A Big Ideas Approach to the Theory of Computation
CMPSCI 401: Spring, 2006

Arnold L. Rosenberg
Distinguished University Professor
Department of Computer Science
University of Massachusetts Amherst
Amherst, MA 01003, USA
`rsnbrg@cs.umass.edu`

March 8, 2006

Contents

Preface and Manifesto

Theoretical computer science in undergraduate academic curricula is traditionally partitioned into two more-or-less disjoint subjects. Under the rubric "Algorithms," one finds courses that range anywhere from "cookbooks" that present the latest and best algorithms for various important computational processes (e.g., sorting, searching, matrix multiplication) to rather abstract and mathematical courses that focus on principles and paradigms for designing and analyzing sophisticated algorithms. No matter where such a course sits along this spectrum, the material covered has rather direct connections to students' experiences with programming and data structures. Whether happy to take the course or not, a student usually understands why the course is valuable (but may not admit as much). The second subject—what one can call "Computation Theory"—is typically more difficult than the "Algorithms" course, being (at least) one level more abstract and usually significantly more demanding of mathematical sophistication. The value of such courses is less evident, not only to most students, but also to many faculties, as evidenced by the frequency with which this course is elective. (Such is the case, for instance, at UMass Amherst.) I contend that the lack of recognition for the value of the "Computation Theory" course is largely a function of what a typical incarnation of the course covers, what it does not cover, and how it presents much of the material that is covered. I believe that all three of the preceding problems can be discerned within the leading "Computation Theory" texts, including: [14, 15, 22, 23, 26, 38].

Since "Computation Theory" courses go by a broad range of titles, and since the very phrase "Computation Theory" has differing meanings for different readers, let me henceforth refer to this second course as "the abstract course." Without arguing the (de)merits of cramming the foundational underpinnings of computer science into only one semester, I do wish to offer a somewhat contrary view concerning what to teach in the abstract course and how to teach it. It should surprise no reader that I offer the current book as a possible implementation of the view I am espousing.

What do we want the abstract course to accomplish? In my opinion, the course should impart to the undergraduate computer-science student:

1. the need for theoretical underpinnings for what is predominantly an engineering disci-

pline;

2. the rudiments of the "theoretical method," as it applies to computer science;
3. an operational command of the basic mathematical tools needed to navigate one's way through (large portions of) theoretical computer science;
4. topics from the abstract branch of theoretical computer science that have a clear (to the student) path to major topics in general computer science.

As a corollary of this presumed agenda, I would advocate that an appropriately designed course in Computation Theory be mandatory for all aspiring computer scientists. With some regret, I would argue that most current curricula for the abstract course (again, as inferred from the standard texts' contents) neither focus on nor satisfy these objectives. Standard texts prescribe a two-module approach to the subject.

Module 1 comprises a smattering of topics that provide a formal-language theory approach to automata and grammars. The main justification for much of the material in this module seems to be the long history of the theory of automata and formal languages. Within the context of this module, I part ways with the major texts: in their inclusion of several topics and their omission of several others, and in the way that they present certain topics.

Module 2 (which is usually the larger one) provides an intense study of one specific topic, Complexity Theory, preceded by some background on its (historical and intellectual) precursor, Computability Theory. This is indisputably important material, which does expose aspects of the intrinsic nature of computation by (digital) computers and does establish the theoretical underpinnings of important topics relating to "Algorithms." That said, I feel that much of what is typically included in this module goes beyond what is essential for general computer science students (as opposed to aspiring theorists); moreover, these topics preclude (because of time demands) the inclusion of several topics that are more relevant to the development of embryonic computer scientists. Additionally, I am troubled by the typical presentation of much of the material via artificial, automata-theoretic models that arouse during the heyday of automata theory in the 1970's.[1]

My proposed alternative to the preceding is a "big-ideas" approach to the abstract component of theoretical computer science. Such an approach allows one to expose the student to all of the major introductory-level ideas covered by present texts and courses, while augmenting these topics with others that are (in my opinion) at least as relevant to an embryonic computer scientist. I contend that, additionally, this approach gives one a chance to expose the student to underlying mathematical ideas that do not arise within the context of the current list of topics. I thus view a "big ideas" approach as strictly improving our progress toward all four educational goals enumerated previously. We thereby (again, in my opinion)

[1]This position echoes that espoused in [10] and in the classical Computability Theory text [31].

enhance students' preparations for their future, in terms of both the material covered and the intellectual tools for thinking about that material. While my commitment to a "big-ideas" approach has philosophical origins, it has been evolving over several decades, as I taught versions of the course material in this book to both graduate and undergraduate students at NYU, Duke and UMass Amherst. Each time I have offered the course, I have made further progress toward my goal of a "big ideas" presentation of the material. My (obviously biased) perception is that my students (who are statistically *very* unlikely to become computer theorists) have been leaving the course with better perspectives and improved technical abilities, as the transition to this approach has progressed.

My dream is that this book, which has been developed around the "big ideas" philosophy will be as approachable to the "theoretical-computer-science student on the street," within the context of the abstract component of theoretical computer science, as David Harel's well-received book [11] has been within the context of the concrete portions of theoretical computer science.

I end this preface with expressions of gratitude to the many colleagues who have debated this educational approach with me and the even greater number of students who have suffered with me through the growing pains of the approach. Both groups are too numerous to list, for fear of missing important names.

Arnold L. Rosenberg

Amherst, MA

Chapter 1

Introduction

My discussion in the Preface concentrated largely on principles. What, however, are the "big ideas" that I want to center this book around? To illustrate both the intended form of the book and to describe one specific, important topic that has been (to me, inexplicably) excluded from present-day texts, an excerpt from my essay, "State" [34], which contains polemic (which the book does not) in addition to technical material.

1.1 A sampler of "big idea" topics

State. Myriad computational systems, both hardware and software, are organized as state-transition systems. Such a system reacts to discrete stimuli. At every "clean" moment, the system is in a well-defined one of its (possibly infinite number of) states. At any such moment, in response to any valid stimulus, the system goes through some process, ending up in another "clean" moment, in some well-defined state. One of the gems of Finite-Automata Theory, the Myhill-Nerode Theorem (see Section 3.3), exposes the mathematical nature of the concept of state within a state-transition system. This characterization often allows one to analyze state-transition systems, with an eye toward improving their designs and/or exposing and quantifying their limitations.

Encoding. One of the greatest intellectual breakthroughs of the 20th century—indeed, possibly of all time—was K. Gödel's 1931 Incompleteness Theorem [9], which, very informally, established that within all but the simplest logical systems, there must be "true" assertions that can not be proved. Underlying this tour de force was the rather simple (to us, today) 1874 result of G. Cantor [2] that established the fact that only simple arithmetic is needed to encode quite elaborate structures—including tuples and strings of integers—as single integers. Quite remarkably, this ability exposed the fact that even primitive formal systems contain (in a sense that can be made precise and formal) sentences that are "self-referential," in the sense that the sentence, "This sentence is false," is. It was well-known in the 19th century that self-referentiality could lead to mathematical paradoxes. Gödel showed that it can also lead to logical incompleteness. A. Turing [39], in his own intellectual breakthrough, showed soon thereafter that it also can be used to expose inherent limitations in the power of digital computing systems. The mathematical underpinnings of this work appear in Section 2.3.2. These underpinnings are fleshed out to a form that applies to Computation Theory in Sections 4.2 and 4.3.

Reduction and Completeness. The notion of encoding did not stop bearing fruit after the work of Gödel and Turing. From the 1940's into the 1970's, and beyond, tremendous advances in broad swath of computer science, from the most abstract theory to the most concrete optimization problems, have benefited from the ability to show that one computational problem, A, is "reducible" to another, B, in the sense that (each instance of) B can be encoded as (an instance of) A in such a way that a solution for (that instance of) B would provide a solution for (the corresponding instance of) A. Given a class of computational problems for which a notion of reducibility has been defined, one can, amazingly, sometimes find a problem C in the class that is a "hardest" one, in the sense that every problem in the class is reducible to C. The famous Halting Problem for Turing Machines [39] is complete for the so-called semi-decidable (or, recursively enumerable) problems. At present, the class-completeness that most affects the work lives of most computer scientists is NP-Completeness. Using pioneering work by Cook [7] which was quickly expanded on by R. Karp [18]—this work was developed essentially contemporaneously by L. Levin [21]—one is now able to show that an enormous class of (intellectually and economically) significant optimization problems are all reducible to one another, via a notion of reducibility/encoding that is built around computational efficiency. Many (most?) computer scientists would count the resulting "P vs. NP" problem as the most important general problem facing comuter science today. Reducibility and Completeness underlie the material in Sections 4.4 and 4.5 and in Chapter 5.

Nondeterminism. Whereas the area of Algorithmics deals with the design and analysis of procedures for accomplishing a broad range of tasks, a mathematical fiction called "nondeterminism" affords one powerful machinery for simplifying the design of certain algorithms; even more importantly, it exposes a feature that is common to many computational problems, that suggests why these problems are computational difficult. An important example of the use of nondeterminism to simplify algorithms resides in the Kleene-Myhill Theorem (or, the "Regular Expression" Theorem) of Finite-Automata Theory. The most important example of the explanatory power of nondeterminism is the theory of NP-Completeness (of the preceding paragraph), aspects of which appear in almost all upper-level computer science courses. This explanatory power can be explained in a nutshell, as follows. Nondeterminism in an "algorithm" (the nondeterminism prevents it from being a "real" algorithm) can be viewed as abbreviating a possibly lengthy, arduous search that is part of an algorithm, by means of a conceptual mechanism embodied in a one-step super-algorithmic primitive of the form "Search for x." The important role of this mechanism is to expose the existence of the search explicitly, which many algorithms' specifications do not do. This exposure has successfully explained the observed computational intransigence of many important computational problems (such the problem of scheduling final exams in the smallest possible number of venues). It has additionally allowed us—via the mechanisms of reduction and completeness—to prove that finding a speedy algorithm for any of myriad important intransigent problems will automatically provide speedy algorithms for all of the problems. Nondeterminism is an indispensable technical tool in Section 3.4; it provides the intellectual raw material for much of the theory of computational complexity, as is evident in Chapter 5.

This book presents the elements of Computation Theory, devoting a chapter to each of the three interrelated, yet distinct, theories of *Finite Automata* (Chapter 3), *Computability* (Chapter 4), and *Complexity* (Chapter 5). Finite Automata theory is developed up to, and including, the two basic theorems that characterize Finite Automata by, respectively, definitively elucidating the notion of "state" (the Myhill-Nerode Theorem, Theorem 3.1) and establishing the "equivalence" of finite automata and *regular expressions* (the Kleene-Myhill Theorem, Theorem 3.4). Computability theory will be built up from the underlying notions of (non)encodability [of one system as another] and reducibility [of one computational problem to another]. The culmination of this development is the notion of the *completeness* of certain individual computational problems within certain classes of such problems; these problems are singled our for their ability to encode, in a sense that is made precise, any other problem in the class. Perhaps the most exciting application of this notion within Computability theory is the Rice-Myhill-Shapiro Theorem (Theorem 4.4) which, informally, asserts the impossibility of effectively [i.e., algorithmically] determining any properties of the dynamic behavior of a program from its static description. We develop Complexity theory as an outgrowth of Computability theory, built upon (computational) resource-bounded versions of (non)encodability and reducibility. This view of Complexity theory would not be supported by many of its practicioners, but I feel that it is a useful pedagogical viewpoint. The culmination of this development, in Section 5.4, is the identification of certain computational problems as being *complete* in terms of time complexity, and the attendant **P**-vs.-**NP** problem. Aside from its intrinsic usefulness—which is manifest in the world of Algorithmics—this identification exposes the importance of nondeterminism as a concept for understanding the inherent time requirements a large variety of computational problems. Section 5.5 provides a useful counterpoint to the development in Section 5.4, by illustrating that nondeterminism affects the space requirements of computations in a very different way than the (apparent) time requirements. The qualifier "apparent" in the preceding sentence exposes the exciting fact that Complexity theory is very much a live discipline: Many of its most sought-after secrets remain to be discovered. In an effort to get the reader to view the "big ideas" within a broader context, we close each of the main chapters with an "enrichment" section devoted to extensions of the central concepts developed within the chapter. Some of this enrichment applies concepts in quite different settings than those in which they were developed; others supplement the "big ideas" with material that may be less "big" in its impact on computation but that round out the reader's perspective.

1.2 The Nature of Computation Theory

The Theory of Computation is a mathematical theory that arose from an attempt to understand the power and limitations of digital computing systems. The most exciting aspects of the Theory are:

Dynamism: The Theory models *dynamic* systems—machines and circuits that compute—rather than just static ones such as one encounters in most mathematical theories, such as algebra and geometry.

Robustness: Several of the systems studied in the Theory are modeled faithfully using quite disparate mathematical formulations that have been developed by practitioners in quite distinct fields, using quite different intuitions and formalisms. This robustness has (at least) two impacts. First, it enhances our ability to navigate the Theory: as we strive to understand various phenomena, we can switch from one formulation to another in our search for perspicuity. Second, it enhances out faith in having discovered something "essential:" a superficiality is unlikely to arrive multiple times in fundamentally different garb.

Applicability: Many aspects of the resulting theory have important applications—either conceptual or computational or both—in quite distinct fields. We find deep implications in the theory of Finite Automata for fields as disparate as compiler-construction, digital circuit design, and linguistics. The theory of Computability has far-reaching consequences in mathematical logic, as well as every aspect of the design and use of digital computers. The theory of Computational Complexity has incisive messages for any field that is concerned with optimizing complex processes.

As noted earlier, this book is dedicated to developing some of the "Big Ideas" in the three major thrusts of the Theory.

In the remainder of this section, we put these thrusts into their historical contexts. Most obviously, such context helps one appreciate the nature of the Theory. Less obviously, it helps one understand (and tolerate) some of the often-obscure and sometimes-conflicting terminology and notation in which much of the Theory is couched.

The dynamic nature of the Theory. Any student of mathematics has encountered multi-component systems such as groups, rings, fields, etc. As complex as these systems are, their specifications are “one dimensional:” each system has a fixed set of objects and a fixed set of operations on the objects, that jointly obey a fixed set of rules. If one understands the objects, the operations, and the rules, then one can (in principle, at least) understand everything about the system. The systems in Computation Theory add to the “syntactic” triumvirate of objects, operations, and rules a “semantic” component that describes how the system evolves over time. The notion of time thereby sneaks into the Theory as a guest that is unexpected because it does not appear explicitly when specifying the system. It may take some time getting used to systems that have both semantic and syntactic components, but the reader will view the time as well spent as the dynamic theory begins to unfold.

Chapter 2

Mathematical Preliminaries

2.1 Sets and Their Operations

Sets are probably the most basic object in mathematical discourse. We assume, therefore, that the reader knows what a set is and recognizes that some sets are finite (e.g., the set of words in this book or the set of characters in any valid C program), while others are infinite (e.g., the set $\mathbb{N}$ of nonnegative integers or the set $\{0,1\}^\star$ of all finite-length binary strings). Given this assumption, we devote this short section to reviewing some basic operations on sets. (Others will appear as needed throughput the text.) One recurring finite set is the *empty set* $\emptyset$ that is defined by the property of having no elements.

In many of our discussions throughout the text, the sets of interest will be subsets of some fixed "universal" set U. Two universal sets that will appear later are the two sample infinite sets just mentioned. Given a universal set U and a *subset* $S \subseteq U$ (the notation meaning that every element of S—if there are any—is also an element of U), we note that the set-inequalities

$$\emptyset \ \subseteq \ S \ \subseteq \ U$$

always hold.

Given two sets S and T, we denote by:

- $S \times T$ the *direct-product* of S and T, which is the set of all ordered pairs whose first coordinate is an element of S and whose second coordinate is an element of T.
- $S \cap T$ the *intersection* of S and T, which is the set of elements that occur in *both* S and T.
- $S \cup T$ the *union* of S and T, which is the set of elements that occur in *either* S or T or *both.* (Because of the "or both" qualifier, this is sometimes called "inclusive union.")

- $S \setminus T$ the *difference* of S and T, which is the set of elements that occur in S but not in T. (Particularly in the US, one often finds "$S - T$" instead of "$S \setminus T$.")

When there is a universal set U that all other sets are subsets of, then we add the operation

- $\overline{T} = U \setminus T$, the *complement* of T (relative to the universal set U).

We note a number of easily verified identities involving sets and operations on them.

- $S \setminus T \ = \ S \cap \overline{T}$
- If $S \subseteq T$, then
  1. $S \setminus T \ = \ \emptyset$
  2. $S \cap T \ = \ S$
  3. $S \cup T \ = \ T$.

Note, in particular, that[1]

$$[S = T] \ \text{ iff } \ [[S \subseteq T] \text{ and } [T \subseteq S]] \ \text{ iff } \ [(S \setminus T) \cup (T \setminus S) = \emptyset].$$

The operations union, intersection, and complementation are usually called the *Boolean (set) operations*, acknowledging the seminal work of the 19th-century English mathematician George Boole. (One sometimes finds the lower-case adjective "boolean;" such is the price of fame.) There are several important identities involving the Boolean set operations. Among the most frequently invoked are the two "laws" attributed to the 19th-century French mathematician Auguste De Morgan.

$$\text{For all sets } S \text{ and } T\text{:} \quad \left\{ \begin{array}{lcl} \overline{S \cup T} & = & \overline{S} \cap \overline{T} \\ \overline{S \cap T} & = & \overline{S} \cup \overline{T}. \end{array} \right. \tag{2.1.1}$$

While we have focused here on Boolean operations on sets, there are analogues of these operations for the logical values 0 and 1. Specifically:

- The logical analogue of complementation is (logical) not: $\mathsf{not}(0) = 1$, and $\mathsf{not}(1) = 0$.
- The logical analogue of union is (logical) or: $X \ \mathsf{or} \ Y = 1$ iff $X = 1$ or $Y = 1$ or both.
- The logical analogue of intersection is (logical) and: $X \ \mathsf{and} \ Y = 1$ iff both $X = 1$ and $Y = 1$

[1] "iff" abbreviates the common mathematical phrase, "if, and only if."

We close with a set-theoretic definition that will recur throughout our study. Let $\mathcal{C}$ be any (finite or infinite) collection of sets, and let S and T be two elements of $\mathcal{C}$. (Note that $\mathcal{C}$ is a set whose elements are sets.) Focus, just for example, on the set-theoretic operation of intersection; you should be able to extrapolate easily to other operations. We say that $\mathcal{C}$ is *closed* under intersection if, whenever set S and T (which could be the same set) both belong to $\mathcal{C}$, the set $S \cap T$ also belongs to $\mathcal{C}$. As one instance of the desired extrapolation, $\mathcal{C}$'s closure under union would mean that the set $S \cup T$ belongs to $\mathcal{C}$.

2.2 Binary Relations

2.2.1 The Formal Notion of Binary Relation

Given sets S and T, a *relation on S and T* (in that order) is any subset

$$R \ \subseteq \ S \times T.$$

When $S = T$, we often call R a *binary relation on (the set) S*. We shall see later (Section 2.3.2) that there is a formal sense in which such *binary* relations (so named because there are two sets being related) are all we ever need consider: 3-set relations (i.e., subsets of $S_1 \times S_2 \times S_3$), 4-set relations (i.e., subsets of $S_1 \times S_2 \times S_3 \times S_4$), etc., can all be expressed via binary relations.

By convention, we often write "sRt" in place of the more conservative "$\langle s, t\rangle \in R$."

The following operation on relations occurs in many guises, in almost all mathematical theories. Let R and R' be binary relations on a set S. The *composition* of R and R' (in that order) is the relation

$$R'' \ \stackrel{\text{def}}{=} \ \{\langle s, u\rangle \in S \times S \mid (\exists t \in S)\,[[sRt] \text{ and } [tR'u]]\}.$$

(Notice how we have used both of our notational conventions here.)

There are two special classes of binary relations that play such a central role in the theory of computation—and elsewhere!—that we must single them out immediately, in the next two subsections.

2.2.2 Equivalence Relations

When the sets S and T are identical ($S = T$), we call a binary relation on S and T a *binary relation on S*. A binary relation R on S is an *equivalence relation* if it enjoys the following three properties.

1. R is *reflexive:* for all $s \in S$, we have sRs.
2. R is *symmetric:* for all $s, s' \in S$, we have sRs' whenever $s'Rs$.
3. R is *transitive:* for all $s, s', s'' \in S$, whenever we have sRs' and $s'Rs''$, we also have sRs''.

Closely related to the notion of an equivalence relation on a set S is the notion of a *partition of* S. A partition of S is a collection of subsets of S: $S_1, S_2, \ldots, S_n$ that are

1. mutually exclusive: for all distinct indices i and j, $S_i \cap S_j = \emptyset$;
2. collectively exhaustive: $S_1 \cup S_2 \cup \cdots \cup S_n = S$.

We call each S_i a *block* of the partition.

It is easy to verify that partitions of a set S and equivalence relations on S are just two ways of looking at the same concept. To see this, note the following.

Given any partition, $S_1, S_2, \ldots, S_n$, define the relation R on S: sRs' if, and only if s and s' belong to the same block of the partition. Relation R is reflexive, symmetric, and transitive because collective exhaustiveness ensures that each $s \in S$ belongs to some block of the partition, while mutual exclusivity ensures that it belongs to only one block.

For the converse, focus on any equivalence relation R on set S. For each $s \in S$, denote by $[s]_R$ the set

$$[s]_R \overset{\text{def}}{=} \{s' \in S \mid sRs'\};$$

and call the set *the equivalence class of s under relation R*. We claim that the equivalence classes under R form a partition of S. R's reflexivity ensures that the equivalence classes collectively exhaust S. Symmetry and transitivity ensure that equivalence classes are mutually disjoint.

The *index* of relation R is its number of classes—which can be finite or infinite.

2.3 Functions

2.3.1 The Formal Notion of Function

A *function from set S to set T* is a relation $F \subseteq S \times T$ that is *uni-valued:* for each $s \in S$, there is at most one $t \in T$ such that sFt. We traditionally write "$F : S \rightarrow T$" as shorthand for the assertion, "F is a function from the set S to the set T;" we also traditionally write

"$F(s) = t$" for the more conservative "sFt." (The uni-value property makes this notation safe.)

Three special classes of functions merit explicit mention. For each, we give both a down-to-earth name and a more scholarly Latinate one.

A function $F : S \to T$ is:

1. *one-to-one* (or, *injective*) if, for each $t \in T$, there is at most one $s \in S$ such that $F(s) = t$;
2. *onto* (or, *surjective*) if, for each $t \in T$, there is at least one $s \in S$ such that $F(s) = t$;
3. *one-to-one, onto* (or, *bijective*) if, for each $t \in T$, there is precisely one $s \in S$ such that $F(s) = t$.

Recasting the preceding adjectives into nominal form: F is an injection (resp., a surjection) (resp., a bijection) if it is injective (resp., surjective) (resp., bijective).

2.3.2 Countability and Uncountability

The theory of Countability (and Uncountability) began with G. Cantor's deliberations on the nature of infinity [3]. Cantor concentrated on the questions that can be framed intuitively as follows: "Are there 'more' rationals than integers? Are there 'more' reals than integers?" Since we are concerned with computable functions rather than numbers, we shall actually treat technically simpler analogues of these questions. But the tools we develop here (which are the ones that Cantor used to answer his questions) can be adapted in very simple ways to answer Cantor's questions directly.

In order to start thinking about Cantor's questions, we must find a formal, precise way to talk about one infinite set's having "more" elements than another. We would like this way to be an *extension* of how we make this comparison with finite sets. Here is the mechanism that we shall use.

Let A and B be (possibly infinite) sets. We write

$$|A| \ \leq \ |B| \tag{2.3.2}$$

just when there is an *injection*—i.e., a one-to-one function

$$f : A \xrightarrow{1-1} B.$$

When A is a finite set, then (2.3.2) means: *set A has no more elements than does set B.* It is important *not* to read (2.3.2) in that way when A is infinite, since "more" is not (yet) defined in that case.

The hallmark of an injection such as f is that, given any $b \in B$, there is at most one $a \in A$ such that $f(a) = b$. Of course, when A and B are *finite* sets, this condition is equivalent to saying that B has at least as many elements as A—so we do, indeed, have an extension of the finite situation.

When

$$|A| \ \le \ |B| \quad \textbf{and} \quad |B| \ \le \ |A|,$$

we write

$$|A| \ = \ |B|.$$

One proves easily—by arguing about composition of injections—that "$\le$" is *reflexive* and *transitive*, while "$=$" is an equivalence relation.

We single out the important case when $B = \mathbb{N}$. When

$$|A| \ \le \ |\mathbb{N}|$$

we say that the set A is *countable*. When

$$|A| \ = \ |\mathbb{N}|$$

we say that the set A is *countably infinite*.

The following important result of Schroeder and Bernstein[2] plays a crucial role in the study of countability.

Theorem 2.1. (The Schroeder-Bernstein Theorem) *If $|A| = |B|$, then there is a* bijection—*i.e., one-to-one, onto function*

$$f : A \overset{1-1,onto}{\longrightarrow} B.$$

The Schroeder-Bernstein Theorem is quite easy to prove for finite sets but is decidedly nontrivial for infinite ones. (Its proof is beyond the scope of these notes.)

In deference to our intended use of the bijections promised by Theorem 2.1, we henceforth view these functions as *encoding* (instances of) set A as (instances of) set B, and we call such functions *encoding functions*.

2.3.2.A Encoding functions and proofs of countability

Theorem 2.2. *The following sets are countable.*

1. *$\Sigma^\star$, for any finite set Σ.*

[2]See, e.g., [1].

2. *The set of all* finite *subsets of* $\mathbb{N}$.
3. $\mathbb{N}^\star$. *Note that this includes all sets* $\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}$, *where the product is performed any finite number of times.*

$\Sigma^\star$ for finite Σ. Let us start by establishing the countability of $\Sigma^\star$, where $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. For our purposes, this is the most important set to prove countable, since *every program in any programming language is a finite string over some finite alphabet.* Since a function must be programmable in order to be computable, we shall, therefore, have the important corollary to our proof.

Corollary 2.1. *The set of computable functions is countable.*

The easiest proof of this result is to interpret strings over Σ as base-$(n+1)$ numerals. (We use $n+1$, rather than n, as the base in order to avoid the vexatious problem of leading 0's.) We thus define the function

$$f_\Sigma : \Sigma^\star \longrightarrow \mathbb{N}$$

by defining $|\sigma_k| = k$ for each $\sigma_k \in \Sigma$, and

$$f_\Sigma(\sigma_{i_m}\sigma_{i_{m-1}} \cdots \sigma_{i_1}) \ \stackrel{\text{def}}{=} \ \sum_{j=1}^{m} |\sigma_{i_j}|(n+1)^{j-1}.$$

Since in any "ary" positional number system (e.g., binary, ternary, octal, decimal, etc.) every numeral that contains no 0's specifies a unique integer, the function f_Σ is one-to-one, hence proves the countability of $\Sigma^\star$.

Finite subsets of $\mathbb{N}$. We reduce the problem of establishing the countability of the set of finite subsets of $\mathbb{N}$ to the preceding problem via the following notion.

The *characteristic vector* $\beta(S)$ of a finite set $S \subset \mathbb{N}$ is the following binary string whose length is 1 greater than the maximum integer in S:

$$\beta(S) \ \stackrel{\text{def}}{=} \ \delta_0\delta_1 \cdots \delta_{\max(S)}$$

where, for each $i \in \{0, 1, \ldots, \max(S)\}$,

$$\delta_i \ = \ \begin{cases} 1 & \text{if } \ i \in S \\ 0 & \text{if } \ i \notin S \end{cases}$$

By using characteristic vectors, we thus have an injection

$$g : [\text{Finite subsets of } \mathbb{N}] \ \stackrel{1-1}{\longrightarrow} \{0,1\}^\star$$

so that

$$|\text{Finite subsets of } \mathbb{N}| \ \leq \ |\{0,1\}^\star|$$

Since we already know that $\{0,1\}^\star$ is countable, and since "$\leq$" is transitive, we conclude that the set of finite subsets of $\mathbb{N}$ is countable.

$\mathbb{N}^\star$**.** The ploy of encoding the objects of interest as numerals will not work here, because there is no natural candidate for the base of the number systems. Therefore, we call in some heavier machinery, the Fundamental Theorem of Arithmetic, sometimes called the Prime-Factorization Theorem, and the Infinite-prime Theorem.[3]

Theorem 2.3. *(***The Fundamental Theorem of Arithmetic***) Every positive integer can be represented as a product of primes in one, and only one, way, up to the order of the primes.*

Theorem 2.4. *(***The Infinite-Prime Theorem***) There are infinitely many primes.*

The proof of the Infinite-Prime Theorem is so simple that we sketch it here. Any finite set of primes, $\{p_1, p_2, \ldots, p_n\}$, is inadequate to provide a prime factorization for the positive integer $1 + \prod_{i=1}^n p_i$.

We use the preceding two theorems to establish the injectiveness of the following function $h : \mathbb{N}^\star \longrightarrow \mathbb{N}$, thereby proving the countability of $\mathbb{N}^\star$. For any finite sequence $m_1, m_2, \ldots, m_k$, of nonnegative integers:

$$h(m_1, m_2, \ldots, m_k) \ = \ \prod_{i=1}^{k} p_k^{m_k},$$

where p_k is the kth smallest prime.

Of course, the function h can be used to establish the countability of an finite cross-product $\mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}$. For instance, h restricted to the set of ordered pairs $\mathbb{N} \times \mathbb{N}$ becomes the injection

$$h(m_1, m_2) \ = \ 2^{m_1} \cdot 3^{m_2}.$$

Of course, all of the cross-products $\mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N}$ are infinite, so, by the Schroeder-Bernstein theorem, there exist *bijections*

$$f : \mathbb{N} \times \mathbb{N} \times \cdots \times \mathbb{N} \ \overset{1-1, onto}{\longrightarrow} \ \mathbb{N}$$

In these special cases, the advertised bijections actually have simple forms. We now present one such bijection for $\mathbb{N} \times \mathbb{N}$, just to indicate its charming form. This bijection is attributed

[3]As with other mathematical tools, we do not prove these results here; see, e.g., [1].

to Cantor, but there is evidence that it was known already to Cauchy [4].

$$d(x,y) \;=\; \binom{x+y+1}{2} + y \;=\; \frac{1}{2}(x+y)(x+y+1)+y$$

(Of course, there is a twin of d that interchanges x and y.)

If we illustrate $\mathbb{N}\times\mathbb{N}$ as follows

$$\begin{array}{cccccc}
(0,0) & (0,1) & (0,2) & (0,3) & (0,4) & \cdots \\
(1,0) & (1,1) & (1,2) & (1,3) & (1,4) & \cdots \\
(2,0) & (2,1) & (2,2) & (2,3) & (2,4) & \cdots \\
(3,0) & (3,1) & (3,2) & (3,3) & (3,4) & \cdots \\
(4,0) & (4,1) & (4,2) & (4,3) & (4,4) & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

then the action of the bijection d can be seen in the following illustration, where each position (x,y) contains $d(x,y)$, and where the "diagonal" $x+y=4$ is highlighted.

$$\begin{array}{rrrrrrrrr}
0 & 2 & 5 & 9 & \boxed{14} & 20 & 27 & 35 & \cdots \\
1 & 4 & 8 & \boxed{13} & 19 & 26 & 34 & 43 & \cdots \\
3 & 7 & \boxed{12} & 18 & 25 & 33 & 42 & 52 & \cdots \\
6 & \boxed{11} & 17 & 25 & 33 & 42 & 52 & 63 & \cdots \\
\boxed{10} & 16 & 23 & 31 & 40 & 50 & 61 & 73 & \cdots \\
15 & 22 & 30 & 39 & 49 & 60 & 72 & 85 & \cdots \\
21 & 29 & 38 & 48 & 59 & 71 & 84 & 98 & \cdots \\
28 & 37 & 47 & 58 & 70 & 83 & 97 & 112 & \cdots \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots
\end{array}$$

The preceding illustration lends us the intuition to prove that d is, indeed, a bijection. To wit:

- Along each "diagonal" of $\mathbb{N}\times\mathbb{N}$—which corresponds to a fixed value of $x+y$—the value of $d(x,y)$ increases by 1 as x decreases (by 1) and y increases (by 1).

- As we leave diagonal $x+y$—at node $(0,x+y)$—and enter diagonal $x+y+1$—at node

$(x+y+1,0)$—the value of $d(x,y)$ increases by 1, because

$$\begin{aligned}
d(x+y+1,0) &= \binom{x+y+2}{2} \\
&= \frac{(x+y+1)(x+y+2)}{2} \\
&= \frac{(x+y)(x+y+1)+2(x+y+1)}{2} \\
&= \frac{(x+y)(x+y+1)}{2} + (x+y) + 1 \\
&= \binom{x+y+1}{2} + (x+y) + 1 \\
&= d(0,x+y)+1.
\end{aligned}$$

Note that the various injections that we have used to prove the countability of sets—numeral evaluation in an "ary" number system, encodings via powers of primes, the Cauchy-Cantor polynomial—are eminently computable. The importance of this fact is that we can use the functions as *encoding mechanisms*. In other words:

> *We can now formally and rigorously encode "everything"—programs, data structures, data—as strings (say, for definiteness, binary strings) or as integers.*

On the one hand, this gives us tremendous power, by "flattening" out our universe of discourse; henceforth, we can discuss *only* functions from $\mathbb{N}$ to $\mathbb{N}$, without losing any generality. On the other hand, we must be very careful from now on, because, as we apparently discuss and manipulate integers, we are also—via the appropriate encodings—discussing and manipulating programs. We shall see before long the far-reaching implications of this new power.

A brief survey of surprising uses of encoding functions (using their old name, "pairing functions") can be found in [33].

2.3.2.B Diagonalization: proofs of uncountability

In this section we introduce the technique of *diagonalization*. Cantor developed this notion to prove that certain sets—notably, the real numbers—are not countable. Our interest in it stems from its being the primary tool for establishing the noncomputability of functions and/or the existence of functions whose complexity exceeds certain limits. The latter, complexity-theoretic, role of diagonalization is hard to talk about until we establish a framework for studying the complexity of computation. In contrast, we shall have our first computability-theoretic result by the end of this section.

Theorem 2.5. *The following sets are not countable (are uncountable):*

1. *the set of functions* $f : \mathbb{N} \longrightarrow \{0,1\}$*;*
2. *the set of all subsets of* N*;*
3. *the set of functions* $f : \mathbb{N} \longrightarrow \mathbb{N}$*;*
4. *the set of (countably) infinite binary strings.*

Assuming that we can establish the uncountability of the set of functions $f : \mathbb{N} \longrightarrow \{0,1\}$, we can immediately infer the uncountability of the set of functions $f : \mathbb{N} \longrightarrow \mathbb{N}$, for the former set can be mapped into the latter via the identity injection. (You should carefully verify this consequence of the transitivity of "$\le$".) We therefore leave this part of the theorem to the reader.

Let's attack the other three parts of the theorem in tandem. We can do this because one can represent any function $f : \mathbb{N} \longrightarrow \{0,1\}$ or any subset of N as a (countably) infinite binary string. In the case of the subsets of N, any such string can be viewed as the characteristic vector of the set, as defined earlier (except now the string is infinite, because the set may be). In the case of the functions, such a string "enumerates" the values of a function. Specifically, the function $f : \mathbb{N} \longrightarrow \{0,1\}$ is represented by the infinite binary string

$$f(0)\ f(1)\ f(2)\ \cdots$$

whose kth bit-value is $f(k)$. So, let's just focus on the set $\mathcal{B}$ of (countably) infinite binary strings.

Assume, for contradiction, that the set $\mathcal{B}$ is countable, so that $|\mathcal{B}| \le |\mathbb{N}|$.

Now, one sees easily that $|\mathbb{N}| \le |\mathcal{B}|$. Just focus on the subset of $\mathcal{B}$ comprising the infinite characteristic vectors of the sets $\{\{k\} \mid k \in \mathbb{N}\}$. The infinite characteristic vector of the set $\{m\}$ is the infinite binary string with precisely one 1, in bit-position m. We thus have $|\mathcal{B}| = |\mathbb{N}|$, so that (by the Schroeder-Bernstein Theorem) there exists a *bijection*

$$h : \mathcal{B} \overset{1-1,onto}{\longrightarrow} \mathbb{N}.$$

It is not hard to view the bijection h as producing an "infinite-by-infinite" binary matrix Δ, whose kth row is the infinite binary string $h^{-1}(k)$. Let's visualize Δ:

$$\Delta \ = \ \begin{array}{cccccc} \delta_{0,0} & \delta_{0,1} & \delta_{0,2} & \delta_{0,3} & \delta_{0,4} & \cdots \\ \delta_{1,0} & \delta_{1,1} & \delta_{1,2} & \delta_{1,3} & \delta_{1,4} & \cdots \\ \delta_{2,0} & \delta_{2,1} & \delta_{2,2} & \delta_{2,3} & \delta_{2,4} & \cdots \\ \delta_{3,0} & \delta_{3,1} & \delta_{3,2} & \delta_{3,3} & \delta_{3,4} & \cdots \\ \delta_{4,0} & \delta_{4,1} & \delta_{4,2} & \delta_{4,3} & \delta_{4,4} & \cdots \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots \end{array}$$

Now let us construct the following infinite binary string.

$$\Psi \ = \ \psi_0 \ \psi_1 \ \psi_2 \ \psi_3 \ \psi_4 \cdots ,$$

where for each index i,

$$\psi_i \ = \ \overline{\delta}_{i,i} \ = \ 1 - \delta_{i,i}.$$

(The term "diagonal argument" stems from the fact that we create the new string Ψ from the *diagonal* elements of the matrix Δ.)

Clearly string Ψ does not occur as a row of matrix Δ since it differs from each row of Δ in at least one element: for each index i, $\psi_i \neq \delta_{i,i}$. But this contradicts our assumption that Δ contains *every* infinite binary string as one of its rows. Where could we have gone wrong? In just one place: our assumption that the set $\mathcal{B}$ is countable. Everything that we did based on that assumption is on solid mathematical ground! We conclude that the set $\mathcal{B}$ is *not* countable.

For the purposes of our tour of the highlights of Computation Theory, the most important consequence of our work in this section is the following

Corollary 2.2. *Since the set of (0-1 valued) integer functions is uncountable, while the set of programs is countable, there must exist noncomputable (0-1 valued) integer functions.*

2.4 Formal Languages

2.4.1 The Computation-Theoretic Notion of Language

Let Σ be a finite set of (atomic) symbols; we often call these symbols **letters** and the set Σ an **alphabet**. For each nonnegative integer k, we denote by Σ^k the set of all length-k strings of elements of Σ. For instance, if $\Sigma = \{a, b\}$, then:

$$\begin{array}{rcl} \Sigma^0 & = & \{\varepsilon\} \ \ (\varepsilon \text{ is the } \textit{null string:} \text{ the unique string of length } 0) \\ \Sigma^1 & = & \Sigma = \{a, b\} \\ \Sigma^2 & = & \{aa, ab, ba, bb\} \\ \Sigma^3 & = & \{aaa, aab, aba, abb, baa, bab, bba, bbb\}. \end{array}$$

We denote by $\Sigma^\star$ the set of all finite-length strings of elements of Σ; symbolically,

$$\Sigma^\star \ = \ \bigcup_{k \in \mathbb{N}} \Sigma^k.$$

A **word over** Σ is any element of $\Sigma^\star$; a **language over** Σ is any subset $L \subseteq \Sigma^\star$.

Equivalence relations on $\Sigma^\star$, specifically, *right-invariant* ones, cast a broad shadow in the theory, hence, in our survey.

An equivalence relation $\equiv$ on $\Sigma^\star$ is **right-invariant** if, for all $z \in \Sigma^\star$, $xz \equiv yz$ whenever $x \equiv y$.

Our particular focus will be on the following specific (right-invariant) equivalence relation on $\Sigma^\star$, which is defined in terms of a given language $L \subseteq \Sigma^\star$.

$$\text{For all } x, y \in \Sigma^\star : \quad [x \equiv_L y] \ \text{ iff } \ (\forall z \in \Sigma^\star)[[xz \in L] \Leftrightarrow [yz \in L]]. \tag{2.4.3}$$

The following important result is a simple exercise.

Lemma 2.1. *For all alphabets Σ and all languages $L \subseteq \Sigma^\star$, the equivalence relation $\equiv_L$ is right-invariant.*

2.4.2 Languages as a Metaphor for Computation Theory

Each language $L \subseteq \Sigma^\star$ has an associated function that allows us to step back and forth between the world of functions and the world of languages. While we may be a bit uncomfortable hopping between formal notions in this way, the history of Computation Theory more or less forces us to, especially if we want access to the primary sources in the development of the Theory.

The *characteristic function* of the set/language A is the function κ_L defined as follows:

$$(\forall x \in \Sigma^\star) : \ \kappa_L(x) \ = \ \begin{cases} 1 & \text{if } \ x \in L \\ 0 & \text{if } \ x \notin L \end{cases}$$

Dually, every function $f : \Sigma^\star \to \{0,1\}$ has an associated language L_f defined as follows:

$$L_f \ = \ \{x \in \Sigma^\star \mid f(x) = 1\}.$$

One can study a large range of computational issues involving two-valued functions by focusing on the languages associated with the functions; and one can study a large range of computational issues involving langauges by focusing on the languages' characteristic functions. One thus finds three distinct notions talked about interchangeably within the Theory:

A *language* L
The *computational problem*: to compute L's characteristic function
The *system property*: to decide, given $x \in \Sigma^\star$, is $x \in L$?

A more concrete example of the duality involves an arbitrary function

$$g : \{0,1\}^\star \times \{0,1\}^\star \to \{0,1\}^\star. \tag{2.4.4}$$

(Think of addition or multiplication, for instance.) One often studies the problem of computing g via the following language-recognition problem. We define the language[4] $L(g)$ as follows. $L(g)$ is a language over the alphabet $\Sigma \overset{\text{def}}{=} \{0,1\} \times \{0,1\}$ whose letters are *ordered pairs* of bits. For each $n \in \mathbb{N}$, the n-letter word

$$\langle \alpha_0, \beta_0 \rangle \langle \alpha_1, \beta_1 \rangle \cdots \langle \alpha_{n-1}, \beta_{n-1} \rangle$$

in Σ^n belongs to the language $L(g)$ precisely when the nth bit of the bit-string

$$g(\alpha_{n-1} \cdots \alpha_0, \beta_{n-1} \cdots \beta_0)$$

is a 1. (Note that we reverse the orders of bit strings.)

2.5 Exercises

1. Prove that all three Boolean operations can be expressed using just the single operation set-difference. In other words, find ways of expressing $\overline{S}$, $S \cap T$, and $S \cup T$ using only expressions of the form $S \setminus T$.

2. Prove Lemma 2.1: For all alphabets Σ and all languages $L \subseteq \Sigma^\star$, the equivalence relation $\equiv_L$ is right-invariant.

3. Prove in detail the fact alluded to in Section 2.2.2, that an equivalence relation on a set S and a partition of S are just two ways to look at the same notion.

4. Prove that the composition of injections is an injection.

5. Prove Corollary 2.2. You may want to attempt this by focusing first on the following simpler version. of the problem.

   Prove that there are functions $f : \mathbb{N} \rightarrow \mathbb{N}$ that are not computed by any C program, even if one can employ arbitrarily long numerals in the program.

6. This problem considers *pairing functions*, i.e., bijections

   $$f(x,y) \ = \ \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N},$$

   that are *additive*, in the following sense. For each $x \in \mathbb{N}$, there exists a positive integer "stride" s_x such that, for all $y \in \mathbb{N}$,

   $$f(x, y+1) \ = \ f(x,y) + s_x.$$

[4]We avoid the notation "L_g" to avoid any confusion with languages and their characteristic functions.

(a) Prove that there exist additive pairing functions.

(b) Prove that there *do not* exist additive pairing functions for which all of the "strides" s_x are equal.

7. Prove that the relation $|A| \leq |B|$ on (finite or infinite) sets as defined in Section 2.3.2 is: (a) reflexive — so that $|A| \leq |A|$; (b) transitive — so that $|A| \leq |C|$ whenever there exists a B such that $|A| \leq |B|$ **and** $|B| \leq |C|$.

8. Using first addition and then multiplication as the function in (2.4.4), present ten strings in the language L_g.

9. Prove that the set of square matrices whose entries are positive integers is countable.

Chapter 3

Finite Automata and Regular Languages

3.1 Introduction

The model of *Finite Automaton* (*FA*, for short) amply illustrates the three features of the Theory of Computation described in Chapter 1. (Note that, in deference to the Greek origins of the word, the plural of *automaton* is *automata*.) Essentially equivalent models of finite-state systems were developed over a span of three decades, to model a large range of "real-life" systems. In roughly chronological order:

1. Researchers (e.g., McCulloch and Pitts) attempting to explain the behavior of *neural systems* (natural and artificial "brains") beginning in the 1940's developed models that were very close to our model of FA. While the neural models studied nowadays have diverged from the standard FA model in many ways, they still share many of its essential features.

2. Engineers seeking to systematize the design and analysis of *synchronous sequential circuits* developed a model of Finite State Machine (FSM) in the 1940's. E.F. Moore's variant of the FSM model [25] is essentially identical to the FA that we study; G. Mealy's variant [24] can be translated to our model very easily. These models still play an essential role in the design of digital systems—from carry-ripple adders to the control units of digital computers (and, often other systems, such as elevator controllers).

3. In the mid-1950's, several *linguists*—N. Chomsky being the best known and most influential—strove for formal models that could explain the acquisition of language by children. Chomsky developed a hierarchy of models, each augmenting the linguistic complexity of its predecessor [5, 6]. Chomsky's lowest-level model—his type 3

systems—are essentially finite automata. His work was later picked up by compiler designers, who could use Chomsky's FA directly as token recognizers.

4. In the late 1950's, researchers (e.g., M.O. Rabin and D. Scott) who were disheartened by the resistance of detailed computational models to algorithmic tractability—we'll see this in the theory of unsolvability—began to study *very coarse models for computers.* The FA is such a model, since the bi-stable devices (transistors) that implement the hardware of a computer—notably, the CPU and the memory—being finite (albeit astronomical) in number, can assume only finitely many distinct configurations. The algorithmic tractability of FA (which we shall see a few examples of) means that, in principle (i.e., modulo the astronomical numbers), one can analyze a lot about the dynamic behavior of programs—as long as the information one seeks is *very* coarse.

5. Toward the mid-to-late 1960's (and beyond), as people began to investigate the potential for *optimizing compilers* (J. Cocke and F. Allen) and for *program verifiers* (R. Floyd), they began to study various graphs that abstracted the behavior of programs. The analytical tools that had been developed for studying FA could be applied—with little or no adaptation—to the analysis of the resulting data- and control-flow graphs for programs.

The high points of this chapter are two theorems—the Myhill-Nerode Theorem (Section 3.3) and the Kleene-Myhill Theorem (Section 3.4), both of which completely characterize the power of Finite Automata, but from very different perspectives—and some of their applications. The former theorem is the more useful computation-theoretically and has broader-ranging hardware-engineering applications; the latter is the more useful language-theoretically and has broader-ranging software-engineering applications. We close the Chapter with a theorem, known widely as the "Pumping Lemma" (Section 3.5), that exploits a simple aspect of the structure of FA—their finiteness. In my estimation, the Pumping Lemma has been promoted in textbooks far beyond its intrinsic importance. We attempt to put the Lemma into perspective by expounding, at greater length than is customary, on its strengths and its limitations within the context of the Theory of FA. As usual, we close the chapter with an "enrichment" section devoted to extensions of the central concepts we have developed.

3.2 Preliminaries

Throughout our tour of Finite Automata theory, we will be talking about sets of *input symbols* to our FA's; due to the linguistic heritage of FA, these finite sets are traditionally called *(input) alphabets.* As we develop the theory of FA, we treat input symbols as atomic (i.e., indivisible) entities. Of course, when we design FA, the symbols usually have intrinsic structure, which is endowed by their meaning.

3.2.1 A motivating example

We introduce our abstract model gently, via the simple example of a sequential carry-ripple adder. The input to the adder will be a sequence/string of pairs of bits. An output bit will be produced upon reading each input symbol: the aggregate output after having read n input symbols is the string comprising the low-order n bits of the sum of the pair of numerals represented by the input. Symbolically, the behavior of the adder at step n can be depicted as follows.

$$\begin{array}{lccccc}
\text{INPUT:} & \langle\alpha_{n-1},\beta_{n-1}\rangle & \langle\alpha_{n-2},\beta_{n-2}\rangle & \ldots & \langle\alpha_1,\beta_1\rangle & \langle\alpha_0,\beta_0\rangle \\
\text{OUTPUT:} & \gamma_{n-1} & \gamma_{n-2} & \ldots & \gamma_1 & \gamma_0
\end{array}$$

where all $\alpha_i, \beta_i, \gamma_i \in \{0, 1\}$ and

$$\begin{array}{ccccccc}
 & \cdots & \alpha_{n-1} & \alpha_{n-2} & \cdots & \alpha_1 & \alpha_0 \\
+ & \cdots & \beta_{n-1} & \beta_{n-2} & \cdots & \beta_1 & \beta_0 \\
= & \cdots & \gamma_{n-1} & \gamma_{n-2} & \cdots & \gamma_1 & \gamma_0.
\end{array}$$

In order to produce the desired behavior, we design a "machine" that has four states. (The right arrow indicates the initial state, where the computation begins.) The states and their roles in the addition process are depicted in Table 3.1. One can depict the actions of the

	State-name	State "meaning"	
→	0, no-C	output 0 is produced	no carry is propagated
	0, C	output 0 is produced	a carry is propagated
	1, no-C	output 1 is produced	no carry is propagated
	1, C	output 1 is produced	a carry is propagated

Table 3.1: The states of our sequential adder.

"machine" in either a tabular or graph-theoretic manner. The tabular format is depicted in Table 3.2.

We can also represent the machine graphically—literally, as an arc-labeled directed graph—as shown in Fig. 3.1.

We can follow the behavior of this machine on a small example: let us add 7 ($= 111_2$) and 6 ($= 110_2$):

Time ⟶							
State:	[0, no-C]		[1, no-C]		[0, C]		[1, C]
Input:		$\langle 1,0\rangle$		$\langle 1,1\rangle$		$\langle 1,1\rangle$	
Output:	(ignored)		1		0		1

State	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 1,1\rangle$
$\rightarrow$ 0, no-C	0, no-C	1, no-C	1, no-C	0, C
1, no-C	0, no-C	1, no-C	1, no-C	0, C
0, C	1, no-C	0, C	0, C	1, C
1, C	1, no-C	0, C	0, C	1, C

Table 3.2: A tabular representation of a 4-state sequential adder.

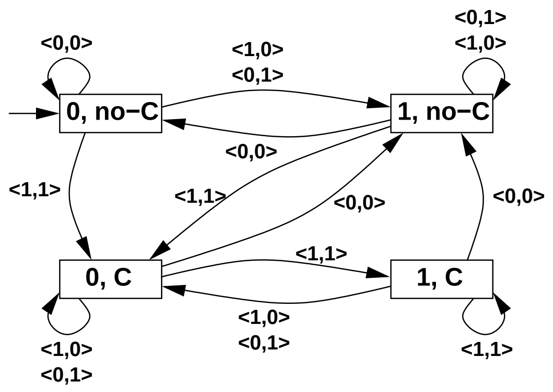

Figure 3.1: A graph-theoretic representation of a 4-state sequential adder.

The reader may have noted that it would be more intuitive to associate the output with the *arcs*, or *transitions*, in our adder, rather than with the states. Indeed, there is a variant of the model that we are developing that operates in just that way. Historically, the model we develop is called a "Moore" machine, in honor of its inventor, E.F. Moore; the competing model is called a "Mealy" machine, in honor of its inventor, G. Mealy. We stick with the Moore model, in which outputs are associated with states rather than transitions because that is the model that is dominant in all studies based on finite automata, save, perhaps, studies that use the model to design synchronous sequential circuits.

Since we are using the Moore model, in which outputs are associated with states, we can somewhat simplify our representation of finite-state machines, as follows. We partition the set of states into those that give output 1 and those that give output 0, and we suppress explicit mention of the output in our representation of the machine. We illustrate this abbreviated notation in Table 3.3, where the states that give output 1 are boxed, and those that give output 0 are not. Of course, we shall have a more elegant mechanism for distinguishing these classes of states when we turn to the formal development of the model.

	State	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 1,1\rangle$
$\rightarrow$	no-C	no-C	$\boxed{\text{no-C}}$	$\boxed{\text{no-C}}$	C
	$\boxed{\text{no-C}}$	no-C	$\boxed{\text{no-C}}$	$\boxed{\text{no-C}}$	C
	C	$\boxed{\text{no-C}}$	C	C	$\boxed{\text{C}}$
	$\boxed{\text{C}}$	$\boxed{\text{no-C}}$	C	C	$\boxed{\text{C}}$

Table 3.3: A tabular representation of a 4-state sequential adder, with "implicit" outputs.

3.2.2 Finite Automata and Their "Languages"

We develop the formal model of Finite Automaton by extrapolating and abstracting from the example that we have just gone through. Since we are now embarking on our mathematical development, this is a good time to review the material in Sections 2.1, 2.2, 2.3.1, and 2.4.1.

FA's as algebraic systems. An FA M is specified as follows.

$$M \;=\; (Q,\ \Sigma,\ \delta,\ q_0,\ F)$$

where

- Q is a finite set of *states*.
- Σ is a finite *alphabet*.
- δ is the *state-transition function:* $\delta : Q \times \Sigma \;\longrightarrow\; Q$.
  (On the basis of the current state and the most recent input symbol, δ specifies the next state of M.)
- q_0 is M's *initial state;* it is the state M is in when you first "switch it on."
- $F \subseteq Q$ is the set of *final* (or, *accepting*) states.

FA's as labeled digraphs. One can view M as a labeled directed graph (*digraph*, for short), in a natural way.

- The nodes of the graph are the states of M.
  One traditionally represents each final state, $q \in F$, of M by a double circle; each nonfinal state, $q \in Q - F$, by a single circle; and one has a "tail-less" arrow point to the initial state q_0.

- The labeled arcs represent the state transitions. For each state $q \in Q$ and each alphabet symbol $\sigma \in \Sigma$, there is an arc labeled σ leading from state/node q to state/node $\delta(q, \sigma)$.

We shall see in the next two subsections that each of our two formulations of FA's lends powerful intuition to the capabilities and limitations of the devices.

The "behavior" of an FA: the language it accepts. In order to make the FA model *dynamic*, we need to talk about how an FA M responds to strings, not just to single symbols. We must therefore, *extend* the state-transition function δ to operate on $Q \times \Sigma^\star$, rather than just on $Q \times \Sigma$. It is crucial that our extension truly *extend* δ, i.e., that it agree with δ on strings of length 1. We call our extended function $\widehat{\delta}$ and define it via the following induction. For all $q \in Q$:

$$\begin{array}{rcll} \widehat{\delta}(q, \varepsilon) & = & q & \text{(no stimulus} \Rightarrow \text{no response)} \\ (\forall \sigma \in \Sigma,\ \forall x \in \Sigma^\star)\ \widehat{\delta}(q, \sigma x) & = & \widehat{\delta}(\delta(q, \sigma), x) & \text{(the single "}\delta\text{" highlights the \underline{extension})} \end{array}$$

Finally, we have the *language accepted* (or, *recognized*) by M (sometimes called the "behavior" of M):

$$L(M) \stackrel{\text{def}}{=} \{x \in \Sigma^\star \mid \widehat{\delta}(q_0, x) \in F\}.$$

A language L is *regular* (is a *regular set*) iff there is an FA M such that $L = L(M)$.

Since it can cause no confusion to "overload" the semantics of δ, we stop embellishing the extended δ with a hat and just write $\delta : Q \times \Sigma^\star \longrightarrow Q$. Note that we have a long history of doing this: we use "+" for addition of integers, rationals, reals, ..., even though, properly speaking, each successive operation extends its predecessors.

In analogy with the equivalence relation $\equiv_L$ of Eq. 2.4.3, which is associated with a language L, we associate with each FA M the following equivalence relation on $\Sigma^\star$.

$$\text{For all } x, y \in \Sigma^\star : \qquad [x \equiv_M y] \ \text{ iff } \ [\delta(q_0, x) = \delta(q_0, y)]. \tag{3.2.1}$$

(You should verify that $\equiv_M$ is always an equivalence relation on the set $\Sigma^\star$—it is reflexive, symmetric, and transitive.) The following is an immediate consequence of how we extended the state-transition function δ to $Q \times \Sigma^\star$, in particular, the fact that $\delta(q_0, xz) = \delta(\delta(q_0, x), z)$.

Lemma 3.1. *For each FA $M = (Q, \Sigma, \delta, q_0, F)$:*

(a) *the equivalence relation $\equiv_M$ is right-invariant;*

(b) $(\forall x, y \in \Sigma^\star)$ $[x \equiv_M y]$ *iff* $[x \equiv_{L(M)} y]$.

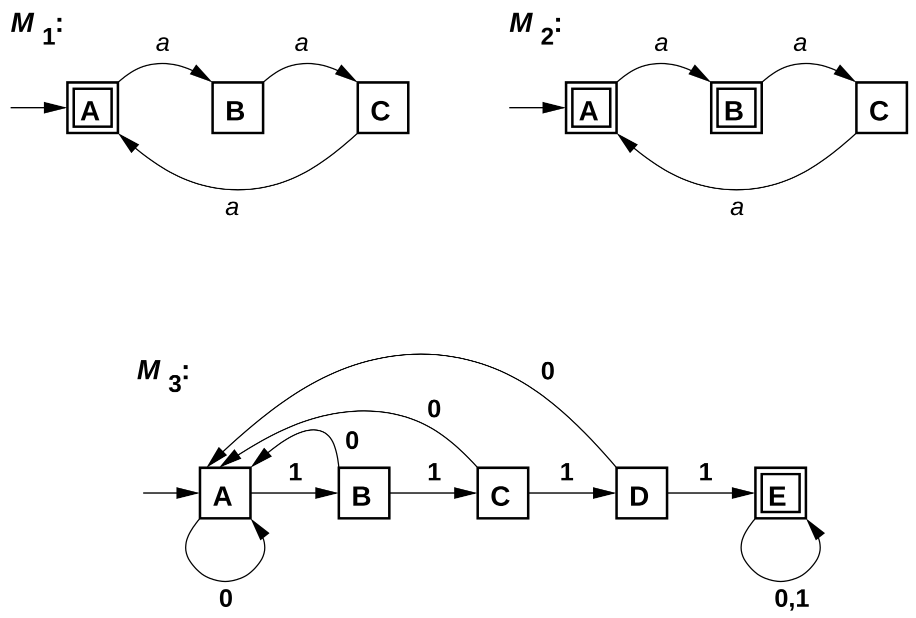

Figure 3.2: A graph-theoretic representation of a 4-state sequential adder.

	M_1	a
$\rightarrow$	$\boxed{\text{A}}$	B
	B	C
	C	A

	M_2	a
$\rightarrow$	$\boxed{\text{A}}$	B
	$\boxed{\text{B}}$	C
	C	A

	M_3	0	1
$\rightarrow$	A	A	B
	B	A	C
	C	A	D
	D	A	E
	$\boxed{\text{E}}$	E	E

Table 3.4: Tabular representations of the FA's of Fig. 3.2.

Reinforcing the preliminaries. Before continuing with our development, we illustrate our definitions with three more simple FA's, to hone the reader's intuition. Fig. 3.2 illustrates the digraph representations of FA's whose structure is specified in Table 3.4. One verifies by inspection that[1]

1. $L(M_1) = \{a^k \mid k \equiv 0 \bmod 3\}$, the set of strings of a's whose length is divisible by 3;

2. $L(M_2) = \{a^k \mid k \not\equiv 2 \bmod 3\}$, the set of strings of a's whose length is congruent to either 0 or 1 modulo 3.

[1] We intentionally use diverse terminology in describing these languages, so that the reader will get familiar with alternative modes of describing the same concept.

3. $L(M_3) = \{x \in \{0,1\}^\star \mid (\exists y \in \{0,1\}^\star)(\exists z \in \{0,1\}^\star)[x = y111z]\}$, the set of binary strings that contain three consecutive 1's.

The preceding examples illustrate an FA's ability to make finitely many discriminations relating to the structure of the string of inputs it has seen thus far. We now show informally, via two examples, that an FA cannot make infinitely many discriminations. We return to these examples with a formal treatment in Section 3.3.2.

Consider the language $L_1 = \{a^n b^n \mid n \in \mathbb{N}\}$ whose constituent strings consist of a block of a's followed by a like-length block of b's. We claim that no FA recognizes L_1. The simplest way of seeing this begins by assuming, for contradiction that there is an FA $M = (Q, \{a,b\}, \delta, q_0, F)$ such that $L_1 = L(M)$. One then notes that, since there are infinitely many finite-length strings of a's, some two distinct ones, say a^i and a^j, must be "confused" by M, in the sense that $\delta(q_0, a^i) = \delta(q_0, a^j) = q$. Since δ is a function, we know that

$$\delta(q, b^i) \ = \ \delta(q_0, a^i b^i) \ = \ \delta(q_0, a^j b^i).$$

The preceding string of equations means that either both $a^i b^i$ and $a^j b^i$ are accepted by M or neither is. In either case, M does not accept the language L_1.

Consider next the language $L_2 = \{a^{n^2} \mid n \in \mathbb{N}\}$ whose constituent strings consist of a block of a's whose length is a perfect square. We claim that no FA recognizes L_1. The simplest way of seeing this begins by assuming, for contradiction that there is an FA $M = (Q, \{a,b\}, \delta, q_0, F)$ such that $L_2 = L(M)$. One then notes that, since there are infinitely many strings of a's of the form a^{k^2}, some two distinct ones, say a^{i^2} and a^{j^2} where $j > i$, must be "confused" by M, in the sense that $\delta(q_0, a^{i^2}) = \delta(q_0, a^{j^2}) = q$. On the one hand, we note that

$$\delta(q, a^{2i+1}) \ = \ \delta(q_0, a^{i^2} a^{2i+1}) \ = \ \delta(q_0, a^{i^2+2i+1}) \ = \ \delta(q_0, a^{(i+1)^2}),$$

which M *should* accept, since $a^{(i+1)^2} \in L_2$. On the other hand, we note that

$$\delta(q, a^{2i+1}) \ = \ \delta(q_0, a^{j^2} a^{2i+1}) \ = \ \delta(q_0, a^{j^2+2i+1}),$$

which M *should not* accept, since $j^2 + 2i + 1 < j^2 + 2j + 1 = (j+1)^2$, so that $a^{j^2+2i+1} \notin L_2$. We thus see that the state $\delta(q, a^{2i+1})$ (which is a unique state because δ is a function) should be an accepting state of M in order to accept $a^{(i+1)^2}$, but it should be a nonaccepting state in order not to accept a^{j^2+2i+1}. We conclude that the FA M cannot exist.

What is the common thread in the arguments about L_1 and L_2? Well, in both cases, we discovered an infinite set of strings that an FA must distinguish in order to accept precisely the strings in the language. The argument is that simple in principle!

We now lay the groundwork to show formally that the *defining characteristic* of regular languages is their requiring only finitely many past histories to be discriminated among in order to act correctly in the future. Each set of *un*-discriminated histories comprises what we have called a "state." We now develop a formal mathematical characterization of "state" that forms the desired groundwork and that has wide-ranging applications within the theory of Finite Automata.

3.3 The Myhill-Nerode Theorem and Applications

The notion of state is fundamental to the design and analysis of virtually all computational systems, from the sequential circuits that underlie sophisticated hardware, to the semantic models that enable optimizing compilers, to leading-edge machine-learning concepts, to the models used in discrete-event simulation. Decades of experience with state-based systems have taught that all but the simplest display a level of complexity that makes them hard—conceptually and/or computationally—to design and analyze. One brilliant candle in this gloomy scenario is the Myhill-Nerode Theorem, which supplies a *rigorous, mathematical*, analogue of the following informal characterization of the notion "state."

> The state of a system comprises that fragment of its history that allows it to behave correctly in the future.

Superficially, it may appear that this definition of "state" is of no greater *operational* significance than is the foundational identification of the number *eight* with the infinitude of sets that contain eight elements. This appearance is illusory. The Myhill-Nerode Theorem turns out to be a conceptual and technical powerhouse when analyzing a surprising range of problems concerning the state-transition systems that occur in so many guises within the field of computation. Indeed, although the Theorem resides most naturally within the theory of Finite Automata—it first appeared in [28]; an earlier, weaker version appeared in [27]; the most accessible presentation appeared in [30]—it has manifold lessons for the analysis of many problems associated with any state-transition system, even those having infinitely many states.

3.3.1 The Theorem

We now prepare for our presentation of the Myhill-Nerode Theorem, which supplies a rigorous mathematical correspondent of the notion of "state." We begin with some basic definitions, facts, and notation. Let $\equiv$ be any equivalence relation on $\Sigma^\star$.

- For each $x \in \Sigma^\star$, the $\equiv$-*class* that x belongs to is $[x]_\equiv \overset{\text{def}}{=} \{y \in \Sigma^\star \mid x \equiv y\}$.

  (When the subject relation $\equiv$ is clear from context, we simplify notation by writing $[x]$ for $[x]_\equiv$.)

- The *classes* of $\equiv$ *partition* $\Sigma^\star$.
- The *index* of $\equiv$ is the number of *classes* that it partitions $\Sigma^\star$ into.

Theorem 3.1 ([27, 28, 30]). **(The Myhill-Nerode Theorem)**
The following statements about a language $L \subseteq \Sigma^\star$ are equivalent.

1. *L is regular.*
2. *L is the union of some of the equivalence classes of a right-invariant equivalence relation over $\Sigma^\star$ of finite index.*
3. *The right-invariant equivalence relation, $\equiv_L$ of Eq. 2.4.3 has finite index.*

Note. *The earliest version of the Theorem, in [27], uses* congruences*—i.e., equivalence relations that are both right- and left-invariant.*

Proof. We prove the (logical) equivalence of the Theorem's three statements by verifying the three cyclic implications: statement 1 implies statement 2, which implies statement 3, which implies statement 1.

(1) $\Rightarrow$ (2). Say that the language L is regular. There is, then, a FA $M = (Q, \Sigma, \delta, q_0, F)$ such that $L = L(M)$. Then the right-invariant equivalence relation $\equiv_M$ of Eq. 3.2.1 clearly has index no greater than $|Q|$. Moreover, L is the union of some of the classes of relation $\equiv_M$:

$$L \;=\; \{x \in \Sigma^\star \mid \delta(q_0, x) \in F\} \;=\; \bigcup_{f \in F} \{x \in \Sigma^\star \mid \delta(q_0, x) = f\}.$$

(2) $\Rightarrow$ (3). We claim that if L is "defined" via some (any) finite-index right-invariant equivalence relation, $\equiv$, on $\Sigma^\star$, in the sense of statement 2, then the specific right-invariant equivalence relation $\equiv_L$ has finite index. We verify the claim by showing that the relation $\equiv$ must *refine* relation $\equiv_L$, in the sense that every equivalence class of $\equiv$ is totally contained in some equivalence class of $\equiv_L$. To see this, consider any strings $x, y \in \Sigma^\star$ such that $x \equiv y$. By right invariance, then, for all $z \in \Sigma^\star$, we have $xz \equiv yz$. Since L is, by assumption, the union of entire classes of relation $\equiv$, we must have

$$[xz \in L] \quad \text{if, and only if,} \quad [yz \in L].$$

We thus have

$$[x \equiv y] \;\Rightarrow\; [x \equiv_L y].$$

Since relation $\equiv$ has only finitely many classes, and since each class of relation $\equiv$ is a subset of some class of relation $\equiv_L$, it follows that relation $\equiv_L$ has finite index.

(3) $\Rightarrow$ (1). Say that L is the union of some of the classes of the finite-index right-invariant equivalence relation $\equiv_L$ on $\Sigma^\star$. Let the distinct classes of $\equiv_L$ be $[x_1]$, $[x_2]$, ..., $[x_n]$, for some n strings $x_i \in \Sigma^\star$. (Note that, because of the transitivity of relation $\equiv_L$, we can identify a class uniquely via any one of its constituent strings. This works, of course, for any equivalence relation.) We claim that these classes form the states of an FA $M = (Q, \Sigma, \delta, q_0, F)$ that accepts L. To wit:

1. $Q = \{[x_1], [x_2], \ldots, [x_n]\}$.
   This set is finite because $\equiv_L$ has finite index.

2. For all $x \in \Sigma^\star$ and all $\sigma \in \Sigma$, define $\delta([x], \sigma) = [x\sigma]$.
   The right-invariance of relation $\equiv_L$ guarantees that δ is a well-defined function.

3. $q_0 = [\varepsilon]$.
   M's start state corresponds to its having read nothing.

4. $F = \{[x] \mid x \in L\}$

One verifies by an easy induction that M is a well-defined FA that accepts L. $\square$

3.3.2 Using the MNT to prove that languages are nonregular

FA's are very limited in their computing power due to the finiteness of their memories, i.e., of their sets of states. Indeed, the standard way to expose the limitations of FA's—by proving that a language L is not regular—is to establish somehow that the structure of L requires distinguishing among infinitely many mutually distinct situations.

The Continuation Lemma and fooling sets. Given the conceptual parsimony and power of Theorem 3.1, it is not surprising that the Theorem affords one a simple, yet powerful tool for proving that a language is not regular. This tool is encapsulated in the following corollary, which follows immediately from the equivalence of statements (1) and (3) in the Theorem. For reasons that we hope will become suggestive imminently, we refer to the corollary as "The Continuation Lemma." We maintain that the ensuing development should be viewed as *the primary tool* for proving that a language is not regular.

Lemma 3.2. (The Continuation Lemma)
Let $L \subseteq \Sigma^\star$ be an infinite regular language. Every sufficiently large set of words over Σ contains at least two words x, y such that $x \equiv_L y$.

Proof. By direct calculation, we find that:

$$\delta(q_0, xz) \;=\; \delta(\delta(q_0, x), z) \;=\; \delta(\delta(q_0, y), z) \;=\; \delta(q_0, yz).$$

(Note that we proceed from expression #2 to expression #3 via the universal algebraic operation of substituting equals for equals.) Underlying this argument is the inductive extension of δ to $Q \times \Sigma^\star$. □

> The Continuation Lemma has a natural interpretation in terms of FA's, namely, that an FA M has no "memory of the past" other than its current state. Specifically, if strings x and y lead M to the same state (from its initial state)—i.e., if $\delta(q_0, x) = \delta(q_0, y)$, or, in our shorthand, $x \equiv_M y$—then no continuation/extension of the input string will ever allow M to determine which of x and y it actually read. (Note that the Lemma tells us that, for any FA M, the equivalence relation $\equiv_M$ is *right-invariant:* If $x \equiv_M y$, then, for all $z \in \Sigma^\star$, $xz \equiv_M yz$.

One applies the Continuation Lemma to the problem of showing that an infinite[2] language $L \subseteq \Sigma^\star$ is not regular by constructing a *fooling set for* L, i.e., an infinite set of words no two of which are equivalent with respect to L. In other words, an infinite set $S \subseteq \Sigma^\star$ is a fooling set for L if for every pair of words $x, y \in S$, there exists a word $z \in \Sigma^\star$ such that precisely one of xz and yz belongs to L.

> This technique has a natural interpretation in terms of FA's. Since any FA M has only finitely many states, any infinite set of words must (by the so-called "pigeonhole principle"[3]) always contain two, x and y, that are indistinguishable to M, in the sense that $x \equiv_M y$ (so that $x \equiv_{L(M)} y$; cf. Lemma 3.1). By the FA version of the Continuation Lemma, no continuation z can ever cause M to distinguish between x and y.

We now consider a few sample proofs of the nonregularity of languages, which suggest how direct and simple such proofs can be when they are based on the Continuation Lemma and fooling sets.

Application 1. *The language*[4] $L_1 \;=\; \{a^n b^n \mid n \in \mathbb{N}\} \;\subset\; \{a, b\}^\star$ *is not regular.*

We claim that the set $S_1 = \{a^k \mid k \in \mathbb{N}\}$ is a fooling set for L_1. To see this, note that, for any distinct words $a^i, a^j \in S_1$, we have $a^i b^i \in L_1$ while $a^j b^i \notin L_1$; hence, $a^i \not\equiv_{L_1} a^j$. By Lemma 3.2, L_1 is not regular. □

[2]You will be asked in the Exercises to show that every finite language is regular.

[3]The "pigeonhole principle" asserts: *If you put* $n + 1$ *balls into* n *bins, some bin must receive more than one ball.* It is sometime called "Dirichlet's Box Principle."

[4]a^n denotes a string of n occurrences of string (or symbol) a.

Application 2. *The language $L_2 = \{a^k \mid k$ is a perfect square$\}$ is not regular.*

This application requires a bit of subtlety. We claim that L_2 is a fooling set for itself! To see this, consider any distinct words $a^{i^2}, a^{j^2} \in L_2$, where $j > i$. On the one hand, $a^{i^2}a^{2i+1} = a^{i^2+2i+1} = a^{(i+1)^2} \in L_2$; on the other hand, $a^{j^2}a^{2i+1} = a^{j^2+2i+1} \notin L_2$, because $j^2 < j^2 + 2i + 1 < (j+1)^2$; hence, $a^{i^2} \not\equiv_{L_2} a^{j^2}$. By Lemma 3.2, L_2 is not regular. $\square$

Applications 3 and 4. *The language*[5]

$$L_3 \ = \ \{x \in \{0,1\}^\star \mid x \text{ reads the same forwards and backwards; symbolically, } x = x^R\}$$

(whose words are often called "palindromes"), and the language

$$L_4 = \{x \in \{0,1\}^\star \mid (\exists y \in \{0,1\}^\star)[x = yy]\}$$

(whose words are often called "squares"), are not regular.

We claim that the set $S_3 = \{10^k1 \mid k \in \mathbb{N}\}$ is a fooling set for both L_3 and L_4. To see this, consider any pair of distinct words, 10^i1 and 10^j1, from S_3. On the one hand, $10^i110^i1 \in L_3 \cap L_4$; on the other hand, $10^j110^i1 \notin L_3 \cup L_4$; hence, $10^i1 \not\equiv_{L_3} 10^j1$, and $10^i1 \not\equiv_{L_4} 10^j1$. By Lemma 3.2, neither L_3 nor L_4 is regular. $\square$

3.3.3 Using the MNT to minimize Finite Automata

Theorem 3.1 and its proof tell us two important things.

1. The notion of "state" underlying the FA model is embodied in the relations $\equiv_M$. More precisely, a state of an FA is a set of input strings that the FA "identifies," because—and so that—any two strings in the set are indistinguishable with respect to the language the FA accepts.

2. The *coarsest*—i.e., smallest-index—equivalence relation that "works" is $\equiv_L$, so that this relation embodies the *smallest* FA that accepts language L.

We can turn the preceding intuition into an algorithm for minimizing the state-set of a given FA. You can look at this algorithm as starting with any given equivalence relation that "defines" L (e.g., with any FA that accepts L) and iteratively "coarsifying" the relation as far as we can, thereby "sneaking" up on the relation $\equiv_L$.

The resulting algorithm for minimizing a FA $M = (Q, \Sigma, \delta, q_0, F)$ essentially computes the following equivalence relation on M's state-set Q. For $p, q \in Q$,

$$[p \equiv_\delta q] \text{ if, and only if } (\forall x \in \Sigma^\star)[[\delta(p,x) \in F] \Leftrightarrow [\delta(q,x) \in F]]$$

[5] x^R denotes string x written backwards; e.g., $(\sigma_1\sigma_2\cdots\sigma_{n-1}\sigma_n)^R = \sigma_n\sigma_{n-1}\cdots\sigma_2\sigma_1$.

This relation says that no input string will allow one to distinguish M's being in state p from M's being in state q. One can, therefore, coalesce states p and q to obtain a smaller FA that accepts $L(M)$. The equivalence classes of $\equiv_\delta$, i.e., the set

$$\{[p]_{\equiv_\delta} \mid p \in Q\}$$

are, therefore, the states of the smallest FA—call it $\widehat{M}$—that accepts $L(M)$. The state-transition function $\widehat{\delta}$ of $\widehat{M}$ is given by

$$\widehat{\delta}([p]_{\equiv_\delta}, \sigma) \ = \ [\delta(p, \sigma)]_{\equiv_\delta}.$$

Finally, the initial state of $\widehat{M}$ is $[q_0]_{\equiv_\delta}$, and the accepting states are $\{[p]_{\equiv_\delta} \mid p \in F\}$. One shows easily that $\widehat{\delta}$ is well defined and that $L(\widehat{M}) = L(M)$.

We simplify our explanation of how to compute $\equiv_\delta$ by describing an example concurrently with our description of the algorithm. We start with a very coarse approximation to $\equiv_\delta$ and iteratively improve the approximation. Fig. 3.3 presents an FA

$$M = (\{a, b, c, d, f, g, h\}, \{0, 1\}, \delta, a, \{c\})$$

in tabular form.

M	q	$\delta(q,0)$	$\delta(q,1)$	$q \in F$?
(start state) $\longrightarrow$	a	b	f	$\notin F$
	b	g	c	$\notin F$
(final state) $\longrightarrow$	c	a	c	$\in F$
	d	c	g	$\notin F$
	e	h	f	$\notin F$
	f	c	g	$\notin F$
	g	g	e	$\notin F$
	h	g	c	$\notin F$

Figure 3.3: The FA M that we minimize.

Our initial partition[6] of Q is $\langle Q - F,\ F\rangle$, to indicate that the null string ε witnesses the fact that no accepting state is equivalent to any nonaccepting state. This yields the initial partition of M's states:

$$[a, b, d, e, f, g, h]_1, \ [c]_1$$

(The subscript "1" indicates that this is the first discriminatory step). State c, being the unique final state, is not equivalent to any other state.

[6]Recalling that partitions and equivalence relations are equivalent notions, we continue to use notation "$[a, b, \ldots, z]$" to denote the set $\{a, b, \ldots, z\}$ viewed as a block of a partition (= equivalence class).

Inductively, we now look at the current, time-t, partition and try to "break apart" time-t blocks. We do this by feeding single input symbols to pairs of states in the same block. If any symbol leads states p and q to different blocks, then, by induction, we have found a string x that discriminates between them. In detail, say that $\delta(p, \sigma) = r$ and $\delta(q, \sigma) = s$. If there is a string x that discriminates between states r and s—by showing them not to be equivalent under $\equiv_\delta$—then the string σx discriminates between states p and q. In our example, we find that input "0" breaks the big time-1 block, so that we get the "time 1.5" partition

$$[a, b, e, g, h]_{1.5},\ [d, f]_{1.5},\ [c]_{1.5}$$

and input "1" further breaks the block down. We end up with the time-2 partition

$$[a, e]_2,\ [b, h]_2,\ [g]_2,\ [d, f]_2,\ [c]_2$$

Let's see how this happens. First, we find that $\delta(d, 0) = \delta(f, 0) = c \in F$, while $\delta(q, 0) \notin F$ for $q \in \{a, b, e, g, h\}$. This leads to the "time-1.5" partition (since we have thus far used only one of the two input symbols). At this point, input "1" leads states a and e to block $\{d, f\}$, and it leads states b and h to block $\{c\}$; it leaves state g in its present block. We thus end up with the indicated time-2 partition. Further single inputs leave this partition unchanged, so it must be the coarsest partition that preserves $L(M)$.

The preceding sentence invokes the fact that, by a simple induction, if a partition persists under (i.e., is unchanged by) all single inputs, then it persists under all input strings. We claim that such a stable partition embodies the relation $\equiv_M$, hence, by Lemma 3.1, the relation $\equiv_{L(M)}$. To see this, consider any two states, p and q, that belong to the same block of a partition that persists under all input strings. Stability ensures that, for all $z \in \Sigma^\star$, the states $\delta(p, z)$ and $\delta(q, z)$ belong to the same block of the partition; hence, either both states belong to F or neither does. In other words: If $\delta(q_0, x) = p$ and $\delta(q_0, y) = q$, for $x, y \in \Sigma^\star$, then for all $z \in \Sigma^\star$, either $\{p, q\} \subseteq F$, in which case $\{xz, yz\} \subseteq L(M)$, or $\{p, q\} \subseteq Q - F$, in which case $\{xz, yz\} \subseteq \Sigma^\star - L(M)$. By definition, then, $x \equiv_M y$.

Returning to the algorithm, we have ended up with the FA $\widehat{M}$ of Fig. 3.4 as the minimum-state version of M.

$\widehat{M}$	q	$\widehat{\delta}(q, 0)$	$\widehat{\delta}(q, 1)$	$q \in F$?
(start state) $\rightarrow$	$[ae]$	$[bh]$	$[df]$	$\notin F$
	$[bh]$	$[g]$	$[c]$	$\notin F$
(final state) $\rightarrow$	$[c]$	$[ae]$	$[c]$	$\in F$
	$[df]$	$[c]$	$[g]$	$\notin F$
	$[g]$	$[g]$	$[ae]$	$\notin F$

Figure 3.4: The FA $\widehat{M}$ that minimizes the FA M of Fig.3.3.

3.4 The Kleene-Myhill Theorem; Regular Expressions

3.4.1 Nondeterministic FA's and their power

As noted earlier, one can view FA as abstract representations of actual circuits or machines or programs. In contrast, the generalization of FA's that we present now is a mathematical abstraction that cannot be realized from conventional hardware or software elements. It is best to view this model either as a purely mathematical convenience—whose utility we shall see imminently—or as a "strategy" that we shall try to realize via sophisticated transformation.

Here is how the abstraction works. One can view an FA as "making a decision" upon receipt of an input symbol: the decision resides in the choice of next state. The new abstraction is endowed with the ability to "hedge its bets" in this decision-making process. One can view this hedging as residing in the new abstraction's ability to create "alternative universes," making a possibly distinct decision in each universe. Thus, whereas a computation by an FA on a string $\sigma_0\sigma_1\sigma_2\cdots\sigma_k$ can be viewed as a linear sequence

$$q_0 \xrightarrow{\sigma_0} q_1 \xrightarrow{\sigma_1} q_2 \xrightarrow{\sigma_2} \cdots \xrightarrow{\sigma_k} q_{k+1} \tag{3.4.2}$$

in our generalized setting, a "computation" must be viewed as a *forest.* The reason that it's a forest rather than a tree is that we allow the FA to "hedge its bets" even before the computation starts—by beginning in a set of start states. The analogue of the computation (3.4.2) now has a form like the following (where we split universes in two only for convenience):

$$\begin{tikzcd}
 & & & & q_0 \arrow[dll] \arrow[drr] & & & & & & & & q'_0 \arrow[dll] \arrow[drr] & & & \\
\sigma_0 & & q_{11} \arrow[dl] \arrow[dr] & & & & q_{12} \arrow[dl] \arrow[dr] & & & & q'_{11} \arrow[dl] \arrow[dr] & & & & q'_{12} \arrow[dl] \arrow[dr] & \\
\sigma_1 & q_{21} & & q_{22} & & q_{23} & & q_{24} & & q'_{21} & & q'_{22} & & q'_{23} & & q'_{24} \\
\vdots & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots & & \vdots
\end{tikzcd} \tag{3.4.3}$$

In order to flesh out the model, we must, of course, indicate when a nondeterministic finite automaton "accepts" a string. In the deterministic setting, acceptance resides in the fact that the final state q_{k+1} in computation (3.4.2) is an accepting state. In the nondeterministic setting, after reading an input string $\sigma_0\sigma_1\sigma_2\cdots\sigma_k$, the automaton is in different states in different universes: in computation (3.4.3), after reading $\sigma_0\sigma_1$, the automaton is in up to eight states, $q_{21}, q_{22}, q_{23}, q_{24}, q'_{21}, q'_{22}, q'_{23}, q'_{24}$ (which need not all be distinct) in its various universes. In this case, we say that the automaton accepts an input string if *at least one of the states that the string leads to is an accepting state.* Nondeterminism as thus-construed is built around an *existential quantifier—"there exists a path to an accepting state."*

Formalizing the preceding discussion, a *nondeterministic finite automaton* (*NFA*, for short) is a system $M = (Q, \Sigma, \delta, Q_0, F)$ where:

1. Q, Σ, and F play the same roles as with a *deterministic* FA (henceforth called a *DFA*);
2. Q_0 is M's *set* of initial states;
3. M's state transitions take sets of states to sets of states.

We elaborate on the third of these points. Letting $\mathcal{P}(Q)$ denote the *power set* of Q—i.e., the set of all subsets of Q (including the empty set $\emptyset$)—we have

$$\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q).$$

We extend δ to sets of states—i.e., to a function

$$\delta : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q),$$

in the natural way, via unions: For any subset $Q' \subseteq Q$ and $\sigma \in \Sigma$:

$$\delta(Q', \sigma) \ = \ \bigcup_{q \in Q'} \delta(q, \sigma).$$

There is a natural inductive extension of δ to $\mathcal{P}(Q) \times \Sigma^\star$. For any subset $Q' \subseteq Q$:

$$\delta(Q', \varepsilon) \ = \ Q',$$

and, for all $\sigma \in \Sigma$ and $x \in \Sigma^\star$,

$$\delta(Q', \sigma x) \ = \ \bigcup_{p \in \delta(Q', \sigma)} \delta(\{p\}, x).$$

Acceptance is now formalized by the following condition.

$$L(M) \ = \ \{x \in \Sigma^\star \mid \delta(Q_0, x) \cap F \neq \emptyset\}.$$

You should make sure that you see the correspondence between the formal setting of an NFA and its language, as just described, and our intuitive description preceding the formalism.

The concept of nondeterminism plays a central role in the theory of computation. In the case of finite automata and regular languages, the concept is just a convenience, as we shall see imminently. In the case of more powerful automata, the concept plays an essential role: Context-Free Languages are precisely the languages accepted by *nondeterministic* pushdown automata [5, 6]; Context-Sensitive Languages are precisely the languages accepted by *non-deterministic* linear-bounded automata [5, 6]; one of the—perhaps *the*—central question in Complexity Theory asks how much computational resource is needed to simulate nondeterminism deterministically. The most famous instance of the last point is the **P** vs. **NP** question [7].

In the theory of finite automata and regular languages, nondeterminism is used as a simplifying conceptual tool for proving the Kleene-Myhill Theorem, which we cover in the next section. (It is just a convenience! Kleene and Myhill proved the Theorem arduously, without the help of nondeterminism.) We show now that nondeterminism is no more than a convenience, by showing that NFA's accept only regular languages.

Theorem 3.2. *Every language accepted by an NFA $M = (Q, \Sigma, \delta, Q_0, F)$ is regular.*

Proof. Our proof relies on the following intuition, which is discernible in an NFA's acceptance criterion. Say that the NFA M has thus far read the string $x \in \Sigma^\star$ (and has bifurcated into some number of independent universes). If we want to determine how having read x will influence M's subsequent behavior, all we need to know is the *set* of states $\delta(Q_0, x)$. The number of occurrences of a state q in this set is immaterial—as long as it is not 0. It follows that we can simulate the computation of M on any string z just by keeping track of the successive sets of states that the successive symbols of z lead M to. Since M's state-set Q is finite, so also is the set $\mathcal{P}(Q)$. Therefore, we can actually construct a *DFA* $M' = (Q', \Sigma, \delta', q_0', F')$ that can simulate *all* of M's computational universes simultaneously, in the sense that $L(M') = L(M)$. The DFA M' need only keep track of the sequence of sets of states that an input string would lead M through. The following so-called *subset construction* achieves this. We define the various components of the DFA M':

- $Q' = \mathcal{P}(Q)$. This allows M' to keep track of sets of M's states.
- For all $R \in \mathcal{P}(Q)$ (equivalently, for all $R \subseteq Q$) and all $\sigma \in \Sigma$,

  $$\delta'(R, \sigma) \ = \ \bigcup_{r \in R} \delta(r, \sigma).$$

  This allows M' to follow M from one set of states to that set's successor under input σ.
- $q_0' = Q_0$. This is because M starts out in the set of states Q_0.
- $F' = \{R \in \mathcal{P}(Q) \mid R \cap F \neq \emptyset\}$. This captures the fact that acceptance by M requires attainment of one or more (accepting) states of F.

Our intuitive justifications for each component of M' can be turned into the inductive proof one finds in HU. You should try to craft the required induction on your own. $\square$

Since NFA's accept only regular languages, nondeterminism is, indeed, at most a mathematical convenience for us. Since we are being kind to ourselves by allowing ourselves this convenience, let us continue to be kind by making the model of NFA even easier to use within the context of the Kleene-Myhill Theorem (specifically, the half of the Theorem that produces an NFA from a Regular Expression). We do this by enhancing NFA's to allow them to have so-called *ε-transitions*.

An *ε-nondeterministic finite automaton* (*ε-NFA*, for short) $M = (Q, \Sigma, \delta, Q_0, F)$ is an NFA whose state-transition function δ is extended to allow *spontaneous* so-called ε-transitions:

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$$

Allowing a transition on "input" ε—which is really the *absence* of an input—will simplify some of our constructions in the next section. We show now that such transitions do not augment the power of NFA's.

Theorem 3.3. *Every language accepted by an ε-NFA $M = (Q, \Sigma, \delta, Q_0, F)$ is regular.*

Proof. For each state $q \in Q$, we define the ε-reachability set of state q, denoted $E(q)$, as follows.

$$E(q) \stackrel{\text{def}}{=} \{p \in Q \mid [p = q] \text{ or } (\exists p_1, p_2, \ldots, p_n \in Q)[q \xrightarrow{\varepsilon} p_1 \xrightarrow{\varepsilon} p_2 \xrightarrow{\varepsilon} \cdots \xrightarrow{\varepsilon} p_n \xrightarrow{\varepsilon} p]\}$$

Essentially, $E(q)$ is the set of states that state q can reach spontaneously, i.e., via ε-transitions. (One sometimes sees $E(q)$ called something like the "ε-closure" of state q.)

Using this construct, we now present an NFA $M'' = (Q, \Sigma, \delta'', Q_0'', F'')$ (that has no ε-transitions) that is equivalent to M.

$$\begin{aligned} (\forall \sigma \in \Sigma,\ Q' \subseteq Q)\ \ \delta''(Q', \sigma) &= \bigcup_{p \in \delta(Q', \sigma)} E(p); \\ Q_0'' &= \bigcup_{q \in Q_0} E(q); \\ F'' &= F \cup \{q \in Q_0 \mid F \cap E(q) \neq \emptyset\} \end{aligned}$$

M'' thus systematically traces through, and collapses, all of M's ε-transitions. Specifically, M'' starts out in all states that M does, either directly or via ε-transitions; M'' makes all state transtions that M does, either directly or via ε-transitions; M'' accepts any string that M does, either directly or via ε-transitions. □

With the results of this section in our toolkit, we can henceforth wander freely within the world of DFA's, NFA's, and ε-NFA's when studying regular languages. We make use of this freedom in the next section.

3.4.2 The Kleene-Myhill Theorem

The Myhill-Nerode Theorem characterizes the regular languages by characterizing the ability of DFA's to make discriminations among input strings. One can also characterize the regular languages via the operations that suffice to build them up from the letters of the input alphabet. This characterization, which culminates in the Kleene-Myhill Theorem, also gives rise to a useful notation, called Regular Expressions, for assigning names to the regular languages.

Three basic operations on languages. The Kleene-Myhill Theorem describes a sense in which the following three operations explain the inherent nature of regular languages over a given alphabet Σ.

Union. The *union* of languages L_1 and L_2 is just the set-theoretic union $L_1 \cup L_2$, as defined in Section 2.1.

Concatenation. The *concatenation*[7] of languages L_1 and L_2 (in that order!) is:

$$L_1 \cdot L_2 \stackrel{\text{def}}{=} \{xy \in \Sigma^\star \mid [x \in L_1] \text{ and } [y \in L_2]\}.$$

Thus, $L_1 \cdot L_2$ consists of all strings that have a prefix in L_1, whose removal leaves a string in L_2. Since languages need not be "prefix-free"—i.e., since all of the strings $u1$, u_1u_2, …, $u_1u_2 \cdots u_k$ may belong to L_1—recognizing concatenations is a "very nondeterministic" operation: each encountered prefix in L_1 could be the x that one is looking for, or just a prefix of that x.

"Powers" of a language. For any language L and integer $k \geq 0$:

$$L^0 \stackrel{\text{def}}{=} \{\varepsilon\} \quad \text{and, inductively,} \quad L^{k+1} \stackrel{\text{def}}{=} L \cdot L^k.$$

Clearly, $L^1 = L$.

Star-closure. The *star-closure* of a language L, denoted $L^\star$, is, informally, all finite concatenations of strings from L. Formally,

$$L^\star \stackrel{\text{def}}{=} \bigcup_{i=0}^{\infty} L^i \;=\; \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \cdots$$

Note the somewhat unintuitive fact that $\emptyset^\star = \{\varepsilon\}$. (In fact, $\emptyset^\star$ is the only finite star-closure language.)

Of course, the fact that star-closure involves iterated concatenation makes it "even more nondeterministic" than concatenation.

[7]Concatenation is the *complex product* operation of algebra, applied to the free semigroup generated by Σ.

Regular Expressions. Always remember that a Regular Expression over an alphabet Σ is just a (finite) string! The Expression *denotes* a (possibly infinite) language $L \subseteq \Sigma^\star$—*but the Expression is not a language!* (It is the *name* of a language.)

Here is the inductive definition of a Regular Expression $\mathcal{R}$ over Σ, accompanied by the "interpretation" of the Expression, in terms of the language $\mathcal{L}(\mathcal{R})$ that it denotes.

	Atomic Regular Expressions	
	Regular Expression $\mathcal{R}$	Associated Language $\mathcal{L}(\mathcal{R})$
	$\emptyset$	$\emptyset$
	ε	$\{\varepsilon\}$
For $\sigma \in \Sigma$:	σ	$\{\sigma\}$
	Composite Regular Expressions	
For RE's $\mathcal{R}_1, \mathcal{R}_2$:	$(\mathcal{R}_1 + \mathcal{R}_2)$	$\mathcal{L}(\mathcal{R}_1) \cup \mathcal{L}(\mathcal{R}_2)$
For RE's $\mathcal{R}_1, \mathcal{R}_2$:	$(\mathcal{R}_1 \cdot \mathcal{R}_2)$	$\mathcal{L}(\mathcal{R}_1) \cdot \mathcal{L}(\mathcal{R}_2)$
For any RE $\mathcal{R}$:	$(\mathcal{R}^\star)$	$(\mathcal{L}(\mathcal{R}))^\star$

Informally, we sometimes violate the formal rules and omit parentheses and dots, as in $a^\star b^\star$ for $((a)^\star) \cdot ((b)^\star)$. But when we are being formal, we can never leave anything out!

The Kleene-Myhill Theorem. We now have the theorem that exposes a sense in which Regular Expressions tell the entire story of regular languages.

Theorem 3.4. (The Kleene-Myhill Theorem) *A language is regular if, and only if, it is definable by a Regular Expression. In other words:*

The family of regular languages over an alphabet Σ is the smallest family of subsets of $\Sigma^\star$ that contains all finite languages over Σ (including $\emptyset$ and $\{\varepsilon\}$) and that is closed under a finite number of applications of the operations of union, concatenation, and star-closure.

We prove Theorem 3.4 via the following two lemmas.

Lemma 3.3. *If the language $L \subseteq \Sigma^\star$ is denoted by a Regular Expression R, then L is a regular language. (This is another way of saying that the family of regular languages is closed under the operations of union, concatenation, and star-closure.)*

Proof. We present schematic intuitive arguments that can easily be turned into inductive proofs. We start with the NFA's M_1 and M_2 in Fig. 3.5.

Union. We build an NFA $M_{1,2}$ that accepts $L(M_1) \cup L(M_2)$, as follows. We take M_1 and M_2 and "defrock" their start states: these states still exist, but they are no longer start

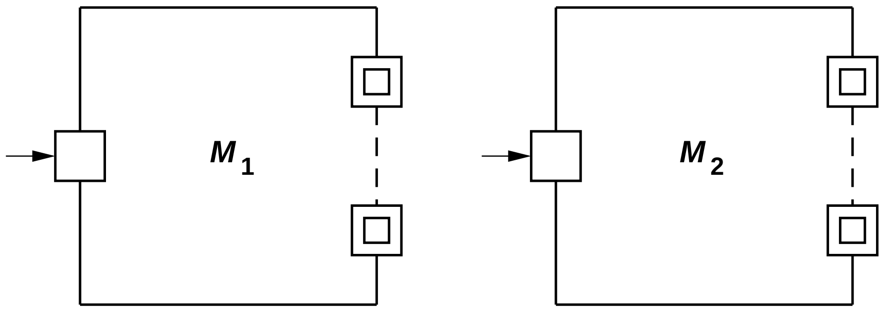

Figure 3.5: A schematic depiction of the NFA's M_1 and M_2. Small squares denote states; squares with inscribed squares are accepting states. The dashed lines indicate that we make no assumptions about the number of accepting states. The arrows point to the start states.

states. We then endow $M_{1,2}$ with new, nonaccepting, start state whose only transitions are ε-transitions to what used to be the start states of M_1 and M_2; see Fig. 3.6. Clearly, the only path in $M_{1,2}$ from the start state to an accepting state consists of the ε-transition from the start state to what was the start state of either M_1 or M_2—say, without loss of generality, M_1—followed by a path from that state to a final state of M_1. It follows that $M_{1,2}$ accepts a string if, and only if, either M_1 or M_2 did.

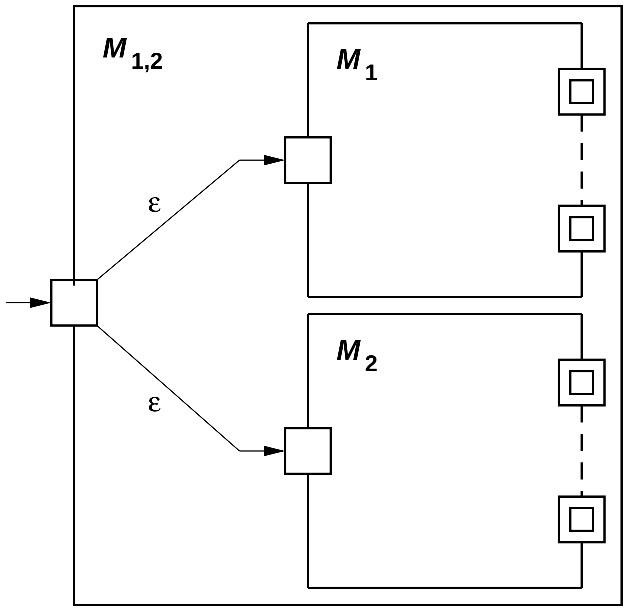

Figure 3.6: A schematic depiction of the "union" NFA $M_{1,2}$.

Concatenation. We build an NFA $M_{1.2}$ that accepts $L(M_1) \cdot L(M_2)$ as follows. We take M_1 and M_2 and "defrock" M_2's start state: it still exists, but it is no longer a start state.

The start state of M_1 becomes $M_{1.2}$'s start state. Next, we "defrock" M_1's accepting states: these states still exist, but they are no longer accepting states. Finally, we add ε-transitions from what used to be M_1's accepting states, to what used to be M_2's start state; see Fig. 3.7. The net effect of this construction is that whenever $M_{1.2}$ has read a string x that would lead M_1 to one of its accepting states—so that x belongs to $L(M_1)$—$M_{1.2}$ continues process any continuation of x within M_1, but it also passes that continuation through M_2. It follows that, if there is any way to parse the input to M into the form xy, where $x \in L(M_1)$ and $y \in L(M_2)$, then $M_{1.2}$ will find it—via the inserted ε-transition. Conversely, if that ε-transition leads $M_{1.2}$ to an accepting state, then the successful input must admit the desired parse.

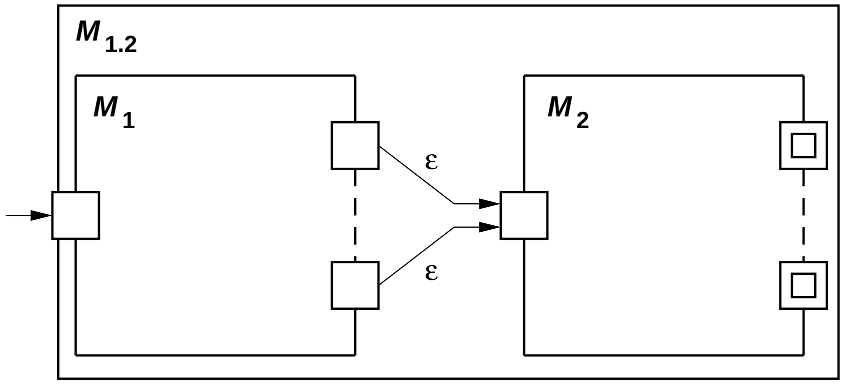

Figure 3.7: A schematic depiction of the "concatenation" NFA $M_{1.2}$.

Star-Closure. We build an NFA M_* that accepts $(L(M_1))^\star$. The only delicate issue here is that we must take care that M_* accepts ε, as well as all positive powers of $L(M_1)$. Taking care of this delicacy first, we give M_* a new start state that is also an accepting state: M_1's start state stays around, but is "defrocked" (as a start state). The sole transition from this new start state is a ε-transition to M_1's (now "defrocked") old start state. Next, we defrock all of M_1's accepting states, and we add an ε-transition from each of these to M_*'s new start state (which, recall, is an accepting state). What we have accomplished is the following. If M_* reads an input x that would accepted by M_1, then it hedges its bets. On the one hand, it keeps reading x, thereby seeking continuations of x that are also in $L(M_1)$; additionally, though, it assumes that the continuation of x is an independent string in $L(M_1)$, so it starts over with the start state of M_1.

Explained in another way, the NFA M_* implements the following identity, which holds for arbitrary languages L.

$$L^\star \;=\; \{\varepsilon\} \cup L \cdot L^\star$$

To see this equation in the construction of M_*, note the following.

1. The new accepting start state that we have endowed M_* with ensures that $\varepsilon \in L(M_*)$.

2. The ε-transitions from M_1's (now "defrocked") accepting states to the new start state ensures that M_* accepts every string in the concatenation of L (since we have arrived at a state that is an accepting state of M_1) and $L^\star$ (by induction, since we are branching back to the start state of M_*).

We thus see that $L(M_*) = (L(M_1))^\star$, as claimed.

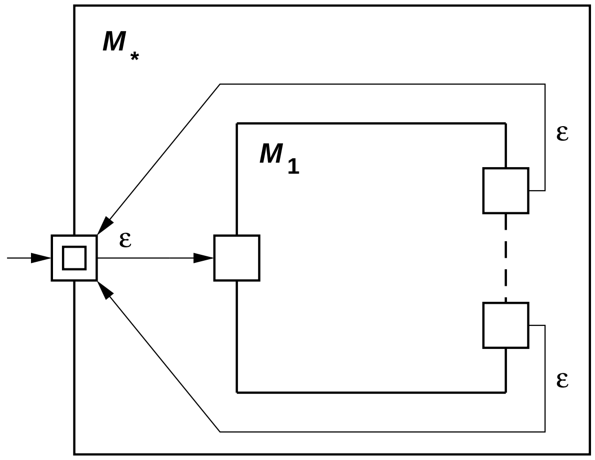

Figure 3.8: A schematic depiction of the "star-closure" NFA M_*.

The preceding material is the intuition underlying a formal proof using induction on the length of the input string. $\square$

Lemma 3.4. *If the language $L \subseteq \Sigma^\star$ is regular, then L is denoted by a Regular Expression R_L.*

Proof. We prove the lemma via a famous dynamic programming algorithm for constructing R_L, that is attributed to Floyd and Warshall and that appears in many Algorithms texts, such as [8] (in the section on transitive closure of graphs).

Let $L = L(M)$ for the DFA $M = (Q, \Sigma, \delta, q_0, F)$. For the purposes of the indexing required by the dynamic programming algorithm we are about to present, let us rename the states of M as $Q = \{s_1, s_2, \ldots, s_n\}$, with $s_1 = q_0$.

For all triples of integers $1 \leq i, j \leq n$ and $0 \leq k \leq n$, define $L_{ij}^{(k)}$ to be the set of all strings $x \in \Sigma^\star$ such that

1. $\delta(s_i, x) \ = \ s_j$
2. Every *intermediate* state encountered as x leads state s_i to state s_j has the form s_ℓ for some $\ell \leq k$. (Note that we are constraining only the *intermediate* states, not s_i or s_j.)

When $k = 0$, there is no intermediate state, so that

$$L_{ij}^{(0)} \;=\; \begin{cases} \{\sigma \mid \delta(s_i, \sigma) \;=\; s_j\} & \text{if } i \neq j \\ \{\varepsilon\} \cup \{\sigma \mid \delta(s_i, \sigma) \;=\; s_j\} & \text{if } i = j \end{cases}$$

Note that the first line of this definition implicitly specifies $L_{ij}^{(0)}$ to be the empty set $\emptyset$ when $i \neq j$ and there is no σ such that $\delta(s_i, \sigma) \;=\; s_j$.

When $k > 0$, we can derive an exact expression for $L_{ij}^{(k)}$ in terms of L's with a smaller upper index, via the following intuition. The set $L_{ij}^{(k)}$ of all strings that lead state s_i to state s_j with all intermediate states of index $\leq k$ consists of

- The set of all strings that lead state s_i to state s_j with all intermediate states of index $< k$—which is the set $L_{ij}^{(k-1)}$—*unioned with* ...
  - the set of all strings that lead state s_i to state s_k with all intermediate states of index $< k$—which is the set $L_{ik}^{(k-1)}$—*concatenated with* ...
  - the set of all strings that lead state s_k back to itself with all intermediate states of index $< k$, *repeated as many times as you want*—which is the set $\left(L_{kk}^{(k-1)}\right)^{\star}$—*concatenated with* ...
  - the set of all strings that lead state s_k to state s_j with all intermediate states of index $< k$—which is the set $L_{kj}^{(k-1)}$.

In other words:

$$L_{ij}^{(k)} \;=\; L_{ij}^{(k-1)} \;\cup\; L_{ik}^{(k-1)} \cdot \left(L_{kk}^{(k-1)}\right)^{\star} \cdot L_{kj}^{(k-1)}.$$

Since L is the (perforce, finite) union of the sublanguages that lead M from its initial state s_1 to some accepting state, we have $L \;=\; \bigcup_{s_\ell \in F} L_{1\ell}^{(n)}$.

We have thus (implicitly) derived a Regular Expression that denotes L, hence have proved the lemma. □

3.5 "Pumping" in Regular Languages

The so-called Pumping Lemma for Regular Languages is the primary tool used in most textbooks for exposing the limitations of FA. As suggested earlier, we disagree with this emphasis. This section is devoted to explaining the Pumping Lemma, its origins, its strengths, and its weaknesses.

The phenomenon of "pumping" that underlies the Lemma is a characteristic of any finite closed system.

Example 1. Consider, for instance, any finite semigroup[8], $S = \{\alpha_1, \alpha_2, \ldots, \alpha_n\}$. Since there are only finitely many *distinct* products in any sequence of the form α_{i_1}, $\alpha_{i_1}\alpha_{i_2}$, $\alpha_{i_1}\alpha_{i_2}\alpha_{i_3}$, ..., where each $\alpha_{i_j} \in S$, there must exist two products in the sequence, say $\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}$ and $\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}\alpha_{i_{k+1}}\cdots\alpha_{i_{k+\ell}}$ such that

$$\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k} \ = \ \alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}\alpha_{i_{k+1}}\cdots\alpha_{i_{k+\ell}}$$

within the semigroup. By associativity, then, for all $h \in \mathbb{N}$,

$$\alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k} \ = \ \alpha_{i_1}\alpha_{i_2}\cdots\alpha_{i_k}(\alpha_{i_{k+1}}\cdots\alpha_{i_{k+\ell}})^h,$$

where the power notation implies iterated multiplication within the semigroup.

Example 2. Consider next, via the following little tale, how the phenomenon of pumping manifests itself in finite graphs. Say that you are in a park in Paris (lucky you!) which is organized as a set of n statues interconnected by paths. (Think of the statues as the nodes/states of an FA and of the paths as its edges.) Say that you take a *long* walk in the park. All we really need is that you walk long enough to traverse n inter-statue paths. By the Pigeonhole Principle—do you see how it applies here?—you must encounter some specific statue at least twice in your walk. Moreover, you can keep repeating the n-path traversal as many times as you want, and it will always return you to that statue.

Now let's talk like automata theorists. In the semigroup example, semigroup elements become states, and sequences of such elements become input strings. In the graph example, statues become states, and paths become input strings. In either case, the phenomenon of pumping ensures the following. Say that we are given an FA M whose language, $L(M)$ contains infinitely many strings. Then, no matter how many states M has, there is a string w in $L(M)$ whose length is at least as large as M's set of states. When we feed this string to M (starting from the initial state q_0, of course), we must pass through some state of M—call it q—at least twice. Let's "parse" w into the form $w = xyz$, where x is the prefix of w that leads us to state q for the *first* time, y takes us back to q for the *last* time (when reading w), and z is the suffix of w that leads us from state q to an accepting state $q^\star$ of M. Clearly, for all integers $k = 0, 1, \ldots$, the string xy^kz acts essentially like w, in the sense that it takes us from q_0 to q, loops around to q k times, and then leads from q to $q^\star$. If we recast this description of "pumping" within the formalism of FA's, we find that any word $w \in \Sigma^\star$ of length[9] $\ell(w) \geq |Q|$ can be parsed into the form $w = xy$, where $y \neq \varepsilon$,[10] in such a way that

[8]A *semigroup* is a set of elements that are closed under an associative binary multiplication (denoted here by juxtaposition).

[9]$\ell(w)$ denotes the *length* of the string w.

[10]Of course, we could have $x = \varepsilon$.

$\delta(q_0, x) = \delta(q_0, xy)$. Since M is deterministic—i.e., since δ is a function—for all $h \in \mathbb{N}$,

$$\delta(q_0, x) = \delta(q_0, xy^h), \tag{3.5.4}$$

where, as earlier, the power notation implies iterated concatenation. Since the "pumping" depicted in Eq. 3.5.4 occurs also with words $w \in \Sigma^\star$ that admit a continuation $z \in \Sigma^\star$ that places them in $L(M)$—i.e., $wz \in L(M)$—we arrive finally at the Pumping Lemma. Acknowledging the importance of the "Lemma," we henceforth call it "the Pumping Theorem for Regular Languages." (As you read along, note the implicit invocation of Lemma 3.2 in our argument.)

Theorem 3.5. (The Pumping Lemma for Regular Languages)
For every infinite regular language L, there exists an integer $n \in \mathbb{N}$ such that: Every word $w \in L$ of length $\ell(w) \geq n$ can be parsed into the form $w = xyz$, where $\ell(xy) \leq n$ and $\ell(y) > 0$, in such a way that, for all $h \in \mathbb{N}$, $xy^hz \in L$.

The reader should easily see how to use Theorem 3.5 to prove that sets are not regular. The technique differs from the fooling set/Continuation Lemma technique of Section 3.3.2.A mainly in the new (and nonintrinsic!) requirement that one of the "fooling" words must be a prefix of the other. I view this extraneous restriction as a sufficient argument *not* to use Theorem 3.5 for proofs of nonregularity. However (inexplicably to me), most standard texts actually mandate looking for undesired "pumping" activity, rather than just for a pair of "fooling" words. For instance, a common pumping-based proof of the nonregularity of the language L_1 of Application 1 (Section 3.3.2.A) notes that the "pumped" word y of Theorem 3.5:

1. cannot consist solely of a's, or else the block of a's becomes longer than the block of b's;

2. cannot consist solely of b's, or else the block of b's becomes longer than the block of a's;

3. cannot contain both an a and a b, or else the pumped word no longer has the form "a block of a's followed by a block of b's.

Even when one judiciously avoids this three-case argument by invoking the Lemma's length limit on the prefix xy, one is inviting/risking excessive complication by seeking a string that pumps. For instance, when proving the nonregularity of the language L_3 of palindromes, one must cope with the fact that any palindrome *does* pump about its center. (That is, for any palindrome w and any integer ℓ, if one parses w into $w = xyz$, where x and z both have length ℓ, then, indeed, for all $h \in \mathbb{N}$, the word xy^hz is a palindrome.) Note that we are not suggesting that any of the problems we raise is insuperable, only that they unnecessarily complicate the proof process, hence violate Occam's Razor. The danger inherent in using Theorem 3.5 to prove that a language is not regular is mentioned explicitly in [23]:

The pumping lemma is difficult for several reasons. Its statement is complicated, and it is easy to go astray in applying it.

We show now that the condition for a language to be regular that is provided in Theorem 3.5 is *necessary* but not sufficient. This contrasts with the *necessary and sufficient* condition provided by Theorem 3.1.

Lemma 3.5 ([40]). *Every string of length > 4 in the nonregular language*

$$L_5 = \{uu^Rv \mid u, v \in \{0,1\}^\star;\ \ell(u), \ell(v) \geq 1\}$$

pumps in the sense of Theorem 3.5.

Proof. We paraphrase from [40]. Each string in L_5 consists of a nonempty even palindrome followed by another nonempty string. Say first that $w = uu^Rv$ and that $\ell(w) \geq 4$. If $\ell(u) = 1$, then we can choose the first character of v as the nonnull "pumping" substring of Theorem 3.5. (Of course, the "pumped" strings are uninteresting in this case.) Alternatively, if $\ell(u) > 1$, then, since a^k is a palindrome for every $k > 1$, where a is the first character of u, we can let this first letter be the nonnull "pumping" substring of Theorem 3.5. In either case, the lemma holds. □

Notably, the discussion in [40] ends with the following comment.

> For a practical test that exactly characterizes regular languages, see the Myhill-Nerode theorem.

For the record, Theorem 3.1 provides a simple proof that L_5 is not regular. Let x and y be distinct strings from the infinite language $L = (01)(01)^\star$, with $\ell(y) > \ell(x)$. (Strings in L consist of a sequence of one or more instances of 01.) Easily, xx^R is an even-length palindrome, hence belongs to L_5 (with $v = \varepsilon$). However, one verifies easily that yx^R does not begin with an even-length palindrome, so that $yx^R \notin L_5$. To wit, if one could write yx^R in the form uu^Rv, then:

- u could not end with a 0, since the "center" substring 00 does not occur in yx^R;
- u could not end with a 1, since the unique occurrence of 11 in yx^R occurs to the right of the center of the string.

By Lemma 3.2, L_5 is not regular. □

For completeness, we end this section with a version of Theorem 3.5 that supplies a condition that is both necessary and sufficient for a language to be regular. This version is rather nonperspicuous and a bit cumbersome, hence, is infrequently taught.

Theorem 3.6 ([16]). **(The Necessary-and-Sufficient Pumping Theorem for Regular Languages)**
A language $L \subseteq \Sigma^\star$ is regular if, and only if, there exists an integer $n \in \mathbb{N}$ such that: Every word $w \in \Sigma^\star$ of length $\ell(w) \geq n$ can be parsed into the form $w = xyz$, where $\ell(y) > 0$, in such a way that, for all $z \in \Sigma^\star$:

- *if $wz \in L$, then for all $h \in \mathbb{N}$, $xy^h z \in L$;*
- *if $wz \notin L$, then for all $h \in \mathbb{N}$, $xy^h z \notin L$;*

3.6 Regular Languages and the Boolean Operations

While developing the machinery for Kleene-Myhill Theorem (Theorem 3.4), we had to prove (in Lemma 3.3) that the regular languages are closed under the "Kleene operations" of union, concatenation, and star-closure. It turns out that the regular languages are also closed under the Boolean operations. This fact and its proof are important in many applications of FA theory, both to other components of Computation Theory and to application areas such as logic design. This section is devoted to developing two proofs of the following theorem.

Theorem 3.7. *The regular languages are closed under the Boolean operations. That is, if L_1 and L_2 are regular languages, then so also are $\overline{L}_1$, $L_1 \cup L_2$, and $L_1 \cap L_2$.*

Proof. **Complementation**. To verify that a language L is regular iff its complement $\overline{L}$ is, we need only have the FA M that accepts L "complement" (or, "flip") its answer regarding the (non)acceptance of all input words. This is achieved formally by switching the roles of the accepting and nonaccepting states. Thus, if $L = L(M)$ for the DFA

$$M \ = \ (Q,\ \Sigma,\ \delta,\ q_0,\ F),$$

then $\overline{L} = L(\overline{M})$ for the DFA

$$\overline{M} \ = \ (Q,\ \Sigma,\ \delta,\ q_0,\ Q \setminus F).$$

Union and Intersection. In Lemma 3.3, we used a construction based on NFA's to verify that the regular languages are closed under union. By De Morgan's Laws (2.1.1), closure under both union and complementation guarantees closure under intersection.

While we have, thus, actually proved the theorem, it turns out that there is an alternative proof of closure under union and intersection, that is a useful element in our kitbag of tools for analyzing DFA's. It is the *direct-product* construction for DFA's.

The direct-product construction. We illustrate both the construction and the use of direct products of DFA's by focusing on an arbitrary "generalized Boolean operation" $\otimes$, i.e., a binary operation that can be formed as a composition of one or more of the three basic set-theoretic Boolean operations, union, intersection, and complementation. (Some of the more important such operations are mentioned in Section 2.1.) We prove, via the direct-product construction that the regular languages are closed under the operation $\otimes$.

The strategy underlying the direct-product construction of DFA's is to run two DFA's "in parallel," state-transition by state-transition, and then combine their answers to each input string via the logical Boolean operation that corresponds to the desired set-theoretic Boolean operation $\otimes$. One achieves this effect as follows. Say that, for $i = 1, 2$, $L_i = L(M_i)$ for the DFA

$$M_i \ = \ (Q_i, \ \Sigma, \ \delta_i, \ q_{i0}, \ F_i).$$

We construct the following direct-product DFA.

$$M_{1,2}^{\otimes} \ = \ (Q_1 \times Q_2, \ \Sigma, \ \delta_{1,2}, \ \langle q_{10}, q_{20} \rangle, \ F_{1,2}^{\otimes}),$$

where

- For all $q_1 \in Q_1$, $q_2 \in Q_2$, and $\sigma \in \Sigma$, $\delta_{1,2}(\langle q_1, q_2 \rangle, \sigma) \ = \ \langle \delta_1(q_1, \sigma), \delta_2(q_2, \sigma) \rangle$.
- $F_{1,2}^{\otimes} \ = \ \{ \langle q_1, q_2 \rangle \in Q_1 \times Q_2 \mid ([q_1 \in F_1] \otimes [q_2 \in F_2] = 1) \}$.

To explain the definition of $F_{1,2}^{\otimes}$: Note that the predicate "$q_i \in F_i$" can be viewed as evaluating to 0 (if the predicate is false) or to 1 (if the predicate is true). Thus intepreted, it is meaningful to combine predicates using the logical versions of the set-theoretic Boolean operations.

We claim that $L(M_{1,2}^{\otimes}) = L_1 \otimes L_2$, whence the latter language is regular. To see this, note that, when we extend the state-transition function $\delta_{1,2}$ of the product DFA $M_{1,2}^{\otimes}$ to act on strings, rather than letters, in the standard way, we find that, for all strings $x \in \Sigma^\star$,

$$\delta_{1,2}(\langle q_{10}, q_{20} \rangle, x) \ = \ \langle \delta_1(q_{10}, x), \delta_2(q_{20}, x) \rangle.$$

By definition, then, the state $\langle \delta_1(q_{10}, x), \delta_2(q_{20}, x) \rangle$ belongs to $F_{1,2}^{\otimes}$—or, equivalently, $x \in L(M_{1,2}^{\otimes})$—precisely when

$$[\delta_1(q_{10}, x) \in F_1] \otimes [\delta_2(q_{20}, x) \in F_2] \ = \ 1,$$

using the logical version of operation $\otimes$. But this is equivalent to saying that

$$L(M_{1,2}^{\otimes}) \ = \ L(M_1) \otimes L(M_2) \ = \ L_1 \otimes L_2,$$

as was claimed. □

3.7 Enrichment Topics

3.7.1 Memory bounds for Online Automata

Our first enrichment topic requires us to generalize the FA model by removing the restriction that its set Q of states be finite. The model is otherwise unchanged. We call the generalized model *Online Automata* (*OA*, for short).

By Theorem 3.1, any OA M that accepts a nonregular language must have infinitely many states. We now present a result from [17] that sharpens this statement via an "infinitely-often" lower bound on the number of states an FA $M^{(n)}$ must have in order to correctly mimic M's (word-acceptance) behavior on all words of length $\leq n$ (thereby providing an "order-n approximation" of M). This bound assumes nothing about M other than its accepting a nonregular language. (Indeed, M's state-transition function δ need not even be computable.) In the context of this survey, this result removes Theorem 3.1 from the confines of the theory of FA's, by adapting it to a broader class of state-transition systems. This adaptation is achieved by converting the *word-relating* equivalence relation $\equiv_M$ to an *automaton-relating* relation that asserts the equivalence of two OA's on all words that are no longer than a chosen parameter.

Let L be a nonregular language, and let M be an OA that accepts L: $L = L(M)$. For any $n \in \mathbb{N}$, an FA $M^{(n)}$ is an *order-n approximate acceptor* of L or, equivalently, an *order-n approximation* of M if

$$\{x \in L(M^{(n)}) \mid \ell(x) \leq n\} \ = \ \{x \in L \mid \ell(x) \leq n\} \ = \ \{x \in L(M) \mid \ell(x) \leq n\}.$$

We denote by $\mathcal{A}_L(n)$ the (obviously monotonic nondecreasing) number of states in the smallest order-n approximate acceptor of L, as a function of n. This quantity can be viewed as a measure of L's "space complexity," in the sense that one needs $\lceil \log_2 \mathcal{A}_L(n) \rceil$ bi-stable devices (say, transistors) in order to implement an order-n approximate acceptor of L in circuitry.

The conceptual framework of Theorem 3.1 affords one easy access to a nontrivial lower bound on the "infinitely-often" behavior of $\mathcal{A}_L(n)$, *for any* nonregular language L.

Theorem 3.8 ([17])**.** *If the language L is nonregular, then, for infinitely many n,*

$$\mathcal{A}_L(n) > \frac{1}{2}n + 1. \tag{3.7.5}$$

Proof. Let M_1 and M_2 be OA's. For any $n \in \mathbb{N}$, we say that M_1 and M_2 are *n-equivalent*, denoted $M_1 \equiv_n M_2$, just when

$$\{x \in L(M_1) \mid \ell(x) \leq n\} \ = \ \{x \in L(M_2) \mid \ell(x) \leq n\}.$$

This relation is, thus, a parameterized extension of the relation $\equiv_M$ that is central to Theorem 3.1.

Our analysis of approximate acceptors of L builds on the following bound on the "degree" of equivalence of pairs of FA's.

Lemma 3.6 ([25]). *Let M_1 and M_2 be FA's with s_1 and s_2 states, respectively, such that $L(M_1) \neq L(M_2)$. Then $M_1 \not\equiv_{s_1+s_2-2} M_2$.*

Proof of Lemma 3.6. We bound from above the number of partition-refinements that suffice for the state-minimization algorithm of Section 3.3.3 to distinguish the initial states of M_1 and M_2 (which, by hypothesis, are distinguishable).

Since the algorithm is actually a "state-equivalence tester," we can apply it to state-transition systems that are not legal FA's, as long as we are careful to keep final and nonfinal state segregated from one another. We therefore apply the algorithm to the following "disconnected" FA M. Say that, for $i = 1, 2$, $M_i = (Q_i, \Sigma, \delta_i, q_{i,0}, F_i)$, where $Q_1 \cap Q_2 = \emptyset$. Then $M = (Q, \Sigma, \delta, \{q_{1,0}, q_{2,0}\}, F)$, where

- $Q = Q_1 \cup Q_2$
- for $q \in Q$ and $\sigma \in \Sigma$: $\delta(q, \sigma) = \begin{cases} \delta_1(q,\sigma) & \text{if } q \in Q_1 \\ \delta_2(q,\sigma) & \text{if } q \in Q_2 \end{cases}$
- $F = F_1 \cup F_2$.

Now, the fact that $L(M_1) \neq L(M_2)$ implies: (a) that $q_{1,0} \not\equiv_M q_{2,0}$; ($b$) that neither $Q - F$ nor F is empty. How many stages of the algorithm would be required, in the worst case, to distinguish states $q_{1,0}$ and $q_{2,0}$ within M, when the algorithm starts with the initial partition $\{Q - F, F\}$? Well, each stage of the algorithm, save the last, must "split" some block of the partition into two nonempty subblocks. Since one "split," namely, the separation of $Q - F$ from F, occurs before the algorithm starts applying input symbols, and since $|Q| = s_1 + s_2$, the algorithm can proceed for no more than $s_1 + s_2 - 2$ stages; after that many stages, all blocks would be singletons! In other words, if $p \not\equiv_M q$, for states $p, q \in Q$, then there is a string of length $\leq s_1 + s_2 - 2$ that witnesses the nonequivalence. Since we know that $q_{1,0} \not\equiv_M q_{2,0}$, this completes the proof. $\square$

Back to the theorem. For each $k \in \mathbb{N}$, Theorem 3.1 guarantees that there is a smallest integer $n > k$ such that $\mathcal{A}_L(k) = \mathcal{A}_L(n-1) < \mathcal{A}_L(n)$. The preceding inequality implies the existence of FA's M_1 and M_2 such that:

1. M_1 has $\mathcal{A}_L(n-1)$ states and is an $(n-1)$-approximate acceptor of L;
2. M_2 has $\mathcal{A}_L(n)$ and is an n-approximate acceptor of L.

By statement 1, $M_1 \equiv_{n-1} M_2$; by statements 1 and 2, $M_1 \not\equiv_n M_2$. By Lemma 3.6, then, $M_1 \not\equiv_{\mathcal{A}_L(n-1)+\mathcal{A}_L(n)-2} M_2$. Since $M_1 \equiv_{n-1} M_2$, we therefore have $\mathcal{A}_L(n-1)+\mathcal{A}_L(n) > n+1$, which yields Ineq. 3.7.5, since $\mathcal{A}_L(n-1) \le \mathcal{A}_L(n) - 1$. □

It is shown in [17] that Theorem 3.8 is as strong as possible, in that: the constants $\frac{1}{2}$ and 1 in Ineq. 3.7.5 cannot be improved; the phrase "infinitely many" cannot be strengthened to "all but finitely many."

3.7.2 Finite Automata with probabilistic transitions

We now consider a rather different genre of FA's, namely, ones whose state-transitions are *probabilistic*, with acceptance decisions depending on the probability of ending up in an final state. This is a very timely model to consider since probabilistic state-transition systems are currently quite in vogue in several areas of artificial intelligence, notably the growing area of machine learning. The main result that we present comes from [29]; it exhibits a nontrivial, somewhat surprising situation in which probabilistic state-transitions add no power to the model: The restricted automata accept only regular sets.

PFA's and their languages. We start with an FA, $M = (Q, \Sigma, \delta, q_0, F)$, and make its state-transitions and acceptance criterion *probabilistic*. We call the resulting model a *Probabilistic Finite Automaton* (*PFA*, for short).

States. We simplify the formal development by positing that the state-set of the PFA M is $Q = \{1, 2, \ldots, n\}$, with $q_0 = 1$, and $F = \{m, m+1, \ldots, n\}$ for some $m \in Q$.

State-transitions. We replace M's state-transition function δ with a set of tables, one for each symbol of Σ. The table associated with $\sigma \in \Sigma$ indicates, for each pair of states $q, q' \in Q$, the probability—call it $\rho_{q,q'}$—that M ends up in state q' when started in state q and "fed" input symbol σ. It is convenient to present the state-transition tables as matrices. The table associated with $\sigma \in \Sigma$ is:

$$\Delta_\sigma = \begin{pmatrix} \rho_{1,1} & \rho_{1,2} & \cdots & \rho_{1,n} \\ \rho_{2,1} & \rho_{2,2} & \cdots & \rho_{2,n} \\ \vdots & \vdots & \ddots & \vdots \\ \rho_{n,1} & \rho_{n,2} & \cdots & \rho_{n,n} \end{pmatrix}$$

where each[11] $\rho_{i,j} \in [0,1]$, and, for each i, $\sum_j \rho_{i,j} = 1$.

States, revisited. The probabilistic nature of M's state-transitions forces us to distinguish between M's set of states—the set Q—and the "state" that reflects M situation at

[11]As usual, $[0,1]$ denotes the closed *real* interval $\{x \mid 0 \le x \le 1\}$.

any point of a computation, which is a probability distribution over Q. We therefore define the *state-distribution* of M to be a vector of probabilities $\vec{q} = \langle \pi_1, \pi_2, \ldots, \pi_n \rangle$, where each π_i is the probability that M is in state i. The *initial state-distribution* is $\vec{q_0} = \langle 1, 0, \ldots, 0 \rangle$.

State-transitions, revisited. Under the preceding formalism, the PFA analogue of the FA single-symbol state-transition $\delta(q, \sigma)$ is the vector-matrix product: $\widehat{\Delta}(\vec{q}, \sigma) \ = \ \vec{q} \times \Delta_\sigma$. By extension, the PFA analogue of the FA string state-transition $\delta(q, \sigma_1\sigma_2 \cdots \sigma_k)$, where each $\sigma_i \in \Sigma$, is

$$\widehat{\Delta}(\vec{q}, \sigma_1\sigma_2 \cdots \sigma_k) \ \stackrel{\text{def}}{=} \ \vec{q} \times \Delta_{\sigma_1} \times \Delta_{\sigma_2} \times \cdots \times \Delta_{\sigma_n}. \qquad (3.7.6)$$

The language accepted by a PFA. The probabilistic analogue of acceptance by final state builds on the notion of an *(acceptance) threshold* $\theta \in [0, 1]$. The string $x \in \Sigma^\star$ is *accepted* by M iff

$$p_M(x) \ \stackrel{\text{def}}{=} \ \sum_{i=m}^{n} \widehat{\Delta}(\vec{q_0}, x)_i \ > \ \theta,$$

where $\widehat{\Delta}(\vec{q}, x)_i$ denotes the ith coordinate of the tuple $\widehat{\Delta}(\vec{q}, x)$. (Recall that M's final states are those whose integer-names are $\geq m$.) Thus, x is accepted iff it leads M from its initial state to its set of final states with probability $> \theta$. As with all FA's, the *language accepted by* M is the set of all strings that M accepts. Acknowledging the crucial role of the acceptance threshold θ, we denote this language by

$$L(M, \theta) \ \stackrel{\text{def}}{=} \ \{x \in \Sigma^\star \mid p_M(x) > \theta\}.$$

$L(M, \theta)$ is regular when θ is "isolated." It is noted in [29] that even simple—e.g., two-state—PFA's can accept nonregular languages, when accompanied by an "unfavorable" acceptance threshold. When thresholds are "favorable," though, all PFA's accept regular languages.

The threshold $\theta \in [0, 1]$ is *isolated* for the PFA M iff there exist a real *constant of isolation* $\varepsilon > 0$ such that, for all $x \in \Sigma^\star$, $|p_M(x) - \theta| \ \geq \ \varepsilon$.

Theorem 3.9 ([29]). *For any PFA M and associated* isolated *acceptance threshold θ, the language $L(M, \theta)$ is regular.*

Proof. We sketch the proof from [29], which is a direct application of Theorem 3.1. Say that M has n states, a of which are final, and let $\varepsilon > 0$ be the constant of isolation. We claim that the relation $\equiv_{L(M,\theta)}$ cannot have more than $\kappa \stackrel{\text{def}}{=} [1 + (a/\varepsilon)]^{n-1}$ classes.

This bound is established by considering a set of k words that are mutually inequivalent under $\equiv_{L(M,\theta)}$, with the aim of showing that k cannot exceed κ. This is accomplished by converting M's language-related problem to a geometric setting, by considering, for each $x \in \Sigma^\star$, the point in n-dimensional space given by $\widehat{\Delta}(\vec{q_0}, w)$ (cf. Eq. 3.7.6).

In the language-related setting, we consider an arbitrary pair of inequivalent words, $x_i, x_j \in \Sigma^\star$, and note that there must exist $y \in \Sigma^\star$ such that (with no loss of generality) $x_i y \in L(M, \theta)$ while $x_j y \notin L(M, \theta)$. In the geometric setting, this translates into the existence of three points:

$$\begin{array}{cl} \langle \xi_1^{(i)}, \xi_2^{(i)}, \ldots, \xi_n^{(i)} \rangle & \text{corresponding to } x_i \\ \langle \xi_1^{(j)}, \xi_2^{(j)}, \ldots, \xi_n^{(j)} \rangle & \text{corresponding to } x_j \\ \langle \eta_1, \eta_2, \ldots, \eta_n \rangle & \text{corresponding to } y \end{array}$$

such that (invoking the conditions for a word to be accepted):

$$\begin{array}{rcl} \theta + \varepsilon & < & \xi_1^{(i)}\eta_1 + \xi_2^{(i)}\eta_2 + \cdots + \xi_n^{(i)}\eta_n; \\ \theta - \varepsilon & \geq & \xi_1^{(j)}\eta_1 + \xi_2^{(j)}\eta_2 + \cdots + \xi_n^{(j)}\eta_n. \end{array}$$

Elementary reasoning then allows us to infer that

$$2(\varepsilon/a) \ \leq \ |\xi_1^{(i)} - \xi_1^{(j)}| + |\xi_2^{(i)} - \xi_2^{(j)}| + \cdots + |\xi_n^{(i)} - \xi_n^{(j)}|.$$

We next consider, for each $i \in \{1, 2, \ldots, k\}$, the set Λ_i comprising all points $\langle \xi_1, \xi_2, \ldots \xi_n \rangle$ such that

- $\xi_l \ \geq \ \xi_l^{(i)}$ for all $l \in \{1, 2, \ldots, n\}$
- $\displaystyle\sum_{l=1}^{n} (\xi_l - \xi_l^{(i)}) \ = \ (\varepsilon/a)$.

By bounding the volumes of the sets Λ_i, and arguing that no two share an internal point, one arrives at the following bounds on the cumulative volumes of the sets.

$$kc(\varepsilon/a)^{n-1} \ = \ \sum_{l=1}^{n} \text{Vol}(\Lambda_l) \ = \ c(1 + (\varepsilon/a))^{n-1}.$$

We infer directly that $k \leq [1 + (a/\varepsilon)]^{n-1}$, as was claimed. $\square$

3.7.3 Pumping in nonregular languages

Our final enrichment topic illustrates a language-theoretic setting—the theory of so-called *context-free* languages—where pumping plays even a more fundamental role than with regular languages, because it is the primary tool for negative proofs in this context. The material in this section has its roots in the "Type-2" grammars and languages of [5, 6].

A *context-free grammar* (*CFG*, for short) is specified as follows.

$$G \ = \ (V, \Sigma, S, \mathcal{P}),$$

where

- V is a finite *vocabulary* of *nonterminal symbols* that play the roles of "syntactic categories;"
- Σ is a finite alphabet of *terminal symbols*;
- $S \in V$ is the *sentence symbol*, the most general "syntactic category;"
- $\mathcal{P} \subseteq V \times (V \cup \Sigma)^\star$ is a relation whose elements are called *productions*.

Conventionally, a production (A, w), where $A \in V$ and $w \in (V \cup \Sigma)^\star$, is written as "$A \rightarrow w$." (The arrow notation suggests, accurately, that productions will be used to rewrite letters as strings.)

Informally, one starts with the sentence symbol S and begins generating strings by rewriting nonterminal symbols in manners allowed by the productions. One continues until one has a string of terminal symbols. Here is one simple example to provide intuition.

- $V = \{$Mathematical-Expression, Sum, Product, Var$\}$
- $\Sigma = \{(,), +, -, \times, \div, X, Y, Z\}$
- $S =$ Mathematical-Expression
- $\mathcal{P}$ consists of the following ten productions

  Mathematical-Expression $\rightarrow$ Sum
  Mathematical-Expression $\rightarrow$ Product
  Mathematical-Expression $\rightarrow$ Var
  Sum $\rightarrow$ (Mathematical-Expression $+$ Mathematical-Expression)
  Sum $\rightarrow$ (Mathematical-Expression $-$ Mathematical-Expression)
  Product $\rightarrow$ (Mathematical-Expression $*$ Mathematical-Expression)
  Product $\rightarrow$ (Mathematical-Expression / Mathematical-Expression)
  Var $\rightarrow X$
  Var $\rightarrow Y$
  Var $\rightarrow Z$

Hopefully, you can intuit how this CFG specifies a "Mathematical-Expression" as either a variable or a fully parenthesized sum/difference or product/ratio of "Mathematical-Expressions." We now turn to formalizing this intuition, by supplying the semantics of CFG's.

Let us be given a CFG $G = (V, \Sigma, S, \mathcal{P})$. Consider any string uAv, where $A \in V$ and $u, v \in (V \cup \Sigma)^\star$. If there is a production $(A, w) \in \mathcal{P}$, then we write

$$uAv \;\Rightarrow_G\; uwv$$

meaning that string uAv can be rewritten as string uwv under G. This defines $\Rightarrow_G$ as a new "rewriting" relation on strings:

$$\Rightarrow_G \ \subseteq \ (V \cup \Sigma)^\star V (V \cup \Sigma)^\star \times (V \cup \Sigma)^\star$$

We are really interested in the *reflexive, transitive closure* of relation $\Rightarrow_G$, which we denote by $\Rightarrow_G^\star$ and define as follows. For strings $u, v \in (V \cup \Sigma)^\star$, we write $u \Rightarrow_G^\star v$, articulated as "v is derivable from u under G," just when the following holds.

$$u \Rightarrow_G^\star v \ \text{ means: } \begin{cases} \text{either } \ u = v \\ \text{or there exist strings } \ w_1, w_2, \ldots, w_n \ \text{ such that} \\ \quad u \ \Rightarrow_G \ w_1 \ \Rightarrow_G \ w_2 \ \Rightarrow_G \ \cdots \ \Rightarrow_G \ w_n \ \Rightarrow_G \ v \end{cases}$$

We can now, finally, define the *context-free language* (*CFL*, for short) $L(G)$ that is *generated* by the CFG G:

$$L(G) \ = \ \{x \in \Sigma^\star \mid S \Rightarrow_G^\star x\}$$

Each derivation

$$S \ \Rightarrow_G \ y_1 \ \Rightarrow_G \ y_2 \ \Rightarrow_G \ \cdots \ \Rightarrow_G \ y_n \ \Rightarrow_G \ x \tag{3.7.7}$$

of a string $x \in L(G)$ can be depicted in a natural way as a rooted, oriented tree where:

- S is the root of the tree;
- for each rewriting $uAv \Rightarrow_G uwv$ in the sequence of rewritings that constitute derivation (3.7.7), all of the letters of strings w are, from left to right, the children of node A;
- the left-to-right sequence of leaves of the tree constitute string x.

Here is an example using our CFG for Mathematical-Expressions. Consider the *derivation tree* of the expression $(X + ((X * Y) - (Y/Z))$ in Fig. 3.9.

Note that every root-to-leaf path in a derivation tree is a string $\beta_1\beta_2 \cdots \beta_m\sigma$, where each $\beta_k \in V$ and $\sigma \in \Sigma$. If we prune this path to eliminate the impact of nonproductive productions of the form $\beta_i \rightarrow \beta_j$, then we are left with a substring $\beta_1'\beta_2' \cdots \beta_\ell'\sigma$, where each β_k' either produces a leaf (i.e., a terminal symbol), or it has more than one child in the tree. We make two simple, yet important observations.

- If the path is long enough, then some nonterminal along the path must repeat. This follows from the pigeonhole principle, because the set V is finite.
- If $L(G)$ is infinite, then the root-to-leaf paths in derivation trees get arbitrarily long. This is verified using the same reasoning as in the famous "Infinity Lemma" of D. König [19], since there is an upper bound on the number of children a node can have in a derivation tree for G—namely, the length of the longest righthand string in a production—and since there is no upper bound on the number of leaves that a derivation tree for G can have (because $L(G)$ is infinite).

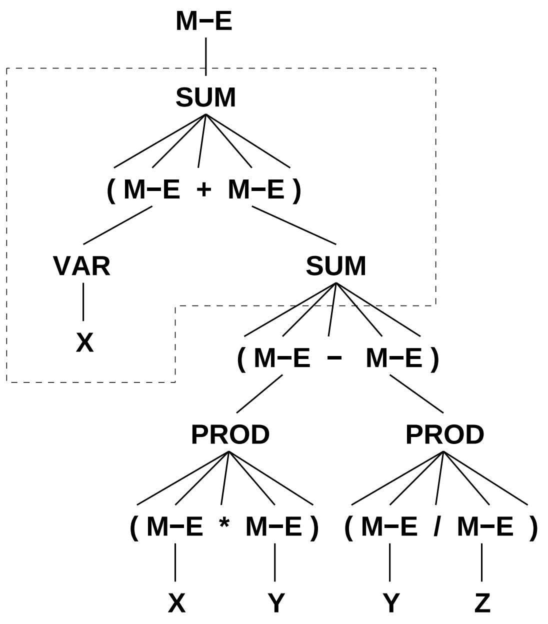

Figure 3.9: A derivation of the expression $(X + ((X * Y) - (Y/Z)))$ under the CFG G.

Consider now the effect of taking the portion of a derivation tree consisting of two occurrences of the same β_i on a path, plus the subtree subtended by these occurrences, and replicating it—which one can do because the grammar G is *context-free*, so that nonterminal β_i can be rewritten via any valid production wherever it occurs. If one repeats this replication, one discerns a pattern of *pumping* in the terminal string derived via the tree! For instance, if one replicates the portion of the derivation tree of Fig. 3.9 that is delineated by the dashed box, then one obtains, via repeated such replications, the "pumped" sequence of expressions

Number of replications	Resulting expression
0	$((X * Y) - (Y/Z))$
1	$(X + ((X * Y) - (Y/Z)))$
2	$(X + (X + ((X * Y) - (Y/Z))))$
3	$(X + (X + (X + ((X * Y) - (Y/Z)))))$
$\vdots$	$\vdots$

Symbolically, after k iterations, one has generated the expression $\xi^k \eta \zeta^k$, where ξ is the string "$(X+$", η is the string "$((X * Y) - (Y/Z))$", and ζ is the string "$)$". The situation we have described illustrates a special case of the general phenomenon of pumping in CFL's, as described in the following, whose detailed proof is left as an exercise.

Theorem 3.10. (The Pumping Lemma for Context-Free Languages)
For every infinite context-free language L, there exists an integer $m \in \mathbb{N}$ such that: Every

word $z \in L$ of length $\ell(z) \geq m$ can be parsed into the form $z = uvwxy$, where $\ell(uv) \leq m$ and $\ell(vx) > 0$, in such a way that, for all $h \in \mathbb{N}$, $uv^h wx^h y \in L$.

As suggested earlier, Theorem 3.10 is the primary tool for proving that languages are not context free. We present just one simple example.

Application 5. *The language $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$ is not context free.*

The proof consists of a case-by-case analysis of where the pumping pair of strings, v, x, of Theorem 3.10 can reside, relative to the blocks of a's, b's, and c's in each string of L.

1. If the strings v and x exist, then each contains instances of only one letter. If either of these strings, say v, contained instances of two or more letters, then after a single pump, the resulting string would no longer belong to the language $a^\star b^\star c^\star$, hence not to the language L.
2. By item 1, each of v and x is contained within a single one of the three blocks of letters that comprises each string of L. This means that after a single pump, at most two of the three blocks will have increased in length. Once this happens, the three blocks will no longer share the same length, so the pumped string will not belong to L.

The language L is particularly simple prey for the Pumping Lemma. The reader is invited to prove (the true fact) that the language of "squares" considered in Application 4 of Section 3.3.2 is not context free.

3.8 Exercises

1. Give DFA's that accept the following languages over the alphabet $\{0, 1\}$.
   (a) The set of all strings that end in 00.
   (b) the set of all strings that contain the string 000 somewhere (not necessarily at the end);
   (c) the set of all strings that contain the string 011, in that order, but not necessarily consecutively.
2. Prove that the following languages are not regular.
   (a) $L = \{a^n \mid n \text{ is a power of } 2\}$;
   (b) $L = \{a^n b^m \mid n \leq m\}$.
   (c) The language L_5 of Lemma 3.5.

3. Use the TRIE (digital search tree) data structure (see any text on data structures or algorithms) to prove that every finite set of binary strings is regular.

4. Prove that there exist three languages $L_1 \subset L_2 \subset L_3$, all over the alphabet $\Sigma = \{a, b\}$ such that: • L_1 and L_3 are regular; • L_2 is not regular.

   Message: *The properties of subsets and supersets cannot be used to prove regularity or nonregularity.*

5. Prove that the language $\{a^n b^n c^n \mid n \geq 0\}$ is not regular:

   (a) using the "ordinary" form of the Pumping Lemma;

   (b) using the "fooling argument" based on the Continuation Lemma.

6. Compute the minimum-state FA that is equivalent to the following one.

M	q	$\delta(q,0)$	$\delta(q,1)$	$q \in F$?
$\rightarrow$	a	b	a	$\notin F$
	b	a	c	$\notin F$
	c	d	b	$\notin F$
	d	d	a	$\in F$
	e	d	f	$\notin F$
	f	g	e	$\notin F$
	g	f	g	$\notin F$
	h	g	d	$\notin F$

7. Prove that if an n-state FA M accepts any string — i.e., $L(M) \neq \emptyset$ — then it accepts a string of length $< n$.

8. In the [(3) $\Rightarrow$ (1)] portion of the proof of the Myhill-Nerode Theorem, we need to show that, for all strings $x \in \Sigma^\star$, $\delta([\varepsilon], x) = [x]$. Prove that these equations hold.

9. Prove that the minimal-state FA $\widehat{M}$ defined in Section 3.3.3 is well-defined, that it accepts the same language as does the original FA M (i.e., $L(\widehat{M}) = L(M)$), and that no FA with fewer states than $\widehat{M}$ accepts $L(M)$.
   (Much of this is just summarizing what we prove in the text.)

10. Present—and justify—an algorithm that decides of a given FA $M = (Q, \Sigma, \delta, q_0, F)$ whether or not $L(M) = \Sigma^\star$. Your algorithm should run in time $O(|Q| \times |\Sigma|)$.
    Hint: Familiarize yourself with the proof that the regular languages are closed under the Boolean operations: union, intersection, complementation.

11. (a) Prove that there is no integer k such that every regular set is accepted by a DFA having $\leq k$ accepting states.

(b) Every nonempty regular set is accepted by some NFA that has just one accepting state, *even when NFAs are not allowed to have ε-transitions.*

12. How many distinct regular sets over the alphabet $\{a\}$ are accepted by a DFA having n states and one accepting state? You should restrict attention to languages that are *not* accepted by any DFA having fewer than n states.

13. Prove that the 4-state FA below adds binary integers, in the following sense. We view the FA as emitting the bit 1 whenever it enters an accepting state, and as emitting the bit 0 whenever it enters a rejecting state. We then design an FA that, on any input string over the 4-symbol alphabet $\{0,1\} \times \{0,1\}$, say,

$$\langle\alpha_0, \beta_0\rangle\langle\alpha_1, \beta_1\rangle \cdots \langle\alpha_n, \beta_n\rangle$$

emits the bit-string

$$\gamma_0\gamma_1 \cdots \gamma_n$$

just when the bit-string $\gamma_n\gamma_{n-1} \cdots \gamma_0$ comprises the low-order $n+1$ bits of the sum of

$$\alpha_n\alpha_{n-1} \cdots \alpha_0 \text{ and } \beta_n\beta_{n-1} \cdots \beta_0.$$

Note the reversed order of the input and output strings of the FA. The adding FA has four states, $\{q_0, q_1, q_2, q_3\}$; q_0 is the start state; q_1 and q_3 are the accepting states. The state-table for the FA is:

δ	$\langle 0,0\rangle$	$\langle 0,1\rangle$	$\langle 1,0\rangle$	$\langle 1,1\rangle$
q_0	q_0	q_1	q_1	q_2
q_1	q_0	q_1	q_1	q_2
q_2	q_1	q_2	q_2	q_3
q_3	q_1	q_2	q_2	q_3

Hint: Interpret the states via outputs and carries.

14. Using the same formal framework as in the preceding problem, prove that finite automata *cannot* multiply. That is, there is no FA which, on every input string

$$\langle\alpha_0, \beta_0\rangle\langle\alpha_1, \beta_1\rangle \cdots \langle\alpha_n, \beta_n\rangle \in (\{0,1\} \times \{0,1\})^\star$$

emits the bit-string

$$\gamma_0\gamma_1 \cdots \gamma_n$$

that comprises (in reverse order) the low-order $n+1$ bits of the *product* of

$$\alpha_n\alpha_{n-1} \cdots \alpha_0 \text{ and } \beta_n\beta_{n-1} \cdots \beta_0.$$

Hint: Note that we allow leading 0's in the input.

15. (a) Prove that the following set *is not* regular.

$$\{x \in \{0,1\}^\star \mid \text{the length of } x \text{ is a power of } 2\}$$

(b) Prove that the following set *is not* regular.

$$\{a^m b^n \mid m \neq n\}$$

(c) Prove that the following set *is* regular.

$$\left\{a^\ell b^m c^n \mid m \text{ can be written as } m = h + k \text{ for some } h \neq \ell \text{ and } k \neq n\right\}$$

16. *Use pumping arguments to prove the following assertions.*

(a) The following language of *perfect squares under concatenation* is not context-free:

$$L = \{xx \mid x \in \{a,b\}^\star\}.$$

Each string in L consists of a string of a's and b's, followed by an identical copy of the same string.

(b) The set of strings over the alphabet $\{a\}$ whose length is a power of 2 is not regular.

(c) The set of strings over the alphabet $\{a\}$ whose length is a power of 2 is not context-free.

(d) The following language is not regular.

$$L = \{a^i b^j c^k \mid i = j \;\; \textbf{or} \;\; j = k\}.$$

17. Consider two FA, M_m and M_n, which share the one-letter input alphabet $\{a\}$, and which accept, respectively, the set of all strings whose length is divisible by m and the set of all strings whose length is divisible by n.

(a) Use the direct-product construction on M_m and M_n to produce a DFA that accepts the set of all strings whose length is divisible by the least common multiple of m and n.

(b) Prove that when m and n are *relatively prime* — i.e., have unit greatest common divisor — then the DFA you just produced from M_m and M_n is the *smallest* DFA (in number of states) that accepts this set.

18. You are given two DFA's, M_1 and M_2. Present—and justify—an algorithm that decides whether or not $L(M_1) = L(M_2)$.
Do this problem in the following two distinct ways.

(a) Use the Myhill-Nerode Theorem to argue that there is a unique smallest DFA equivalent to each DFA M_i. Then use the preceding fact to craft an algorithm that decides whether or not $L(M_1) = L(M_2)$.
Remark. The "uniqueness" of the smallest DFA is only up to possible renamings of the states. Your algorithm should take this into account.

(b) Use the fact that the regular languages are closed under the Boolean operations to reduce the equivalence problem to an easier decision problem.

19. Recall the NFA M that accepts all strings over $\{a, b\}$ whose penultimate letter is a. When we converted M to an equivalent DFA M', we generated the states of M' in a *lazy* manner — since only states accessible from $\{q_0\}$ are of interest. List the states that we did *not* generate.

20. Explain why there is no analogue of the Myhill-Nerode Theorem for NFA's. In particular, why can one *not* identify the "states" of an NFA as the classes of an equivalence relation. Your answer should be short and to-the-point, not a rambling essay.
(The answer is not abstruse. Look at some sample NFA's, and *think* about what the various terms in this question mean.)

21. Prove that every ε-free NFA M for which $\varepsilon \notin L(M)$ can be converted to an equivalent ε-free NFA M' that has only one accepting state. (It is easy to prove that this is *not* possible in general for DFAs.)

22. Prove that an ε-free NFA that is allowed to have multiple start states can be converted to an equivalent ε-free NFA M' that has only one start state.

23. The subset construction converts an (ε-free) n-state NFA M to an equivalent DFA M', potentially with 2^n states. Prove, via the following example, that this explosion in number of states is sometimes inevitable. The vehicle for your proof should be the following language over the alphabet $\Sigma = \{a, b\}$.

$$L_n \overset{\text{def}}{=} \{x \in \Sigma^\star \mid \text{ the } n\text{th symbol from the end of } x \text{ is an } a\},$$

where n is a positive integer.

Prove that, for all n, any DFA M' that accepts L_n has $\Omega(2^n)$ states.

Hint: Show that the Myhill-Nerode equivalence relation $\equiv_{L_n}$ has $\Omega(2^n)$ classes, by considering the continuation-lemma consequences of the "guesses" by the NFA.

24. We have shown that the language $L = \{a^k \mid k \text{ is a perfect square}\}$ is not regular. It follows that the Myhill-Nerode relation $\equiv_L$ has infinitely many classes. Describe all of the classes.

25. Prove whether or not the following sets are regular.

(a) $\{x \in \{0,1\}^\star \mid \text{Length}(x) \text{ is a power of } 2\}$

(b) $\{x \in \{0,1\}^\star \mid \text{Length}(x) \text{ is the sum of two perfect squares}\}$
(**Hint:** Consult any standard number theory book to see which integers are sums of so-and-so many squares.)

(c) $\{x \in \{0,1\}^\star \mid x \text{ begins with a palindrome of length} \geq 3\}$

(d) $\{x \in \{0,1\}^\star \mid x \text{ contains a subword that is a palindrome}\}$

(e) $\{a^m b^n \mid m = n^2\}$

(f) $\{a^m b^n \mid m \equiv n^2 \pmod 7\}$

26. Prove or disprove:

(a) Every subset of a regular set is regular.

(b) There is a nonregular set L such that $L^\star$ is regular.

(c) There are nonregular sets L_1 and L_2 such that both $L_1 \cup L_2$ and $L_1 \cap L_2$ are regular.

27. Use the direct-product DFA $M_{1,2}^{\otimes}$ from the proof of Theorem 3.7 to *prove* that the regular languages are closed under the set-theoretic operation of *symmetric difference* $\oplus$ defined as follows. $L_1 \oplus L_2$ is the set of all strings that belong *either* to L_1 *or* to L_2, but *not to both.*

28. Prove that the following language L_1 *is not* context free, but that language L_2 *is* context free.

$$\begin{aligned} L_1 &= \{xy \in \{0,1\}^\star \mid x = y\} \\ L_2 &= \{xy \in \{0,1\}^\star \mid \mathit{Length}(x) = \mathit{Length}(y) \;\text{ and }\; x \neq y\} \end{aligned}$$

29. Consider the following family of finite (hence, regular) languages: $\mathcal{L}^{\neq} = \{L_i^{\neq} \mid i \in \mathbb{N}\}$. For each $n \in \mathbb{N}$,

$$L_n^{\neq} = \{xy \mid x, y \in \{0,1\}^n \;\text{ and }\; x \neq y\}\,.$$

Prove the following assertions.

(a) For all n, there is an NFA M, having $O(n^2)$ states, such that $L(M) = L_n^{\neq}$.
(You may describe M in English, but be sure that you describe in detail the states of M and the transitions among them.)

(b) Any *deterministic* FA that accepts $L_n^{\neq}$ must have at least 2^n states.

30. Contrast the family $\mathcal{L}^{\neq}$ of the preceding problem with the family of finite (hence, regular) sets $\mathcal{L}^{=} = \{L_i^{=} \mid i \in \mathbb{N}\}$, where, for each $n \in \mathbb{N}$,

$$L_n^{=} = \{xy \mid x, y \in \{0,1\}^n \ \text{ and } \ x = y\}\,.$$

Prove that any NFA that accepts $L_n^{=}$ must have at least 2^n states.

31. Present algorithms that decide, of given FA M_1 and M_2 which share the input alphabet Σ, whether or not:

   (a) $L(M_1) = \emptyset$

   (b) $L(M_1) = \Sigma^{\star}$

   (c) $L(M_1) = L(M_2)$

   (d) $L(M_1) \subseteq L(M_2)$.

32. Prove Theorem 3.10.

Chapter 4

Computability Theory

4.1 Introduction and History

Mathematics has been "practiced" for thousands of years, yet it was not until the 19th century that people strove to formalize the notion of *proof*. (The advent of computers and computer-assisted proofs has led to a re-thinking of the formal notion that is still ongoing.) The formal notion that led to the birth of the field of mathematical logic made a proof a kind of rewriting system. One started with a set of *axioms*, or "self-evident truths," which were automatically granted the status of *theorem*. One then added a set of *rules of inference*, or, rewriting rules, that allowed one to convert existing theorems into new ones. Inspired by this apparently "mechanistic" notion of "proving a theorem," at the very end of the 19th century, the great German mathematician D. Hilbert challenged mathematicians to devise automatic procedures—what we would now call *algorithms*—that would either prove or refute purported theorems in the elementary theory of numbers. The hope for such procedures was dashed in 1931 by the famous Incompleteness Theorem of K. Gödel. Informally, the Theorem said that in any mathematical system—i.e., axioms plus rules of inference—that was powerful enough to "talk" about elementary properties of the positive integers, the notion of "theorem" could never be powerful enough to encompass the notion of "true statement." The proof of the Theorem employed mathematical tools developed by G. Cantor for comparing the relative "sizes" of infinite sets. We have seen these tools in Section 2.3.2. In the mid-1930's, A.M. Turing essentially adapted the logical framework of Gödel to a computational setting, leading to what we now call the Theory of Computability. Turing showed that there existed functions $f : \mathbb{N} \rightarrow \{0, 1\}$ ($\mathbb{N}$ is the set of nonnegative integers.) that were quite simple to specify informally but that could not be computed by any reasonable notion of digital computer. (Recall that digital computers did not exist in the 1930's, so Turing had to use a lot of imagination to craft a formal "reasonable notion of digital computer.") Turing's landmark work led to an incredible number of competitive attempts to craft a formal "reasonable notion of digital computer," with an attendant Theory of Computability. Remarkably, no such attempt produced a formal model that was more powerful than Turing's eponymous *Turing machine*, but many of these attempts gave rise to equally powerful, quite different, alternative formulations of the Theory. Acknowledging the many contributions of Turing and his competitors, the Theory has freely absorbed notions from competing theories based on: *Turing machines*, the *lambda calculus*, the theory of *recursive functions*, *combinatory logics*, Type-0 grammars, *Markov algorithms*, and on and on.

The confluence of the many attempts to formulate a Theory of Computability has led all mainstream computer scientists and mathematicians to accept the extra-mathematical *Church-Turing Thesis:*

> *The informal notion "computable by a digital computer" is equivalent to the formal notion "computable by a Turing machine.*

This chapter is incredibly rich intellectually. It deals with the very underpinnings of the mathematical theory of computation that was initiated by Turing [39] and others with kindred concerns. We start our study with a section that appears to be of purely mathematical interest, yet develops the main mathematical tools for the entire theory of computability. Once we turn to the details of Computability Theory, we employ the "model-independent" approach that seems to have originated in the classical text of [31]. When you think about the process of computing, you should think about concepts using your own favorite (real!) programming language—anything from APL to BASIC to C to C++ to Java to . . . ; this should allow you both to employ intuitions that you have developed over your years of programming and to appreciate the relevance of the Theory to real computing. The mathematical underpinnings of the next section should convince you that the specific language you choose is irrelevant: any language you think of can be "encoded" as any another. (Our noble pursuit of "model-independence" notwithstanding, we shall have to introduce the classical *Turing Machine* (*TM*, for short) as a "standard" model for algorithm/digital computer in Chapter 5. Of course, we already saw a variant of the model in Chapter 3.3.4.B.)

4.2 Preliminaries

4.2.1 Representing computational problems as "languages"

In Section 2.4.2, we discussed briefly how one could talk about a variety of computational problems using the medium of formal languages. There are historically unavoidable terminological corollaries of this unusual way of talking about computation.

A *set* (of integers or strings) is:	*decidable* or *recursive*	or	*undecidable* or *nonrecursive*
A *computational problem* is:	*solvable*	or	*unsolvable*
A *property of a system* is:	*decidable*	or	*undecidable.*

All of these notions are equivalent to the assertion that the characteristic function of A is (in the case of decidability)—or is not (in the case of undecidability)—computable by a program.

In order to develop our theory, we must extend the preceding notions in a way that you may not have anticipated.

The *semi-characteristic function* of the set/language A is the function κ'_A defined as follows:

$$(\forall x \in \mathbb{N}) : \ \kappa'_A(x) \ = \ \begin{cases} 1 & \text{if } \ x \in A \\ \text{is} & \text{undefined if } \ x \notin A \end{cases}$$

probably the best concrete way to think about semi-characteristic functions in a computational setting is as follows. If the semi-characteristic function κ'_A is computed by a program P, then P would halt and say "YES" when presented with an input that belongs to set A, and P would never halt when presented with an input that does not belong to set A. To emphasize the possibility of P's not halting, we would call the function κ'_A *semi-computable.*

We now have (weaker) companion notion to decidability.

A *set* (of integers or strings) is:	*semi-decidable* or *recursively enumerable*
A *computational problem* is:	*partially solvable*
A *property of a system* is:	*semi-decidable.*

All of these notions are equivalent to the assertion that the *semi*-characteristic function of A is *semi*-computable by a program, in the sense just discussed.

4.2.2 Functions and Partial Functions

This is a good time to review the material in Section 2.3.1.

We shall select a fixed countable universe of discourse when we talk about functions. While the encodings we presented when discussing countability give us a broad range of choices for the fixed universal set, typical choices found in the literature are the set $\mathbb{N}$ of natural numbers, the set $\{0,1\}^\star$ of finite-length binary strings, and the generic set $\Sigma^\star$ of finite-length strings over a generic finite alphabet $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_n\}$. We shall choose one such universal set U—making sure that U contains 0 and 1, to simplify our lives—and always talk either about functions $f : U \rightarrow U$ or about functions $g : U \rightarrow \{0,1\}$; the latter class of functions is mandated by our focus on *languages.* Since we know about pairing functions, we never have to widen our focus to accommodate multivariate functions; we just use a pairing function to map $U \times U$ one-to-one onto U.

Our insistence on a fixed universe is mostly a happy decision, as it allows us to talk about functions and compositions of functions without worrying about questions of compatibility between domains and ranges. One unhappy consequence of this insistence, though, is that we have to develop our theory of computability (or of computational complexity) as a theory of *partial* functions.

By default, every function $f : U \to U$ is a *partial* function; i.e., "partial" is the default terminology. In the special case when, for each $u \in U$, there exists a value $f(u) \in U$, we call f a *total* function. Although we seldom talk explicitly about partial functions in everyday discourse, as computer scientists, we deal with such functions all the time; for instance, we are all aware that both $f(n) = \sqrt{n}$ and $g(n) = n/2$ are *partial functions* on the natural numbers, as f is defined only when n is a perfect square, and g is defined only when n is even. (Although it is not relevant to the point we are making here, it is worth noting that we often—but not always—opt to simplify our lives by extending such partial functions to be total. In the cited cases, we use floors and ceilings for this, as in $\lceil\sqrt{n}\rceil$ and $\lfloor n/2\rfloor$.)

4.2.3 Self-Referential Programs: Interpreters and Compilers

The major concepts of computability theory were developed before programmable computers existed. It is quite remarkable, then, to note that the people who developed the theory came up with the notion of an *interpreter*: a program P that takes as arguments strings x and y, that interprets x as a program, and that simulates program x step by step on input y. There is, of course, no reason that P, x, and y could not all be the same string—in which case, the interpreter would be simulating itself operating on itself .... Such *self reference* plays havoc with our intuitions, as you can see by pondering whether or not the following sentence is true: "This sentence is false."

Like it or not, self reference is part of our lives as computer scientists. We may not be happy about this, since self reference has some ugly consequences, the first of which we see in the next section.

4.3 The Unsolvability of the Halting Problem

This is a good time to review the material in Section 2.3.2.

The *Halting Problem*, HP, is the following set of ordered pairs of strings:

$$HP \overset{\text{def}}{=} \{\langle x, y\rangle \mid \text{ Program } x \text{ halts when presented with input } y\}.$$

By using a pairing function, we turn HP into a language. The *Diagonal Halting Problem*, DHP is the set of all programs that halt when supplied with their own descriptions as input.

Symbolically:

$$DHP \overset{\text{def}}{=} \{x \mid \text{ Program } x \text{ halts when presented with input } x\}.$$

If you look back at our discussion of *un*countability, you should understand the adjective "Diagonal" here.

Theorem 4.1. (Turing, 1936) *The Halting Problem is not solvable. In other words: the set HP is not decidable.*

Proof. We shall, in fact, focus on proving the undecidability of the Diagonal Halting Problem, since trivially, if HP were decidable, then so also would be DHP, since $x \in DHP$ if, and only if, $\langle x, x\rangle \in HP$. (We shall see later that the preceding sentence is actually asserting the *mapping-reducibility* of the Diagonal Halting Problem to the Halting Problem.)

Assume, for contradiction, that DHP were decidable. There would, then, be a program—call it p—that behaved as follows.

$$\text{On input } x,\ p \ \text{ outputs} \begin{cases} 1 & \text{if Program } x \text{ halts when presented with input } x \\ 0 & \text{if Program } x \text{ does not halt when presented with input } x \end{cases}$$

(Program p would compute the characteristic function of DHP.)

I now want to draw on your experience writing programs. You should agree that, if you were presented with the preceding program p, then you could modify the program to a program p' that behave as follows.

$$\text{On input } x,\ p' \ \text{ outputs} \begin{cases} 1 & \text{if Program } x \textit{ does not halt} \text{ when presented with input } x \\ \text{loops} & \text{if Program } x \text{ halts when presented with input } x \end{cases}$$

We have placed no restriction on the input to either program p or program p'. In particular, this input could be the string p' itself—shades of self reference! How does program p' respond to being presented with its own description? The following sequence of biconditionals ("if-and-only-if" statements) tells the story.

Program p' *halts* when presented with input p'
if, and only if,
Program p' outputs 1 when presented with input p'
if, and only if,
Program p' *does not halt* when presented with input p'.

You can "play" this sequence in either direction, in either case arriving at a final statement that contradicts the first statement.

What can be wrong here? The only unsubstantiable step in our development was the very first one, namely, the assumption that the program p existed. This proves the theorem. $\square$

Although Theorem 4.1 shows that problem HP is "hard," the news is not totally bad.

Theorem 4.2. *The Halting Problem HP is semi-decidable.*

Proof. To semi-decide if the input $\langle x, y\rangle \in HP$, construct a Program P that behaves as follows:

- Program P simulates Program x on input y.
- If Program x ever halts and gives an output, then have Program P halt and give the same output.

Note that we have given no indication of what to do if Program x never halts. This is fine since we are *semi*-deciding HP; cf. Section 4.2.1. $\square$

4.4 Reducibility and Completeness

4.4.1 Reducing one problem to another

At an intuitive level, the ability to "reduce Problem A to Problem B" means that we can use any solution to Problem B to "help" us solve Problem A. The major source of informality here is the meaning of the word "help." In the context of computability theory, we mean that we can convert any program that decides language B into a program that decides language A. When we deal with complexity theory later, we shall want to insert the qualifier "efficiently" before the word "convert" and the qualifier "efficient" before both occurrences of the word "program" in the preceding sentence. Let's stick with computability theory for now, and let's say that all the languages of interest are over the finite alphabet Σ. The following is one of the main concepts (one could argue *the* main concept) in all of computation theory.

> *Language A is mapping-reducible (m-reducible, for short) to Language B*, written $A \le_{\rm m} B$, if, and only if, there exists a total computable function $f : \Sigma^\star \to \Sigma^\star$ such that, for all $x \in \Sigma^\star$, $[x \in A]$ if, and only if, $[f(x) \in B]$.

It is fruitful to view the function f as a mechanism for "encoding" instances of Problem A as instances of Problem B.

It is easy to verify that m-reducibility plays the desired role with respect to "helping" one decide or semi-decide languages.

Lemma 4.1. *Let A and B be languages over the alphabet Σ, and say that $A \leq_{\mathrm{m}} B$.*

(a) *If language B is semi-decidable (resp., decidable), then language A is semi-decidable (resp., decidable).*

(b) *Contrapositively, if language A is* not *semi-decidable (resp.,* not *decidable), then language B is* not *semi-decidable (resp.,* not *decidable).*

Proof. It clearly suffices to deal only with part (a). Let f be the total computable function that m-reduces A to B, and let φ be a program that computes f.

If language B is semi-decidable, then there is a program p that, when presented with a string $x \in \Sigma^\star$, halts and outputs 1 precisely when $x \in B$; program p loops forever if $x \notin B$.

If language B is semi-decidable, then there is a program p' that, when presented with a string $x \in \Sigma^\star$, always halts. It outputs 1 when $x \in B$ and 0 when $x \notin B$.

In either case, we can use the program φ as a preprocessor to either program p or to program p' (if the latter exists). Now, φ converts any input $y \in \Sigma^\star$ whose membership in language A is of interest to an input $f(y) \in \Sigma^\star$ that belongs to language B iff y belongs to language A. Therefore, composite program φ-then-p is a semi-decider for language A; and, the composite program φ-then-p' (if p' exists) is a decider for language A. □

It is of great importance for the development of the theory that the reducibility relation is transitive.

Lemma 4.2. *The relation "is mapping reducible to," $\leq_{\mathrm{m}}$, is transitive. In other words, for any three languages $A, B, C \subseteq \Sigma^\star$, if $A \leq_{\mathrm{m}} B$ and $B \leq_{\mathrm{m}} C$, then $A \leq_{\mathrm{m}} C$.*

Proof. Say that:

- $A \leq_{\mathrm{m}} B$ via the total computable function f; i.e.,

$$(\forall x \in \Sigma^\star)\ [x \in A] \Longleftrightarrow [f(x) in B].$$

- $B \leq_{\mathrm{m}} C$ via the total computable function g; i.e.,

$$(\forall x \in \Sigma^\star)\ [x \in B] \Longleftrightarrow [g(x) in C].$$

Then, easily,

$$(\forall x \in \Sigma^\star)\ [x \in A] \Longleftrightarrow [f(g(x)) in C]. \tag{4.4.1}$$

Since the composition of total computable functions is another total computable function, condition (4.4.1) is equivalent to saying that $A \leq_{\mathrm{m}} C$. □

4.4.2 Complete, or “hardest,” problems

One of the most exciting features of mapping-reducibility is that there exist semi-decidable problems/languages that, in a precise, formal sense, are the *hardest* semi-decidable problems. We call these hardest problems *m-complete*, where, as in the term “m-reducibility,” the “m” stands for “mapping.”

A problem $A \subseteq \mathbb{N}$ is *m-complete* iff:

1. A is semi-decidable.
2. Every semi-decidable problem B is mapping-reducible to A: $B \leq_{\mathrm{m}} A$.

The reason we apply the sobriquet “hardest” to m-complete semi-decidable problems is that (by Lemma 4.1):

> *If any semi-decidable problem A were decidable, then* all *semi-decidable problems would be decidable.*

It is not clear a priori that there exist m-complete problems. In fact, though, we have been dealing with two of them throughout this section.

Theorem 4.3. *The set HP (the Halting Problem) is m-complete.*

Proof. The semi-decidability of HP having been established in Theorem 4.2, we concentrate only on the fact that every semi-decidable problem reduces to HP.

Let A be an arbitrary semi-decidable problem. By definition—see Section 4.2.1—there is a program P such that, for all integers x,

$P(x)$ halts precisely when $x \in A$.

If y is a string or integer “name” of program P—hence, of set A—then the preceding condition can be rewritten as:

$\langle y, x \rangle \in HP$ if, and only if $x \in A$.

Easily, this last assertion implies that $A \leq_{\mathrm{m}} HP$ via the total computable function f_y defined by: $f_y(x) = \langle y, x \rangle$.

Since A was an arbitrary semi-decidable problem, the Theorem follows. $\square$

Corollary 4.1. *The set DHP (the Diagonal Halting Problem) is m-complete.*

Proof. The semi-decidability of DHP having been established in Theorem 4.2, we concentrate only on the fact that every semi-decidable problem reduces to DHP. Further, since mapping reducibility is transitive (Lemma 4.2), it suffices to show that $HP \leq_{\mathrm{m}} DHP$ and then invoke Theorem 4.3, which shows that HP is m-complete.

We begin our demonstration that $HP \leq_{\mathrm{m}} DHP$ by revisiting Section 2.3.2 in the light of what we know now about computability. We invoke the countability of the set $\Sigma^\star$ (for any finite set Σ) to note that, not only is there an injection $F : \Sigma^\star \times \Sigma^\star \rightarrow \Sigma^\star$, but there is actually a *computable* such injection. (We leave the easy verification to the reader.) This means that we can computably deal with ordered pairs of strings, $\langle x, y\rangle$, as though they were strings, $F(x, y)$.

Consider now the following program.

```
Program   Simulate.1
Input     $x$
Input     $y$
Input     $z$
if        Prog. $x$ halts on input $y$
then      output $z$
else      loop forever
```

Note that, whenever the first two inputs to **Program** Simulate.1 form a pair $\langle x, y\rangle$ that belongs to the language HP, the program computes the identity function on its third input—hence, halts for every value of the third input. When the first two inputs form a pair $\langle x, y\rangle$ that belongs to the language $\overline{HP}$, the program computes the empty function—hence, never halts for any value of the third input.

As we did in the proof of the Rice-Myhill-Shapiro Theorem, Theorem 4.4, we can replace **Program** Simulate.1 by an infinite family of programs—one for each ordered pair of strings $\langle x, y\rangle$. The family member that corresponds to the specific ordered pair $\langle x_0, y_0\rangle$ reads as follows.

```
Program   Simulate.2:$\langle x_0, y_0\rangle$
Input     $z$
if        Prog. $x_0$ halts on input $y_0$
then      output $z$
else      loop forever
```

Note that each program **Program** Simulate.2:$\langle x_0, y_0\rangle$ in this family either computes the identity function (when $\langle x_0, y_0\rangle \in HP$), *which is total*, or the empty function (when $\langle x_0, y_0\rangle \in \overline{HP}$), *which is nowhere defined.*

Once again, we invoke our knowledge of how interpreters work to assert that we can write a program P that always halts—hence, computes a total computable function—and that, on any input pair $\langle x, y\rangle$ produces the string that is the program **Program** Simulate.2:$\langle x, y\rangle$. (In other words, this program is the value $P(x, y)$.)

We can now ask, for any pair of strings $\langle x, y\rangle$, whether or not the string $P(x, y) \in DHP$, i.e., whether or not Program $P(x, y)$ halts when it is run with a copy of itself as input. From what we have said earlier, $P(x, y) \in DHP$ if, and only if, Program $P(x, y)$ halts on all inputs—and this happens if, and only if, $\langle x, y\rangle \in HP$.

We have, thus, shown that, for all pairs of strings $\langle x, y\rangle \in \Sigma^\star \times \Sigma^\star$, $\langle x, y\rangle \in HP$ iff $P(x, y) \in DHP$. In other words, we have shown that $HP \leq_{\rm m} DHP$, whence the latter language is m-complete. □

4.5 The Rice-Myhill-Shapiro Theorem

The theorem we are about to prove states—informally!—that there is nothing "nontrivial" that one can determine about the function computed by a program from its static description. The word "nontrivial" here precludes behavioral properties that are true either of no program or of every program. Now, on to the formal statement of the Theorem.

A language A (whose constituent strings are interpreted as programs) is a *Property of Functions* (*PoF*, for short) if the following is true. If the programs x and y compute the same function (say, from $\Sigma^\star$ to $\Sigma^\star$), then either both x and y belong to A, or neither does. In other words, all programs that compute the same function lie on the same side of the metaphorical line that separates the language A from its complement $\bar{A}$. PoF's are our formal mechanism for talking about functions within the Theory of Computability: we identify a property of functions with the set of all programs that compute functions that enjoy the desired property. A few examples:

- The property "total function" is embodied in the set of all programs that halt on every input: $\{x \mid P_x \text{ halts on all inputs}\}$.
- The property "empty function" is embodied in the set of all programs that never halt on any input: $\{x \mid P_x \text{ never halts on any input}\}$.
- The property "constant function" is embodied in the set of all programs that halt and produce the same answer, no matter what the input is: $\{x \mid P_x \text{ halts, and produces the same result, o}$
- The property "the square root" is embodied in the set of all programs that halt precisely when their input is an integer that is a perfect square and that produce, when they halt, the output $\sqrt{n}$ (or, really, a numeral therefor) on input n.

A PoF A is *nontrivial* if there exists some program x that belongs to A and there exists some program y that does not belong to A. In other words, neither A nor $\bar{A}$ is empty.

Theorem 4.4. (The Rice-Myhill-Shapiro Theorem) *Every nontrivial PoF is undecidable. In other words: If a language A is a nontrivial Property of Functions, then A is not decidable. Furthermore, if a program for the* empty function *belongs to the PoF A, then A is not semi-decidable.*

Proof. Let us concentrate, for definiteness, on programs that compute (partial) functions from $\{0,1\}^\star$ to $\{0,1\}^\star$. As we now know, this is really no restriction because of our ability to encode any finite set as any other finite set.

Let us denote by Prog. x the program specified by the string x and by F_x the function computed by Prog. x. In particular, let e be a string such that Prog. e loops forever on every input, so that F_e is the empty (i.e., nowhere defined) function.

It should be clear how to supply the details necessary to turn the following pseudoprogram into a real interpreter program, in a real programming language (of your choice).

```
Program   Simulate.1
Inputs    $x$, $y$, $z$
if        Prog. $x$ halts on input $x$
then      Simulate Prog. $y$ on input $z$
else      loop forever
```

You should be able to verify that

$$F_{\mathsf{Simulate.1}}(x,y,z) \;=\; \begin{cases} F_y(z) & \text{if } x \in DHP \\ F_e(z) & \text{if } x \notin DHP \end{cases}$$

If you are comfortable with the development thus far, then you should agree that we can replace the input y in **Program** Simulate.1 by a specific string—call it y_0—that is fixed once and for all. We thereby get the following pseudoprogram, which you can convert into a real interpreter program, in a real programming language.

```
Program   Simulate.2
Inputs    $x$, $z$
if        Prog. $x$ halts on input $x$
then      Simulate Prog. $y_0$ on input $z$
else      loop forever
```

You should now be able to verify that

$$F_{\mathsf{Simulate.2}}(x,z) \;=\; \begin{cases} F_{y_0}(z) & \text{if } x \in DHP \\ F_e(z) & \text{if } x \notin DHP \end{cases}$$

Now, the dependence of **Program** Simulate.2 on input x is so formulaic that we could actually supply x to a *preprocessor* for our interpreter program, that automatically inserts a value for x into **Program** Simulate.2. Indeed, we can design this preprocessor so that, in response to any string x, it produces the following program which will be the input to our interpreter program.

```
Program  Simulate.3
Input    z
=====================================
if       Prog. x halts on input x
then     Simulate Prog. y_0 on input z
else     loop forever
```

In more detail, the preprocessor is specified by the following program.

```
Program  Preprocessor
Input    x
==============================================
         "Input  z
          ====================================
Output    if     Prog. x halts on input x
          then   Simulate Prog. y_0 on input z
          else   loop forever"
```

Note that **Program** Preprocessor just outputs a string that is fixed except for the indicated inclusion of the input x. This string is **Program** Simulate.3 with the appropriate value of x in the indicated place. Thus, **Program** Preprocessor *always halts* on every input x; i.e., it computes a *total computable* function, $\mathcal{F} : \{0,1\}^\star \to \{0,1\}^\star$. You should now be able to verify that

$$F_{\mathcal{F}(x)}(z) \;=\; F_{\mathsf{Simulate.3}}(z) \;=\; \begin{cases} F_{y_0}(z) & \text{if } x \in DHP \\ F_e(z) & \text{if } x \notin DHP \end{cases} \tag{4.5.2}$$

Let us now shift gears and start thinking about an arbitrary but fixed nontrivial PoF A. Now, the program e for the empty function belongs either to A or to $\bar{A}$. We shall assume that $e \notin A$, leaving to an assignment the determination of what happens when $e \in A$. (The required changes in the argument should be clear.) Since A is a PoF, we know that *no program $y \in A$ is equivalent to Prog. e*—which means that *every program $y \in A$ halts on some input.* Since A is nontrivial, there must be some program $y_0 \in A$. Let's see what happens when we let this program y_0 be the y_0 mentioned in **Program** Simulate.3. When this happens, we can infer from (4.5.2) that

$$[x \in DHP] \;\Leftrightarrow\; [\mathcal{F}(x) \in A]. \tag{4.5.3}$$

Why is this true?

1. If $x \in DHP$, then $F_{\mathcal{F}(x)} \equiv F_{y_0}$, as functions. This means that programs $\mathcal{F}(x)$ and y_0 compute the same function. Since $y_0 \in A$ (by hypothesis) *and* since A is a PoF, this means that $\mathcal{F}(x) \in A$.

2. If $x \notin DHP$, then $F_{\mathcal{F}(x)} \equiv F_e$, as functions. This means that programs $\mathcal{F}(x)$ and e compute the same function (the empty function in this case). Since $e \in \bar{A}$ *and* since $\bar{A}$ is a PoF, this means that $\mathcal{F}(x) \in \bar{A}$. We have used here the following fact, which you should verify: the set A is a PoF iff its complement $\bar{A}$ is a PoF.

We have now verified (4.5.3).

What have we shown here? Looking at (4.5.3) and comparing it to the formula for mapping reductions, we find that we have proved that $DHP \leq_{\mathrm{m}} A$, for any nontrivial PoF A that *does not contain* Prog. e. By Lemma 4.1, this means that any such A is undecidable.

You should be able to wend your way through our argument to prove that, for any nontrivial PoF A that *does contain* Prog. e, we can show that $\overline{DHP} \leq_{\mathrm{m}} A$. In this case, Lemma 4.1 tells us that the set A is not semi-decidable! □

4.6 Exercises

1. A *paper-tape* TM (PTM) is a one-tape off-line TM whose work-tape alphabet Γ is *partially ordered*. A PTM operates much like a TM, except that, when a PTM overwrites a symbol γ on its work tape, it must do so with some symbol γ' that is *greater than* γ in the partial order.
   (Intuitively, a PTM can "add holes" to the representation of γ, but it cannot "remove holes.")

   (a) Prove that a PTM can simulate a classical TM.

   (b) Estimate the time required by your simulation, proving a result of the form:

   A PTM can simulate T steps of an off-line TM in at most $f(T)$ steps.

2. A *Register Machine* (*RM*, for short) is a TM which, in place of work tapes, has a number of *registers*, each capable of holding *any* nonnegative integer (no matter how large). During each step of its computation, an RM performs the following actions.

- It reads the currently scanned input symbol and tests its registers to determine which are zero and which are nonzero.
- On the basis of its internal state and the outcome of the polling in action 1, the RM:
  - changes state
  - *independently* adds 0, +1, or −1 to each register
  - moves its read-only input head at most one square left or right.

Thus, the "syntax" of the (one-step) transition function of a k-register RM (abbreviated, k-RM) is:

$$\delta : Q \times \Sigma \times \{\text{zero, nonzero}\}^k \rightarrow Q \times \{-1,\ 0,\ +1\}^k \times \{\text{N, L, R}\}$$

The notions of *computation*, *acceptance*, and *rejection* by a k-RM are as with a TM. You should be able to supply details.

(a) Prove that a 3-RM can simulate a 1-tape off-line TM, via the following sequence of simulations.

  a. Prove that an automaton whose work-storage is two pushdown stacks can simulate a 1-tape off-line TM, *with no time loss.*
  (**Note**: This is basically a data-structure question: Two stacks can simulate a single tape. A stack operation is a POP followed by a PUSH.)

  b. Prove that a TM can use 2 registers to simulate the action of a single stack. (The TM is allowed to use the registers precisely as an RM uses its registers—poll by testing for zero; alter by adding 0, +1, or −1.)

  c. Complete the proof that 3 registers can simulate a tape.

(b) Estimate the time required by your entire simulation, proving a result of the form:

A 3-RM can simulate T steps of an off-line TM in at most $f(T)$ steps.

3. This problem is devoted to establishing that classical TM's and "queue-based TM's" (QTM's, for short) are equivalent in power. A QTM M looks much like a classical TM, except that it has *a single FIFO queue* as its only unbounded data structure. Initially, M's input is written on the queue in an order that allows it to be read via a sequences of dequeue operations. M reads its tape by *dequeuing* the leftmost symbol; it writes its tape by *enqueuing* a new rightmost symbol.

   Say that we are given a classical TM M_1, and a QTM M_2. Show the following, via a combination of prose and pictures.

   (a) There exists a QTM M_1' such that $L(M_1) = L(M_1')$.

   (b) There exists a TM M_2' such that $L(M_2') = L(M_2)$.

Hint: Consider how to use the queue to generate and process the sequence of configurations/instantaneous descriptions of a given classical TM M, starting with M's initial configuration.
Note: This problem shows that, in some sense, a queue is a "more powerful" data structure than a stack.

4. Say that a language L is *recursively enumerable* if there is a surjective total recursive function $f : \mathbb{N} \to L$. (Note that $f(\mathbb{N}) \subseteq L$.) Prove that a language L is recursively enumerable if, and only if, it is semi-decidable. (One direction is quite challenging.)

5. Say that the language $L \subseteq \Sigma^\star$ is *enumerable in increasing order*, in the following sense. There is a surjective total recursive function $f : \mathbf{N} \to L$ such that, for all integers i, $f(i+1) > f(i)$ when the strings $f(k)$ are interpreted as numerals (in base $|\Sigma|$). (Note that $f(\mathbf{N}) \subseteq L$.) Prove that L is decidable.

6. Prove that every *infinite* recursively enumerable language L contains an infinite decidable (a/k/a recursive) subset.

7. The relation "is mapping-reducible to" (symbolically, $\leq_{\mathrm{m}}$) satisfies the following. For any sets A and B, $[A \leq_{\mathrm{m}} B]$ iff $[\bar{A} \leq_{\mathrm{m}} \bar{B}]$.

8. If A is decidable and $B \not\in \{\emptyset, \mathbb{N}\}$, then $A \leq \mathrm{m}B$.
(Intuitively, a set that needs no "help" is "helped" by any set.)

9. Prove the following theorem. *Every nontrivial Property of Functions that contains a program for the empty function—i.e., a program that never halts on any input—is not semi-decidable.*

10. Prove that the following set is not semi-decidable: $\{\langle x, y\rangle \mid \text{Prog. } x \equiv \text{Prog. } y\}$. The defining condition here is that programs x and y compute the same function.

11. State—with a justifying proof—if each of the following sets is semi-decidable, decidable, or neither.
   (a) The set S_1 of programs that halt when started with the null string as input.
   (b) The set S_2 of programs that halt when the input is the *reversal* of their descriptions.
   (c) The set S_3 of strings that are (halting) *computations* by a Turing Machine (TM). (Review the definitions in your notes before attempting this problem.)
   (d) The set $S_4 = \{\langle x, y\rangle \mid x \text{ and } y \text{ are equivalent Finite Automata}\}$.
   (e) The set $S_5 = \{\langle w, x, y, z\rangle\}$ such that:
      - y is the inital configuration of Program w on input x;

- z is a terminal (i.e., halting) configuration of Program w when w is presented with input x.

Before you attempt this problem, consider the implications of any ordered quadruple $\langle w, x, y, z\rangle$'s membership in S_5.

12. Classify—with proofs—the following languages as being (*i*) decidable or (*ii*) semi-decidable but not decidable or (*iii*) not semi-decidable.

    (a) The language $\{x \mid \text{Program } x \text{ never halts on any input}\}$.

    (b) The language $\{x \mid \text{Program } x \text{ halts on at least one input}\}$.

    (c) The language $\{x \mid \text{Program } x \text{ halts on infinitely many inputs}\}$.
    Hint. Prove that one can reduce $\overline{DHP}$ to L.

13. Classify—with proofs—the following languages as being (*i*) decidable or (*ii*) semi-decidable but not decidable or (*iii*) not semi-decidable.

    (a) The language $\{\langle x, y\rangle \mid \text{Programs } x \text{ and } y \text{ halt on precisely the same inputs}\}$.
    Hint. Can you think of any specific programs y for which this problem is undecidable?

    (b) The language $\{x \mid \text{Program } x \text{ halts on the blank input tape}\}$.

14. We noted in Corollary 2.2 that one could infer the existence of noncomputable functions $f : \mathbf{N} \rightarrow \{0, 1\}$ from the fact that the class $\mathbf{F}$ of such functions is uncountable, while the set $\mathbf{P}$ of programs in any programming language is countable. Present a *standalone (direct)* proof of the existence of noncomputable such functions. *What you need to present is a formal proof that there is no injection from* $\mathbf{F}$ *into* $\mathbf{P}$.

15. Prove that a language $L \subseteq \Sigma^\star$ is decidable if, and only if, both L and its complement $\overline{L} = \Sigma^\star - L$ are semi-decidable.

16. Prove that the "complementary Halting Problem" $\overline{HP}$ is not semi-decidable.

17. Prove the following theorem.
    Say that the language A is mapping-complete (= m-complete, = complete with respect to mapping reductions), and say that the language B is decidable. Then it is not possible that $A \leq_{\mathrm{m}} B$.

    You should supply a complete proof of this fact, from definitions and first principles. For instance, it is *not* adequate for you to invoke vague intuitive principles such as "Good things travel up; bad things travel down."

18. In this Problem, $T = \{x \mid \text{Progr. } x \text{ halts on every input}\}$.

    (a) Present an infinite family $\mathcal{F}$ of sets such that $S \leq \bar{S}$ for each $S \in \mathcal{F}$.

(b) Is every semi-decidable set reducible to T?

(c) Prove that the set T is *productive*, in the sense that there is a total recursive function f with the following property. For every program x whose *domain* is a subset of T, we have

$$f(x) \in T - \text{Domain}(\text{Progr. } x).$$

(This problem is not as hard as it looks if you get beyond the formalism.)

Chapter 5

Complexity Theory

5.1 Introduction

While the major strength of Computability Theory can be viewed as its robustness across widely varying models, the major weakness of the Theory is its accepting as "computable" many functions that cannot be computed using all of the resources—both time and matter—in the known Universe. Parts of the Theory are, thus, mathematical abstractions that can never be realized physically. By the early 1960's—soon after the development of "real" digital computers—both theoreticians and practitioners began looking for a way to bring the strengths of Computability Theory to bear on real computational problems. Perhaps the most influential practitioner in this regard was the expert in mathematical optimization, J. Edmonds. Edmonds noticed that there were myriad computationally—and economically—significant problems for which the only known algorithms took time *exponential in the size of the problem.* Just to suggest the range of these "practically intractable" problems we mention the following two; the list could be expanded into the thousands.

1. the *Traveling Salesman Problem:* One is given a set of n cities that the "Salesman" must visit, along with a matrix of inter-city travel costs. One seeks a minimum-cost tour of the cities, in which each city is visited precisely once.

2. the Boolean Minimization Problem: One is given a boolean expression E that specifies a logic function F. One seeks the shortest expression $\widehat{E}$ that is equivalent to E, in the sense that it also specifies the function F.

From a quite different point of departure, a group of mathematical logicians were making use of the new tool, the digital computer, to extend (in a sense) the theory inherent in Gödel's earlier cited work. These logicians were trying to find efficient algorithms that would prove or refute purported theorems in the Propositional Calculus, a weak logical system that *is* complete in Gödel's sense of the word. Their attempts were running into the same problem as Edmond's optimization problems: every algorithm they crafted took time exponential in the size of the purported theorem. Around 1970, S. Cook propounded a theory that was based formally on Computability Theory, but that refined that Theory by incorporating measures of the time and space needed to compute a function. Remarkably, Cook [7], followed within a year by R.M. Karp [18] and L. Levin [21], and thereafter by myriad other computation theorists, used his theory to show that all of the problems mentioned in this section—and a host of other ones—are, in a sense that can be formalized, translates of one another. The question of whether Edmonds's optimization problems, and the problem of proving theorems in the Propositional Calculus, could be solved in time *polynomial in the size of the problem* was morphed in this new theory in the now-famous **P**-vs.-**NP** Problem that dominates this chapter.

5.2 Complexity theory as resource-bounded computability theory

5.3 Nondeterminism as unbounded search

5.4 Cook's Theorem

[7]

the importance of nondeterminism as a concept for understanding a large variety of computational problems.

5.5 Savitch's Theorem

[36]

5.6 Enrichment Topic: Online Turing Machines

Section 3.7.1.A derives a lower bound on the size of any OA M that accepts a nonregular language L, by bounding the number of classes of $\equiv_L$. In this section, we present lower bounds from [13] on the *time* a specific genre of OA requires to accept a language L, based on the "structure" of the OA's infinitely many states. Specifically, we analyze the behavior of "online" Turing Machines (TM's) whose infinitely many states arise from a collection of read-write "work tapes" of unbounded capacities. As in Section 3.7.1, the desired bound is achieved by adapting Theorem 3.1 to a broad class of infinite-state OA's. This adaptation is achieved here by parameterizing the word-relating equivalence relation $\equiv_M$; for each integer $t > 0$, the parameter-t relation $\equiv_M^{(t)}$ behaves like $\equiv_M$, but exposes only discriminations that M can make in t or fewer steps.

A word about TM's is in order, to explain why the study in this section is relevant to computer scientists. The TM model originated in the monumental study [39] that planted the seeds of computability theory, hence, also, of complexity theory. Lacking real digital computers as exemplars of the genre, Turing devised a model that served his purposes but that would be hard to justify today as a way for thinking about either computers or algorithms. Seen in this light, one surmises that TM's persists in today's textbooks on computation theory only because of their mathematical simplicity. However, I believe there is an alternative role for the TM model, which justifies continued attention—in certain contexts. Specifically, one can often devise varieties of TM that allow one to expose the impact of data-structure topology on the efficiency of certain computations. These TM's abstract the control portion of an algorithm down to a finite state-transition system and use the TM's "tapes" to model access to data structures. The study in [13] uses TM's in this way, focusing on the impact of tape topology on efficiency of retrieving sets of words. As such, the bounds here can be viewed as an early contribution to the theory of data structures. This perspective underlay both my "data graph" model [32] and Schönhage's "storage modification machine" model [37]. The interesting features here are the formulation of an information-retrieval problem as a formal language, and the use of the concepts underlying Theorem 3.1 to analyze the problem.

The online TM model. A *d-dimensional tape* is a linked data structure with an array-like topology, termed an *orthogonal list* in [20]. A tape is accessed via a *read-write head*—the TM-oriented name for a pointer. Each cell of a tape holds one symbol from a finite set Γ that contains a designated "blank" symbol; e.g., in a 32-bit computer, Γ could be the set of 32-bit binary words, and the "blank" symbol could be the word of all 1's. Access to cells within a tape is sequential: one can move the head at most one cell in any of the $2d$ permissible directions in a step.

An **online TM** M with t d-dimensional "work tapes" can be viewed as an FA that has access to t d-dimensional tapes. As with any FA, M has an *input port* via which it scans symbols from its input alphabet Σ; it also has a designated initial state and a designated set of final states.

Let me explain the role of the input port in M's "online" computing, by analogy with FA's. One can view an FA as a device that is passive until a symbol $\sigma \in \Sigma$

is “dropped into” its input port. If the FA is in a stable configuration at that moment—meaning that all bi-stable devices in its circuitry have stabilized—then the FA responds to input σ. The most interesting aspect of this response is that the FA indicates whether the entire sequence of input symbols that it has been presented up to that point—i.e., up to and including the last instance of symbol σ—is accepted. Note that the FA responds to input symbols in an *online* manner, making acceptance/rejection decisions about each prefix of the input string as that prefix has been read. Of course, once the FA has “digested” the last instance of symbol σ, by again reaching a stable configuration, then it is ready to “digest” another input symbol, when and if one is “dropped into” its input port.

The TM M uses its input port in much the manner just described. There is, however, a fundamental difference between an FA and an online TM. During the “passive” periods in which an FA does not accept new input symbols at its input port, the FA is typically waiting for its logic to stabilize, hence is usually not considered to be doing valuable computation. In contrast, during the “passive” periods in which an online TM does not accept new input symbols at its input port, the TM may be doing quite valuable subcomputations using its work tapes. Indeed, the study in [13] can be viewed as bounding (from below) the cumulative time that must be devoted to these “introspective” subcomputations when performing certain computations. With this intuitive background in place, a computational step by M depends on:

- its current state,
- the current input symbol, *if M's program reads the input at this step*,
- the t symbols (elements of Γ) currently scanned by the pointers on the t tapes.

On the basis of these, M:

- enters a new state (which may be the same as the current one),
- independently rewrites the symbols currently scanned on the t tapes (possibly with the same symbol as the current one),
- independently moves the read-write head on each tape at most[1] one square in one of the $2d$ allowable directions.

Notes. **(a)** When $d = 1$, we have a TM with t linear (i.e., one-dimensional) tapes. **(b)** Our tapes have array-like topologies because of the focus in [13]. It is easy to specify tapes with other regular topologies, such as trees of various arities.

[1]The qualifier “at most” indicates that a read-write head is allowed to remain stationary.

One extends M's one-step computation to a multistep computation (whose goal is language recognition, as usual) as follows. To determine if a word $w = \sigma_1\sigma_2 \cdots \sigma_n \in \Sigma^\star$ is accepted by M—i.e., is in the language $L(M)$—one makes w's n symbols available, in sequence, at M's input port. If M starts in its initial state with all cells of all tapes containing the "blank" symbol, and it proceeds through a sequence of N steps that:

- includes n steps during which M "reads" an input symbol,
- ends with a step in which M is programmed to "read" an input symbol,

then M is said to *decide w in N steps*; if, moreover, M's state at step N is an accepting state, then M is said to *accept w in N steps.*

The somewhat complicated double condition for acceptance ("includes ..." and "ends with ...") ensures that, if M accepts w, then it does so unambiguously. Specifically, after reading the last symbol of w, M does not "give its answer" until it prepares to read the next input symbol (if that ever happens). This means that M cannot oscillate between accepting and nonaccepting states after reading the last symbol of w.

The impact of tape structure on memory locality. The *configuration* of an online TM M having t d-dimensional tapes, at any step of a computation, is the $(t+1)$-tuple $\langle q, \tau_1, \tau_2, \ldots, \tau_t \rangle$ defined as follows.

- q is the state of M's finite-state control (its associated FA);
- each τ_i is the d-dimensional configuration of symbols from Γ that comprises the non-"blank" portion of tape i, with one symbol highlighted (in some way) to indicate the current position of M's read-write head on tape i.

(M's configuration is often called its "total state.") The importance of this concept resides in the following. Say that, for $i = 1, 2$, the database-string $x_i \in \Sigma^\star$ leads M to configuration $C_M(x_i) = \langle q_i, \tau_{i1}, \tau_{i2}, \ldots, \tau_{it} \rangle$. If:

- $q_1 = q_2$; i.e., the configurations share the same state;
- for some integer $r \geq 1$, and all $i \in \{1, 2, \ldots, t\}$, tape configurations τ_{1i} and τ_{2i} are identical within r symbols of their highlighted symbols (which indicate where M's read-write heads reside),

then we say that the databases specified by x_1 and x_2 are *r-indistinguishable* by M, denoted $x_1 \equiv_M^{(r)} x_2$. This relation is an important parameterization of the FA-oriented relation $\equiv_M$ that is central to Theorem 3.1. Specifically, by analyzing relation $\equiv_M^{(r)}$, one can sometimes bound the time-complexity of various subcomputations by M, in the following sense.

Lemma 5.1. *Say that $x_1 \equiv_M^{(r)} x_2$. If there exists a $y \in \Sigma^\star$ such that one of x_1y, x_2y belongs to $L(M)$, while the other does not, then, having read either of x_1 or x_2, M must compute for more than r steps while reading y.*

An information-retrieval problem formulated as a language. The following problem is used in [13] to expose the potential effect of tape structure on computational efficiency. We feed an online TM M a set of equal-length binary words, which we term a *database*. We then feed M a sequence of binary words, each of which is termed a *query*. After reading each query, M must respond "YES" if the query word occurs in the database, and "NO" if not.

The *database language* $L_{DB} \subseteq \Sigma^\star$, where, $\Sigma = \{0, 1, :\}$, and ":" is a symbol distinct from "0" and "1," formalizes the preceding problem. Each word in L_{DB} has the form

$$\xi_1 : \xi_2 : \cdots : \xi_m :: \eta_1 : \eta_2 : \cdots : \eta_n$$

where, for some $k \in \mathbb{N}$,

- each ξ_i $(1 \le i \le m)$ and each η_j $(1 \le j \le n)$ is a length-k binary string;
- $m = 2^k$;
- $\eta_n \in \{\xi_1, \xi_2, \ldots, \xi_m\}$.

Both the sequence of ξ_i's and the sequence of η_j's can contain repetitions. In particular, we are interested only in the *set* of words $\{\xi_1, \xi_2, \ldots, \xi_m\}$ (the database). The *database string* "$\xi_1 : \xi_2 : \cdots : \xi_m$" is just the mechanism we use to present the database to M. Each word η_j is a *query*. In each word $x \in L_{DB}$, the double colon "::" separates the database from the queries, while the single colon ":" separates consecutive binary words.

The fact that we are interested only in whether or not *the last* query appears in the database reflects the *online* nature of the computation: M must respond to each query as it appears, with no knowledge of which is the last, hence, the important one. (This is essentially the challenge faced by all online algorithms.)

Tape dimensionality and the time to recognize L_{DB}. For simplicity, we focus henceforth on the sublanguages of L_{DB} that are parameterized by the common lengths of their binary words. For each $k \in \mathbb{N}$, $L_{DB}^{(k)}$ denotes the sublanguage in which each ξ_i and each η_j has length k. Note that each database-string in $L_{DB}^{(k)}$ has length $(k+1)2^k - 1$.

Focus on any fixed $L_{DB}^{(k)}$. If the database-strings x_1 and x_2 specify distinct databases, then there exists a query η that appears in the database specified by one of the x_i but not the other—so, precisely one of $x_1 :: \eta$ and $x_2 :: \eta$ belongs to $L_{DB}^{(k)}$. Database-strings that specify distinct databases must, thus, lead M to distinct configurations.

How "big" must these configurations be? On the one hand, there are $2^{2^k} - 1$ distinct databases (corresponding to each nonempty set of length-k ξ_i's). On the other hand, for any M with t d-dimensional tapes, there is an $\alpha_M > 0$ that depends only on M's structure, such that M has $\leq \alpha_M^{dtr}$ distinct configurations of "radius" r—meaning that all non-"blank" symbols on all tapes reside within r cells of the read-write heads. Thus, in order for each database to get a distinct configuration (so that $\equiv_M^{(r)}$ has $\geq 2^{2^k} - 1$ equivalence classes), the "radius" r must exceed $\beta_M \cdot 2^{k/d}$, for some $\beta_M > 0$ that depends only on M's structure.

Combining this bound with Lemma 5.1, we arrive at the following time bound.

Lemma 5.2. *If $L(M) = L_{DB}^{(k)}$, then, for some length-k query η, M must take*[2] $> \beta_M \cdot (2^{1/d})^k$ *steps while reading η, for some $\beta_M > 0$ that depends only on M's structure.*

The reasoning behind Lemma 5.2 is *information theoretic,* depending only on the fact that the number of databases in $L_{DB}^{(k)}$ is doubly exponential in k, while the number of bounded-"radius" TM configurations is singly exponential. Therefore, no matter how M reorganizes its tape contents while responding to one bad query, there must be a query that is bad for the new configuration! By focusing on strings with 2^k bad queries, we thus obtain:

Theorem 5.1 ([13])**.** *Any online TM M with d-dimensional tapes that recognizes the language L_{DB} must, for infinitely many N, take time $> \beta_M \cdot (N/\log N)^{1+1/d}$ to process inputs of length N, for some constant $\beta_M > 0$ that depends only on M's structure.*

One finds in [13] a companion upper bound of $O(N^{1+1/d})$ for the problem of recognizing L_{DB}. Hence, Theorem 5.1 does expose the potential of nontrivial impact of data-structure topology on computational efficiency. In its time, the theorem also exposed one of the earliest examples of the cost of the *online* requirement. Specifically, L_{DB} can clearly be accepted *in linear time* by a TM M that has just a single, linear work tape, but that operates in an *offline* manner—meaning that M gets to see the entire input string before it must give an answer (so that it knows which query is important before it starts computing).

5.7 Exercises

1. Prove that the following sets are in NP:
   (a) The set of binary representations of composite (i.e., nonprime) integers.
   (b) The set of state-transition tables of DFAs that accept at least one string.
2. *Prove:* The relation "is polynomial-reducible to" ($\leq_{\text{poly}}$) is reflexive and transitive.

[2] We write $2^{k/d}$ in the unusual form $(2^{1/d})^k$ to emphasize that the dimensionality of M's tapes (which is a fixed constant) appears only in the base of the exponential.

3. As usual, call a set A *nontrivial* if $A \notin \{\emptyset, \mathbb{N}\}$. *Prove:* If there is a nontrivial set A that is not NP-hard, then $P \neq NP$.

   Hint: Prove and then use the following fact: If $A \in$ P and B is nontrivial, then $A \leq_{\text{poly}} B$. (This fact is a complexity-theoretic instance of the intuition that a set that needs no "help" is "helped" by any set.)

4. As usual, let *co*-NP denote the class of languages whose complements are in NP. *Prove:* If *co*-NP contains an NP-complete set, then $NP = co\text{-}NP$.

5. *Prove:* For any sets A and B, $[A \leq_{\text{poly}} B]$ iff $[\overline{A} \leq_{\text{poly}} \overline{B}]$.

6. We saw in Theorem 4.3 that the Halting Problem, HP, is m-complete. Using the same intuition that underlies the proof proof of that result, prove that the set

   $$HP^{(\text{poly})} \stackrel{\text{def}}{=} \{\langle x, y, \sigma^n \rangle \mid \text{TM } x \text{ halts on input } y \text{ in} \leq n \text{ steps}\}$$

   is NP-complete.

   In the definition of this set, you may assume that both x and y are strings over the alphabet $\{0, 1\}$ and that σ is special symbol. (The last clause impies that σ^n is a string of n occurrences of the letter σ.)

   Hints: **(a)** Since we know that a multitape TM that operates in time $t(m)$ can be simulated by a one-tape TM in time polynomial in $t(m)$, you may want to start out by placing the three components of an alleged element of $HP^{(\text{poly})}$, namely, x, y, and σ^n, on separate tapes. **(b)** Once having done this, how long does it take the TM that you must construct to simulate a single step of the TM x? When you now take into account that you must simulate TM x on input y for only n steps, you should be able to show that your TM operates in time polynomial in the size of *its* input (not the size of TM x's input).

7. Prove that the language HP is NP-Hard.

8. (a) What is the "size" (in the complexity-theoretic sense of the term) of a 3-CNF formula that has n clauses and m literals?

   (b) What is the size of a graph that has n vertices and m edges? (There are at least two correct answers.)

   Hint. How would you design a TM that could process *any* 3-CNF formula or *any* graph? The only "tough" issue is how to represent "names." You should use the most compact (to within constant factors) fixed-length encoding of "names."

9. *Prove:* For any pair of languages A and B such that neither B nor $\overline{B}$ is empty: If $A \in \mathcal{P}$, then $A \leq_{\text{poly}} B$.

10. Prove the following. *For any pair of languages A and B such that*

    - *$A \in \mathcal{P}$;*
    - *neither B nor $\overline{B}$ is empty.*

    we have $A \leq_{\text{poly}} B$.

11. A language L is accepted by an offline TM that operates in space $S(n) = O(1)$ if, and only if, L is regular.

12. The following space functions are fully tape constructible. (You may describe your construction algorithms via English-language descriptions. Do not consult any book.)

$$\begin{aligned} S(n) &= \lfloor \log_2 n \rfloor \\ S(n) &= n^2 \\ S(n) &= 2^n \end{aligned}$$

13. Let $T(n) \geq n$ be any nondecreasing integer function. Prove that any *off-line* TM M that operates in time $T(n)$ can be simulated by an *on-line* TM M' which has the same work-tape repertoire (number and structure) as M and which operates in time $T'(n) \leq T^2(n)$.

14. Repeat our lower-bound argument for Hennie's database language, for an on-line TM with one *two-dimensional* tape (with move repertoire UP, DOWN, LEFT, RIGHT, NO-MOVE). Since Hennie's Lemma 2 is independent of tape structure, you may cite it with no proof. (Hence, you must prove a version of Hennie's Lemma 1 and do the final timing analysis.)

15. Let L be the following "unary" version of Hennie's database language. Each $w \in L$ has the form

$$w = 0^{n_1}10^{n_2}1 \cdots 10^{n_r}20^{m_1}10^{m_2}1 \cdots 10^{m_s}$$

    where

    - each $m_i, n_i \geq 1$
    - $m_s \in \{n_1, n_2, \cdots, n_r\}$.

    Find (and verify) upper and lower bounds on the time required for an on-line TM with one linear work-tape to accept L. For full credit, your bounds should be only a constant factor apart.

Bibliography

[1] G. Birkhoff and S. MacLane (1953): *A Survey of Modern Algebra*, Macmillan, New York.

[2] G. Cantor (1874): Über eine Eigenschaft des Inbegriffes aller reellen algebraischen Zahlen. *J. für die Reine und Angewandte Mathematik, 77*, 258–262.

[3] G. Cantor (1878): Ein Beitrag zur Begrundung der transfiniter Mengenlehre. *J. Reine Angew. Math. 84*, 242–258.

[4] A.L. Cauchy (1821): *Cours d'analyse de lÉcole Royale Polytechnique, 1ère partie: Analyse algébrique*. l'Imprimerie Royale, Paris. Reprinted: Wissenschaftliche Buchgesellschaft, Darmstadt, 1968.

[5] N. Chomsky (1956): Three models for the description of language. *IRE Trans. Information Theory 2*, 113–124.

[6] N. Chomsky (1959): On certain formal properties of grammars. *Inform. Contr. 2*, 137–167.

[7] S.A. Cook (1971): The complexity of theorem-proving procedures. *ACM Symp. on Theory of Computing (STOC71)*, 151–158.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, C. Stein (2001): *Introduction to Algorithms (2nd ed.)*. MIT Press, Cambridge, MA.

[9] K. Gödel (1931): Über Formal Unentscheidbare Sätze der Principia Mathematica und Verwandter Systeme, I. *Monatshefte für Mathematik u. Physik 38*, 173–198.

[10] O. Goldreich (2006 [expected]): On teaching the basics of complexity theory. In *Essays in Theoretical Computer Science in Memory of Shimon Even* (O. Goldreich, A.L. Rosenberg, A. Selman, eds.) Springer-Verlag, Heidelberg.

[11] D. Harel (1987): *Algorithmics: The Spirit of Computing.* Addison-Wesley, Reading, MA.

[12] J. Hartmanis and R.E. Stearns (1966): *Algebraic Structure Theory of Sequential Machines*. Prentice Hall, Englewood Cliffs, NJ.

[13] F.C. Hennie (1966): On-line Turing machine computations. *IEEE Trans. Electronic Computers, EC-15*, 35–44.

[14] J.E. Hopcroft, R. Motwani, J.D. Ullman (2001): *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison-Wesley, Reading, MA.

[15] J.E. Hopcroft and J.D. Ullman (1979): *Introduction to Automata Theory, Languages, and Computation* (1st ed.) Addison-Wesley, Reading, MA.

[16] J. Jaffe (1978): A necessary and sufficient pumping lemma for regular languages. *SIGACT News*, 48–49.

[17] R.M. Karp (1967): Some bounds on the storage requirements of sequential machines and Turing machines. *J. ACM 14*, 478–489.

[18] R.M. Karp (1972): Reducibility among combinatorial problems. In *Complexity of Computer Computations* (R.E. Miller and J.W. Thatcher, eds.) Plenum Press, NY, pp. 85–103.

[19] D. König (1936): *Theorie der Endlichen und Unendlichen Graphen.* Teubner.

[20] D.E. Knuth (1973): *The Art of Computer Programming: Fundamental Algorithms* (2nd ed.) Addison-Wesley, Reading, MA.

[21] L. Levin (1973): Universal search problems. *Problemy Peredachi Informatsii 9*, 265–266. Translated in, B.A. Trakhtenbrot (1984): A survey of Russian approaches to perebor (brute-force search) algorithms. *Annals of the History of Computing 6*, 384–400.

[22] H.R. Lewis and C.H. Papadimitriou (1981): *Elements of the Theory of Computation.* Prentice-Hall, Englewood Cliffs, NJ.

[23] P. Linz (2001): *An Introduction to Formal Languages and Automata* (3rd ed.) Jones and Bartlett Publ., Sudbury, MA.

[24] G.H. Mealy (1955): A method for synthesizing sequential circuits. *Bell Syst. Tech. J. 34*, 1045–1079.

[25] E.F. Moore (1956): Gendanken experiments on sequential machines. In *Automata Studies* (C.E. Shannon and J. McCarthy, Eds.) *[Ann. Math. Studies 34]*, Princeton Univ. Press, Princeton, NJ, pp. 129–153.

[26] B.M. Moret (1997): *The Theory of Computation.* Addison-Wesley, Reading, MA.

[27] J. Myhill (1957): Finite automata and the representation of events. WADD TR-57-624, Wright Patterson AFB, Ohio, pp. 112–137.

[28] A. Nerode (1958): Linear automaton transformations. *Proc. AMS 9*, 541–544.

[29] M.O. Rabin (1963): Probabilistic automata. *Inform. Control 6*, 230–245.

[30] M.O. Rabin and D. Scott (1959): Finite automata and their decision problems. *IBM J. Res. Develop. 3*, 114–125.

[31] H. Rogers, Jr. (1967): *Theory of Recursive Functions and Effective Computability.* McGraw Hill, New York. Reprinted in 1987 by MIT Press, Cambridge, MA.

[32] A.L. Rosenberg (1971): Data graphs and addressing schemes. *J. CSS 5*, 193–238.

[33] A.L. Rosenberg (2003): Efficient pairing functions—and why you should care. *Intl. J. Foundations of Computer Science 14*, 3–17.

[34] A.L. Rosenberg (2006 [expected]): State. In *Essays in Theoretical Computer Science in Memory of Shimon Even* (O. Goldreich, A.L. Rosenberg, A. Selman, eds.) Springer-Verlag, Heidelberg.

[35] A.L. Rosenberg and L.S. Heath (2001): *Graph Separators, with Applications.* Kluwer Academic/Plenum Publishers, New York.

[36] W. Savitch (1969): Deterministic simulation of non-deterministic Turing machines. *1st ACM Symp. on Theory of Computing*, 247–248.

[37] A. Schönhage (1980): Storage modification machines. *SIAM J. Computing 9*, 490–508.

[38] M. Sipser (1997): *Introduction to the Theory of Computation.* PWS Publishing Co., Boston, MA.

[39] A.M. Turing (1936): On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.* (ser. 2, vol. 42) 230–265; Correction *ibid.* (vol. 43) 544–546.

[40] Wikipedia: The Free Encyclopedia (2005):
`http://en.wikipedia.org/wiki/Pumping_lemma`