

3. $C$ true and $H$ false.

4. $H$ and $C$ both false.

There is only one way to make an if-then statement false; the hypothesis must be true and the conclusion false, as in case (2). For the other three cases, including case (4) where the conclusion is false, the if-then statement itself is true.

Now, consider for which cases the contrapositive "if not $C$ then not $H$" is false. In order for this statement to be false, its hypothesis (which is "not $C$") must be true, and its conclusion (which is "not $H$") must be false. But "not $C$" is true exactly when $C$ is false, and "not $H$" is false exactly when $H$ is true. These two conditions are again case (2), which shows that in each of the four cases, the original statement and its contrapositive are either both true or both false; i.e., they are logically equivalent.

### Saying "If-And-Only-If" for Sets

As we mentioned, theorems that state equivalences of expressions about sets are if-and-only-if statements. Thus, Theorem 1.10 could have been stated: an element $x$ is in $R \cup (S \cap T)$ if and only if $x$ is in

$$(R \cup S) \cap (R \cup T)$$

Another common expression of a set-equivalence is with the locution "all-and-only." For instance, Theorem 1.10 could as well have been stated "the elements of $R \cup (S \cap T)$ are all and only the elements of

$$(R \cup S) \cap (R \cup T)$$

### The Converse

Do not confuse the terms "contrapositive" and "converse." The *converse* of an if-then statement is the "other direction"; that is, the converse of "if $H$ then $C$" is "if $C$ then $H$." Unlike the contrapositive, which is logically equivalent to the original, the converse is *not* equivalent to the original statement. In fact, the two parts of an if-and-only-if proof are always some statement and its converse.

**Example 1.11:** Recall Theorem 1.3, whose statement was: "if $x \geq 4$, then $2^x \geq x^2$." The contrapositive of this statement is "if not $2^x \geq x^2$ then not $x \geq 4$." In more colloquial terms, making use of the fact that "not $a \geq b$" is the same as $a < b$, the contrapositive is "if $2^x < x^2$ then $x < 4$." □

When we are asked to prove an if-and-only-if theorem, the use of the contrapositive in one of the parts allows us several options. For instance, suppose we want to prove the set equivalence $E = F$. Instead of proving "if $x$ is in $E$ then $x$ is in $F$ and if $x$ is in $F$ then $x$ is in $E$," we could also put one direction in the contrapositive. One equivalent proof form is:

- If $x$ is in $E$ then $x$ is in $F$, and if $x$ is not in $E$ then $x$ is not in $F$.

We could also interchange $E$ and $F$ in the statement above.

### 1.3.3 Proof by Contradiction

Another way to prove a statement of the form "if $H$ then $C$" is to prove the statement

- "$H$ and not $C$ implies falsehood."

That is, start by assuming both the hypothesis $H$ and the negation of the conclusion $C$. Complete the proof by showing that something known to be false follows logically from $H$ and not $C$. This form of proof is called *proof by contradiction.*

**Example 1.12:** Recall Theorem 1.5, where we proved the if-then statement with hypothesis $H$ = "$U$ is an infinite set, $S$ is a finite subset of $U$, and $T$ is the complement of $S$ with respect to $U$." The conclusion $C$ was "$T$ is infinite." We proceeded to prove this theorem by contradiction. We assumed "not $C$"; that is, we assumed $T$ was finite.

Our proof was to derive a falsehood from $H$ and not $C$. We first showed from the assumptions that $S$ and $T$ are both finite, that $U$ also must be finite. But since $U$ is stated in the hypothesis $H$ to be infinite, and a set cannot be both finite and infinite, we have proved the logical statement "false." In logical terms, we have both a proposition $p$ ($U$ is finite) and its negation, not $p$ ($U$ is infinite). We then use the fact that "$p$ and not $p$" is logically equivalent to "false." □

To see why proofs by contradiction are logically correct, recall from Section 1.3.2 that there are four combinations of truth values for $H$ and $C$. Only the second case, $H$ true and $C$ false, makes the statement "if $H$ then $C$" false. By showing that $H$ and not $C$ leads to falsehood, we are showing that case 2 cannot occur. Thus, the only possible combinations of truth values for $H$ and $C$ are the three combinations that make "if $H$ then $C$" true.

### 1.3.4 Counterexamples

In real life, we are not told to prove a theorem. Rather, we are faced with something that seems true -- a strategy for implementing a program for example -- and we need to decide whether or not the "theorem" is true. To resolve the question, we may alternately try to prove the theorem, and if we cannot, try to prove that its statement is false.

Theorems generally are statements about an infinite number of cases, perhaps all values of its parameters. Indeed, strict mathematical convention will only dignify a statement with the title "theorem" if it has an infinite number of cases; statements that have no parameters, or that apply to only a finite number of values of its parameter(s) are called *observations.* It is sufficient to show that an alleged theorem is false in any one case in order to show it is not a theorem. The situation is analogous to programs, since a program is generally considered to have a bug if it fails to operate correctly for even one input on which it was expected to work.

It often is easier to prove that a statement is not a theorem than to prove it *is* a theorem. As we mentioned, if $S$ is any statement, then the statement "$S$ is not a theorem" is itself a statement without parameters, and thus can

be regarded as an observation rather than a theorem. The following are two examples, first of an obvious nontheorem, and the second a statement that just misses being a theorem and that requires some investigation before resolving the question of whether it is a theorem or not.

**Alleged Theorem 1.13:** All primes are odd. (More formally, we might say: if integer $x$ is a prime, then $x$ is odd.)

**DISPROOF:** The integer 2 is a prime, but 2 is even. □

Now, let us discuss a "theorem" involving modular arithmetic. There is an essential definition that we must first establish. If $a$ and $b$ are positive integers, then $a$ mod $b$ is the remainder when $a$ is divided by $b$, that is, the unique integer $r$ between 0 and $b-1$ such that $a = qb + r$ for some integer $q$. For example, 8 mod 3 = 2, and 9 mod 3 = 0. Our first proposed theorem, which we shall determine to be false, is:

**Alleged Theorem 1.14:** There is no pair of integers $a$ and $b$ such that

$$a \bmod b = b \bmod a$$

□

When asked to do things with pairs of objects, such as $a$ and $b$ here, it is often possible to simplify the relationship between the two by taking advantage of symmetry. In this case, we can focus on the case where $a < b$, since if $b < a$ we can swap $a$ and $b$ and get the same equation as in Alleged Theorem 1.14. we must be careful, however, not to forget the third case, where $a = b$. This case turns out to be fatal to our proof attempts.

Let us assume $a < b$. Then $a$ mod $b = a$, since in the definition of $a$ mod $b$ we have $q = 0$ and $r = a$. That is, when $a < b$ we have $a = 0 \times b + a$. But $b$ mod $a < a$, since anything mod $a$ is between 0 and $a - 1$. Thus, when $a < b$, $b$ mod $a < a$ mod $b$, so $a$ mod $b = b$ mod $a$ is impossible. Using the argument of symmetry above, we also know that $a$ mod $b \neq b$ mod $a$ when $b < a$.

However, consider the third case: $a = b$. Since $x$ mod $x = 0$ for any integer $x$, we *do* have $a$ mod $b = b$ mod $a$ if $a = b$. We thus have a disproof of the alleged theorem:

**DISPROOF:** (of Alleged Theorem 1.14) Let $a = b = 2$. Then

$$a \bmod b = b \bmod a = 0$$

□

In the process of finding the counterexample, we have in fact discovered the exact conditions under which the alleged theorem holds. Here is the correct version of the theorem, and its proof.

**Theorem 1.15:** $a$ mod $b = b$ mod $a$ if and only if $a = b$.

**PROOF:** (If part) Assume $a = b$. Then as we observed above, $x \bmod x = 0$ for any integer $x$. Thus, $a \bmod b = b \bmod a = 0$ whenever $a = b$.

(Only-if part) Now, assume $a \bmod b = b \bmod a$. The best technique is a proof by contradiction, so assume in addition the negation of the conclusion; that is, assume $a \neq b$. Then since $a = b$ is eliminated, we have only to consider the cases $a < b$ and $b < a$.

We already observed above that when $a < b$, we have $a \bmod b = a$ and $b \bmod a < a$. Thus, these statements, in conjunction with the hypothesis $a \bmod b = b \bmod a$ lets us derive a contradiction.

By symmetry, if $b < a$ then $b \bmod a = b$ and $a \bmod b < b$. We again derive a contradiction of the hypothesis, and conclude the only-if part is also true. We have now proved both directions and conclude that the theorem is true. □

## 1.4 Inductive Proofs

There is a special form of proof, called "inductive," that is essential when dealing with recursively defined objects. Many of the most familiar inductive proofs deal with integers, but in automata theory, we also need inductive proofs about such recursively defined concepts as trees and expressions of various sorts, such as the regular expressions that were mentioned briefly in Section 1.1.2. In this section, we shall introduce the subject of inductive proofs first with "simple" inductions on integers. Then, we show how to perform "structural" inductions on any recursively defined concept.

### 1.4.1 Inductions on Integers

Suppose we are given a statement $S(n)$, about an integer $n$, to prove. One common approach is to prove two things:

1. The *basis*, where we show $S(i)$ for a particular integer $i$. Usually, $i = 0$ or $i = 1$, but there are examples where we want to start at some higher $i$, perhaps because the statement $S$ is false for a few small integers.

2. The *inductive step*, where we assume $n \geq i$, where $i$ is the basis integer, and we show that "if $S(n)$ then $S(n + 1)$."

Intuitively, these two parts should convince us that $S(n)$ is true for every integer $n$ that is equal to or greater than the basis integer $i$. We can argue as follows. Suppose $S(n)$ were false for one or more of those integers. Then there would have to be a smallest value of $n$, say $j$, for which $S(j)$ is false, and yet $j \geq i$. Now $j$ could not be $i$, because we prove in the basis part that $S(i)$ is true. Thus, $j$ must be greater than $i$. We now know that $j - 1 \geq i$, and $S(j - 1)$ is true.

However, we proved in the inductive part that if $n \geq i$, then $S(n)$ implies $S(n + 1)$. Suppose we let $n = j - 1$. Then we know from the inductive step that $S(j - 1)$ implies $S(j)$. Since we also know $S(j - 1)$, we can conclude $S(j)$.

We have assumed the negation of what we wanted to prove; that is, we assumed $S(j)$ was false for some $j \geq i$. In each case, we derived a contradiction, so we have a "proof by contradiction" that $S(n)$ is true for all $n \geq i$.

Unfortunately, there is a subtle logical flaw in the above reasoning. Our assumption that we can pick a least $j \geq i$ for which $S(j)$ is false depends on our believing the principle of induction in the first place. That is, the only way to prove that we can find such a $j$ is to prove it by a method that is essentially an inductive proof. However, the "proof" discussed above makes good intuitive sense, and matches our understanding of the real world. Thus, we generally take as an integral part of our logical reasoning system:

- *The Induction Principle*: If we prove $S(i)$ and we prove that for all $n \geq i$, $S(n)$ implies $S(n+1)$, then we may conclude $S(n)$ for all $n \geq i$.

The following two examples illustrate the use of the induction principle to prove theorems about integers.

**Theorem 1.16:** For all $n \geq 0$:

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \tag{1.1}$$

**PROOF**: The proof is in two parts: the basis and the inductive step; we prove each in turn.

**BASIS**: For the basis, we pick $n = 0$. It might seem surprising that the theorem even makes sense for $n = 0$, since the left side of Equation (1.1) is $\sum_{i=1}^{0}$ when $n = 0$. However, there is a general principle that when the upper limit of a sum (0 in this case) is less than the lower limit (1 here), the sum is over no terms and therefore the sum is 0. That is, $\sum_{i=1}^{0} i^2 = 0$.

The right side of Equation (1.1) is also 0, since $0 \times (0+1) \times (2 \times 0+1)/6 = 0$. Thus, Equation (1.1) is true when $n = 0$.

**INDUCTION**: Now, assume $n \geq 0$. We must prove the inductive step, that Equation (1.1) implies the same formula with $n+1$ substituted for $n$. The latter formula is

$$\sum_{i=1}^{[n+1]} i^2 = \frac{[n+1]([n+1]+1)(2[n+1]+1)}{6} \tag{1.2}$$

We may simplify Equations (1.1) and (1.2) by expanding the sums and products on the right sides. These equations become:

$$\sum_{i=1}^{n} i^2 = (2n^3 + 3n^2 + n)/6 \tag{1.3}$$

$$\sum_{i=1}^{n+1} i^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.4}$$

We need to prove (1.4) using (1.3), since in the induction principle, these are statements $S(n+1)$ and $S(n)$, respectively. The "trick" is to break the sum to $n+1$ on the right of (1.4) into a sum to $n$ plus the $(n+1)$st term. In that way, we can replace the sum to $n$ by the left side of (1.3) and show that (1.4) is true. These steps are as follows:

$$\left(\sum_{i=1}^{n} i^2\right) + (n+1)^2 = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.5}$$

$$(2n^3 + 3n^2 + n)/6 + (n^2 + 2n + 1) = (2n^3 + 9n^2 + 13n + 6)/6 \tag{1.6}$$

The final verification that (1.6) is true requires only simple polynomial algebra on the left side to show it is identical to the right side. □

**Example 1.17:** In the next example, we prove Theorem 1.3 from Section 1.2.1. Recall this theorem states that if $x \geq 4$, then $2^x \geq x^2$. We gave an informal proof based on the idea that the ratio $x^2/2^x$ shrinks as $x$ grows above 4. We can make the idea precise if we prove the statement $2^x \geq x^2$ by induction on $x$, starting with a basis of $x = 4$. Note that the statement is actually false for $x < 4$.

**BASIS**: If $x = 4$, then $2^x$ and $x^2$ are both 16. Thus, $2^4 \geq 4^2$ holds.

**INDUCTION**: Suppose for some $x \geq 4$ that $2^x \geq x^2$. With this statement as the hypothesis, we need to prove the same statement, with $x+1$ in place of $x$, that is, $2^{[x+1]} \geq [x+1]^2$. These are the statements $S(x)$ and $S(x+1)$ in the induction principle; the fact that we are using $x$ instead of $n$ as the parameter should not be of concern; $x$ or $n$ is just a local variable.

As in Theorem 1.16, we should rewrite $S(x+1)$ so it can make use of $S(x)$. In this case, we can write $2^{[x+1]}$ as $2 \times 2^x$. Since $S(x)$ tells us that $2^x \geq x^2$, we can conclude that $2^{x+1} = 2 \times 2^x \geq 2x^2$.

But we need something different; we need to show that $2^{x+1} \geq (x+1)^2$. One way to prove this statement is to prove that $2x^2 \geq (x+1)^2$ and then use the transitivity of $\geq$ to show $2^{x+1} \geq 2x^2 \geq (x+1)^2$. In our proof that

$$2x^2 \geq (x+1)^2 \tag{1.7}$$

we may use the assumption that $x \geq 4$. Begin by simplifying (1.7):

$$x^2 \geq 2x + 1 \tag{1.8}$$

Divide (1.8) by $x$, to get:

### Integers as Recursively Defined Concepts

We mentioned that inductive proofs are useful when the subject matter is recursively defined. However, our first examples were inductions on integers, which we do not normally think of as "recursively defined." However, there is a natural, recursive definition of when a number is a nonnegative integer, and this definition does indeed match the way inductions on integers proceed: from objects defined first, to those defined later.

**BASIS**: 0 is an integer.

**INDUCTION**: If $n$ is an integer, then so is $n + 1$.

$$x \geq 2 + \frac{1}{x} \tag{1.9}$$

Since $x \geq 4$, we know $1/x \leq 1/4$. Thus, the left side of (1.9) is at least 4, and the right side is at most 2.25. We have thus proved the truth of (1.9). Therefore, Equations (1.8) and (1.7) are also true. Equation (1.7) in turn gives us $2x^2 \geq (x+1)^2$ for $x \geq 4$ and lets us prove statement $S(x+1)$, which we recall was $2^{x+1} \geq (x+1)^2$. □

## 1.4.2 More General Forms of Integer Inductions

Sometimes an inductive proof is made possible only by using a more general scheme than the one proposed in Section 1.4.1, where we proved a statement $S$ for one basis value and then proved that "if $S(n)$ then $S(n+1)$." Two important generalizations of this scheme are:

1. We can use several basis cases. That is, we prove $S(i), S(i+1), \ldots, S(j)$ for some $j > i$.

2. In proving $S(n+1)$, we can use the truth of all the statements

   $$S(i), S(i+1), \ldots, S(n)$$

   rather than just using $S(n)$. Moreover, if we have proved basis cases up to $S(j)$, then we can assume $n \geq j$, rather than just $n \geq i$.

The conclusion to be made from this basis and inductive step is that $S(n)$ is true for all $n \geq i$.

**Example 1.18**: The following example will illustrate the potential of both principles. The statement $S(n)$ we would like to prove is that if $n \geq 8$, then $n$ can be written as a sum of 3's and 5's. Notice, incidentally, that 7 cannot be written as a sum of 3's and 5's.

**BASIS**: The basis cases are $S(8)$, $S(9)$, and $S(10)$. The proofs are $8 = 3 + 5$, $9 = 3 + 3 + 3$, and $10 = 5 + 5$, respectively.

**INDUCTION**: Assume that $n \geq 10$ and that $S(8), S(9), \ldots, S(n)$ are true. We must prove $S(n+1)$ from these given facts. Our strategy is to subtract 3 from $n+1$, observe that this number must be writable as a sum of 3's and 5's, and add one more 3 to the sum to get a way to write $n+1$.

More formally, observe that $n - 2 \geq 8$, so we may assume $S(n-2)$. That is, $n - 2 = 3a + 5b$ for some integers $a$ and $b$. Then $n + 1 = 3 + 3a + 5b$, so $n+1$ can be written as the sum of $a+1$ 3's and $b$ 5's. That proves $S(n+1)$ and concludes the inductive step. □

### 1.4.3 Structural Inductions

In automata theory, there are several recursively defined structures about which we need to prove statements. The familiar notions of trees and expressions are important examples. Like inductions, all recursive definitions have a basis case, where one or more elementary structures are defined, and an inductive step, where more complex structures are defined in terms of previously defined structures.

**Example 1.19**: Here is the recursive definition of a tree:

**BASIS**: A single node is a tree, and that node is the *root* of the tree.

**INDUCTION**: If $T_1, T_2, \ldots, T_k$ are trees, then we can form a new tree as follows:

1. Begin with a new node $N$, which is the root of the tree.
2. Add copies of all the trees $T_1, T_2, \ldots, T_k$.
3. Add edges from node $N$ to the roots of each of the trees $T_1, T_2, \ldots, T_k$.

Figure 1.7 shows the inductive construction of a tree with root $N$ from $k$ smaller trees. □

**Example 1.20**: Here is another recursive definition. This time we define *expressions* using the arithmetic operators $+$ and $*$, with both numbers and variables allowed as operands.

**BASIS**: Any number or letter (i.e., a variable) is an expression.

**INDUCTION**: If $E$ and $F$ are expressions, then so are $E + F$, $E * F$, and $(E)$.

For example, both 2 and $x$ are expressions by the basis. The inductive step tells us $x + 2$, $(x + 2)$, and $2 * (x + 2)$ are all expressions. Notice how each of these expressions depends on the previous ones being expressions. □

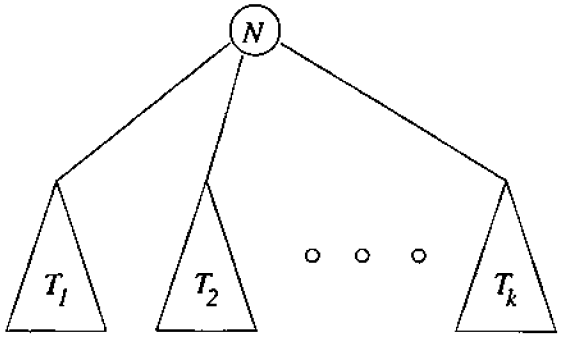


Figure 1.7: Inductive construction of a tree

## Intuition Behind Structural Induction

We can suggest informally why structural induction is a valid proof method. Imagine the recursive definition establishing, one at a time, that certain structures $X_1, X_2, \ldots$ meet the definition. The basis elements come first, and the fact that $X_i$ is in the defined set of structures can only depend on the membership in the defined set of structures that precede $X_i$ on the list. Viewed this way, a structural induction is nothing but an induction on integer $n$ of the statement $S(X_n)$. This induction may be of the generalized form discussed in Section 1.4.2, with multiple basis cases and an inductive step that uses all previous instances of the statement. However, we should remember, as explained in Section 1.4.1, that this intuition is not a formal proof, and in fact we must assume the validity this induction principle as we did the validity of the original induction principle of that section.

When we have a recursive definition, we can prove theorems about it using the following proof form, which is called *structural induction*. Let $S(X)$ be a statement about the structures $X$ that are defined by some particular recursive definition.

1. As a basis, prove $S(X)$ for the basis structure(s) $X$.
2. For the inductive step, take a structure $X$ that the recursive definition says is formed from $Y_1, Y_2, \ldots, Y_k$. Assume that the statements $S(Y_1), S(Y_2), \ldots, S(Y_k)$, and use these to prove $S(X)$.

Our conclusion is that $S(X)$ is true for all $X$. The next two theorems are examples of facts that can be proved about trees and expressions.

**Theorem 1.21:** Every tree has one more node than it has edges.

**PROOF**: The formal statement $S(T)$ we need to prove by structural induction is: "if $T$ is a tree, and $T$ has $n$ nodes and $e$ edges, then $n = e + 1$."

**BASIS**: The basis case is when $T$ is a single node. Then $n = 1$ and $e = 0$, so the relationship $n = e + 1$ holds.

**INDUCTION**: Let $T$ be a tree built by the inductive step of the definition, from root node $N$ and $k$ smaller trees $T_1, T_2, \ldots, T_k$. We may assume that the statements $S(T_i)$ hold for $i = 1, 2, \ldots, k$. That is, let $T_i$ have $n_i$ nodes and $e_i$ edges; then $n_i = e_i + 1$.

The nodes of $T$ are node $N$ and all the nodes of the $T_i$'s. There are thus $1 + n_1 + n_2 + \cdots + n_k$ nodes in $T$. The edges of $T$ are the $k$ edges we added explicitly in the inductive definition step, plus the edges of the $T_i$'s. Hence, $T$ has

$$k + e_1 + e_2 + \cdots + e_k \tag{1.10}$$

edges. If we substitute $e_i + 1$ for $n_i$ in the count of the number of nodes of $T$ we find that $T$ has

$$1 + [e_1 + 1] + [e_2 + 1] + \cdots + [e_k + 1] \tag{1.11}$$

nodes. Since there are $k$ of the "+1" terms in (1.10), we can regroup (1.11) as

$$k + 1 + e_1 + e_2 + \cdots + e_k \tag{1.12}$$

This expression is exactly 1 more than the expression of (1.10) that was given for the number of edges of $T$. Thus, $T$ has one more node than it has edges. □

**Theorem 1.22**: Every expression has an equal number of left and right parentheses.

**PROOF**: Formally, we prove the statement $S(G)$ about any expression $G$ that is defined by the recursion of Example 1.20: the numbers of left and right parentheses in $G$ are the same.

**BASIS**: If $G$ is defined by the basis, then $G$ is a number or variable. These expressions have 0 left parentheses and 0 right parentheses, so the numbers are equal.

**INDUCTION**: There are three rules whereby expression $G$ may have been constructed according to the inductive step in the definition:

1. $G = E + F$.
2. $G = E * F$.
3. $G = (E)$.

We may assume that $S(E)$ and $S(F)$ are true; that is, $E$ has the same number of left and right parentheses, say $n$ of each, and $F$ likewise has the same number of left and right parentheses, say $m$ of each. Then we can compute the numbers of left and right parentheses in $G$ for each of the three cases, as follows:

1. If $G = E + F$, then $G$ has $n + m$ left parentheses and $n + m$ right parentheses; $n$ of each come from $E$ and $m$ of each come from $F$.

2. If $G = E * F$, the count of parentheses for $G$ is again $n + m$ of each, for the same reason as in case (1).

3. If $G = (E)$, then there are $n+1$ left parentheses in $G$ — one left parenthesis is explicitly shown, and the other $n$ are present in $E$. Likewise, there are $n + 1$ right parentheses in $G$; one is explicit and the other $n$ are in $E$.

In each of the three cases, we see that the numbers of left and right parentheses in $G$ are the same. This observation completes the inductive step and completes the proof. □

### 1.4.4 Mutual Inductions

Sometimes, we cannot prove a single statement by induction, but rather need to prove a group of statements $S_1(n), S_2(n), \ldots, S_k(n)$ together by induction on $n$. Automata theory provides many such situations. In Example 1.23 we sample the common situation where we need to explain what an automaton does by proving a group of statements, one for each state. These statements tell under what sequences of inputs the automaton gets into each of the states.

Strictly speaking, proving a group of statements is no different from proving the *conjunction* (logical AND) of all the statements. For instance, the group of statements $S_1(n), S_2(n), \ldots, S_k(n)$ could be replaced by the single statement $S_1(n)$ AND $S_2(n)$ AND $\cdots$ AND $S_k(n)$. However, when there are really several independent statements to prove, it is generally less confusing to keep the statements separate and to prove them all in their own parts of the basis and inductive steps. We call this sort of proof *mutual induction*. An example will illustrate the necessary steps for a mutual recursion.

**Example 1.23:** Let us revisit the on/off switch, which we represented as an automaton in Example 1.1. The automaton itself is reproduced as Fig. 1.8. Since pushing the button switches the state between *on* and *off*, and the switch starts out in the *off* state, we expect that the following statements will together explain the operation of the switch:

$S_1(n)$: The automaton is in state *off* after $n$ pushes if and only if $n$ is even.

$S_2(n)$: The automaton is in state *on* after $n$ pushes if and only if $n$ is odd.

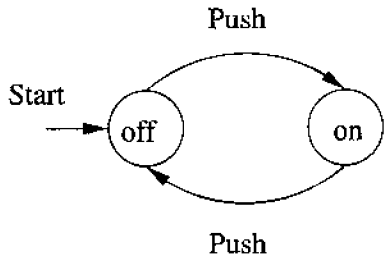


Figure 1.8: Repeat of the automaton of Fig. 1.1

We might suppose that $S_1$ implies $S_2$ and vice-versa, since we know that a number $n$ cannot be both even and odd. However, what is not always true about an automaton is that it is in one and only one state. It happens that the automaton of Fig. 1.8 is always in exactly one state, but that fact must be proved as part of the mutual induction.

We give the basis and inductive parts of the proofs of statements $S_1(n)$ and $S_2(n)$ below. The proofs depend on several facts about odd and even integers: if we add or subtract 1 from an even integer, we get an odd integer, and if we add or subtract 1 from an odd integer we get an even integer.

**BASIS**: For the basis, we choose $n = 0$. Since there are two statements, each of which must be proved in both directions (because $S_1$ and $S_2$ are each "if-and-only-if" statements), there are actually four cases to the basis, and four cases to the induction as well.

1. [$S_1$; If] Since 0 is in fact even, we must show that after 0 pushes, the automaton of Fig. 1.8 is in state *off*. Since that is the start state, the automaton is indeed in state *off* after 0 pushes.

2. [$S_1$; Only-if] The automaton is in state *off* after 0 pushes, so we must show that 0 is even. But 0 is even by definition of "even," so there is nothing more to prove.

3. [$S_2$; If] The hypothesis of the "if" part of $S_2$ is that 0 is odd. Since this hypothesis $H$ is false, any statement of the form "if $H$ then $C$" is true, as we discussed in Section 1.3.2. Thus, this part of the basis also holds.

4. [$S_2$; Only-if] The hypothesis, that the automaton is in state *on* after 0 pushes, is also false, since the only way to get to state *on* is by following an arc labeled *Push*, which requires that the button be pushed at least once. Since the hypothesis is false, we can again conclude that the if-then statement is true.

**INDUCTION**: Now, we assume that $S_1(n)$ and $S_2(n)$ are true, and try to prove $S_1(n+1)$ and $S_2(n+1)$. Again, the proof separates into four parts.

1. [$S_1(n+1)$; If] The hypothesis for this part is that $n+1$ is even. Thus, $n$ is odd. The "if" part of statement $S_2(n)$ says that after $n$ pushes, the automaton is in state *on*. The arc from *on* to *off* labeled *Push* tells us that the $(n+1)$st push will cause the automaton to enter state *off*. That completes the proof of the "if" part of $S_1(n+1)$.

2. [$S_1(n+1)$; Only-if] The hypothesis is that the automaton is in state *off* after $n+1$ pushes. Inspecting the automaton of Fig. 1.8 tells us that the only way to get to state *off* after one or more moves is to be in state *on* and receive an input *Push*. Thus, if we are in state *off* after $n+1$ pushes, we must have been in state *on* after $n$ pushes. Then, we may use the "only-if" part of statement $S_2(n)$ to conclude that $n$ is odd. Consequently, $n+1$ is even, which is the desired conclusion for the only-if portion of $S_1(n+1)$.

3. [$S_2(n+1)$; If] This part is essentially the same as part (1), with the roles of statements $S_1$ and $S_2$ exchanged, and with the roles of "odd" and "even" exchanged. The reader should be able to construct this part of the proof easily.

4. [$S_2(n+1)$; Only-if] This part is essentially the same as part (2), with the roles of statements $S_1$ and $S_2$ exchanged, and with the roles of "odd" and "even" exchanged.

□

We can abstract from Example 1.23 the pattern for all mutual inductions:

- Each of the statements must be proved separately in the basis and in the inductive step.
- If the statements are "if-and-only-if," then both directions of each statement must be proved, both in the basis and in the induction.

## 1.5 The Central Concepts of Automata Theory

In this section we shall introduce the most important definitions of terms that pervade the theory of automata. These concepts include the "alphabet" (a set of symbols), "strings" (a list of symbols from an alphabet), and "language" (a set of strings from the same alphabet).

### 1.5.1 Alphabets

An *alphabet* is a finite, nonempty set of symbols. Conventionally, we use the symbol $\Sigma$ for an alphabet. Common alphabets include:

1. $\Sigma = \{0, 1\}$, the *binary* alphabet.

2. $\Sigma = \{a, b, \ldots, z\}$, the set of all lower-case letters.

3. The set of all ASCII characters, or the set of all printable ASCII characters.

## 1.5.2 Strings

A *string* (or sometimes *word*) is a finite sequence of symbols chosen from some alphabet. For example, 01101 is a string from the binary alphabet $\Sigma = \{0, 1\}$. The string 111 is another string chosen from this alphabet.

### The Empty String

The *empty string* is the string with zero occurrences of symbols. This string, denoted $\epsilon$, is a string that may be chosen from any alphabet whatsoever.

### Length of a String

It is often useful to classify strings by their *length*, that is, the number of positions for symbols in the string. For instance, 01101 has length 5. It is common to say that the length of a string is "the number of symbols" in the string; this statement is colloquially accepted but not strictly correct. Thus, there are only two symbols, 0 and 1, in the string 01101, but there are five *positions* for symbols, and its length is 5. However, you should generally expect that "the number of symbols" can be used when "number of positions" is meant.

The standard notation for the length of a string $w$ is $|w|$. For example, $|011| = 3$ and $|\epsilon| = 0$.

### Powers of an Alphabet

If $\Sigma$ is an alphabet, we can express the set of all strings of a certain length from that alphabet by using an exponential notation. We define $\Sigma^k$ to be the set of strings of length $k$, each of whose symbols is in $\Sigma$.

**Example 1.24:** Note that $\Sigma^0 = \{\epsilon\}$, regardless of what alphabet $\Sigma$ is. That is, $\epsilon$ is the only string whose length is 0.

If $\Sigma = \{0, 1\}$, then $\Sigma^1 = \{0, 1\}$, $\Sigma^2 = \{00, 01, 10, 11\}$,

$$\Sigma^3 = \{000, 001, 010, 011, 100, 101, 110, 111\}$$

and so on. Note that there is a slight confusion between $\Sigma$ and $\Sigma^1$. The former is an alphabet; its members 0 and 1 are symbols. The latter is a set of strings; its members are the strings 0 and 1, each of which is of length 1. We shall not try to use separate notations for the two sets, relying on context to make it clear whether $\{0, 1\}$ or similar sets are alphabets or sets of strings. □

### Type Convention for Symbols and Strings

Commonly, we shall use lower-case letters at the beginning of the alphabet (or digits) to denote symbols, and lower-case letters near the end of the alphabet, typically $w$, $x$, $y$, and $z$, to denote strings. You should try to get used to this convention, to help remind you of the types of the elements being discussed.

The set of all strings over an alphabet $\Sigma$ is conventionally denoted $\Sigma^*$. For instance, $\{0,1\}^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \ldots\}$. Put another way,

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \Sigma^2 \cup \cdots$$

Sometimes, we wish to exclude the empty string from the set of strings. The set of nonempty strings from alphabet $\Sigma$ is denoted $\Sigma^+$. Thus, two appropriate equivalences are:

- $\Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \Sigma^3 \cup \cdots$.
- $\Sigma^* = \Sigma^+ \cup \{\epsilon\}$.

#### Concatenation of Strings

Let $x$ and $y$ be strings. Then $xy$ denotes the *concatenation* of $x$ and $y$, that is, the string formed by making a copy of $x$ and following it by a copy of $y$. More precisely, if $x$ is the string composed of $i$ symbols $x = a_1 a_2 \cdots a_i$ and $y$ is the string composed of $j$ symbols $y = b_1 b_2 \cdots b_j$, then $xy$ is the string of length $i + j$: $xy = a_1 a_2 \cdots a_i b_1 b_2 \cdots b_j$.

**Example 1.25:** Let $x = 01101$ and $y = 110$. Then $xy = 01101110$ and $yx = 11001101$. For any string $w$, the equations $\epsilon w = w\epsilon = w$ hold. That is, $\epsilon$ is the *identity for concatenation*, since when concatenated with any string it yields the other string as a result (analogously to the way 0, the identity for addition, can be added to any number $x$ and yields $x$ as a result). □

### 1.5.3 Languages

A set of strings all of which are chosen from some $\Sigma^*$, where $\Sigma$ is a particular alphabet, is called a *language*. If $\Sigma$ is an alphabet, and $L \subseteq \Sigma^*$, then $L$ is a *language over* $\Sigma$. Notice that a language over $\Sigma$ need not include strings with all the symbols of $\Sigma$, so once we have established that $L$ is a language over $\Sigma$, we also know it is a language over any alphabet that is a superset of $\Sigma$.

The choice of the term "language" may seem strange. However, common languages can be viewed as sets of strings. An example is English, where the

collection of legal English words is a set of strings over the alphabet that consists of all the letters. Another example is C, or any other programming language, where the legal programs are a subset of the possible strings that can be formed from the alphabet of the language. This alphabet is a subset of the ASCII characters. The exact alphabet may differ slightly among different programming languages, but generally includes the upper- and lower-case letters, the digits, punctuation, and mathematical symbols.

However, there are also many other languages that appear when we study automata. Some are abstract examples, such as:

1. The language of all strings consisting of $n$ 0's followed by $n$ 1's, for some $n \geq 0$: $\{\epsilon, 01, 0011, 000111, \ldots\}$.

2. The set of strings of 0's and 1's with an equal number of each:

$$\{\epsilon, 01, 10, 0011, 0101, 1001, \ldots\}$$

3. The set of binary numbers whose value is a prime:

$$\{10, 11, 101, 111, 1011, \ldots\}$$

4. $\Sigma^*$ is a language for any alphabet $\Sigma$.

5. $\emptyset$, the empty language, is a language over any alphabet.

6. $\{\epsilon\}$, the language consisting of only the empty string, is also a language over any alphabet. Notice that $\emptyset \neq \{\epsilon\}$; the former has no strings and the latter has one string.

The only important constraint on what can be a language is that all alphabets are finite. Thus languages, although they can have an infinite number of strings, are restricted to consist of strings drawn from one fixed, finite alphabet.

### 1.5.4 Problems

In automata theory, a *problem* is the question of deciding whether a given string is a member of some particular language. It turns out, as we shall see, that anything we more colloquially call a "problem" can be expressed as membership in a language. More precisely, if $\Sigma$ is an alphabet, and $L$ is a language over $\Sigma$, then the problem $L$ is:

- Given a string $w$ in $\Sigma^*$, decide whether or not $w$ is in $L$.

**Example 1.26:** The problem of testing primality can be expressed by the language $L_p$ consisting of all binary strings whose value as a binary number is a prime. That is, given a string of 0's and 1's, say "yes" if the string is the binary representation of a prime and say "no" if not. For some strings, this

### Set-Formers as a Way to Define Languages

It is common to describe a language using a "set-former":

$$\{w \mid \text{something about } w\}$$

This expression is read "the set of words $w$ such that (whatever is said about $w$ to the right of the vertical bar)." Examples are:

1. $\{w \mid w$ consists of an equal number of 0's and 1's $\}$.
2. $\{w \mid w$ is a binary integer that is prime $\}$.
3. $\{w \mid w$ is a syntactically correct C program $\}$.

It is also common to replace $w$ by some expression with parameters and describe the strings in the language by stating conditions on the parameters. Here are some examples; the first with parameter $n$, the second with parameters $i$ and $j$:

1. $\{0^n1^n \mid n \geq 1\}$. Read "the set of 0 to the $n$ 1 to the $n$ such that $n$ is greater than or equal to 1," this language consists of the strings $\{01, 0011, 000111, \ldots\}$. Notice that, as with alphabets, we can raise a single symbol to a power $n$ in order to represent $n$ copies of that symbol.
2. $\{0^i1^j \mid 0 \leq i \leq j\}$. This language consists of strings with some 0's (possibly none) followed by at least as many 1's.

decision is easy. For instance, 0011101 cannot be the representation of a prime, for the simple reason that every integer except 0 has a binary representation that begins with 1. However, it is less obvious whether the string 11101 belongs to $L_p$, so any solution to this problem will have to use significant computational resources of some kind: time and/or space, for example. □

One potentially unsatisfactory aspect of our definition of "problem" is that one commonly thinks of problems not as decision questions (is or is not the following true?) but as requests to compute or transform some input (find the best way to do this task). For instance, the task of the parser in a C compiler can be thought of as a problem in our formal sense, where one is given an ASCII string and asked to decide whether or not the string is a member of $L_c$, the set of valid C programs. However, the parser does more than decide. It produces a parse tree, entries in a symbol table and perhaps more. Worse, the compiler as a whole solves the problem of turning a C program into object code for some

### Is It a Language or a Problem?

Languages and problems are really the same thing. Which term we prefer to use depends on our point of view. When we care only about strings for their own sake, e.g., in the set $\{0^n1^n \mid n \geq 1\}$, then we tend to think of the set of strings as a language. In the last chapters of this book, we shall tend to assign "semantics" to the strings, e.g., think of strings as coding graphs, logical expressions, or even integers. In those cases, where we care more about the thing represented by the string than the string itself, we shall tend to think of a set of strings as a problem.

machine, which is far from simply answering "yes" or "no" about the validity of a program.

Nevertheless, the definition of "problems" as languages has stood the test of time as the appropriate way to deal with the important questions of complexity theory. In this theory, we are interested in proving lower bounds on the complexity of certain problems. Especially important are techniques for proving that certain problems cannot be solved in an amount of time that is less than exponential in the size of their input. It turns out that the yes/no or language-based version of known problems are just as hard in this sense, as their "solve this" versions.

That is, if we can prove it is hard to decide whether a given string belongs to the language $L_X$ of valid strings in programming language $X$, then it stands to reason that it will not be easier to translate programs in language $X$ to object code. For if it were easy to generate code, then we could run the translator, and conclude that the input was a valid member of $L_X$ exactly when the translator succeeded in producing object code. Since the final step of determining whether object code has been produced cannot be hard, we can use the fast algorithm for generating the object code to decide membership in $L_X$ efficiently. We thus contradict the assumption that testing membership in $L_X$ is hard. We have a proof by contradiction of the statement "if testing membership in $L_X$ is hard, then compiling programs in programming language $X$ is hard."

This technique, showing one problem hard by using its supposed efficient algorithm to solve efficiently another problem that is already known to be hard, is called a "reduction" of the second problem to the first. It is an essential tool in the study of the complexity of problems, and it is facilitated greatly by our notion that problems are questions about membership in a language, rather than more general kinds of questions.

## 1.6 Summary of Chapter 1

- *Finite Automata*: Finite automata involve states and transitions among states in response to inputs. They are useful for building several different kinds of software, including the lexical analysis component of a compiler and systems for verifying the correctness of circuits or protocols, for example.

- *Regular Expressions*: These are a structural notation for describing the same patterns that can be represented by finite automata. They are used in many common types of software, including tools to search for patterns in text or in file names, for instance.

- *Context-Free Grammars*: These are an important notation for describing the structure of programming languages and related sets of strings; they are used to build the parser component of a compiler.

- *Turing Machines*: These are automata that model the power of real computers. They allow us to study decidabilty, the question of what can or cannot be done by a computer. They also let us distinguish tractable problems — those that can be solved in polynomial time — from the intractable problems — those that cannot.

- *Deductive Proofs*: This basic method of proof proceeds by listing statements that are either given to be true, or that follow logically from some of the previous statements.

- *Proving If-Then Statements*: Many theorems are of the form "if (something) then (something else)." The statement or statements following the "if" are the hypothesis, and what follows "then" is the conclusion. Deductive proofs of if-then statements begin with the hypothesis, and continue with statements that follow logically from the hypothesis and previous statements, until the conclusion is proved as one of the statements.

- *Proving If-And-Only-If Statements*: There are other theorems of the form "(something) if and only if (something else)." They are proved by showing if-then statements in both directions. A similar kind of theorem claims the equality of the sets described in two different ways; these are proved by showing that each of the two sets is contained in the other.

- *Proving the Contrapositive*: Sometimes, it is easier to prove a statement of the form "if $H$ then $C$" by proving the equivalent statement: "if not $C$ then not $H$." The latter is called the contrapositive of the former.

- *Proof by Contradiction*: Other times, it is more convenient to prove the statement "if $H$ then $C$" by proving "if $H$ and not $C$ then (something known to be false)." A proof of this type is called proof by contradiction.

- *Counterexamples*: Sometimes we are asked to show that a certain statement is not true. If the statement has one or more parameters, then we can show it is false as a generality by providing just one counterexample, that is, one assignment of values to the parameters that makes the statement false.

- *Inductive Proofs*: A statement that has an integer parameter $n$ can often by proved by induction on $n$. We prove the statement is true for the basis, a finite number of cases for particular values of $n$, and then prove the inductive step: that if the statement is true for values up to $n$, then it is true for $n + 1$.

- *Structural Inductions*: In some situations, including many in this book, the theorem to be proved inductively is about some recursively defined construct, such as trees. We may prove a theorem about the constructed objects by induction on the number of steps used in its construction. This type of induction is referred to as structural.

- *Alphabets*: An alphabet is any finite set of symbols.

- *Strings*: A string is a finite-length sequence of symbols.

- *Languages and Problems*: A language is a (possibly infinite) set of strings, all of which choose their symbols from some one alphabet. When the strings of a language are to be interpreted in some way, the question of whether a string is in the language is sometimes called a problem.

## 1.7 References for Chapter 1

For extended coverage of the material of this chapter, including mathematical concepts underlying Computer Science, we recommend [1].

1. A. V. Aho and J. D. Ullman, *Foundations of Computer Science*, Computer Science Press, New York, 1994.