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 Σ 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 ϵ, 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 Σ 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 Σ^k to be the set of strings of length k, each of whose symbols is in Σ.

Example 1.24: Note that $\Sigma^0 = \{\epsilon\}$, regardless of what alphabet Σ is. That is, ϵ 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 Σ and Σ^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. □

The set of all strings over an alphabet Σ is conventionally denoted Σ^*. 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 Σ is denoted Σ^+. Thus, two appropriate equivalences are:

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.

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

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 $x = a_1a_2 \cdots a_i$ and y is the string composed of j symbols $y = b_1b_2 \cdots b_j$, then xy is the string of length $i + j$: $xy = a_1a_2 \cdots a_ib_1b_2 \cdots b_j$.

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

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. Σ^* is a language for any alphabet Σ.

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 Σ is an alphabet, and L is a language over Σ, then the problem L is:

- Given a string w in Σ^*, 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 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

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.

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 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

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.

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 be 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 Gradiance Problems for Chapter 1

The following is a sample of problems that are available on-line through the Gradiance system at `www.gradiance.com/pearson`. Each of these problems is worked like conventional homework. The Gradiance system gives you four choices that sample your knowledge of the solution. If you make the wrong choice, you are given a hint or advice and encouraged to try the same problem again.

Problem 1.1: Find in the list below the expression that is the contrapositive of $A\ AND\ (NOT\ B)\ \rightarrow\ C\ OR\ (NOT\ D)$. Note: the hypothesis and conclusion of the choices in the list below may have some simple logical rules applied to them, in order to simplify the expressions.

Problem 1.2: To prove $A\ AND\ (NOT\ B)\ \rightarrow\ C\ OR\ (NOT\ D)$ by contradiction, which of the statements below would we prove? Note: each of the choices is simplified by pushing NOT's down until they apply only to atomic statements A through D.

Problem 1.3: Suppose we want to prove the statement $S(n)$: "If $n \geq 2$, the sum of the integers 2 through n is $(n+2)(n-1)/2$" by induction on n. To prove the inductive step, we can make use of the fact that

$$2 + 3 + 4 + \ldots + (n+1) = (2 + 3 + 4 + \ldots + n) + (n+1)$$

Find, in the list below an equality that we may prove to conclude the inductive part.

Problem 1.4: The length of the string X [shown on-line by the Gradiance system from a stock of choices] is:

Problem 1.5: What is the concatenation of X and Y? [strings shown on-line by the Gradiance system from a stock of choices]

Problem 1.6: The binary string X [shown on-line by the Gradiance system] is a member of which of the following problems? Remember, a "problem" is a language whose strings represent the cases of a problem that have the answer "yes." In this question, you should assume that all languages are sets of binary strings interpreted as base-2 integers. The exception is the problem of finding *palindromes*, which are strings that are identical when reversed, like 0110110, regardless of their numerical value.

1.8 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.

Chapter 2

Finite Automata

This chapter introduces the class of languages known as "regular languages." These languages are exactly the ones that can be described by finite automata, which we sampled briefly in Section 1.1.1. After an extended example that will provide motivation for the study to follow, we define finite automata formally.

As was mentioned earlier, a finite automaton has a set of states, and its "control" moves from state to state in response to external "inputs." One of the crucial distinctions among classes of finite automata is whether that control is "deterministic," meaning that the automaton cannot be in more than one state at any one time, or "nondeterministic," meaning that it may be in several states at once. We shall discover that adding nondeterminism does not let us define any language that cannot be defined by a deterministic finite automaton, but there can be substantial efficiency in describing an application using a nondeterministic automaton. In effect, nondeterminism allows us to "program" solutions to problems using a higher-level language. The nondeterministic finite automaton is then "compiled," by an algorithm we shall learn in this chapter, into a deterministic automaton that can be "executed" on a conventional computer.

We conclude the chapter with a study of an extended nondeterministic automaton that has the additional choice of making a transition from one state to another spontaneously, i.e., on the empty string as "input." These automata also accept nothing but the regular languages. However, we shall find them quite important in Chapter 3, when we study regular expressions and their equivalence to automata.

The study of the regular languages continues in Chapter 3. There, we introduce another important way to describe regular languages: the algebraic notation known as regular expressions. After discussing regular expressions, and showing their equivalence to finite automata, we use both automata and regular expressions as tools in Chapter 4 to show certain important properties of the regular languages. Examples of such properties are the "closure" properties, which allow us to claim that one language is regular because one or more

other languages are known to be regular, and "decision" properties. The latter are algorithms to answer questions about automata or regular expressions, e.g., whether two automata or expressions represent the same language.

2.1 An Informal Picture of Finite Automata

In this section, we shall study an extended example of a real-world problem whose solution uses finite automata in an important role. We investigate protocols that support "electronic money" — files that a customer can use to pay for goods on the internet, and that the seller can receive with assurance that the "money" is real. The seller must know that the file has not been forged, nor has it been copied and sent to the seller, while the customer retains a copy of the same file to spend again.

The nonforgeability of the file is something that must be assured by a bank and by a cryptography policy. That is, a third player, the bank, must issue and encrypt the "money" files, so that forgery is not a problem. However, the bank has a second important job: it must keep a database of all the valid money that it has issued, so that it can verify to a store that the file it has received represents real money and can be credited to the store's account. We shall not address the cryptographic aspects of the problem, nor shall we worry about how the bank can store and retrieve what could be billions of "electronic dollar bills." These problems are not likely to represent long-term impediments to the concept of electronic money, and examples of its small-scale use have existed since the late 1990's.

However, in order to use electronic money, protocols need to be devised to allow the manipulation of the money in a variety of ways that the users want. Because monetary systems always invite fraud, we must verify whatever policy we adopt regarding how money is used. That is, we need to prove the only things that can happen are things we intend to happen — things that do not allow an unscrupulous user to steal from others or to "manufacture" money. In the balance of this section, we shall introduce a very simple example of a (bad) electronic-money protocol, model it with finite automata, and show how constructions on automata can be used to verify protocols (or, in this case, to discover that the protocol has a bug).

2.1.1 The Ground Rules

There are three participants: the customer, the store, and the bank. We assume for simplicity that there is only one "money" file in existence. The customer may decide to transfer this money file to the store, which will then redeem the file from the bank (i.e., get the bank to issue a new money file belonging to the store rather than the customer) and ship goods to the customer. In addition, the customer has the option to cancel the file. That is, the customer may ask the bank to place the money back in the customer's account, making the money

no longer spendable. Interaction among the three participants is thus limited to five events:

1. The customer may decide to *pay*. That is, the customer sends the money to the store.
2. The customer may decide to *cancel*. The money is sent to the bank with a message that the value of the money is to be added to the customer's bank account.
3. The store may *ship* goods to the customer.
4. The store may *redeem* the money. That is, the money is sent to the bank with a request that its value be given to the store.
5. The bank may *transfer* the money by creating a new, suitably encrypted money file and sending it to the store.

2.1.2 The Protocol

The three participants must design their behaviors carefully, or the wrong things may happen. In our example, we make the reasonable assumption that the customer cannot be relied upon to act responsibly. In particular, the customer may try to copy the money file, use it to pay several times, or both pay and cancel the money, thus getting the goods "for free."

The bank must behave responsibly, or it cannot be a bank. In particular, it must make sure that two stores cannot both redeem the same money file, and it must not allow money to be both canceled and redeemed. The store should be careful as well. In particular, it should not ship goods until it is sure it has been given valid money for the goods.

Protocols of this type can be represented as finite automata. Each state represents a situation that one of the participants could be in. That is, the state "remembers" that certain important events have happened and that others have not yet happened. Transitions between states occur when one of the five events described above occur. We shall think of these events as "external" to the automata representing the three participants, even though each participant is responsible for initiating one or more of the events. It turns out that what is important about the problem is what sequences of events can happen, not who is allowed to initiate them.

Figure 2.1 represents the three participants by automata. In that diagram, we show only the events that affect a participant. For example, the action *pay* affects only the customer and store. The bank does not know that the money has been sent by the customer to the store; it discovers that fact only when the store executes the action *redeem*.

Let us examine first the automaton (c) for the bank. The start state is state 1; it represents the situation where the bank has issued the money file in question but has not been requested either to redeem it or to cancel it. If a