CMPSCI 401: Introduction to the Theory of Computation
Spring, 2003

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

April 30, 2003

Abstract

This course presents the elements of the theory of computation, focusing on the theories of *Finite Automata*, *Computability*, and *Complexity*. The Theory of Finite Automata will be developed up to, and including, the two basic theorems that characterize finite automata by, respectively, definitively elucidating the notion of "state" and establishing the "equivalence" of finite automata and *regular expressions*. 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 will be the notion of the *completeness* of certain computational problems and the theorem that, informally, asserts the impossibility of effectively [i.e., algorithmically] determining properties of the dynamic behavior of a program from its static description. Complexity theory will be developed as an outgrowth of Computability theory, built upon (computational) resource-bounded versions of (non)encodability and reducibility. The culmination of this development will be the identification of certain computational problems that are *complete* in a resource-bounded sense, and the attendant **P**-vs.-**NP** problem.

1 Introduction

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 resulting theory are:

Dynamism: The theory models *dynamic* systems—machines and circuits that compute—rather than just static ones such as are encountered in, say, 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.

Applicability: Many aspects of the resulting theory have important applications—either conceptual or computational or both—in quite distinct fields.

1.1 Finite Automata and Context-Free Grammars

Finite Automata. The model of Finite Automaton (FA) amply illustrates these three features. 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 (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, 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. Moore's variant of the FSM model is essentially identical to the FA that we study; Mealy's variant 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*—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. 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 (Rabin and 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 just remark on) 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* (Cocke and Allen) and for *program verifiers* (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, almost without adaptation, to the analysis of the resulting data- and control-flow graphs for programs.

Context-Free Grammars and Languages. Context-free grammars (CFGs) and the languages they generate (CFLs) illustrate our three features in quite a different way than do finite automata. CFG's and CFL's have their origins in the 4-level hierarchy that Chomsky developed in the 1950s, in his attempt to model language acquisition in humans at various levels of sophistication. The most exciting aspect of Chomsky's study, from a mathematical perspective, is that, for each of his four levels, Chomsky was able to present a grammatical structure and an associated family of automata that defined precisely the same family of formal languages. Thus,

1. *Regular* (or, *type-3*) grammars generate precisely the same family of languages—the regular sets—that *finite automata* accept.

2. *Context-free* (or, *type-2*) grammars generate precisely the same family of languages—the CFL's—that *nondeterministic pushdown automata* accept. These automata are essentially FA's augmented with a stack of unbounded capacity. They were one of the first illustrations of the utility and the power of the stack data structure.

3. *Context-sensitive* (or, *type-1*) grammars generate precisely the same family of languages—the context-sensitive languages—that *nondeterministic linear-bounded automata* accept.

4. *Unrestricted* (or, *type-0*) grammars generate precisely the same family of languages—the recursively enumerable sets—that *Turing machines* accept.

CFGs were reinvented in the early 1960s, under the name *Backus Normal Form* (BNF)—later renamed *Backus-Naur Form*—by John Backus, leader of the original FORTRAN project at

IBM. BNF was adopted as the formal mechanism for specifying the (core) syntax of programming languages for the development of *syntax-directed compilers*.

CFGs never completely lost their interest for linguists either, as they were embellished with "transformations," in order to explain the "complete" syntax of natural languages. (I put "transformations" in quotes because I am not telling you what they are; I put "complete" in quotes, because the claim underlying this assertion defies rigorous verification.)

1.2 Computability Theory

Coming attractions ...

1.3 Complexity Theory

Coming attractions ...

2 Finite Automata and Regular Languages

The high points of this section will be two theorems: the Myhill-Nerode Theorem and the Kleene-Myhill Theorem. Both theorems completely characterize the power of Finite Automata (FA, for short)—from very different perspectives. The former 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.

2.1 Preliminaries

Throughout, 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 inherent structure, which is endowed by their meaning.

For any alphabet Σ, we denote by $\Sigma^\star$ the set of all finite strings of elements of Σ—including the *null symbol* ε, which si the unique string of length 0. Note that the set $\{\varepsilon\}$ contains one element—albeit a null one: the set is different from the *null set* $\emptyset$, which contains no elements.

For any letter a and nonnegative integer k, we define a^k to be a string of k a's. In particular, $a^0 = \varepsilon$.

2.2 Finite Automata and their "languages"

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.)
- $\mathcal{Q}_0$ 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 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 $q \in Q$:

$$\widehat{\delta}(q, \varepsilon) = q \qquad \text{(no stimulus} \Rightarrow \text{no response)}$$

$$(\forall \sigma \in \Sigma,\ \forall x \in \Sigma^\star)\ \widehat{\delta}(q, \sigma x) = \widehat{\delta}(\delta(q, \sigma), x) \qquad \text{(the single "}\delta\text{" highlights the extension)}$$

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.

2.3 What FA's cannot do: proofs of non-regularity

The following "Continuation Lemma" is/should be the primary tool for proving that a set is not regular.

Lemma 2.1 *(***The Continuation Lemma***) If $\delta(q_0, x) = \delta(q_0, y)$, for some $x, y \in \Sigma^\star$, then $(\forall z \in \Sigma^\star)\ \delta(q_0, xz) = \delta(q_0, yz)$.*

Proof. By direct calculation, we find that:

$$\begin{aligned} \delta(q_0, xz) &= \delta(\delta(q_0, x), z) \\ &= \delta(\delta(q_0, y), z) \\ &= \delta(q_0, yz). \end{aligned}$$

(Note that we proceed from expression #2 to expression #3 via the universal algebraic operation of substituting equals for equals.) The basic tool we use here is the inductive extension of δ to $Q \times \Sigma^\star$. ■

The way you should think of the Continuation Lemma's message is that an FA M has no "memory" other than its current state. If strings x and y lead M to the same state (from its initial state)—i.e., if $\delta(q_0, x) = \delta(q_0, y)$—then no subsequent input string will ever allow M to determine which of x and y it actually read. An algebra-inspired way to say this is:

> If $\delta(q_0, x) = \delta(q_0, y)$, then strings x and y are *equivalent* to the FA M. We can denote this fact by the notation $x \equiv_M y$. Symbolically,
>
> $$[x \equiv_M y] \quad \text{if, and only if,} \quad [\delta(q_0, x) = \delta(q_0, y)]. \tag{2.1}$$
>
> (You should verify that $\equiv_M$ is always an equivalence relation on the set $\Sigma^\star$—it is reflexive, symmetric, and transitive.)
>
> The Continuation Lemma tells us that, for any FA M, the equivalence relation $\equiv_M$ is *right-invariant:* If $x \equiv_M y$, then, for all $z \in \Sigma^\star$, $xz \equiv_M yz$.

Finding the appropriate strings x and y to "fool" M when it is presented with strings from a nonregular language is largely an art. But the art always builds on the legendary "Pigeonhole Principle" (sometime called "Dirichlet's Box Principle").

> **The Pigeonhole Principle**: *If you put $n + 1$ balls into n bins, some bin must receive more than one ball.*

Applying the Continuation Lemma, 1. The first language we prove to be nonregular is known fondly as "a to the n, b to the n." It is the language $L_1 = \{a^n b^n \mid n \in \mathsf{N}\} \subset \{a, b\}^\star$. We claim that L_1 is not regular.

As with all such proofs, we assume, for contradiction, that L_1 is regular, and is, in fact, $L(M)$ for the FA $M = (Q,\ \{a, b\},\ \delta,\ q_0,\ F)$.

Now, by the Pigeonhole Principle, as we range over the infinite set of strings of the form a^k, for $k = 0, 1, 2, \ldots$, there must exist two distinct nonnegative integers, i and $j > i$, such that $\delta(q_0, a^i) = \delta(q_0, a^j)$. By the Continuation Lemma, then, for all nonnegative integers ℓ, we

must have $\delta(q_0, a^i b^\ell) = \delta(q_0, a^j b^\ell)$. In particular, we must have $\delta(q_0, a^i b^i) = \delta(q_0, a^j b^i)$. But this means that $L(M) \neq L_1$! To wit, either $\delta(q_0, a^i b^i)$ and $\delta(q_0, a^j b^i)$—which are the same state!— both belong to F, or neither does. But $a^i b^i \in L_1$, and $a^i b^j \notin L_1$, so we *should* have $\delta(q_0, a^i b^i) \in F$ but $\delta(q_0, a^j b^i) \notin F$. This is the contradiction we were looking for. Since we have assumed nothing about M other than its being an FA, we conclude that L_1 is not regular. ■

Applying the Continuation Lemma, 2. We turn to a somewhat subtler application of the Continuation Lemma: we prove that the set $L_2 = \{a^k \mid k \text{ is a perfect square}\}$ is not regular. Again, assume that $L_2 = L(M)$ for some FA $M = (Q,\ \{a\},\ \delta,\ q_0,\ F)$.

The subtlety here is that we need to know where to look for our pigeons! The simplest proof—which is what we should always be seeking!—looks for its pigeons in the infinite sequence $a^0, a^1, a^4, a^9, \ldots, a^{k^2}, \ldots$. In this Application, we note that there exist two distinct *positive* integers, i and $j > i$, such that $\delta(q_0, a^{i^2}) = \delta(q_0, a^{j^2})$. By the Continuation Lemma, then, for all nonnegative integers ℓ, we must have

$$\begin{aligned} \delta(q_0, a^{i^2+\ell}) &= \delta(q_0, a^{i^2} a^\ell) \\ &= \delta(q_0, a^{j^2} a^\ell) \\ &= \delta(q_0, a^{j^2+\ell}) \end{aligned}$$

Choosing $\ell = 2i + 1$ closes the trap on L_2, because $i^2 + 2i + 1 = (i+1)^2$ is a perfect square, but $j^2 + 2i + 1$ is *not* a perfect square—because $j^2 < j^2 + 2i + 1 < (j+1)^2$.

As before, we have assumed nothing about M, so we can conclude that L_2 is not regular. ■

2.4 "Pumping" regular sets: a hint at the inherent power of FA's

It is intellectually wasteful to use the famous so-called Pumping Lemma to prove that sets are no regular, but the Lemma does elucidate an important aspect of the structure of FA, hence, of regular sets.

The statement of the so-called Pumping Lemma for regular sets may be hard to remember, but the Lemma's proof is incredibly simple if one thinks of FA's via their labeled digraph formulation.

In fact, the Pumping Lemma embodies a characteristic of arbitrary 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. Statues become states. Traversals become input strings. 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 $q^\star$. Symbolically:

$$\begin{aligned}
\delta(q_0, xy^kz) &= \delta(\delta(q_0, x), y^kz) \\
&= \delta(q, y^kz) \\
&= \delta(q, y^{k-1}z) \\
&\vdots \quad \vdots \\
&= \delta(q, yz) \\
&= \delta(q, z) \\
&= q^\star.
\end{aligned}$$

What we have proved is:

Theorem 2.1 (The Pumping Lemma for Regular Sets)
For every infinite regular set L, there is a constant n such that every string $w \in L$ of length $\geq n$ can be written in the form $w = xyz$, where $|x| \leq n$ and $y \neq \varepsilon$, in such as way that for all $k \in \mathsf{N}$, the string $w = xy^kz \in L$.

Sipser uses the Pumping Lemma to prove that "a to the n, b to the n" is not regular. Note how much more complicated his proof is than our proof using the Continuation Lemma.

2.5 The Myhill-Nerode Theorem: The full story of "state"

We have already mentioned right-invariant equivalence relations on $\Sigma^\star$ when we discussed the Continuation Lemma. We show now that, in some sense, such relations underlie the entire story of FA's. Crucial in this story is the notion of the *index* of an equivalence relation, call it $\equiv$, on $\Sigma^\star$. The *index* of $\equiv$ is the number of *classes* that it partitions $\Sigma^\star$ into. (You can verify that partitions and equivalence relations are just two ways of looking at the same concept. We

call the blocks of the partition induced by an equivalence relation *equivalence classes*, or just *classes* for short.)

Theorem 2.2 (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 following right-invariant equivalence relation, $\equiv_L$, over $\Sigma^\star$ has finitely many equivalence classes (i.e., has finite index. For all $x, y \in \Sigma^\star$:*

$$[x \equiv_L y] \quad \textit{iff} \quad (\forall z \in \Sigma^\star)[[xz \in L] \Leftrightarrow [yz \in L]].$$

Proof. We prove the (logical) equivalence of the Theorem's three statements by verifying three cyclic implications: statement 1 implies statement 2, statement 2 implies statement 3, statement 3 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 $L = L(M)$. Consider the equivalence relation $\equiv_M$ of (2.1).

1. Relation $\equiv_M$ is a right-invariant equivalence relation.
   Its being an equivalence relation follows from the definition of FA; its being right invariant follows from the Continuation Lemma.
2. L is the union of some of the classes of relation $\equiv_M$.
   By definition, L is the set of all strings $x \in \Sigma^\star$ that lead M from its initial state to a final state. Thus,

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

3. Relation $\equiv_M$ has finite index.
   Its index is no larger than $|Q|$; its index is exactly $|Q|$ if every state of M is reachable from the initial state.

(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. $\mathcal{Q}_0 = [\varepsilon]$.
   M's start state corresponds to its having read nothing.
4. $F = \{[x] \mid x \in L\}$

You can verify easily that M is a well-defined FA that accepts L. ■

2.6 Minimizing FA's: Applying our understanding of "state"

The Myhill-Nerode Theorem really tells us two important things. First, the notion of "state" that makes the FA model "work" is embodied in the equivalence relations on $\Sigma^\star$ defined by the reachability of the states of an FA. 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 equivalent with respect to the language the FA accepts. Second, the *coarsest* (i.e., smallest-index) relation that works is $\equiv_L$, so that this relation embodies the *smallest* FA that accepts language L. We can turn this 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 (= any FA that accepts L) and iteratively "coarsifying" the relation as far as we can, thereby sneaking up on the relation $\equiv_L$.

Try to recognize how the preceding rationale underlies the algorithmic strategy in the following paragraph.

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

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

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 the same language as M. The equivalence classes of $\equiv_M$—i.e., $\{[p] \mid p \in Q\}$—are therefore the states of the smallest FA—call it $\widehat{M}$—that accepts the same language as M. The state-transition function $\widehat{\delta}$ of $\widehat{M}$ is given by

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

Finally, the initial state of $\widehat{M}$ is $[q_0]$, and the accepting states are $\{[p] \mid p \in F\}$. You should prove that $\widehat{\delta}$ is a well-defined function and that $\widehat{M}$ is a well-defined FA that accepts the same language as M.

We ease into our computation of $\equiv_M$ by describing an example concurrently with our description of the algorithm. We are going to start with a very coarse approximation to $\equiv_M$ and iteratively improve the approximation. Here is the FA for our example, given in tabular form (to save me from drawing pictures).

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

Our initial partition of Q is $Q - F,\ F$, to indicate that the null string ε witnesses the fact that no accepting state can be equivalent to any nonaccepting state, or vice versa. In our example, this yields the initial partition

$$\{a, b, d, e, f, g, h\}_1,\ \{c\}_1$$

(the subscript "1" indicating that this is the first discriminatory step. State c, being the unique state in F, is not equivalent to any other state.

Inductively, we now look at the current, time t, partition and try to "break apart" time-t blocks. We do this by feeding pairs of states in the same block single input symbols. 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—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

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

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. At this point, input "1" leads: states a and e to block $\{d,f\}$; states b and h to block $\{c\}$; and leaves state g in its present block. We thus end up with the indicated time-2 partition. Further single inputs leave this partition stable, so it must be the finest partition that preserves $L(M)$. We thus end up with the following FA as the minimum-state version of M.

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

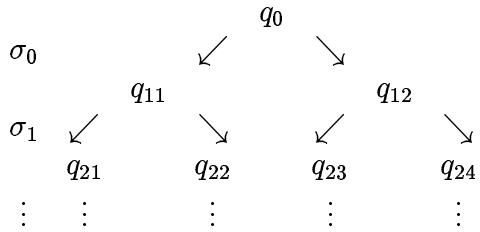
2.7 Nondeterministic FA's and their power

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

$$q_0 \stackrel{\sigma_0}{\rightarrow} q_1 \stackrel{\sigma_1}{\rightarrow} q_2 \stackrel{\sigma_2}{\rightarrow} \cdots \stackrel{\sigma_k}{\rightarrow} q_{k+1} \tag{2.2}$$

in our generalized setting, a "computation" must be viewed as a tree (drawn here as binary only for convenience):



(2.3)

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 (2.2) 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 (2.3), after reading $\sigma_0\sigma_1$, the automaton is in up to four states, $q_{21}, q_{22}, q_{23}, q_{24}$ (which need not 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 *standard nondeterministic finite automaton* (*s-NFA*, for short) is a system $M = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0, and F play the same roles as with a deterministic FA, and where state transitions are *to sets of states.* Letting $\mathcal{P}(Q)$ denote the *power set—i.e., set of all subsets—of* Q, we have

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

There is a natural nondeterministic extension of δ to $Q \times \Sigma^\star$:

$$\begin{aligned} \delta(q, \varepsilon) &= \{q\} \\ \delta(q, \sigma x) &= \bigcup_{p \in \delta(q,\sigma)} \delta(p, x) \quad [\sigma \in \Sigma] \text{ and } [x \in \Sigma^\star] \end{aligned}$$

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 s-NFA and its language and the 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 [Chomsky]; Context-Sensitive Languages are precisely the languages accepted by

nondeterministic linear-bounded automata [Chomsky]; 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 discovered by Cook.

In the theory of finite automata and regular languages, (standard) nondeterminism is used as a convenience 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 standard nondeterminism is no more than a convenience, by showing that s-NFA's accept only regular languages.

Theorem 2.3 *Every language accepted by an s-NFA $M = (Q, \Sigma, \delta, q_0, F)$ is regular.*

Proof. Our proof relies on the following intuition, which is discernible in an s-NFA's acceptance criterion. Say that the s-NFA M has thus far read the string $x \in \Sigma^\star$. 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 $\mathcal{P}(Q)$ of all subsets of Q. Therefore, we can actually construct a DFA $M' = (Q', \Sigma, \delta', q_0', F')$ to simulate *all* computations of M, 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 construction achieves this.

- $Q' = \mathcal{P}(Q)$. 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 state 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 Sipser. ■

Since s-NFA's accept only regular languages, standard nondeterminism is, 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. We do this by enhancing s-NFA's to allow them to have so-called *ε-transitions*.

A *nondeterministic finite automaton* (*NFA*, for short) $M = (Q, \Sigma, \delta, q_0, F)$ is an s-NFA whose state-transition function δ is extended to allow *spontaneous* so-called ε-transitions:

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

Allowing a transition on "input" ε, which is really the absence of an input, will simplify some of our constructions, yet such transitions do not augment the power of NFA's, as we see now.

Theorem 2.4 *Every language accepted by an NFA $M = (Q, \Sigma, \delta, q_0, F)$ is accepted by an s-NFA, hence is regular.*

Proof. For each state $q \in Q$, define

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

The s-NFA that is equivalent to M is $M'' = (Q, \Sigma, \delta'', q_0, F)$, where, for all $q \in Q$ and $\sigma \in \Sigma$,

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

Each state-transition of M'' thus traces through, and collapses, all of M's ε-transitions. ■

We can, therefore, wander freely within the world of DFA's, NFA's, and s-NFA's when studying regular languages. We make use of this freedom in the next section.

2.8 Regular Expressions and 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:

$$L_1 \cup L_2 \stackrel{\text{def}}{=} \{x \mid [x \in L_1] \text{ or } [x \in L_2]\}.$$

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

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

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

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

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 $\emptyset^\star = \{\varepsilon\}$.

Regular Expressions. I am using the same notation here as in Sipser's book, although I think that this notation can lead to confusion—so be careful! 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!*

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 $\mathcal{R}_1, \mathcal{R}_2$:	$(\mathcal{R}_1 \cup \mathcal{R}_2)$	$\mathcal{L}(\mathcal{R}_1) \cup \mathcal{L}(\mathcal{R}_2)$
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 2.5 (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 sets over an alphabet Σ is the smallest family that contains all subsets of Σ and that is closed under a finite number of applications of the operations of union, concatenation, and star-closure.

Proof. We prove the theorem via the following two lemmas.

Lemma 2.2 *If the language $L \subseteq \Sigma^\star$ is denoted by a Regular Expression R, then L is a regular language.*

Proof. See Sipser for a complete proof, including pictures. We present only schematic intuitive arguments.

Union. Let us be given NFA's M_1 and M_2. We build an NFA M that accepts $L(M_1) \cup L(M_2)$ as follows. We take the state diagrams of M_1 and M_2, and we "defrock" their start states: these states still exist, but they are no longer start states. We then add a new, nonaccepting, start state whose only transitions are ε-transitions to what used to be the start states of M_1 and M_2. Clearly, M accepts a string opnly if either M_1 or M_2 did.

Concatenation. Let us be given NFA's M_1 and M_2. We build an NFA M that accepts $L(M_1) \cdot L(M_2)$ as follows. We take the state diagrams of M_1 and M_2, and we "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'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. The net effect is that whenever M 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 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 will find it, via the inserted ε-transition. Conversely, if that ε-transition leads M to an accepting state, then the successful input must admit the desired parse.

Star-Closure. Let us be given an NFA M. We build an NFA M' that accepts $(L(M))^\star$. The only delicate issue here is that we must take care that M' accepts ε, as well as all powers of $L(M)$. Taking care of this delicacy first, we give M' a new start state that is also an accepting state: M's start state stays around, but is "defrocked." Then, we

add ε-transitions to M's (now "defrocked") start state from: the new start state and from all of M's accepting states (which remain accepting states in M'). What we have accomplished is the following. If M' reads an input x that would accepted by M, it hedges its bets. On the one hand, it keeps reading x seeking continuations of x that are also in $L(M)$; on the other hand, it assumes that the continuation of x is an independent string in $(L(M))^\star$, so it starts over with the start state of M.

The preceding material is the intuition underlying Sipser's complete proof. ■

Lemma 2.3 *If the language $L \subseteq \Sigma^\star$ is regular, then L is denoted by a Regular Expression R_L.*

Proof. You can see one algorithm that proves this lemma in Sipser. We present here a famous dynamic programming algorithm for constructing R_L that is attributed to Floyd and Warshall and that one can find, for instance, in the CMPSCI 311 text by Cormen, Leiserson, Rivest, and Stein (in the section on transitive closure of graphs. You should compare the two algorithms and use the one that appeals more to you. But you should make certain that you understand both!

Let $L = L(M)$ for the deterministic FA $M = (Q, \Sigma, \delta, q_0, F)$. 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.)

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

$$L_{ij}^{(0)} \;=\; \begin{cases} \{\sigma \mid \delta(s_i, \sigma) \;=\; s_j\} & \text{if } \; i \neq j \\ \{\varepsilon\} \cup \{\sigma \mid \delta(s_i, \sigma) \;=\; s_j\} & \text{if } \; i = j \\ \emptyset & \text{if } \; i \neq j, \;\; \text{and there is no } \sigma \text{ such that } \; \delta(s_i, \sigma) \;=\; s_j \end{cases}$$

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_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 $(L_{kk}^{(k-1)})^\star$—*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:

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

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 Computability Theory

This section is incredibly rich intellectually. It deals with the very underpinnings of the mathematical theory of computation that was initiated by Turing [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] and others with kindred concerns. We shall 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. We then introduce the *Turing Machine* (*TM*, for short) as the "standard" model for algorithm/digital computer. Although this model is unrealistically primitive, it will be convenient to have one single model to fall back upon at various moments in the development. When you think about the process of computing, you should keep the TM in mind but derive your intuition via your own favorite programming language—anything from APL to BASIC to C to .... The mathematical underpinnings of the next subsection should convince you that the specific language you choose is irrelevant: any language you think of can be "encoded" as any another.

3.1 Countability and Uncountability

The theory of Countability (and Uncountability) began with Cantor's deliberations [G. Cantor (1878): Ein Beitrag zur Begrundung der transfiniter Mengenlehre. *J. Reine Angew. Math. 84*, 242–258] on the nature of infinity. 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

$$|A| \ \leq \ |B|$$

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

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

The hallmark of such an injection 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

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

we write

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

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

We single out the important case when $B = \mathsf{N}$, the set of nonnegative integers. When

$$|A| \leq |\mathsf{N}|$$

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

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

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

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

Theorem 3.1 (The Schroeder-Bernstein Theorem) *If* $|A| = |B|$, *then there is a* bijection*—i.e., one-to-one, onto function*

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

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

3.1.1 Pairing functions: proofs of countability

Theorem 3.2 *The following sets are countable.*

1. $\Sigma^\star$, *for any finite set* Σ.
2. *The set of all* finite *subsets of* N.
3. $\mathsf{N}^\star$. *Note that this includes all sets* $\mathsf{N} \times \mathsf{N} \times \mathsf{N} \times \cdots \times \mathsf{N}$, *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 3.1 *The set of computable functions is countable.*

The easiest proof of this result is to intepret 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

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

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

$$f_\Sigma(\sigma_{i_1}\sigma_{i_2}\cdots\sigma_{i_m}) \stackrel{\text{def}}{=} \sum_{j=1}^{m} |\sigma_j|(n+1)^{j-1}.$$

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 N. We reduce the problem of establishing the countability of the set of finite subsets of N to the preceding problem via the following notion.

The *characteristic vector* $\beta(S)$ of a finite set $S \subset \mathsf{N}$ 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

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

so that

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

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

$\mathsf{N}^\star$. 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. (As with other mathematical tools, we do not prove this result here.)

Theorem 3.3 *(***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 3.4 *(***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 $h : \mathsf{N}^\star \longrightarrow \mathsf{N}$, thereby proving the countability of $\mathsf{N}^\star$. 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 $\mathsf{N} \times \mathsf{N} \times \cdots \times \mathsf{N}$. For instance, h restricted to the set of ordered pairs $\mathsf{N} \times \mathsf{N}$ becomes the injection

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

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

$$f : \mathsf{N} \times \mathsf{N} \times \cdots \times \mathsf{N} \;\stackrel{1-1,onto}{\longrightarrow}\; \mathsf{N}$$

In these special cases, the advertised bijections actually have simple forms. We now present one such bijection for $\mathsf{N} \times \mathsf{N}$, just to indicate its charming form. This bijection is attributed to Cantor, but there is evidence that it was known already to Cauchy [A.L. Cauchy (1821): *Cours d'analyse de lÉcole Royale Polytechnique, 1ère partie: Analyse algébrique.* l'Imprimerie Royale, Paris. Reprinted: Wissenschaftliche Buchgesellschaft, Darmstadt, 1968.]

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

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

If we illustrate $\mathsf{N} \times \mathsf{N}$ 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.

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

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

- Along each "diagonal" of $\mathsf{N} \times \mathsf{N}$—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 N to 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.

3.1.2 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 3.5 *The following sets are not countable (are uncountable):*

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

Assuming that we can establish the uncountability of the set of functions $f : \mathsf{N} \longrightarrow \{0, 1\}$, we can immediately infer the uncountability of the set of functions $f : \mathsf{N} \longrightarrow \mathsf{N}$, for the former set can be mapped into the latter via the identity injection. (You should carefully verify this consequence of the transitivity of "$\leq$".) 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 $f : \mathsf{N} \longrightarrow \{0, 1\}$ 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 $f : \mathsf{N} \longrightarrow \{0, 1\}$ 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 $|\mathcal{B}| \leq |\mathsf{N}|$.

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

$$h : \mathcal{B} \stackrel{1-1,onto}{\longrightarrow} \mathsf{N}.$$

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 this course, the most important consequence of our work thus far in this section is the following

Corollary 3.2 *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.*

3.2 Representing computational problems as "languages"

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

$$(\forall x \in \mathsf{N}): \ \kappa_A(x) \ = \ \begin{cases} 1 & \text{if } \ x \in A \\ 0 & \text{if } \ x \notin A \end{cases}$$

Three (sort of) interchangeable notions:

The *set* A (of integers or strings)
The *computational problem*: to compute the characteristic function of the set A
The *system property*: to decide, given $x \in \mathsf{N}$, is $x \in A$?

The terminological corollaries of the interchange:

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.*

3.3 Functions and partial functions

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 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 $f : U \to U$ or about functions $g : U \to \{0,1\}$; 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 $\lfloor n/2 \rfloor$.)

3.4 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.

3.5 The unsolvability of the Halting Problem

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

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

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:

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

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

Theorem 3.6 (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 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. ■

3.6 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 $A \leq_{\mathrm{m}} B$, 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 recognize/decide languages.

Lemma 3.1 *Let A and B be languages over the alphabet Σ, and say that $A \leq_{\mathrm{m}} B$.*

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

(b) *Contrapositively, if language A is* not *recognizable (resp.,* not *decidable), then language B is* not *recognizable (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 recognizable, 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 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 recognizer for language A; and, the composite program φ-then-p' (if p' exists) is a decider for language A. ■

3.7 The Rice-Myhill-Shapiro Theorem

The theorem we are abpout 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 set (of programs) A is a *Property of Functions* (*PoF*, for short) if the following is true. If the programs x and y compute the same function, then either both x and y belong to A or neither does. 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.
- The property "empty function" is embodied in the set of all programs that never halt 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.
- 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 PofF 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 3.7 (The Rice-Myhill-Shapiro Theorem) *Every nontrivial PoF is undecidable. In other words: If a set A is a 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 recognizable.*

Proof. Let us concentrate 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 sets as other sets.

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

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

If you are comfortable with the development thus far, 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

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

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

Note that our program-producing preprocessor *always halts* on an input x; therefore, it computes a *total computable* function, $\mathcal{F} : \{0,1\}^\star \to \{0,1\}^\star$. You should now be able to verify that

$$F_{\mathcal{F}(x)}(z) \;=\; F_{\mathsf{Simulate.3}}(z) \;=\; \begin{cases} F_{y_0}(z) & \text{if } x \in DHP \\ F_e(z) & \text{if } x \notin DHP \end{cases} \tag{3.4}$$

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 $e \in A$. 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 (3.4) that

$$[x \in DHP] \;\Leftrightarrow\; [\mathcal{F}(x) \in A]. \tag{3.5}$$

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 $x \in DHP$, 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 (3.5).

What have we shown here? Looking at (3.5) 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 3.1, 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 3.1 tells us that the set A is not recognizable! ■

3.8 The "Completeness" of the DHP

4 Complexity Theory

4.1 Complexity theory as resource-bounded computability theory

4.2 Nondeterminism as unbounded search

4.3 Cook's Theorem

5 Context-Free Grammars and Languages

This section will be quite sketchy, since it goes in a direction—pure language theory—that is orthogonal to the thrust of the course.

5.1 Context-free grammars and their languages

5.2 "Pumping" CFL's: proofs of non-context-freeness

5.3 Miscellany: ambiguity and unsolvability