Introduction to the Theory of Computing

Lecture notes for CS 360

John Watrous
School of Computer Science and Institute for Quantum Computing
University of Waterloo

June 27, 2017

List of Lectures

Lecture 1

Course overview and mathematical foundations

1.1 Course overview

This course is about the *theory of computation*, which deals with mathematical properties of abstract models of computation and the problems they solve. An important idea to keep in mind as we begin the course is this:

> *Computational problems, devices, and processes can themselves be viewed as mathematical objects.*

We can, for example, think about each program written in a particular programming language as a single element in the set of all programs written in that language, and we can investigate not only those programs that might be interesting to us, but also properties that must hold for all programs. We can also consider problems that some computational models can solve and that others cannot.

The notion of a *computation* is very general. Examples of things that can be viewed or described as computations include the following.

- Computers running programs (of course).
- Networks of computers running protocols.
- People performing calculations with a pencil and paper.
- Proofs of theorems (in a sense to be discussed from time to time throughout this course).
- Certain biological processes.

One could debate the definition of a computation (which is not something we will do), but a reasonable starting point for a definition is that a computation is a manipulation of symbols according to a fixed set of rules.

One interesting connection between computation and mathematics, which is particularly important from the viewpoint of this course, is that *mathematical proofs* and *computations* performed by the models we will discuss throughout this course have a similarity at the lowest level: they both involve symbolic manipulations according to fixed sets of rules. Indeed, fundamental questions about proofs and mathematical logic have played a critical role in the development of theoretical computer science.

We will begin the course working with very simple models of computation (finite automata, regular expressions, context-free grammars, and related models), and later on we will discuss more powerful computational models (especially the Turing machine model). Before we get to any of these models, however, it is appropriate that we discuss some of the mathematical foundations and definitions upon which our discussions will be based.

1.2 Sets and countability

It is assumed throughout these notes that you are familiar with naive set theory and basic propositional logic.

Naive set theory treats the concept of a set to be self-evident, and this will not be problematic for the purposes of this course—but it does lead to problems and paradoxes, such as Russell's paradox, when it is pushed to its limits. Here is one formulation of Russell's paradox, in case you are interested:

> **Russell's paradox.** Let S be the set of all sets that are not elements of themselves:
>
> $$S = \{T : T \notin T\}.$$
>
> Is it the case that S is an element of itself?
>
> If $S \in S$, then we see by the condition that a set must satisfy to be included in S that it must be that $S \notin S$. On the other hand, if $S \notin S$, then the definition of S says that S is to be included in S. It therefore holds that $S \in S$ if and only if $S \notin S$, which is a contradiction.

If you want to avoid this sort of paradox, you need to replace naive set theory with *axiomatic set theory*, which is quite a bit more formal and disallows objects such as *the set of all sets* (which is what opens the door to let in Russell's paradox).

Set theory is the foundation on which mathematics is built, so axiomatic set theory is the better choice for making this foundation sturdy. Moreover, if you really wanted to reduce mathematical proofs to a symbolic form that a computer can handle, along the lines of what was mentioned above, something along the lines of axiomatic set theory is needed.

On the other hand, axiomatic set theory is quite a bit more complicated than naive set theory and well outside the scope of this course, and there is no point in this course where the advantages of axiomatic set theory over naive set theory will appear explicitly. So, we are safe in thinking about set theory from the naive point of view—and meanwhile we can trust that everything would work out the same way if axiomatic set theory had been used instead.

The *size* of a *finite* set is the number of elements if contains. If A is a finite set, then we write $|A|$ to denote this number. For example, the empty set is denoted $\varnothing$ and has no elements, so $|\varnothing| = 0$. A couple of simple examples are

$$|\{a, b, c\}| = 3 \quad \text{and} \quad |\{1, \ldots, n\}| = n. \tag{1.1}$$

In the second example, we are assuming n is a positive integer, and $\{1, \ldots, n\}$ is the set containing the positive integers from 1 to n.

Sets can also be *infinite*. For example, the set of *natural numbers*[1]

$$\mathbb{N} = \{0, 1, 2, \ldots\} \tag{1.2}$$

is infinite, as are the sets of *integers*

$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}, \tag{1.3}$$

rational numbers

$$\mathbb{Q} = \left\{\frac{n}{m} \,:\, n, m \in \mathbb{Z},\ m \neq 0\right\}, \tag{1.4}$$

as well as the *real* and *complex numbers* (which we won't define here because these sets don't really concern us and the definitions are a bit more complicated than one might initially expect).

While it is sometimes sufficient to say that a set is infinite, we will require a more refined notion, which is that of a set being *countable* or *uncountable*.

Definition 1.1. A set A is *countable* if either (i) A is empty, or (ii) there exists an onto (or surjective) function of the form $f : \mathbb{N} \to A$. If a set is not countable, then we say that it is *uncountable*.

[1] Some people choose not to include 0 in the set of natural numbers, but for this course we will include 0 as a natural number. It is not right or wrong to make such a choice, it is only a definition. What is important is that the meaning is clear, and now that we're clear on the meaning we can move on.

These three statements are equivalent for any choice of a set A:

1. A is countable.
2. There exists a one-to-one (or injective) function of the form $g : A \to \mathbb{N}$.
3. Either A is finite or there exists a one-to-one and onto (or bijective) function of the form $h : \mathbb{N} \to A$.

It is not obvious that these three statements are actually equivalent, but it can be proved. We will, however, not discuss the proof.

Example 1.2. The set of natural numbers $\mathbb{N}$ is countable. Of course this is not surprising, but it is sometimes nice to start out with an extremely simple example. The fact that $\mathbb{N}$ is countable follows from the fact that we may take $f : \mathbb{N} \to \mathbb{N}$ to be the identity function, meaning $f(n) = n$ for all $n \in \mathbb{N}$, in Definition 1.1. We may also notice that by substituting f for the function g in statement 2 makes that statement true, and likewise for statement 3 when f is substituted for the function h.

Example 1.3. The set $\mathbb{Z}$ of integers is countable. To prove that this is so, it suffices to show that there exists an onto function of the form

$$f : \mathbb{N} \to \mathbb{Z}. \tag{1.5}$$

There are many possible choices of f that work—one good choice of a function is this one:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \frac{n+1}{2} & \text{if } n \text{ is odd} \\ -\frac{n}{2} & \text{if } n \text{ is even.} \end{cases} \tag{1.6}$$

Thus, we have

$$f(0) = 0, \quad f(1) = 1, \quad f(2) = -1, \quad f(3) = 2, \quad f(4) = -2, \tag{1.7}$$

and so on. This is a well-defined function[2] of the correct form $f : \mathbb{N} \to \mathbb{Z}$, and it is onto; for every integer m, there is a natural number $n \in \mathbb{N}$ so that $f(n) = m$, as is quite evident from the pattern in (1.7).

Example 1.4. The set $\mathbb{Q}$ of rational numbers is countable, which we can prove by defining an onto function taking the form $f : \mathbb{N} \to \mathbb{Q}$. Again, there are many choices of functions that would work, and we'll pick just one.

[2] We can think of *well-defined* as meaning that there are no "undefined" values, and moreover that every reasonable person that understands the definition would agree on the values the function takes, irrespective of when and where they lived.

First, imagine that we create a sequence of ordered lists of numbers, starting like this:

List 0:	0
List 1:	$-1, 1$
List 2:	$-2, -\frac{1}{2}, \frac{1}{2}, 2$
List 3:	$-3, -\frac{3}{2}, -\frac{2}{3}, -\frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{3}{2}, 3$
List 4:	$-4, -\frac{4}{3}, -\frac{3}{4}, -\frac{1}{4}, \frac{1}{4}, \frac{3}{4}, \frac{4}{3}, 4$
List 5:	$-5, -\frac{5}{2}, -\frac{5}{3}, -\frac{5}{4}, -\frac{4}{5}, -\frac{3}{5}, -\frac{2}{5}, -\frac{1}{5}, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, \frac{5}{4}, \frac{5}{3}, \frac{5}{2}, 5$

and so on. In general, for $n \geq 1$ we let the n-th list be the sorted list of all numbers that can be written as

$$\frac{k}{m}, \tag{1.8}$$

where $k, m \in \{-n, \ldots, n\}$, $m \neq 0$, and the value of the number k/m does not already appear in one of the previous lists. The lists get longer and longer, but for every natural number n it is surely the case that the corresponding list is finite.

Now consider the single list obtained by concatenating all of the lists together, starting with List 0, then List 1, and so on. Because the lists are finite, we have no problem defining the concatenation of all of them, and every number that appears in any one of the lists above will also appear in the single concatenated list. For instance, this single list begins as follows:

$$0, -1, 1, -2, -\frac{1}{2}, \frac{1}{2}, 2, -3, -\frac{3}{2}, -\frac{2}{3}, -\frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{3}{2}, 3, -4, -\frac{4}{3}, -\frac{3}{4}, \ldots \tag{1.9}$$

and naturally the complete list is infinitely long. Finally, let $f : \mathbb{N} \rightarrow \mathbb{Q}$ be the function we obtain by setting $f(n)$ to be the number in position n in the infinitely long list we just defined, starting with position 0. For example, we have $f(0) = 0$, $f(1) = -1$, and $f(8) = -3/2$.

Even though we didn't write down an explicit formula for the function f, it is a well-defined function of the proper form $f : \mathbb{N} \rightarrow \mathbb{Q}$. Moreover, it is an onto function: for any rational number you choose, you will eventually find that rational number in the list constructed above. It therefore holds that $\mathbb{Q}$ is countable. The function f also happens to be one-to-one, although we don't need to know this to conclude that $\mathbb{Q}$ is countable.

It is natural at this point to ask a question: Is every set countable? The answer is "no," and we will now see an example of an uncountable set. We will need the following definition.

Definition 1.5. For any set A, the *power set* of A is the set $\mathcal{P}(A)$ containing all subsets of A:

$$\mathcal{P}(A) = \{B : B \subseteq A\}. \tag{1.10}$$

For example, the power set of $\{1,2,3\}$ is

$$\mathcal{P}(\{1,2,3\}) = \big\{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\big\}. \tag{1.11}$$

Notice in particular that the empty set $\varnothing$ and the set $\{1,2,3\}$ itself are contained in the power set $\mathcal{P}(\{1,2,3\})$. For any finite set A, the power set $\mathcal{P}(A)$ always contains $2^{|A|}$ elements, which is one of the reasons why it is called the power set.

Also notice that there is nothing that prevents us from taking the power set of an infinite set. For instance, $\mathcal{P}(\mathbb{N})$, the power set of the natural numbers, is the set containing all subsets of $\mathbb{N}$. This set, in fact, is our first example of an uncountable set.

Theorem 1.6 (Cantor). *The power set of the natural numbers, $\mathcal{P}(\mathbb{N})$, is uncountable.*

Proof. Assume toward contradiction that $\mathcal{P}(\mathbb{N})$ is countable, which implies that there exists an onto function of the form $f : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N})$. From this function we may define[3] a subset of natural numbers as follows:

$$S = \{n \in \mathbb{N} : n \notin f(n)\}. \tag{1.12}$$

Because S is a subset of $\mathbb{N}$, it is the case that $S \in \mathcal{P}(\mathbb{N})$. We have assumed that f is onto, so there must therefore exist a natural number $m \in \mathbb{N}$ such that $f(m) = S$. Fix such a choice of m for the remainder of the proof.

Now we may ask ourselves a question: Is m contained in S? We have

$$[m \in S] \Leftrightarrow [m \in f(m)] \tag{1.13}$$

because $S = f(m)$. On the other hand, by the definition of the set S we have

$$[m \in S] \Leftrightarrow [m \notin f(m)]. \tag{1.14}$$

It therefore holds that

$$[m \in f(m)] \Leftrightarrow [m \notin f(m)], \tag{1.15}$$

or, equivalently,

$$[m \in S] \Leftrightarrow [m \notin S], \tag{1.16}$$

which is a contradiction.

Having obtained a contradiction, we conclude that our assumption that $\mathcal{P}(\mathbb{N})$ is countable was wrong, so the theorem is proved. □

[3] This definition makes sense because, for each $n \in \mathbb{N}$, $f(n)$ is an element of $\mathcal{P}(\mathbb{N})$, which means it is a subset of $\mathbb{N}$. It therefore holds that either $n \in f(n)$ or $n \notin f(n)$, so if you knew what the function f was, you could determine whether or not a given number n is contained in S or not.

The method used in the proof above is called *diagonalization*, for reasons we will discuss later in the course. This is a fundamentally important proof technique in the theory of computation. Using this technique, one can prove that the sets $\mathbb{R}$ and $\mathbb{C}$ of real and complex numbers are uncountable—the central idea of the proof is the same, but the fact that some real numbers have multiple decimal representations (or, for any other choice of a base b, that some real numbers have multiple base b representations) gives a slight complication to the proof.

1.3 Alphabets, strings, and languages

The last thing we will do for this lecture is to introduce some terminology that you may already be familiar with from other courses.

First let us define what we mean by an *alphabet*. Intuitively speaking, when we refer to an alphabet, we mean a collection of symbols that could be used for writing or performing calculations. Mathematically speaking, there is not much to say—there is nothing to be gained by defining what is meant by the words *symbol*, *writing*, or *calculation* in this context, so instead we keep things as simple as possible and stick to the mathematical essence of the concept.

Definition 1.7. An *alphabet* is a finite and nonempty set.

Typical names for alphabets in this course are capital Greek letters such as Σ, Γ, and Δ. We typically refer to elements of alphabets as *symbols*, and we will often use lower-case Greek letters, such as σ and τ, as variable names when referring to symbols. Our favorite alphabet throughout this course will be the *binary alphabet* $\Sigma = \{0, 1\}$.

Next we have *strings*, which are defined with respect to a particular alphabet as follows.

Definition 1.8. Let Σ be an alphabet. A *string* over the alphabet Σ is a finite, ordered sequence of symbols from Σ. The *length* of a string is the total number of symbols in the sequence.

For example, 11010 is a string of length 5 over the binary alphabet $\Sigma = \{0, 1\}$. It is also a string over the alphabet $\Gamma = \{0, 1, 2\}$ that doesn't happen to include the symbol 2. On the other hand,

$$0101010101 \cdots \quad \text{(repeating forever)} \tag{1.17}$$

is not a string because it is not finite. There are situations where it is interesting or useful to consider infinitely long sequences of symbols like this, and on a couple of occasions in this course we will encounter such sequences—but we won't call them strings.

There is a special string, called the *empty string* and denoted ε, that has no symbols in it (and therefore it has length 0). It is a string over every alphabet.

We will typically use lower-case Roman letters at the end of the alphabet, such as u, v, w, x, y, and z, as names that refer to strings. Saying that these are *names that refer to strings* is just meant to clarify that we're not thinking about u, v, w, x, y, and z as being single symbols from the Roman alphabet in this context. Because we're essentially using symbols and strings to communicate ideas about symbols and strings, there is hypothetically a chance for confusion, but once we establish some simple conventions this will not be an issue. If w is a string, we denote the length of w as $|w|$.

Finally, the term *language* refers to any collection of strings over some alphabet.

Definition 1.9. Let Σ be an alphabet. A *language* over Σ is a set of strings, with each string being a string over the alphabet Σ.

Notice that there has to be an alphabet associated with a language—we would not, for instance, consider a set of strings that included infinitely many different symbols appearing among all of the strings to be a language.

A simple but nevertheless important example of a language over a given alphabet Σ is the set of *all* strings over Σ. We denote this language as Σ^*. Another simple and important example of a language is the *empty language*, which is the set containing no strings at all. The empty language is denoted $\varnothing$ because it is the same thing as the empty set—there is no point in introducing any new notation here because we already have a notation for the empty set. The empty language is a language over an arbitrary choice of an alphabet.

In this course we will typically use capital Roman letters near the beginning of the alphabet, such as A, B, C, and D, to refer to languages.

We will see many other examples of laguages throughout the course. Here are a few examples involving the binary alphabet $\Sigma = \{0,1\}$:

$$A = \bigl\{0010, 110110, 011000010110, 111110000110100010010\bigr\}. \tag{1.18}$$

$$B = \bigl\{x \in \Sigma^* \,:\, x \text{ starts with 0 and ends with 1}\bigr\}. \tag{1.19}$$

$$C = \bigl\{x \in \Sigma^* \,:\, x \text{ is a binary representation of a prime number}\bigr\}. \tag{1.20}$$

$$D = \bigl\{x \in \Sigma^* \,:\, |x| \text{ and } |x| + 2 \text{ are prime numbers}\bigr\}. \tag{1.21}$$

The language A is finite, B and C are not finite (they both have infinitely many strings), and at this point in time nobody knows if D is finite or infinite (because the so-called *twin primes conjecture* remains unproved).

Lecture 2

Countability for languages and deterministic finite automata

The main goal of this lecture is to introduce the finite automata model, but first we will finish our introductory discussion of alphabets, strings, and languages by connecting them with the notion of countability.

2.1 Countability and languages

We discussed a few examples of languages last time, and considered whether or not those languages were finite or infinite. Now let us think about countability for languages.

Languages are countable

We will begin with the following proposition.[1]

Proposition 2.1. *For every alphabet Σ, the language Σ^* is countable.*

Let us focus on how this proposition may be proved just for the binary alphabet $\Sigma = \{0, 1\}$ for simplicity—the argument is easily generalized to any other alphabet. To prove that Σ^* is countable, it suffices to define an onto function

$$f : \mathbb{N} \rightarrow \Sigma^*. \tag{2.1}$$

[1] In mathematics, names including *proposition*, *theorem*, *corollary*, and *lemma* refer to facts, and which name you use depends on the nature of the fact. Informally speaking, *theorems* are important facts that we're proud of and *propositions* are also important facts, but we're embarrassed to call them theorems because they were too easy to prove. *Corollaries* are facts that follow easily from theorems, and *lemmas* (or *lemmata* for Latin purists) are boring technical facts that nobody would care about except for the fact that they are useful for proving certain theorems.

In fact, we can easily obtain a one-to-one and onto function f of this form by considering the *lexicographic ordering* of strings. This is what you get by ordering strings by their length, and using the "dictionary" ordering among strings of equal length. The lexicographic ordering of Σ^* begins like this:

$$\varepsilon,\ 0,\ 1,\ 00,\ 01,\ 10,\ 11,\ 000,\ 001,\ \ldots \tag{2.2}$$

From the lexicographic order we can define a function f of the form (2.1) by setting $f(n)$ to be the n-th string in the lexicographic ordering of Σ^* starting from 0. Thus, we have

$$f(0) = \varepsilon,\ f(1) = 0,\ f(2) = 1,\ f(3) = 00,\ f(4) = 01, \tag{2.3}$$

and so on. An explicit method for finding $f(n)$ is to write $n+1$ in binary and throw away the leading 1.

It is not hard to see that the function f we've just defined is an onto function—every binary string appears as an output value of the function f. It therefore follows that Σ^* is countable. It is also the case that f is a one-to-one function.

It is easy to generalize this argument to any other alphabet. This first thing we need to do is to decide on an ordering of the alphabet symbols themselves. For the binary alphabet we order the symbols in the way we were trained: first 0, then 1. If we started with a different alphabet, such as $\Gamma = \{\heartsuit, \diamondsuit, \clubsuit, \spadesuit\}$, it might not be clear how to order the symbols, but it doesn't matter as long as we pick a single ordering and stay consistent with it. Once we've ordered the symbols in a given alphabet Γ, the lexicographic ordering of the language Γ^* is defined in a similar way to what we did above, using the ordering of the alphabet symbols to determine what is meant by "dictionary" ordering. From the resulting lexicographic ordering we obtain a one-to-one and onto function $f : \mathbb{N} \rightarrow \Gamma^*$.

Remark 2.2. A brief remark is in order concerning the term *lexicographic order*. Some people use this term to mean something different (namely, dictionary ordering *without* first ordering strings according to length), and they use the term *quasi-lexicographic order* to refer to what we are calling lexicographic order. This isn't a problem—there are plenty of cases in which people don't all agree on what terminology to use. What is important is that everyone is clear about what they mean. In this course, *lexicographic order* means strings are ordered first by length, and by "dictionary" ordering among strings of the same length.

It follows from the fact that the language Σ^* is countable, for any choice of an alphabet Σ, that every language $A \subseteq \Sigma^*$ is countable. This is because every subset of a countable set is also countable, which is a simple fact that you will be asked to prove as a homework problem.

The set of all languages over any alphabet is uncountable

Next we will consider the set of all languages over a given alphabet. If Σ is an alphabet, then saying that A is a language over Σ is equivalent to saying that A is a subset of Σ^*, and being a subset of Σ^* is the same thing as being an element of the power set of Σ^*. These three statements are therefore equivalent, for any choice of an alphabet Σ:

1. A is a language over the alphabet Σ.
2. $A \subseteq \Sigma^*$.
3. $A \in \mathcal{P}(\Sigma^*)$.

We have observed, for any choice of an alphabet Σ, that every language $A \subseteq \Sigma^*$ is countable, and it is natural to consider next if the set of all languages over Σ is countable. It is not.

Proposition 2.3. *Let Σ be an alphabet. The set $\mathcal{P}(\Sigma^*)$ is uncountable.*

To prove this proposition, we don't need to repeat the same sort of diagonalization argument used to prove that $\mathcal{P}(\mathbb{N})$ is uncountable—we can simply combine that theorem with the fact that there exists a one-to-one and onto function from $\mathbb{N}$ to Σ^*.

In greater detail, let

$$f : \mathbb{N} \rightarrow \Sigma^* \tag{2.4}$$

be a one-to-one and onto function, such as the function we obtained from the lexicographic ordering of Σ^* like before. We can extend this function, so to speak, to the power sets of $\mathbb{N}$ and Σ^* as follows. Let

$$g : \mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\Sigma^*) \tag{2.5}$$

be the function defined as

$$g(A) = \{f(n) \,:\, n \in A\} \tag{2.6}$$

for all $A \subseteq \mathbb{N}$. In words, the function g simply applies f to each of the elements in a given subset of $\mathbb{N}$. It is not hard to see that g is one-to-one and onto—we can express the inverse of g directly, in terms of the inverse of f, as follows.

$$g^{-1}(B) = \{f^{-1}(w) \,:\, w \in B\} \tag{2.7}$$

for every $B \subseteq \Sigma^*$.

Now, because there exists a one-to-one and onto function of the form (2.5), we conclude that $\mathcal{P}(\mathbb{N})$ and $\mathcal{P}(\Sigma^*)$ have the "same size." That is, because $\mathcal{P}(\mathbb{N})$ is uncountable, the same must be true of $\mathcal{P}(\Sigma^*)$. To be more formal about this statement, one may assume toward contradiction that $\mathcal{P}(\Sigma^*)$ is countable, which implies that there exists an onto function of the form

$$h : \mathbb{N} \rightarrow \mathcal{P}(\Sigma^*). \tag{2.8}$$

By composing this function with the inverse of the function g specified above, we obtain an onto function

$$g^{-1} \circ h : \mathbb{N} \rightarrow \mathcal{P}(\mathbb{N}), \tag{2.9}$$

which contradicts what we already know, which is that $\mathcal{P}(\mathbb{N})$ is uncountable.

2.2 Deterministic finite automata

The first model of computation we will discuss in this course is a very simple one, called the *deterministic finite automata* model. You should have already learned something about finite automata (also called *finite state machines*) in CS 241, so we aren't necessarily starting from the very beginning—but we do of course need a formal definition to proceed mathematically.

Two points to keep in mind as you consider the definition are the following.

1. The definition is based on sets (and functions, which can be formally described in terms of sets, as you may have learned in a discrete mathematics course). This is not surprising: set theory provides the foundation for much of mathematics, and it is only natural that we look to sets as we formulate definitions.
2. Although deterministic finite automata are not very powerful in computational terms, the model is important nevertheless, and it is just the start. Do not be bothered if it seems like a weak and useless model—we're not trying to model general purpose computers at this stage, and finite automata are far from useless.

Definition 2.4. A *deterministic finite automaton* (or *DFA*, for short) is a 5-tuple

$$M = (Q, \Sigma, \delta, q_0, F), \tag{2.10}$$

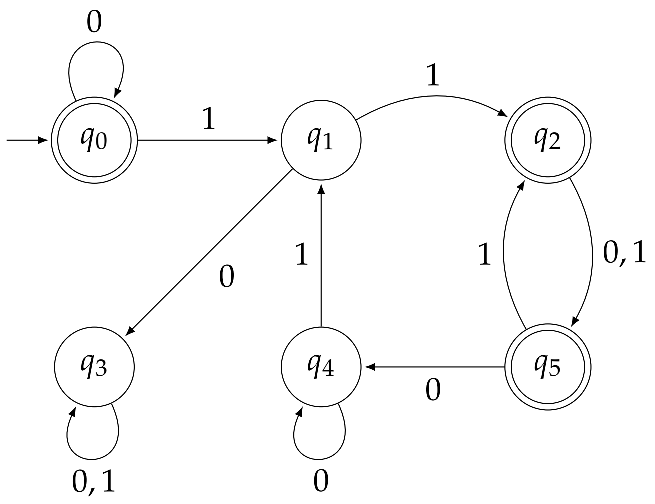
where Q is a finite and nonempty set (whose elements we will call *states*), Σ is an alphabet, δ is a function (called the *transition function*) having the form

$$\delta : Q \times \Sigma \rightarrow Q, \tag{2.11}$$

$q_0 \in Q$ is a state (called the *start state*), and $F \subseteq Q$ is a subset of states (whose elements we will call *accept states*).

State diagrams

It is common that DFAs are expressed using *state diagrams*, such as this one:



State diagrams express all 5 parts of the formal definition of DFAs:

1. States are denoted by circles.
2. Alphabet symbols label the arrows.
3. The transition function is determined by the arrows and the circles they connect.
4. The start state is determined by the arrow coming in from nowhere.
5. The accept states are those with double circles.

For the example above we have the following. The state set is

$$Q = \{q_0, q_1, q_2, q_3, q_4, q_5\}, \tag{2.12}$$

the alphabet is

$$\Sigma = \{0, 1\}, \tag{2.13}$$

the start state is q_0, the set of accepts states is

$$F = \{q_0, q_2, q_5\}, \tag{2.14}$$

and the transition function $\delta : Q \times \Sigma \to Q$ is as follows:

$$\begin{array}{lll} \delta(q_0,0) = q_0, & \delta(q_1,0) = q_3, & \delta(q_2,0) = q_5, \\ \delta(q_0,1) = q_1, & \delta(q_1,1) = q_2, & \delta(q_2,1) = q_5, \\ \delta(q_3,0) = q_3, & \delta(q_4,0) = q_4, & \delta(q_5,0) = q_4, \\ \delta(q_3,1) = q_3, & \delta(q_4,1) = q_1, & \delta(q_5,1) = q_2. \end{array} \tag{2.15}$$

In order for a state diagram to correspond to a DFA, it must be that for every state and every symbol, there is exactly one arrow exiting from that state labeled by that symbol.

Of course you can also go the other way and write down a state diagram from a formal description of a 5-tuple $(Q, \Sigma, \delta, q_0, F)$. It is a routine exercise to do this.

DFA computations

I imagine you already know what it means for a DFA $M = (Q, \Sigma, \delta, q_0, F)$ to *accept* or *reject* a given input string $w \in \Sigma^*$, or that at least you have guessed it based on a moment's thought about the definition. It is easy enough to say it in words, particularly when we think in terms of state diagrams: we start on the start state, follow transitions from one state to another according to the symbols of w (reading one at a time, left to right), and we accept if and only if we end up on an accept state (and otherwise we reject).

This all makes sense, but it is useful nevertheless to think about how it is expressed formally. That is, how do we define in precise, mathematical terms what it means for a DFA to accept or reject a given string? In particular, phrases like "follow transitions" and "end up on an accept state" can be replaced by more precise mathematical notions.

Here is one way to define acceptance and rejection more formally. Notice again that the definition focuses on sets and functions.

Definition 2.5. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $w \in \Sigma^*$ be a string. The DFA M *accepts* the string w if one of the following statements holds:

1. $w = \varepsilon$ and $q_0 \in F$.
2. $w = \sigma_1 \cdots \sigma_n$ for a positive integer n and symbols $\sigma_1, \ldots, \sigma_n \in \Sigma$, and there exist states $r_0, \ldots, r_n \in Q$ such that $r_0 = q_0$, $r_n \in F$, and
$$r_{k+1} = \delta(r_k, \sigma_{k+1}) \tag{2.16}$$
for all $k \in \{0, \ldots, n-1\}$.

If m does not accept w, then M *rejects* w.

In words, the formal definition of acceptance is that there exists a sequence of states $r_0, \ldots, r_n$ such that the first state is the start state, the last state is an accept state, and each state in the sequence is determined from the previous state and the corresponding symbol read from the input as the transition function describes: if we are in the state q and read the symbol σ, the new state becomes $p = \delta(q, \sigma)$. The first statement in the definition is simply a special case that handles the empty string.

It is natural to consider why we would prefer a formal definition like this to what is perhaps a more human-readable definition. Of course, the human-readable version beginning with "Start on the start state, follow transitions ..." is effective for explaining the concept of a DFA, but the formal definition has the benefit that it reduces the notion of acceptance to elementary mathematical statements about sets and functions. It is also quite succinct and precise, and leaves no ambiguities about what it means for a DFA to accept or reject.

It is sometimes useful to define a new function

$$\delta^* : Q \times \Sigma^* \rightarrow Q, \tag{2.17}$$

based on a given transition function

$$\delta : Q \times \Sigma \rightarrow Q, \tag{2.18}$$

in the following recursive way:

1. $\delta^*(q, \varepsilon) = q$ for every $q \in Q$, and
2. $\delta^*(q, w\sigma) = \delta(\delta^*(q, w), \sigma)$ for all $q \in Q$, $\sigma \in \Sigma$, and $w \in \Sigma^*$.

Intuitively speaking, $\delta^*(q, w)$ is the state you end up on if you start at state q and follow the transitions specified by the string w.

It is the case that a DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepts a string $w \in \Sigma^*$ if and only if $\delta^*(q_0, w) \in F$. A natural way to argue this formally, which we will not do in detail, is to prove by induction on the length of w that $\delta^*(q, w) = p$ if and only if one of these two statements is true:

1. $w = \varepsilon$ and $p = q$.
2. $w = \sigma_1 \cdots \sigma_n$ for a positive integer n and symbols $\sigma_1, \ldots, \sigma_n \in \Sigma$, and there exist states $r_0, \ldots, r_n \in Q$ such that $r_0 = q$, $r_n = p$, and

$$r_{k+1} = \delta(r_k, \sigma_{k+1}) \tag{2.19}$$

for all $k \in \{0, \ldots, n-1\}$.

Once that equivalence is proved, the statement $\delta^*(q_0, w) \in F$ can be equated to M accepting w.

Remark 2.6. By now it is evident that we will not formally prove every statement we make in this course. If we did, we wouldn't get very far, and even then we might look back and feel as if we could probably have been even more formal. If we insisting on proving everything with more and more formality, we could in principle reduce every mathematical claim we make to axiomatic set theory—but then we would have covered very little material about computation in a one-term course, and our proofs would most likely be incomprehensible (and quite possibly would contain as many errors as you would expect to find in a complicated and untested program written in assembly language). Naturally we won't take this path, but from time to time we will discuss the nature of proofs, how we would prove something if we took the time to do it, and how certain high-level statements and arguments could be reduced to more basic and concrete steps pointing in the general direction of completely formal proofs that could be verified by a computer.

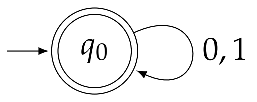
Languages recognized by DFAs and regular languages

Suppose $M = (Q, \Sigma, \delta, q_0, F)$ is a DFA. We may then consider the set of all strings that are accepted by M. This language is denoted $\mathrm{L}(M)$, so that

$$\mathrm{L}(M) = \{ w \in \Sigma^* : M \text{ accepts } w \}. \tag{2.20}$$

We refer to this as the *language recognized by* M.[2] It is important to understand that this is a single, well-defined language consisting precisely of those strings accepted by M and not containing any strings rejected by M.

For example, here is a very simple DFA over the binary alphabet $\Sigma = \{0, 1\}$:



If we call this DFA M, then it is easy to describe the language recognized by M; it is

$$\mathrm{L}(M) = \Sigma^*. \tag{2.21}$$

That's because M accepts exactly those strings in Σ^*. Now, if you were to consider a different language over Σ, such as

$$A = \{ w \in \Sigma^* : |w| \text{ is a prime number} \}, \tag{2.22}$$

then of course it is true that M accepts every string in A. However, M also accepts some strings that are not in A, so A is not the language recognized by M.

We have one more definition for this lecture, which introduces some very important terminology.

Definition 2.7. Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language over Σ. The language A is *regular* if there exists a DFA M such that $A = \mathrm{L}(M)$.

[2] Alternatively, we might also refer to $\mathrm{L}(M)$ as the *language accepted by* M. Unfortunately this terminology sometimes leads to confusion because it overloads the term *accept*.

We have not seen too many DFAs thus far, so we don't have too many examples of regular languages to mention at this point, but we will see plenty of them throughout the course.

Let us finish off the lecture with a question.

Question 1. For a given alphabet Σ, is the number of regular languages over the alphabet Σ countable or uncountable?

The answer is "countable." The reason is that there are countably many DFAs over any alphabet Σ, and we can combine this fact with the observation that the function that maps each DFA to the regular language it recognizes is, by definition, an onto function to obtain the answer to the question.

When we say that there are countably many DFAs, we really should be a bit more precise. In particular, we are not considering two DFAs to be different if they are exactly the same except for the names we have chosen to give the states. This is reasonable because the names we give to different states of a DFA has no influence on the language recognized by that DFA—we may as well assume that the state set of a DFA is $Q = \{q_0, \ldots, q_{m-1}\}$ for some choice of a positive integer m. (In fact, people often don't even bother assigning names to states when drawing state diagrams of DFAs, because the state names are irrelevant to the way DFAs works.)

To see that there are countably many DFAs over a given alphabet Σ, we can use a similar strategy to what we did when proving that the set rational numbers $\mathbb{Q}$ is countable. First imagine that there is just one state: $Q = \{q_0\}$. There are only finitely many DFAs with just one state over a given alphabet Σ. (In fact there are just two, one where q_0 is an accept state and one where q_0 is a reject state.) Now consider the set of all DFAs with two states: $Q = \{q_0, q_1\}$. Again, there are only finitely many (although now it is more than two). Continuing on like this, for any choice of a positive integer m, there will be only finitely many DFAs with m states for a given alphabet Σ. Of course the number of DFAs with m states grows exponentially with m, but this is not important—it is enough to know that the number is finite. If you chose some arbitrary way of sorting each of these finite lists of DFAs, and then you concatenated the lists together starting with the 1 state DFAs, then the 2 state DFAs, and so on, you would end up with a single list containing every DFA. From such a list you can obtain an onto function from $\mathbb{N}$ to the set of all DFAs over Σ in a similar way to what we did for the rational numbers.

Because there are uncountably many languages $A \subseteq \Sigma^*$, and only countably many regular languages $A \subseteq \Sigma^*$, we can immediately conclude that some languages are not regular. This is just an existence proof—it doesn't give us a specific language that is not regular, it just tells us that there is one. We'll see methods later that allow us to conclude that certain specific languages are not regular.

Lecture 3

Nondeterministic finite automata

This lecture is focused on the *nondeterministic finite automata* (NFA) model and its relationship to the DFA model.

Nondeterminism is an important concept in the theory of computing. It refers to the possibility of having multiple choices for what can happen at various points in a computation. We then consider all of the possible outcomes that these choices can have, usually focusing on whether or not a sequence of choices *exists* that leads to acceptance. This may sound like a fantasy mode of computation not likely to be relevant from a practical viewpoint, because real computers don't make nondeterministic choices: each step a computer makes is uniquely determined by its configuration at any given moment. Our interest in nondeterminism is, however, not meant to suggest otherwise. Throughout this course we will see that nondeterminism is a powerful analytic tool (in the sense that it helps us to design things and prove facts), and its close connection with proofs and verification has fundamental importance.

3.1 Nondeterministic finite automata basics

Let us begin our discussion of the NFA model with its definition. The definition is similar to the definition of the DFA model, but with a key difference.

Definition 3.1. A *nondeterministic finite automaton* (or *NFA*, for short) is a 5-tuple

$$N = (Q, \Sigma, \delta, q_0, F), \tag{3.1}$$

where Q is a finite and nonempty set of *states*, Σ is an *alphabet*, δ is a *transition function* having the form

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q), \tag{3.2}$$

$q_0 \in Q$ is a *start state*, and $F \subseteq Q$ is a subset of *accept states*.

The key difference between this definition and the analogous definition for DFAs is that the transition function has a different form. For a DFA we had that $\delta(q, \sigma)$ was a *state*, for any choice of a state q and a symbol σ, representing the next state that the DFA would move to if it was in the state q and read the symbol σ. For an NFA, each $\delta(q, \sigma)$ is not a state, but rather a *subset of states*, which is equivalent to $\delta(q, \sigma)$ being an element of the power set $\mathcal{P}(Q)$. This subset represents all of the *possible states* that the NFA could move to when in state q and reading symbol σ. There could be just a single state in this subset, or there could be multiple states, or there might even be no states at all—it is possible to have $\delta(q, \sigma) = \varnothing$.

We also have that the transition function of an NFA is not only defined for every pair $(q, \sigma) \in Q \times \Sigma$, but also for every pair (q, ε). Here, as always in this course, ε denotes the empty string. By defining δ for such pairs we are allowing for so-called *ε-transitions*, where an NFA may move from one state to another without reading a symbol from the input.

State diagrams

Similar to DFAs, we sometimes represent NFAs with state diagrams. This time, for each state q and each symbol σ, there may be multiple arrows leading out of the circle representing the state q labeled by σ, which tells us which states are contained in $\delta(q, \sigma)$, or there may be no arrows like this when $\delta(q, \sigma) = \varnothing$. We may also label arrows by ε, which indicates where the ε-transitions lead. Figure 3.1 gives an example of a state diagram for an NFA. In this case, we see that $Q = \{q_0, q_1, q_2, q_3\}$,

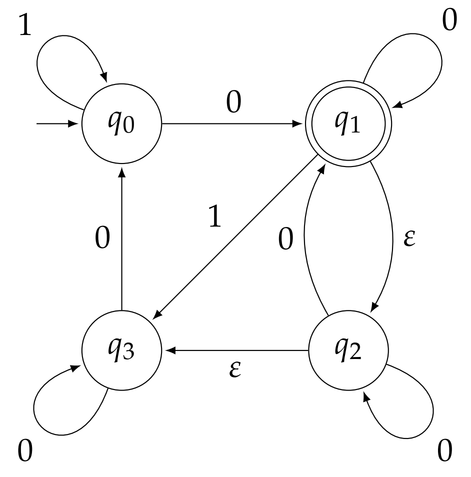


Figure 3.1: An NFA state diagram

$\Sigma = \{0,1\}$, q_0 is the start state, and $F = \{q_1\}$, just like we would have if this diagram were representing a DFA. The transition function, which must take the form

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q) \tag{3.3}$$

as the definition states, is given by

$$\begin{array}{lll} \delta(q_0,0) = \{q_1\}, & \delta(q_0,1) = \{q_0\}, & \delta(q_0,\varepsilon) = \varnothing, \\ \delta(q_1,0) = \{q_1\}, & \delta(q_1,1) = \{q_3\}, & \delta(q_1,\varepsilon) = \{q_2\}, \\ \delta(q_2,0) = \{q_1,q_2\}, & \delta(q_2,1) = \varnothing, & \delta(q_2,\varepsilon) = \{q_3\}, \\ \delta(q_3,0) = \{q_0,q_3\}, & \delta(q_3,1) = \varnothing, & \delta(q_3,\varepsilon) = \varnothing. \end{array} \tag{3.4}$$

NFA computations

Next let us consider the definition of acceptance and rejection for NFAs. This time we'll start with the formal definition and then try to understand what it says.

Definition 3.2. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and let $w \in \Sigma^*$ be a string. The NFA N *accepts* w if there exists a natural number $m \in \mathbb{N}$, a sequence of states $r_0, \ldots, r_m$, and a sequence of either symbols or empty strings $\sigma_1, \ldots, \sigma_m \in \Sigma \cup \{\varepsilon\}$ such that the following statements all hold:

1. $r_0 = q_0$.
2. $r_m \in F$.
3. $w = \sigma_1 \cdots \sigma_m$.
4. $r_{k+1} \in \delta(r_k, \sigma_{k+1})$ for every $k \in \{0, \ldots, m-1\}$.

If N does not accept w, then we say that N *rejects* w.

As you may already know, we can think of the computation of an NFA N on an input string w as being like a single-player game, where the goal is to start on the start state, make moves from one state to another, and end up on an accept state. If you want to move from a state q to a state p, there are two possible ways to do this: you can move from q to p by reading a symbol σ from the input provided that $p \in \delta(q, \sigma)$, or you can move from q to p without reading a symbol provided that $p \in \delta(q, \varepsilon)$ (i.e., there is an ε-transition from q to p). In order to win the game, you must not only end on an accept state, but you must also have read every symbol from the input string w. To say that N accepts w means that *it is possible* to win the corresponding game.

Definition 3.2 essentially formalizes the notion of winning the game we just discussed: the natural number m represents the number of moves you make and

$r_0, \ldots, r_m$ represent the states that are visited. In order to win the game you have to start on state q_0 and end on an accept state, which is why the definition requires $r_0 = q_0$ and $r_m \in F$, and it must also be that every symbol of the input is read by the end of the game, which is why the definition requires $w = \sigma_1 \cdots \sigma_m$. The condition $r_{k+1} \in \delta(r_k, \sigma_{k+1})$ for every $k \in \{0, \ldots, m-1\}$ corresponds to every move being a legal move in which a valid transition is followed.

We should take a moment to note how the definition works when $m = 0$. Of course, the natural numbers (as we have defined them) include 0, so there is nothing that prevents us from considering $m = 0$ as one way that a string might potentially be accepted. If we begin with the choice $m = 0$, then we must consider the existence of a sequence of states $r_0, \ldots, r_0$ and a sequence of symbols of empty strings $\sigma_1, \ldots, \sigma_0 \in \Sigma \cup \{\varepsilon\}$, and whether or not these sequences satisfy the four requirements listed in the definition. There is nothing wrong with a sequence of states having the form $r_0, \ldots, r_0$, by which we really just mean the sequence r_0 having a single element. The sequence $\sigma_1, \ldots, \sigma_0 \in \Sigma \cup \{\varepsilon\}$, on the other hand, looks like it doesn't make any sense. It does, however, actually make sense: it is simply an *empty* sequence having no elements in it. The sensible way to interpret the condition $w = \sigma_1 \cdots \sigma_0$ in this case, which is a concatenation of an empty sequence of symbols or empty strings, is that it means $w = \varepsilon$.[1] Asking that the condition $r_{k+1} \in \delta(r_k, \sigma_{k+1})$ should hold for every $k \in \{0, \ldots, m-1\}$ is a vacuous statement, and therefore trivially true, because there are no values of k to worry about.

Thus, if it is the case that the initial state q_0 of the NFA we are considering happens to be an accept state and our input is the empty string, then the NFA accepts—for we can take $m = 0$ and $r_0 = q_0$, and the definition is satisfied. (It is worth mentioning that we could have done something similar in our definition for when a DFA accepts: if we allowed $n = 0$ in the second statement of that definition, it would be equivalent to the first statement, and so we really didn't need to take the two possibilities separately.)

Along similar lines to what we did for DFAs, we can define an extended version of the transition function of an NFA. In particular, if

$$\delta : Q \times \Sigma \rightarrow \mathcal{P}(Q) \tag{3.5}$$

is a transition of an NFA, we define a new function

$$\delta^* : Q \times \Sigma^* \rightarrow \mathcal{P}(Q) \tag{3.6}$$

[1] Note that it is a *convention*, and not something you can deduce, that the concatenation of an empty sequence of symbols gives you the empty string. It is similar to the convention that the sum of an empty sequence of numbers is 0 and the product of an empty sequence of numbers is 1.

as follows. First, we define the *ε-closure* of any set $R \subseteq Q$ as

$$\varepsilon(R) = \left\{ q \in Q \,:\, \begin{array}{l} q \text{ is reachable from some } r \in R \text{ by following} \\ \text{zero or more } \varepsilon\text{-transitions} \end{array} \right\}. \tag{3.7}$$

Another way of defining $\varepsilon(R)$ is to say that it is the intersection of all subsets $T \subseteq Q$ satisfying these conditions:

1. $R \subseteq T$.
2. $\delta(q, \varepsilon) \subseteq T$ for every $q \in T$.

We can interpret this alternative definition as saying that $\varepsilon(R)$ is the *smallest* subset of Q that contains R and is such that you can never get out of this set by following an ε-transition. With the notion of the ε-closure in hand, we define δ^* recursively as follows:

1. $\delta^*(q, \varepsilon) = \varepsilon(\{q\})$ for every $q \in Q$, and
2. $\delta^*(q, w\sigma) = \varepsilon\Big(\bigcup_{r \in \delta^*(q,w)} \delta(r, \sigma)\Big)$ for every $q \in Q$, $\sigma \in \Sigma$, and $w \in \Sigma^*$.

Intuitively speaking, $\delta^*(q, w)$ is the set of all states that you could potentially reach by starting on the state q, reading w, and making as many ε-transitions along the way as you like. To say that an NFA $N = (Q, \Sigma, \delta, q_0, F)$ accepts a string $w \in \Sigma^*$ is equivalent to the condition that $\delta^*(q_0, w) \cap F \neq \varnothing$.

Also similar to DFAs, the notation $\mathrm{L}(N)$ denotes the language *recognized* by an NFA N:

$$\mathrm{L}(N) = \big\{ w \in \Sigma^* \,:\, N \text{ accepts } w \big\}. \tag{3.8}$$

3.2 Equivalence of NFAs and DFAs

It seems like NFAs might potentially be more powerful than DFAs because NFAs have the option to use nondeterminism. Perhaps you already know from a previous course, however, that this is not the case, as the following theorem states.

Theorem 3.3. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language A is regular (i.e., recognized by a DFA) if and only if $A = \mathrm{L}(N)$ for some NFA N.*

Let us begin by breaking this theorem down, to see what needs to be shown in order to prove it. First, it is an "if and only if" statement, so there are two things to prove:

1. If A is regular, then $A = \mathrm{L}(N)$ for some NFA N.

2. If $A = \mathrm{L}(N)$ for some NFA N, then A is regular.

If you were in a hurry and had to choose one of these two statements to prove, you would likely choose the first—it's the easier of the two by far. In particular, suppose A is regular, so by definition there exists a DFA $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes A. The goal is to define an NFA N that also recognizes A. This is simple, we can just take N to be the NFA whose state diagram is the same as the state diagram for M. At a formal level, N isn't *exactly* the same as M; because N is an NFA, its transition function will have a different form from a DFA transition function, but in this case the difference is only cosmetic. More formally speaking, we can define $N = (Q, \Sigma, \mu, q_0, F)$ where the transition function $\mu : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ is defined as

$$\mu(q, \sigma) = \{\delta(q, \sigma)\} \quad \text{and} \quad \mu(q, \varepsilon) = \varnothing \tag{3.9}$$

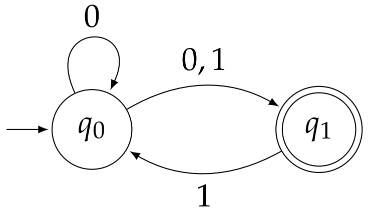
for all $q \in Q$ and $\sigma \in \Sigma$. It holds that $\mathrm{L}(N) = \mathrm{L}(M) = A$, and so we're done.

Now let us consider the second statement listed above. We assume $A = \mathrm{L}(N)$ for some NFA $N = (Q, \Sigma, \delta, q_0, F)$, and our goal is to show that A is regular. That is, we must prove that there exists a DFA M such that $\mathrm{L}(M) = A$. The most direct way to do this is to argue that, by using the description of N, we are able to come up with an *equivalent* DFA M. That is, if we can show how an arbitrary NFA N can be used to define a DFA M such that $\mathrm{L}(M) = \mathrm{L}(N)$, then we'll be done.

We will use the description of an NFA N to define an equivalent DFA M using a simple idea: each *state* of M will keep track of a *subset of states* of N. After reading any part of its input string, there will always be some subset of states that N could possibly be in, and we will design M so that after reading the same part of its input string it will be in a state corresponding to this subset of states of N.

A simple example

Let us see how this works for a simple example before we describe it in general. Consider the following NFA N:



If we describe this NFA formally, according to the definition of NFAs, it is given by

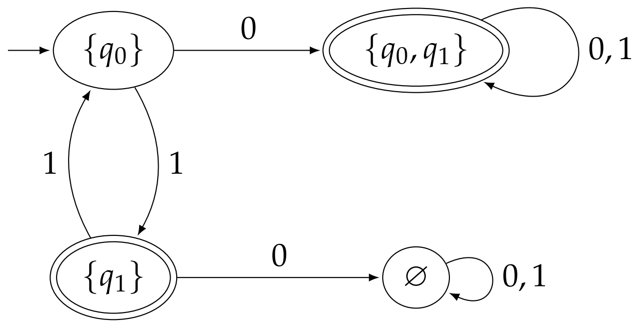
$$N = (Q, \Sigma, \delta, q_0, F) \tag{3.10}$$

where $Q = \{q_0, q_1\}$, $\Sigma = \{0,1\}$, $F = \{q_1\}$, and $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(Q)$ is defined as follows:

$$\begin{array}{lll} \delta(q_0,0) = \{q_0, q_1\}, & \delta(q_0,1) = \{q_1\}, & \delta(q_0,\varepsilon) = \varnothing, \\ \delta(q_1,0) = \varnothing, & \delta(q_1,1) = \{q_0\}, & \delta(q_1,\varepsilon) = \varnothing. \end{array} \tag{3.11}$$

We are going to define an NFA M having one state for every subset of states of N. We can name the states of M however we like, so we may as well name them directly with the subsets of Q. In other words, the state set of M will be the power set $\mathcal{P}(Q)$.

Have a look at the following state diagram and think about if it makes sense as a good choice for M:



Formally speaking, this DFA is given by

$$M = (\mathcal{P}(Q), \Sigma, \mu, \{q_0\}, \{\{q_1\}, \{q_0, q_1\}\}), \tag{3.12}$$

where the transition function $\mu : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q)$ is defined as

$$\begin{array}{ll} \mu(\{q_0\},0) = \{q_0, q_1\}, & \mu(\{q_0\},1) = \{q_1\}, \\ \mu(\{q_1\},0) = \varnothing, & \mu(\{q_1\},1) = \{q_0\}, \\ \mu(\{q_0,q_1\},0) = \{q_0, q_1\}, & \mu(\{q_0,q_1\},1) = \{q_0, q_1\}, \\ \mu(\varnothing,0) = \varnothing, & \mu(\varnothing,1) = \varnothing. \end{array} \tag{3.13}$$

One can verify that this DFA description indeed makes sense, one transition at a time.

For instance, suppose at some point in time N is in the state q_0. If a 0 is read, it is possible to either follow the self-loop and remain on state q_0 or follow the other transition and end on q_1. This is why there is a transition labeled 0 from the state $\{q_0\}$ to the state $\{q_0, q_1\}$ in M—the state $\{q_0, q_1\}$ in M is representing the fact that N could be either in the state q_0 or the state q_1. On the other hand, if N is in the state q_1 and a 0 is read, there are no possible transitions to follow, and this is why

M has a transition labeled 0 from the state $\{q_1\}$ to the state $\varnothing$. The state $\varnothing$ in M is representing the fact that there aren't any states that N could possibly be in (which is sensible because N is an NFA). The self-loop on the state $\varnothing$ in M labeled by 0 and 1 represents the fact that if N cannot be in any states at a given moment, and a symbol is read, there still aren't any states it could be in. You can go through the other transitions and verify that they work in a similar way.

There is also the issue of which state is chosen as the start state of M and which states are accept states. This part is simple: we let the start state of M correspond to the states of N we could possibly be in without reading any symbols at all, which is $\{q_0\}$ in our example, and we let the accept states of M be those states corresponding to any subset of states of N that includes at least one element of F.

The construction in general

Now let us think about the idea suggested above in greater generality. That is, we will specify a DFA M satisfying $\mathrm{L}(M) = \mathrm{L}(N)$ for an *arbitrary* NFA

$$N = (Q, \Sigma, \delta, q_0, F). \tag{3.14}$$

One thing to keep in mind as we do this is that N could have ε-transitions, whereas our simple example did not. It will, however, be easy to deal with ε-transitions by referring to the notion of the *ε-closure* that we discussed earlier. Another thing to keep in mind is that N really is arbitrary—maybe it has 1,000,000 states or more. It is therefore hopeless for us to describe what's going on using state diagrams, so we'll do everything abstractly.

First, we know what the state set of M should be based on the discussion above: the power set $\mathcal{P}(Q)$ of Q. Of course the alphabet is Σ because it has to be the same as the alphabet of N. The transition function of M should therefore take the form

$$\mu : \mathcal{P}(Q) \times \Sigma \rightarrow \mathcal{P}(Q) \tag{3.15}$$

in order to be consistent with these choices. In order to define the transition function μ precisely, we must therefore specify the output subset

$$\mu(R, \sigma) \subseteq Q \tag{3.16}$$

for every subset $R \subseteq Q$. One way to do this is as follows:

$$\mu(R, \sigma) = \bigcup_{q \in R} \varepsilon(\delta(q, \sigma)). \tag{3.17}$$

In words, the right-hand side of (3.17) represents every state in N that you can get to by (i) starting at any state in R, then (ii) following a transition labeled σ, and finally (iii) following any number of ε-transitions.

The last thing we need to do is to define the initial state and the accept states of M. The initial state is $\varepsilon(\{q_0\})$, which is every state you can reach from q_0 by just following ε-transitions, while the accept states are those subsets of Q containing at least one accept state of N. If we write $G \subseteq \mathcal{P}(Q)$ to denote the set of accept states of M, then we may define this set as

$$G = \bigl\{R \in \mathcal{P}(Q) \,:\, R \cap F \neq \varnothing\bigr\}. \tag{3.18}$$

The DFA M can now be specified formally as

$$M = (\mathcal{P}(Q), \Sigma, \mu, \varepsilon(\{q_0\}), G). \tag{3.19}$$

Now, if we are being honest with ourselves, we cannot say that we have *proved* that for every NFA N there is an equivalent DFA M satisfying $\mathrm{L}(M) = \mathrm{L}(N)$. All we've done is to define a DFA M from a given DFA N that *seems* like it should satisfy this equality. We won't go through a formal proof that it really is the case that $\mathrm{L}(M) = \mathrm{L}(N)$, but it is worthwhile to think about how we would do this if we had to. (It is, in fact, true that $\mathrm{L}(M) = \mathrm{L}(N)$, so you shouldn't worry that we're trying to prove something that might be false.)

First, if we are to prove that the two languages $\mathrm{L}(M)$ and $\mathrm{L}(N)$ are equal, the natural way to do it is to split it into two statements: $\mathrm{L}(M) \subseteq \mathrm{L}(N)$ and $\mathrm{L}(N) \subseteq \mathrm{L}(M)$. This is often the way to prove the equality of two sets. Nothing tells us that the two statements need to be proved in the same way, and by doing them separately we give ourselves more options about how to approach the proof. Let's start with the subset relation $\mathrm{L}(N) \subseteq \mathrm{L}(M)$, which is equivalent to saying that if $w \in \mathrm{L}(N)$, then $w \in \mathrm{L}(M)$. We can now fall back on the definition of what it means for N to accept a string w, and try to conclude that M must also accept w. It's a bit tedious to write everything down carefully, but it is possible and maybe you can convince yourself that this is so. The other relation $\mathrm{L}(M) \subseteq \mathrm{L}(N)$ is equivalent to saying that if $w \in \mathrm{L}(M)$, then $w \in \mathrm{L}(N)$. The basic idea here is similar in spirit, although the specifics are a bit different. This time we start with the definition of acceptance for a DFA, applied to M, and then try to reason that N must accept w.

A different way to prove that the construction works correctly is to make use of the functions δ^* and μ^*, which are defined from δ and μ as we discussed in the previous lecture and earlier in this lecture. In particular, using induction on the length of w, it can be proved that

$$\mu^*(\varepsilon(R), w) = \bigcup_{q \in R} \delta^*(q, w) \tag{3.20}$$

for every string $w \in \Sigma^*$ and every subset $R \subseteq Q$. Once we have this, we see that $\mu^*(\varepsilon(\{q_0\}), w)$ is contained in G if and only if $\delta^*(q_0, w) \cap F \neq \varnothing$, which is equivalent to $w \in \mathrm{L}(M)$ if and only if $w \in \mathrm{L}(N)$.

In any case, I do not ask that you try to verify one of these proofs—but only that you *think about* how it would be done.

On the process of converting NFAs to DFAs

It is a very typical type of exercise in courses such as CS 360 that students are presented with an NFA and asked to come up with an equivalent DFA using the construction described above. I will not ask you to perform such a mindless tasks as this. Indeed, it will be helpful later in the course for us to observe that the construction itself can be implemented by a computer, and therefore requires absolutely no creativity whatsoever.

My primary aim when teaching CS 360 is to introduce you to theoretical computer science, and memorizing a simple technique like the one we just saw is not what theoretical computer science is all about. Theoretical computer science is about coming up with the proof in the first place, not behaving like a computer by performing the technique over and over. In this case it was all done in the 1950s by Michael Rabin and Dana Scott.

Instead of asking you to implement simple constructions, such as converting NFAs into DFAs, I am likely to ask you to solve problems that require creativity and reasoning about new ideas that you haven't seen yet. For example, I may tell you that A is a regular language, that B is obtained from A in a certain way, and ask you to prove that B is regular. Now that you know that NFAs and DFAs are equivalent, you can use this fact to your advantage: if you want to prove that B is regular, it suffices for you to give an NFA N satisfying $\mathrm{L}(N) = B$. Maybe it is possible to do this starting from the assumption that A is regular, and therefore recognized by a DFA.

If, for some reason, you find that you really do want to take a given NFA N and construct a DFA M recognizing the same language as N, it is worth pointing out that you really don't need to write down every subset of states of N and then draw the arrows. There will be exponentially many more states in M than in N, and it will often be that many of these states are useless, possibly unreachable from the start state of M. A better option is to first write down the start state of M, which corresponds to the ε-closure of the set containing just the start state of N, and only draw new states of M as you need them.

In the worst case, however, you might actually need all of those states. There are examples known of languages that have an NFA with n states, while the smallest DFA for the same language has 2^n states, for every choice of a positive integer n. So, while NFAs and DFAs are equivalent in computational power, there is sometimes a significant cost to be paid in converting an NFA into a DFA, which is that this might require the DFA to have a huge number of states.

Lecture 4

Regular operations and regular expressions

In this lecture we will discuss the *regular operations*, as well as *regular expressions* and their relationship to regular languages.

4.1 Regular operations

The *regular operations* are three operations on languages, as the following definition describes.

Definition 4.1. Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be languages. The *regular operations* are as follows:

1. *Union.* The language $A \cup B \subseteq \Sigma^*$ is defined as

$$A \cup B = \{w \,:\, w \in A \text{ or } w \in B\}. \tag{4.1}$$

   In words, this is just the ordinary union of two sets that happen to be languages.

2. *Concatenation.* The language $AB \subseteq \Sigma^*$ is defined as

$$AB = \{wx \,:\, w \in A \text{ and } x \in B\}. \tag{4.2}$$

   In words, this is the language of all strings obtained by concatenating together a string from A and a string from B, with the string from A on the left and the string from B on the right.

   Note that there is nothing about a string of the form wx that indicates where w stops and x starts—it is just the sequence of symbols you get by putting w and x together.

3. *Kleene star* (or just *star*, for short). The language A^* is defined as

$$A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup \cdots \tag{4.3}$$

In words, A^* is the language obtained by selecting any finite number of strings from A and concatenating them together. (This includes the possibility to select no strings at all from A, where we follow the convention that concatenating together no strings at all gives the empty string.)

Note that the name *regular operations* is just a name that has been chosen for these three operations—they are special operations and they do indeed have a close connection to the regular languages, but naming them *the regular operations* is a choice we've made and not something mandated in a mathematical sense.

Closure of regular languages under regular operations

Next let us prove a theorem connecting the regular operations with the regular languages.

Theorem 4.2. *The regular languages are closed with respect to the regular operations: if $A, B \subseteq \Sigma^*$ are regular languages, then the languages $A \cup B$, AB, and A^* are also regular.*

Proof. First let us observe that, because the languages A and B are regular, there must exist DFAs

$$M_A = (P, \Sigma, \delta, p_0, F) \quad \text{and} \quad M_B = (Q, \Sigma, \mu, q_0, G) \tag{4.4}$$

such that $\mathrm{L}(M_A) = A$ and $\mathrm{L}(M_B) = B$. We will make use of these DFAs as we prove that the languages $A \cup B$, AB, and A^* are regular. Because we are free to give whatever names we like to the states of a DFA without influencing the language it recognizes, there is no generality lost in assuming that P and Q are disjoint sets (meaning that $P \cap Q = \varnothing$).

Let us begin with $A \cup B$. From last lecture we know that if there exists an NFA N such that $\mathrm{L}(N) = A \cup B$, then $A \cup B$ is regular. With that fact in mind, our goal will be to define such an NFA. We will define this NFA N so that its states include all elements of both P and Q, as well as an additional state r_0 that is in neither P nor Q. This new state r_0 will be the start state of N. The transition function of N is to be defined so that all of the transitions among the states P defined by δ and all of the transitions among the states Q defined by μ are present, as well as two ε-transitions, one from r_0 to p_0 and one from r_0 to q_0.

Figure 4.1 illustrates what the NFA N looks like in terms of a state diagram. You should imagine that the shaded rectangles labeled M_A and M_B are the state

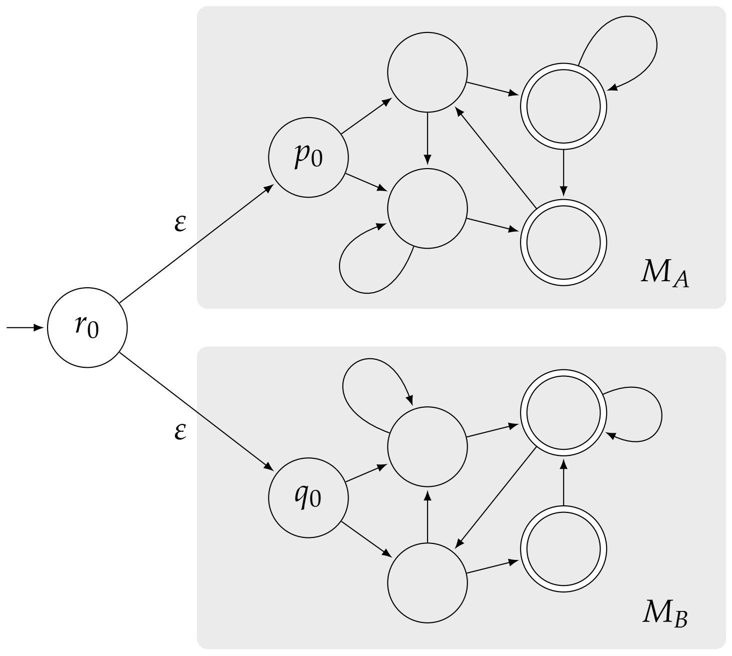


Figure 4.1: DFAs M_A and M_B are combined to form an NFA for the language $\mathrm{L}(M_A) \cup \mathrm{L}(M_B)$.

diagrams of M_A and M_B. (The illustrations in the figure are meant to suggest the state diagrams for these two DFAs.)

We can specify N more formally as follows:

$$N = (R, \Sigma, \eta, r_0, F \cup G) \tag{4.5}$$

where $R = P \cup Q \cup \{r_0\}$ (and we assume P, Q, and $\{r_0\}$ are disjoint sets as suggested above) and the transition function

$$\eta : R \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(R) \tag{4.6}$$

is defined as follows:

$$\begin{aligned}
\eta(p,\sigma) &= \{\delta(p,\sigma)\} && \text{(for all } p \in P \text{ and } \sigma \in \Sigma\text{)},\\
\eta(p,\varepsilon) &= \varnothing && \text{(for all } p \in P\text{)},\\
\eta(q,\sigma) &= \{\mu(q,\sigma)\} && \text{(for all } q \in Q \text{ and } \sigma \in \Sigma\text{)},\\
\eta(q,\varepsilon) &= \varnothing && \text{(for all } q \in Q\text{)},\\
\eta(r_0,\sigma) &= \varnothing && \text{(for all } \sigma \in \Sigma\text{)},\\
\eta(r_0,\varepsilon) &= \{p_0, q_0\}.
\end{aligned}$$

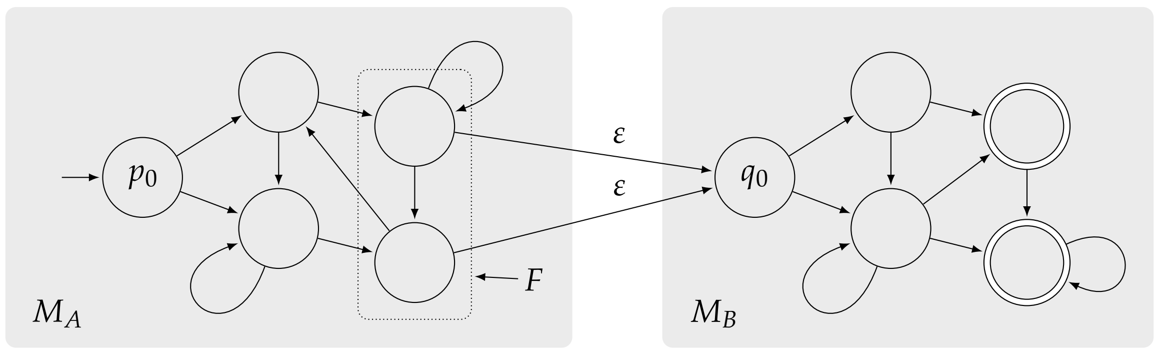


Figure 4.2: DFAs M_A and M_B are combined to form an NFA for the language $\mathrm{L}(M_A)\,\mathrm{L}(M_B)$.

The accept states of N are $F \cup G$.

Every string that is accepted by M_A is also accepted by N because we can simply take the ε-transition from r_0 to p_0 and then follow the same transitions that would be followed in M_A to an accept state. By similar reasoning, every string accepted by M_B is also accepted by N. Finally, every string that is accepted by N must be accepted by either M_A or M_B (or both), because every accepting computation of N begins with one of the two ε-transitions and then necessarily mimics an accepting computation of either M_A or M_B depending on which ε-transition was taken. It therefore follows that

$$\mathrm{L}(N) = \mathrm{L}(M_A) \cup \mathrm{L}(M_B) = A \cup B, \tag{4.7}$$

and so we conclude that $A \cup B$ is regular.

Next we will prove that AB is regular. The idea is similar to the proof that $A \cup B$ is regular: we will use the DFAs M_A and M_B to define an NFA N for the language AB. This time we will take the state set of N to be the union $P \cup Q$, and the start state p_0 of M_A will be the start state of N. All of the transitions defined by M_A and M_B will be included in N, and in addition we will add an ε-transition from each accept state of M_A to the start state of M_B. Finally, the accept states of N will be just the accept states G of M_B (and not the accept states of M_A). Figure 4.2 illustrates the construction of N based on M_A and M_B.

In formal terms, N is the NFA defined as

$$N = (P \cup Q, \Sigma, \eta, p_0, G) \tag{4.8}$$

where the transition function

$$\eta : (P \cup Q) \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(P \cup Q) \tag{4.9}$$

is given by

$$\begin{aligned}
\eta(p,\sigma) &= \{\delta(p,\sigma)\} && \text{(for all } p \in P \text{ and } \sigma \in \Sigma\text{),}\\
\eta(q,\sigma) &= \{\mu(q,\sigma)\} && \text{(for all } q \in Q \text{ and } \sigma \in \Sigma\text{),}\\
\eta(p,\varepsilon) &= \{q_0\} && \text{(for all } p \in F\text{),}\\
\eta(p,\varepsilon) &= \varnothing && \text{(for all } p \in P\backslash F\text{),}\\
\eta(q,\varepsilon) &= \varnothing && \text{(for all } q \in Q\text{).}
\end{aligned}$$

Along similar lines to what was done in the proof that $A \cup B$ is regular, one can argue that N recognizes the language AB, from which it follows that AB is regular.

Finally we will prove that A^* is regular, and once again the proof proceeds along similar lines. (This time we will just consider M_A and not M_B because the language B is not involved.) Let us start with the formal specification of N this time—we will let

$$N = (R, \Sigma, \eta, r_0, \{r_0\}) \tag{4.10}$$

where $R = P \cup \{r_0\}$ and the transition function

$$\eta : R \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(R) \tag{4.11}$$

is defined as

$$\begin{aligned}
\eta(r_0,\sigma) &= \varnothing && \text{(for all } \sigma \in \Sigma\text{),}\\
\eta(r_0,\varepsilon) &= p_0, && \\
\eta(p,\sigma) &= \{\delta(p,\sigma)\} && \text{(for every } p \in P \text{ and } \sigma \in \Sigma\text{),}\\
\eta(p,\varepsilon) &= \{r_0\} && \text{(for every } p \in F\text{),}\\
\eta(p,\varepsilon) &= \varnothing && \text{(for every } p \in P\backslash F\text{).}
\end{aligned}$$

In words, we take N to be the NFA whose states are the states of M_A along with an additional state r_0, which is both the start state of N and its only accept state. The transitions of N include all of the transitions of M_A, along with an ε-transition from r_0 to the start state p_0 of M_A, and ε-transitions from all of the accept states of M_A back to r_0. Figure 4.3 provides an illustration of how N relates to M_A.

It is evident that N recognizes the language A^*. This is because the strings it accepts are precisely those strings that cause N to start at r_0 and loop back to r_0 zero or more times, with each loop corresponding to some string that is accepted by M_A. As $\mathrm{L}(N) = A^*$, it follows that A^* is regular, and so the proof is complete. □

It is natural to ask why we could not easily conclude, for a regular language A, that A^* is regular using the facts that the regular languages are closed under union and concatenation. In more detail, we have that

$$A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup \cdots \tag{4.12}$$

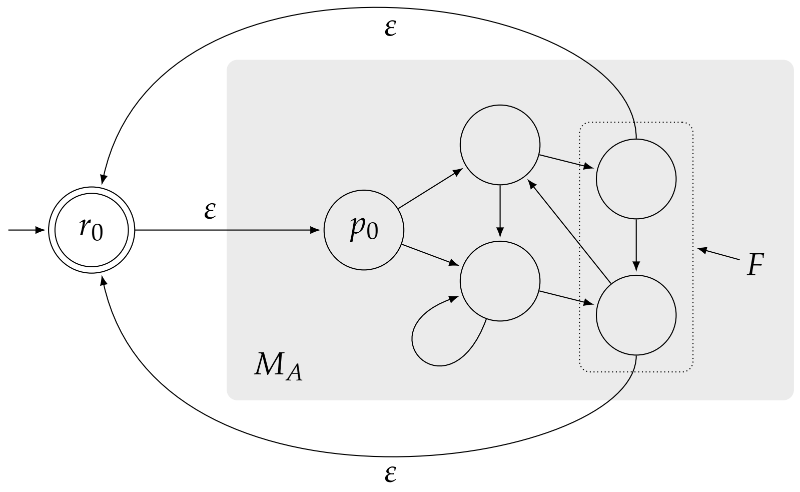
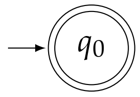


Figure 4.3: The DFA M_A is modified to form an NFA for the language $\mathrm{L}(M_A)^*$.

It is easy to see that the language $\{\varepsilon\}$ is regular—here is the state diagram for an NFA that recognizes the language $\{\varepsilon\}$ (for any choice of an alphabet):



The language $\{\varepsilon\} \cup A$ is therefore regular because the union of two regular languages is also regular. We also have that AA is regular because the concatenation of two regular languages is regular, and therefore $\{\varepsilon\} \cup A \cup AA$ is regular because it is the union of the two regular languages $\{\varepsilon\} \cup A$ and AA. Continuing on like this we find that the language

$$\{\varepsilon\} \cup A \cup AA \cup AAA \tag{4.13}$$

is regular, the language

$$\{\varepsilon\} \cup A \cup AA \cup AAA \cup AAAA \tag{4.14}$$

is regular, and so on. Does this imply that A^* is regular?

The answer is "no." Although it is true that A^* is regular whenever A is regular, as we proved earlier, the argument just suggested based on combining unions and concatenations alone does not establish it. This is because we can never conclude from this argument that the infinite union (4.12) is regular, but only that finite unions such as (4.14) are regular.

If you are still skeptical or uncertain, consider this statement:

If A is a finite language, then A^* is also a finite language.

This statement is false in general. For example, $A = \{0\}$ is finite, but

$$A^* = \{\varepsilon, 0, 00, 000, \ldots\} \tag{4.15}$$

is infinite. On the other hand, it is true that the union of two finite languages is finite, and the concatenation of two finite languages is finite, so something must go wrong when you try to combine these facts in order to conclude that A^* is finite. The situation is similar when the property of being finite is replaced by the property of being regular.

4.2 Other closure properties of regular languages

There are many other operations on languages aside from the regular operations under which the regular languages are closed. For example, the *complement* of a regular language is also regular. Just to be sure the terminology is clear, here is the definition of the complement of a language:

Definition 4.3. Let $A \subseteq \Sigma^*$ be a language over the alphabet Σ. The *complement* of A, which is denoted $\overline{A}$, is the language consisting of all strings over Σ that are not contained in A:

$$\overline{A} = \Sigma^* \backslash A. \tag{4.16}$$

(In this course, we will use a backslash to denote set difference: $S \backslash T$ is the set of all elements in S that are not in T.)

Proposition 4.4. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language over the alphabet Σ. The language $\overline{A}$ is also regular.*

This proposition is very easy to prove: because A is regular, there must exist a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $\mathrm{L}(M) = A$. We obtain a DFA for the language $\overline{A}$ simply by swapping the accept and reject states of M. That is, the DFA $K = (Q, \Sigma, \delta, q_0, Q \backslash F)$ recognizes $\overline{A}$.

While it is easy to obtain a DFA for the complement of a language if you have a DFA for the original language simply by swapping the accept and reject states, this does not work for NFAs. You might, for instance, swap the accept and reject states of an NFA and end up with an NFA that recognizes something very different from the complement of the language you started with. This is due to the asymmetric nature of accepting and rejecting for nondeterministic models.

Within the next few lectures we will see more examples of operations under which the regular languages are closed. Here is one more for this lecture.

Proposition 4.5. *Let Σ be an alphabet and let A and B be regular languages over the alphabet Σ. The intersection $A \cap B$ is also regular.*

This time we can just combine closure properties we already know to obtain this one. This is because De Morgan's laws implies that

$$A \cap B = \overline{\overline{A} \cup \overline{B}}. \tag{4.17}$$

If A and B are regular, then it follows that $\overline{A}$ and $\overline{B}$ are regular, and therefore $\overline{A} \cup \overline{B}$ is regular, and because the complement of this regular language is $A \cap B$ we have that $A \cap B$ is regular.

There is another way to conclude that $A \cap B$ is regular, which is arguably more direct. Because the languages A and B are regular, there must exist DFAs

$$M_A = (P, \Sigma, \delta, p_0, F) \quad \text{and} \quad M_B = (Q, \Sigma, \mu, q_0, G) \tag{4.18}$$

such that $\mathrm{L}(M_A) = A$ and $\mathrm{L}(M_B) = B$. We can obtain a DFA M recognizing $A \cap B$ using Cartesian products like this:

$$M = (P \times Q, \Sigma, \eta, (p_0, q_0), F \times G) \tag{4.19}$$

where

$$\eta((p,q),\sigma) = (\delta(p,\sigma), \mu(q,\sigma)) \tag{4.20}$$

for every $p \in P$, $q \in Q$, and $\sigma \in \Sigma$. In essence, the DFA M is what you get if you build a DFA that runs M_A and M_B in parallel, and accepts if and only if both M_A and M_B accept. You could also get a DFA for $A \cup B$ using a similar idea (but accepting if and only if M_A accepts or M_B accepts).

4.3 Regular expressions

I expect that you already have some experience with regular expressions—they are commonly used in programming languages and other applications to specify patterns for searching and string matching. When we use regular expressions in practice, we typically endow them with a rich set of operations, but in this class we take a minimal definition of regular expressions allowing only the three regular operations (and not other operations like negation or special symbols marking the first or last characters of an input).

Here is a formal definition. It is an example of an *inductive definition*, which will be commented on shortly.

Definition 4.6. Let Σ be an alphabet. It is said that R is a *regular expression* over the alphabet Σ if any of these properties holds:

1. $R = \varnothing$.
2. $R = \varepsilon$.
3. $R = \sigma$ for some choice of $\sigma \in \Sigma$.
4. $R = (R_1 \cup R_2)$ for regular expressions R_1 and R_2.
5. $R = (R_1 R_2)$ for regular expressions R_1 and R_2.
6. $R = (R_1^*)$ for a regular expression R_1.

When you see an inductive definition such as this one, you should interpret it in the most sensible way, as opposed to thinking of it as something circular or paradoxical. For instance, when it is said that $R = (R_1^*)$ for a regular expression R_1, it is to be understood that R_1 is *already* well-defined as a regular expression. We cannot, for instance, take R_1 to be the regular expression R that we are defining; for then we would have $R = (R)^*$, which might be interpreted as a strange, fractal-like expression that looks like this:

$$R = (((\cdots(\cdots)^*\cdots)^*)^*)^* \tag{4.21}$$

Such a thing makes no sense as a regular expression, and is not valid according to a sensible interpretation of the definition. Here are some valid examples of regular expressions over the binary alphabet $\Sigma = \{0, 1\}$:

$$\begin{gathered} \varnothing \\ \varepsilon \\ 0 \\ 1 \\ (0 \cup 1) \\ ((0 \cup 1)^*) \\ (((0 \cup \varepsilon)^*)1) \end{gathered}$$

It is helpful to think about these expressions as strings over the alphabet

$$\{0\,,\, 1\,,\, (\,,\,)\,,\, *\,,\, \cup,\, \varepsilon,\, \varnothing\} \tag{4.22}$$

at this stage.

Next we will define the *language recognized* (or *matched*) by a given regular expression. Again it is an inductive definition, and it directly parallels the regular expression definition itself. If it looks to you like it is stating something obvious, then your impression is correct—we require a formal definition, but it essentially says that we should define the language matched by a regular expression in the most straightforward and natural way.

Definition 4.7. Let R be a regular expression over the alphabet Σ. The *language recognized* by R, which is denoted $\mathrm{L}(R)$, is defined as follows:

1. If $R = \varnothing$, then $\mathrm{L}(R) = \varnothing$
2. If $R = \varepsilon$, then $\mathrm{L}(R) = \{\varepsilon\}$.
3. If $R = \sigma$ for $\sigma \in \Sigma$, then $\mathrm{L}(R) = \{\sigma\}$
4. $R = (R_1 \cup R_2)$ for regular expressions R_1 and R_2, then $\mathrm{L}(R) = \mathrm{L}(R_1) \cup \mathrm{L}(R_2)$
5. $R = (R_1 R_2)$ for regular expressions R_1 and R_2, then $\mathrm{L}(R) = \mathrm{L}(R_1)\,\mathrm{L}(R_2)$.
6. $R = (R_1^*)$ for a regular expression R_1, then $\mathrm{L}(R) = \mathrm{L}(R_1)^*$.

Order of precedence for regular operations

It may seem to you that regular expressions arising from the formal definition of regular expressions have an unusually large number of parentheses. For instance, the regular expression $(((0 \cup \varepsilon)^*)1)$ has more parentheses than it has non-parenthesis symbols. The parentheses ensure that every regular expression has an unambiguous meaning.

We can, however, reduce the need for so many parentheses by introducing an *order of precedence* for the regular operations. The order is as follows:

1. star (highest precedence)
2. concatenation
3. union (lowest precedence).

To be more precise, we're not changing the formal definition of regular expressions, we're just introducing a convention that allows some parentheses to be implicit, which makes for simpler-looking regular expressions. For example, we write

$$10^* \cup 1 \tag{4.23}$$

rather than

$$((1(0^*)) \cup 1). \tag{4.24}$$

Having agreed upon the order of precedence above, the simpler-looking expression is understood to mean the second expression.

A simple way to remember the order of precedence is to view the regular operations as being analogous to algebraic operations that you are already familiar with:

star looks like exponentiation, concatenation looks like multiplication, and unions are similar to additions. So, just as the expression $xy^2 + z$ has the same meaning as $((x(y^2)) + z)$, the expression $10^* \cup 1$ has the same meaning as $((1(0^*)) \cup 1)$.

Regular expressions characterize the regular languages

At this point it is natural to ask which languages have regular expressions. The answer is that the class of languages having regular expressions is precisely the class of regular languages. (If it were otherwise, you would have to wonder why the regular languages were named in this way!)

There are two implications needed to establish that the regular languages coincides with the class of languages having regular expressions. Let us start with the first implication, which is the content of the following proposition.

Proposition 4.8. *Let Σ be an alphabet and let R be a regular expression over the alphabet Σ. It holds that the language $\mathrm{L}(R)$ is regular.*

The idea behind a proof of this proposition is simple enough; we can easily build DFAs for the languages $\varnothing$, $\{\varepsilon\}$, and $\{\sigma\}$ (for a given symbol $\sigma \in \Sigma$), and by repeatedly using the constructions described in the proof of Theorem 4.2 one can combine together such DFAs to build an NFA recognizing the same language as any given regular expression.

The other implication is the content of the following theorem, which is more difficult to prove than the proposition above. I've included a proof in case you are interested, but you can also feel free to skip it if you prefer—there won't be questions on any exams or homeworks that require you to have studied the proof.

Theorem 4.9. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language. There exists a regular expression over the alphabet Σ such that $\mathrm{L}(R) = A$.*

Proof. Because A is regular, there must exist a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $\mathrm{L}(M) = A$. We are free to use whatever names we like for the states of a DFA, so no generality is lost in assuming $Q = \{1, \ldots, n\}$ for some positive integer n.

We are now going to define a language $B^k_{p,q} \subseteq \Sigma^*$, for every choice of states $p, q \in \{1, \ldots, n\}$ and an integer $k \in \{0, \ldots, n\}$. The language $B^k_{p,q}$ is the set of all strings w causing M to operate as follows:

> If we start M in the state p, then by reading w the DFA M moves to the state q. Moreover, aside from the beginning state p and the ending state q, the DFA M only touches states contained in the set $\{1, \ldots, k\}$ when reading w in this way.

For example, the language $B^n_{p,q}$ is simply the set of all strings causing M to move from p to q (because restricting the intermediate states that M touches to those contained in the set $\{1,\ldots,n\}$ is no restriction whatsoever). At the other extreme, the set $B^0_{p,q}$ must be a finite set; it could be the empty set if there are no direct transitions from p to q, it includes the empty string in the case $p = q$, and it includes a length-one string corresponding to each symbol that causes M to transition from p to q.

Now, we will prove by induction on k that there exists a regular expression $R^k_{p,q}$ satisfying $\mathrm{L}(R^k_{p,q}) = B^k_{p,q}$, for every choice of $p,q \in \{1,\ldots,n\}$ and $k \in \{0,\ldots,n\}$. The base case is $k = 0$. The language $B^0_{p,q}$ is finite for every $p,q \in \{1,\ldots,n\}$, consisting entirely of strings of length 0 or 1, so it is straightforward to define a corresponding regular expression $R^0_{p,q}$ that matches $B^0_{p,q}$.

For the induction step, we assume $k \geq 1$, and that there exists a regular expression $R^{k-1}_{p,q}$ satisfying $\mathrm{L}(R^{k-1}_{p,q}) = B^{k-1}_{p,q}$ for every $p,q \in \{1,\ldots,n\}$. It is the case that

$$B^k_{p,q} = B^{k-1}_{p,q} \cup B^{k-1}_{p,k}\big(B^{k-1}_{k,k}\big)^* B^{k-1}_{k,q} \tag{4.25}$$

This equality reflects the fact that the strings that cause M to move from p to q through the intermediate states $\{1,\ldots,k\}$ are precisely those strings that either (i) cause M to move from p to q without visiting k as an intermediate state, or (ii) cause M to move from p to q visiting state k as an intermediate state one or more times. We may therefore define a regular expression $R^k_{p,q}$ satisfying $\mathrm{L}(R^k_{p,q}) = B^k_{p,q}$ for every $p,q \in \{1,\ldots,n\}$ as

$$R^k_{p,q} = R^{k-1}_{p,q} \cup R^{k-1}_{p,k}\big(R^{k-1}_{k,k}\big)^* R^{k-1}_{k,q}. \tag{4.26}$$

Finally, we can now conclude that there exists a regular expression R satisfying $\mathrm{L}(R) = A$ by defining

$$R = \bigcup_{q\in F} R^n_{q_0,q}. \tag{4.27}$$

(In words, R is the regular expression we obtain by forming the union over all regular expressions $R^n_{q_0,q}$ where q is an accept state.) This completes the proof. □

There is a procedure that can be used to convert a given DFA into an equivalent regular expression. The idea behind this conversion process is similar to the idea of the proof above. It tends to get messy, producing rather large and complicated-looking regular expressions from relatively simple DFAs. It can be implemented by a computer, just like the conversion of an NFA to an equivalent DFA. Because I would like you to *reason about* computations, not *implement* computations, I will never test your ability to perform routine (and sometimes tedious) conversions like this.

Lecture 5

Proving languages to be nonregular

We already know that there exist languages $A \subseteq \Sigma^*$ that are nonregular, for any choice of an alphabet Σ. This is because there are *uncountably* many languages in total and only *countably* many regular languages, over any chosen alphabet Σ.

However, this observation does not give us a method to prove that specific nonregular languages are indeed nonregular when that is the case. In this lecture we will discuss a method that can be used to prove that a fairly wide selection of languages are nonregular.

5.1 The pumping lemma (for regular languages)

We will begin by proving a fairly simple fact—known as the *pumping lemma*—that must hold for all regular languages. A bit later in the lecture (in the section following this one) we will then use this fact to conclude that certain languages are nonregular.

Lemma 5.1 (Pumping lemma for regular languages)**.** *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language. There exists a positive integer n (called a* pumping length *of A) that possesses the following property. For every string $w \in A$ with $|w| \geq n$, it is possible to write $w = xyz$ for some choice of strings $x, y, z \in \Sigma^*$ such that*

1. *$y \neq \varepsilon$,*
2. *$|xy| \leq n$, and*
3. *$xy^iz \in A$ for all $i \in \mathbb{N}$.*

The pumping lemma is essentially a precise, technical way of expressing one simple consequence of the following fact:

> If a DFA with n or fewer states reads n or more symbols from an input string, at least one of its states must have been visited more than once.

This means that if a DFA with n states reads a particular string having length at least n, then there must be a substring of that input string that causes a loop—meaning that the DFA starts and ends on the same state. If the DFA accepts the original string, then by repeating that substring that caused a loop multiple times, or alternatively removing it altogether, we obtain a different string that is also accepted by the DFA. It may be helpful to try to match this intuition to the proof that follows.

Proof of Lemma 5.1. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA that recognizes A and let $n = |Q|$ be the number of states of M. We will prove that the property stated in the pumping lemma is satisfied for this choice of n.

Let us note first that if there is no string contained in A that has length n or larger, then there is nothing more we need to do: the property stated in the lemma is trivially satisfied in this case. We may therefore move on to the case in which A does contain at least one string having length at least n. In particular, suppose that $w \in A$ is a string such that $|w| \geq n$. We may write

$$w = \sigma_1 \cdots \sigma_m \tag{5.1}$$

for $m = |w|$ and $\sigma_1, \ldots, \sigma_m \in \Sigma$. Because $w \in A$ it must be the case that M accepts w, and therefore there exist states

$$r_0, r_1, \ldots, r_m \in Q \tag{5.2}$$

such that $r_0 = q_0$, $r_m \in F$, and

$$r_{k+1} = \delta(r_k, \sigma_{k+1}) \tag{5.3}$$

for every $k \in \{0, \ldots, m-1\}$.

Now, the sequence $r_0, r_1, \ldots, r_n$ has $n+1$ members, but there are only n different elements in Q, so at least one of the states of Q must appear more than once in this sequence. (This is an example of the so-called *pigeon hole principle*: if $n+1$ pigeons fly into n holes, then at least one of the holes must contain two or more pigeons.) Thus, there must exist indices $s, t \in \{0, \ldots, n\}$ satisfying $s < t$ such that $r_s = r_t$.

Next, define strings $x, y, z \in \Sigma^*$ as follows:

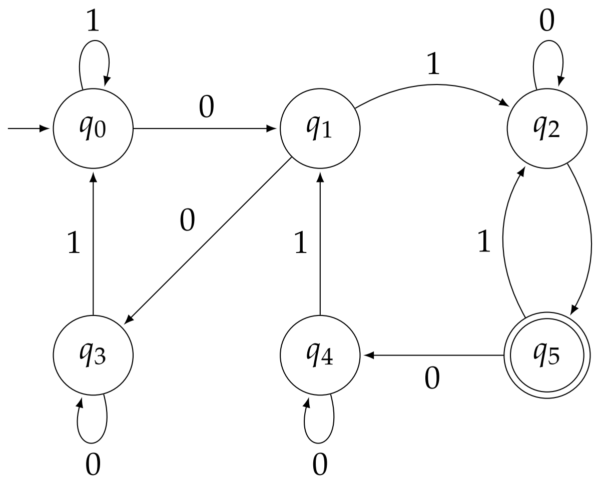
$$x = \sigma_1 \cdots \sigma_s, \qquad y = \sigma_{s+1} \cdots \sigma_t, \qquad z = \sigma_{t+1} \cdots \sigma_m. \tag{5.4}$$

It is the case that $w = xyz$ for this choice of strings, so to complete the proof, we just need to demonstrate that these strings fulfill the three conditions that are listed in the lemma. The first two conditions are immediate: we see that y has length $t - s$, which is at least 1 because $s < t$, and therefore $y \neq \varepsilon$; and we see that xy has length t, which is at most n because t was chosen from the set $\{0,\dots,n\}$. It remains to verify that $xy^iz \in A$, which is equivalent to M accepting xy^iz, for every $i \in \mathbb{N}$. That fact that xy^iz is accepted by M follows from the verification that the sequence of states

$$r_0,\ \dots,\ r_s,\ \underbrace{r_{s+1},\ \dots,\ r_t,}_{\text{repeated } i \text{ times}}\ r_{t+1},\ \dots,\ r_m \tag{5.5}$$

satisfies the definition of acceptance of the string xy^iz by the DFA M. □

If the proof of the pumping lemma, or the idea behind it, is not clear, it may be helpful to see it in action for an actual DFA and a long enough string accepted by that DFA. For instance, let us take M to be the DFA having the following state diagram:



Now consider any string w having length at least 6 (which is the number of states of M) that is accepted by M. For instance, let us take $w = 0110111$. This causes M to move through this sequence of states:

$$q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2 \xrightarrow{1} q_5 \xrightarrow{0} q_4 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{1} q_5 \tag{5.6}$$

(The arrows represent the transitions and the symbols above the arrows indicate which input symbol has caused this transition.) Sure enough, there is at least one state that appears multiple times in the sequence—in this particular case there are three such states: q_1, q_2, and q_5, each of which appear twice. Let us focus on the two appearances of the state q_1, just because this state happens to be the one that gets revisited first. It is the substring 1101 that causes M to move in a loop starting and ending on the state q_1. In the statement of the pumping lemma this corresponds to taking

$$x = 0, \qquad y = 1101, \qquad \text{and} \qquad z = 11. \tag{5.7}$$

Because the substring y causes M to move from the state q_1 back to q_1, it is as if reading y when M is in the state q_1 has no effect—so given that x causes M to move from the initial state q_0 to the state q_1, and z causes M to move from q_1 to an accept state, it holds that M must not only accept $w = xyz$, but it must also accept xz, $xyyz$, $xyyyz$, and so on.

There's nothing special about the example just described—something similar always happens. Pick any DFA whatsoever, and any string accepted by that DFA that has length at least the number of states of the DFA, and you will be able to find a loop like we did above, and by repeating input symbols in the most natural way so that the loop is followed multiple times (or no times) you will obtain different strings accepted by the DFA. This is essentially all that the pumping lemma is saying.

5.2 Using the pumping lemma to prove nonregularity

The pumping lemma is a statement about regular languages; it establishes a property that must always hold for any chosen regular language. We can use the pumping lemma to prove that certain languages are *not* regular using the technique of *proof by contradiction*. In particular, we take the following steps:

1. For A being the language we hope to prove is nonregular, we make the assumption that A *is* regular. Operating under the assumption that the language A is regular, we are free to apply the pumping lemma to it.
2. Using the property that the pumping lemma establishes for A, we derive a contradiction. Just about always the contradiction will be that we conclude that some particular string is contained in A that we know is actually not contained in A.
3. Having derived a contradiction, we conclude that it was our assumption that A is regular that led to the contradiction, and so we conclude that A is nonregular.

Let us see this method in action for a few examples. These examples will be stated as propositions, with the proofs showing you how the argument works.

Proposition 5.2. *Let $\Sigma = \{0,1\}$ be the binary alphabet and define a language over Σ as follows:*

$$A = \big\{0^m 1^m \,:\, m \in \mathbb{N}\big\}. \tag{5.8}$$

The language A is not regular.

Proof. Assume toward contradiction that A is regular. By the pumping lemma for regular languages, there must exist a pumping length $n \geq 1$ for A for which the property stated by that lemma holds. We will fix such a pumping length n for the remainder of the proof.

Define $w = 0^n 1^n$ (where n is the pumping length we just fixed). It holds that $w \in A$ and $|w| = 2n \geq n$, so the pumping lemma tells us that there exist strings $x, y, z \in \Sigma^*$ so that $w = xyz$ and the following conditions hold:

1. $y \neq \varepsilon$,
2. $|xy| \leq n$, and
3. $xy^i z \in A$ for all $i \in \mathbb{N}$.

Now, because $xyz = 0^n 1^n$ and $|xy| \leq n$, the substring y cannot have any 1s in it—the string xy is not long enough to reach the 1s. This means that $y = 0^k$ for some choice of $k \in \mathbb{N}$. Because $y \neq \varepsilon$, we may conclude moreover that $k \geq 1$. We may also conclude that

$$xy^2 z = xyyz = 0^{n+k} 1^n. \tag{5.9}$$

This is because $xyyz$ is the string obtained by inserting $y = 0^k$ somewhere in the initial portion of the string $xyz = 0^n 1^n$, before any 1s have appeared. More generally it holds that

$$xy^i z = 0^{n+(i-1)k} 1^n \tag{5.10}$$

for each $i \in \mathbb{N}$; we don't actually need this more general formula for the sake of the current proof, but in other similar cases it can be helpful.

However, because $k \geq 1$, we see that the string $xy^2 z = 0^{n+k} 1^n$ is *not* contained in A. This contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^i z \in A$ for all $i \in \mathbb{N}$. Having obtained a contradiction, we conclude that our assumption that A is regular was wrong. The language A is therefore nonregular, as required. □

Proposition 5.3. *Let $\Sigma = \{0, 1\}$ be the binary alphabet and define a language over Σ as follows:*

$$B = \big\{ 0^m 1^r \,:\, m, r \in \mathbb{N},\ m > r \big\}. \tag{5.11}$$

The language B is not regular.

Proof. Assume toward contradiction that B is regular. By the pumping lemma for regular languages, there must exist a pumping length $n \geq 1$ for B for which the property stated by that lemma holds. We will fix such a pumping length n for the remainder of the proof.

Define $w = 0^{n+1} 1^n$. It holds that $w \in B$ and $|w| = 2n + 1 \geq n$, so the pumping lemma tells us that there exist strings $x, y, z \in \Sigma^*$ so that $w = xyz$ and the following conditions hold:

1. $y \neq \varepsilon$,
2. $|xy| \leq n$, and
3. $xy^i z \in B$ for all $i \in \mathbb{N}$.

Now, because $xyz = 0^{n+1}1^n$ and $|xy| \leq n$, it must be that $y = 0^k$ for some choice of $k \geq 1$. (The reasoning here is just like in the previous proposition.) This time we have

$$xy^i z = 0^{n+1+(i-1)k}1^n \tag{5.12}$$

for each $i \in \mathbb{N}$. In particular, if we choose $i = 0$, then we have

$$xy^0 z = xz = 0^{n+1-k}1^n. \tag{5.13}$$

However, because $k \geq 1$, and therefore $n + 1 - k \leq n$, we see that the string xy^0z is *not* contained in B. This contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^i z \in B$ for all $i \in \mathbb{N}$.

Having obtained a contradiction, we conclude that our assumption that B is regular was wrong. The language B is therefore nonregular, as required. □

Remark 5.4. In the previous proof, it was important that we could choose $i = 0$ to get a contradiction—no other choice of i would have worked.

Proposition 5.5. *Let $\Sigma = \{0\}$ and define a language over Σ as follows:*

$$C = \big\{0^m \,:\, m \text{ is a perfect square (i.e., } m = k^2 \text{ for } k \in \mathbb{N})\big\}. \tag{5.14}$$

The language C is not regular.

Proof. Assume toward contradiction that C is regular. By the pumping lemma for regular languages, there must exist a pumping length $n \geq 1$ for C for which the property stated by that lemma holds. We will fix such a pumping length n for the remainder of the proof.

Define $w = 0^{n^2}$. It holds that $w \in C$ and $|w| = n^2 \geq n$, so the pumping lemma tells us that there exist strings $x, y, z \in \Sigma^*$ so that $w = xyz$ and the following conditions hold:

1. $y \neq \varepsilon$,
2. $|xy| \leq n$, and
3. $xy^i z \in C$ for all $i \in \mathbb{N}$.

There is only one symbol in the alphabet Σ, so this time it is immediate that $y = 0^k$ for some choice of $k \in \mathbb{N}$. Because $y \neq \varepsilon$ and $|y| \leq |xy| \leq n$ it must be the case that $1 \leq k \leq n$. It holds that

$$xy^iz = 0^{n^2+(i-1)k} \tag{5.15}$$

for each $i \in \mathbb{N}$. In particular, if we choose $i = 2$, then we have

$$xy^2z = xyyz = 0^{n^2+k}. \tag{5.16}$$

However, because $1 \leq k \leq n$, it cannot be that $n^2 + k$ is a perfect square—this is because $n^2 + k$ is larger than n^2, but the next perfect square after n^2 is

$$(n+1)^2 = n^2 + 2n + 1, \tag{5.17}$$

which is strictly larger than $n^2 + k$ because $k \leq n$. The string xy^2z is therefore *not* contained in C, which contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^iz \in C$ for all $i \in \mathbb{N}$.

Having obtained a contradiction, we conclude that our assumption that C is regular was wrong. The language C is therefore nonregular, as required. □

For the next example we will use some notation that will appear from time to time throughout the course. For a given string w, the string w^{R} denotes the *reverse* of the string w. Formally speaking, we may define the string reversal operation inductively as follows:

1. $\varepsilon^{\mathrm{R}} = \varepsilon$, and
2. $(\sigma w)^{\mathrm{R}} = w^{\mathrm{R}}\sigma$ for every $w \in \Sigma^*$ and $\sigma \in \Sigma$.

You can also just think of this operation intuitively as the string you get by reversing the order of the symbols in a string.

Proposition 5.6. *Let $\Sigma = \{0, 1\}$ and define a language over Σ as follows:*

$$D = \{w \in \Sigma^* : w = w^{\mathrm{R}}\}. \tag{5.18}$$

The language D is not regular.

Proof. Assume toward contradiction that D is regular. By the pumping lemma for regular languages, there must exist a pumping length $n \geq 1$ for D for which the property stated by that lemma holds. We will fix such a pumping length n for the remainder of the proof.

Define $w = 0^n10^n$. It holds that $w \in D$ and $|w| = 2n + 1 \geq n$, so the pumping lemma tells us that there exist strings $x, y, z \in \Sigma^*$ so that $w = xyz$ and the following conditions hold:

1. $y \neq \varepsilon$,
2. $|xy| \leq n$, and
3. $xy^iz \in D$ for all $i \in \mathbb{N}$.

As in the first two propositions in this section, we may conclude that $y = 0^k$ for $k \geq 1$. It holds that

$$xy^iz = 0^{n+(i-1)k}10^n \tag{5.19}$$

for each $i \in \mathbb{N}$. In particular, if we choose $i = 2$, then we have

$$xy^2z = xyyz = 0^{n+k}10^n. \tag{5.20}$$

Because $k \geq 1$, this string is not equal to its own reverse, and therefore xy^2z is therefore *not* contained in D. This contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^iz \in D$ for all $i \in \mathbb{N}$.

Having obtained a contradiction, we conclude that our assumption that D is regular was wrong. The language D is therefore nonregular, as required. □

The four propositions above should give you an idea of how the pumping lemma can be used to prove languages are nonregular. The set-up is always the same: we assume toward contradiction that a particular language is regular, and observe that the pumping lemma gives us a pumping length n. At that point it is time to choose the string w, try to use some reasoning, and derive a contradiction.

It may not always be clear what string w to choose or how exactly to get a contradiction—these steps will depend on the language you're working with, there may be multiple good choices for w, and there may be some creativity and/or insight involved in getting it all to work. One thing you can always try is to make a reasonable guess for what string might work, try to get a contradiction, and if you don't succeed then make another choice for w based on whatever insight you've gained by failing to get a contradiction from your first choice. Of course you should always be convinced by your own arguments and actively look for ways they might be going wrong—if you don't believe your own proof, it's not likely anyone else will believe it either.

5.3 Nonregularity from closure properties

Sometimes you can prove that a particular language is nonregular by combining together closure properties for regular languages with your knowledge of other languages being nonregular. Here are two examples, again stated as propositions. (In the proofs of these propositions, the languages B and D refer to the languages proved to be nonregular in the previous section.)

Proposition 5.7. *Let $\Sigma = \{0,1\}$ and define a language over Σ as follows:*

$$E = \bigl\{ w \in \Sigma^* \,:\, w \neq w^{\mathrm{R}} \bigr\}. \tag{5.21}$$

The language E is not regular.

Proof. Assume toward contradiction that E is regular. The regular languages are closed under complementation, and therefore $\overline{E}$ is regular. However, $\overline{E} = D$, which we already proved is nonregular. This is a contradiction, and therefore our assumption that E is regular was wrong. We conclude that E is nonregular, as claimed. □

Proposition 5.8. *Let $\Sigma = \{0,1\}$ and define a language over Σ as follows:*

$$F = \bigl\{ w \in \Sigma^* \,:\, w \text{ has more 0s than 1s} \bigr\}. \tag{5.22}$$

The language F is not regular.

Proof. Assume toward contradiction that F is regular. We know that the language $\mathrm{L}(0^*1^*)$ is regular because it is the language matched by a regular expression. The regular languages are closed under intersection, so $F \cap \mathrm{L}(0^*1^*)$ is regular. However, we have that

$$F \,\cap\, \mathrm{L}(0^*1^*) = B, \tag{5.23}$$

which we already proved is nonregular. This is a contradiction, and therefore our assumption that F is regular was wrong. We conclude that F is nonregular, as claimed. □

It is important to remember, when using this method, that it is the regular languages that are closed under operations such as complementation, intersection, union, and so on, not the nonregular languages. For instance, it is not generally the case that the intersection of two nonregular languages is nonregular—so your proof would not be valid if you were to rely on such a claim.

Lecture 6

Further discussion of regular languages

This is the last lecture to be devoted to regular languages, but we will refer back to regular languages frequently and relate them to various computational models and classes of languages throughout the course. For the most part we will use this lecture to relate some of the different concepts we have already discussed, introduce a few new concepts along the way, and go over some examples of problems concerning regular languages.

6.1 Other operations on languages

We've discussed some basic operations on languages, including the regular operations (union, concatenation, and star) and a few others (such as complementation and intersection). There are many other operations that one can consider—you could probably sit around all day thinking of increasingly obscure examples if you wanted to—but for now we'll take a look at just a few more.

Reverse

Suppose Σ is an alphabet and $w \in \Sigma^*$ is a string. The *reverse* of the string w, which we denote by w^{R}, is the string obtained by rearranging the symbols of w so that they appear in the opposite order. As we observed in the previous lecture, the reverse of a string may be defined inductively as follows:

1. If $w = \varepsilon$, then $w^{\mathrm{R}} = \varepsilon$.
2. If $w = \sigma x$ for $\sigma \in \Sigma$ and $x \in \Sigma^*$, then $w^{\mathrm{R}} = x^{\mathrm{R}}\sigma$.

Now suppose that $A \subseteq \Sigma^*$ is a language. We define the *reverse* of A, which we denote by A^{R}, to be the language obtained by taking the reverse of each element of A. That is, we define

$$A^{\mathrm{R}} = \bigl\{ w^{\mathrm{R}} \,:\, w \in A \bigr\}. \tag{6.1}$$

You can check that the following identities hold that relate the reverse operation to the regular operations:

$$(A \cup B)^{\mathrm{R}} = A^{\mathrm{R}} \cup B^{\mathrm{R}}, \quad (AB)^{\mathrm{R}} = B^{\mathrm{R}} A^{\mathrm{R}}, \quad \text{and} \quad (A^*)^{\mathrm{R}} = (A^{\mathrm{R}})^*. \tag{6.2}$$

A natural question concerning the reverse of a languages is this one:

If a language A is regular, must its reverse A^{R} also be regular?

The answer to this question is "yes." Let us state this fact as a proposition and then consider two ways to prove it.

Proposition 6.1. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language. The language A^{R} is regular.*

First proof. There is a natural way of defining the reverse of a regular expression that mirrors the identities (6.2) above. In particular, if S is a regular expression, then its reverse regular expression can be defined inductively as follows:

1. If $S = \varnothing$ then $S^{\mathrm{R}} = \varnothing$.
2. If $S = \varepsilon$ then $S^{\mathrm{R}} = \varepsilon$.
3. If $S = \sigma$ for some choice of $\sigma \in \Sigma$, then $S^{\mathrm{R}} = \sigma$.
4. If $S = (S_1 \cup S_2)$ for regular expressions S_1 and S_2, then $S^{\mathrm{R}} = (S_1^{\mathrm{R}} \cup S_2^{\mathrm{R}})$.
5. If $S = (S_1 S_2)$ for regular expressions S_1 and S_2, then $S^{\mathrm{R}} = (S_2^{\mathrm{R}} S_1^{\mathrm{R}})$.
6. If $S = (S_1^*)$ for a regular expression S_1, then $S^{\mathrm{R}} = \bigl((S_1^{\mathrm{R}})^*\bigr)$.

It is evident that $\mathrm{L}(S^{\mathrm{R}}) = \mathrm{L}(S)^{\mathrm{R}}$; for any regular expression S, the reverse regular expression S^{R} matches the reverse of the language matched by S.

Now, under the assumption that A is regular, there must exist a regular expression S such that $\mathrm{L}(S) = A$, because every regular language is matched by some regular expression. The reverse of the regular expression S is S^{R}, which is also a valid regular expression. The language matched by any regular expression is regular, and therefore $\mathrm{L}(S^{\mathrm{R}})$ is regular. Because $\mathrm{L}(S^{\mathrm{R}}) = A^{\mathrm{R}}$, we have that A^{R} is regular, as required. □

Second proof (sketch). (We'll consider this as a proof "sketch" because it just summarizes the main idea without covering the details of why it works.) Under the assumption that A is regular, there must exist a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $\mathrm{L}(M) = A$. We can design an NFA N such that $\mathrm{L}(N) = A^{\mathrm{R}}$, thereby implying that A^{R} is regular, by effectively running M backwards in time (using the power of nondeterminism to do this because deterministic computations are generally not reversible).

Here is the natural way to define an NFA N that does what we want:

$$N = (Q \cup \{r_0\}, \Sigma, \mu, r_0, \{q_0\}), \tag{6.3}$$

where it is assumed that r_0 is not contained in Q (i.e., we are letting N have the same states as M along with a new start state r_0), and we take the transition function μ to be defined as follows:

$$\begin{aligned} \mu(r_0, \varepsilon) &= F, & \mu(r_0, \sigma) &= \varnothing, \\ \mu(q, \varepsilon) &= \varnothing, & \mu(q, \sigma) &= \{p \in Q : \delta(p, \sigma) = q\}, \end{aligned} \tag{6.4}$$

for all $q \in Q$ and $\sigma \in \Sigma$.

The way N works is to first nondeterministically guess an accepting state of M, then it reads symbols from the input and nondeterministically chooses to move to a state for which M would allow a move in the opposite direction on the same input symbol, and finally it accepts if it ends on the start state of M.

The most natural way to formally prove that $\mathrm{L}(N) = \mathrm{L}(M)^{\mathrm{R}}$ is to refer to the definitions of acceptance for N and M, and to check that a sequence of states satisfies the definition for M accepting a string w if and only if the reverse of that sequence of states satisfies the definition of acceptance for N accepting w^{R}. □

Symmetric difference

Given two sets A and B, we define the *symmetric difference* of A and B as

$$A \triangle B = (A \backslash B) \cup (B \backslash A). \tag{6.5}$$

In words, the elements of the symmetric difference $A \triangle B$ are those objects that are contained in either A or B, but not both. Figure 6.1 illustrates the symmetric difference in the form of a Venn diagram.

It is not hard to conclude that if Σ is an alphabet and $A, B \subseteq \Sigma^*$ are regular languages, then the symmetric difference $A \triangle B$ of these two languages is also regular. This is because the regular languages are closed under the operations union, intersection, and complementation, and the symmetric difference can be described in

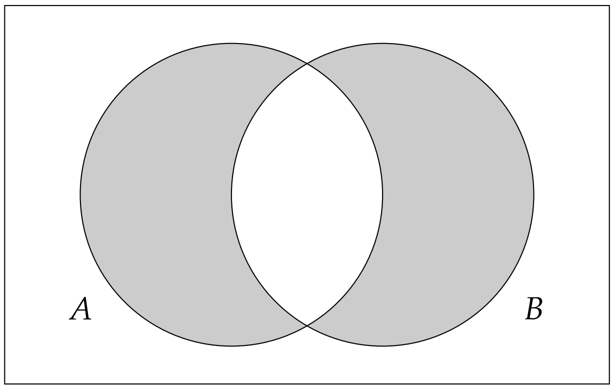


Figure 6.1: The shaded region denotes the symmetric difference $A \triangle B$ of two sets A and B.

terms of these operations. More specifically, if we assume that A and B are regular, then their complements $\overline{A}$ and $\overline{B}$ are also regular; which implies that the intersections $A \cap \overline{B}$ and $\overline{A} \cap B$ are also regular; and therefore the union $\bigl(A \cap \overline{B}\bigr) \cup \bigl(\overline{A} \cap B\bigr)$ of these two intersections is regular as well. Observing that we have

$$A \triangle B = \bigl(A \cap \overline{B}\bigr) \cup \bigl(\overline{A} \cap B\bigr), \tag{6.6}$$

we see that the symmetric difference of A and B is regular.

Prefix, suffix, and substring

Let Σ be an alphabet and let $w \in \Sigma^*$ be a string. A *prefix* of w is any string you can obtain from w by removing zero or more symbols from the right-hand side of w; a *suffix* of w is any string you can obtain by removing zero or more symbols from the left-hand side of w; and a *substring* of w is any string you can obtain by removing zero or more symbols from either or both the left-hand side and right-hand side of w. We can state these definitions more formally as follows: (i) a string $x \in \Sigma^*$ is a *prefix* of $w \in \Sigma^*$ if there exists $v \in \Sigma^*$ such that $w = xv$, (ii) a string $x \in \Sigma^*$ is a *suffix* of $w \in \Sigma^*$ if there exists $u \in \Sigma^*$ such that $w = ux$, and (iii) a string $x \in \Sigma^*$ is a *substring* of $w \in \Sigma^*$ if there exist $u, v \in \Sigma^*$ such that $w = uxv$.

For any language $A \subseteq \Sigma^*$, we will write $\operatorname{Prefix}(A)$, $\operatorname{Suffix}(A)$, and $\operatorname{Substring}(A)$ to denote the languages containing all prefixes, suffixes, and substrings (respectively) that can be obtained from any choice of a string $w \in A$. That is, we define

$$\operatorname{Prefix}(A) = \bigl\{x \in \Sigma^* \,:\, \text{there exists } v \in \Sigma^* \text{ such that } xv \in A\bigr\}, \tag{6.7}$$

$$\operatorname{Suffix}(A) = \bigl\{x \in \Sigma^* \,:\, \text{there exists } u \in \Sigma^* \text{ such that } ux \in A\bigr\}, \tag{6.8}$$

$$\operatorname{Substring}(A) = \bigl\{x \in \Sigma^* \,:\, \text{there exist } u, v \in \Sigma^* \text{ such that } uxv \in A\bigr\}. \tag{6.9}$$

Again we have a natural question concerning these concepts:

> If a language A is regular, must the languages $\text{Prefix}(A)$, $\text{Suffix}(A)$, and $\text{Substring}(A)$ also be regular?

The answer is "yes" as the following proposition states.

Proposition 6.2. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language over the alphabet Σ. The languages* $\text{Prefix}(A)$, $\text{Suffix}(A)$, *and* $\text{Substring}(A)$ *are regular.*

Proof. Because A is regular, there must exist a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $\text{L}(M) = A$. Some of the states in Q are *reachable* from the start state q_0, by following zero or more transitions specified by the transition function δ.[1] We may call this set R, so that

$$R = \{q \in Q : \text{there exists } w \in \Sigma^* \text{ such that } \delta^*(q_0, w) = q\}. \tag{6.10}$$

Also, from some of the states in Q, it is *possible to reach* an accept state of M, by following zero or more transitions specified by the transition function δ. We may call this set P, so that

$$P = \{q \in Q : \text{there exist } w \in \Sigma^* \text{ and } r \in F \text{ such that } \delta^*(q, w) = r\}. \tag{6.11}$$

(See Figure 6.2 for a simple example illustrating the definitions of these sets.)

First, define a DFA $M_{\text{Prefix}} = (Q, \Sigma, \delta, q_0, P)$. In words, M_{Prefix} is the same as M except that its accept states are all of the states in M from which it is possible to reach an accept state of M. It holds that $\text{L}(M_{\text{Prefix}}) = \text{Prefix}(A)$, and therefore $\text{Prefix}(A)$ is regular.

Next, define an NFA $N_{\text{Suffix}} = (Q \cup \{r_0\}, \Sigma, \eta, r_0, F)$, where the transition function η is defined as

$$\begin{aligned} \eta(r_0, \varepsilon) &= R, \\ \eta(q, \sigma) &= \{\delta(q, \sigma)\} \quad \text{(for each } q \in Q \text{ and } \sigma \in \Sigma\text{)}, \end{aligned}$$

and η takes the value $\varnothing$ in all other cases. In words, we define N_{Suffix} from M by adding a new start state r_0, along with ε-transitions from r_0 to every reachable state in M. It holds that $\text{L}(N_{\text{Suffix}}) = \text{Suffix}(A)$, and therefore $\text{Suffix}(A)$ is regular.

Finally, define an NFA $N_{\text{Substring}} = (Q \cup \{r_0\}, \Sigma, \eta, r_0, P)$, for η defined precisely as above. It holds that $\text{L}(N_{\text{Substring}}) = \text{Substring}(A)$ and therefore $\text{Substring}(A)$ is regular. □

[1] If you were defining a DFA for some purpose, there would be no point in having states that are not reachable from the start state—but there is nothing in the definition of DFAs that forces all states to be reachable.

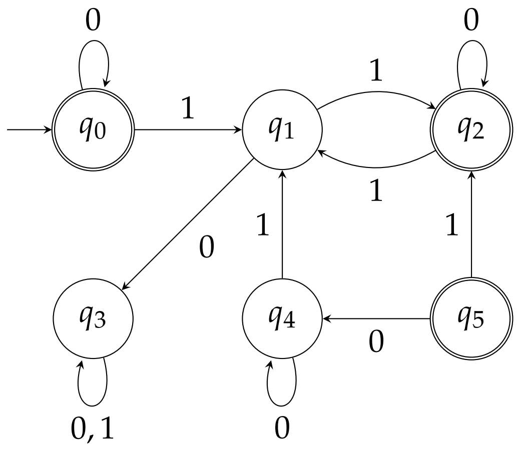


Figure 6.2: An example of a DFA M. In this case, the set R of reachable states is $R = \{q_0, q_1, q_2, q_3\}$ while the set P of states from which it is possible to reach an accepting state of M is $P = \{q_0, q_1, q_2, q_4, q_5\}$.

6.2 Example problems concerning regular languages

We will conclude with a few other examples of problems concerning regular languages along with their solutions.

Problem 6.1. Let $\Sigma = \{0, 1\}$ and let $A \subseteq \Sigma^*$ be an arbitrarily chosen regular language. Prove that the language

$$B = \{uv : u, v \in \Sigma^* \text{ and } u\sigma v \in A \text{ for some choice of } \sigma \in \Sigma\} \tag{6.12}$$

is regular. (Note that the language B can be described in intuitive terms as follows: it is the language of all strings that can be obtained by choosing a nonempty string w from A and deleting one symbol of w.)

Solution. A natural way to solve this problem is to describe an NFA for B, based on a DFA for A, which must exist by the assumption that A is regular. This will imply that B is regular, as every language recognized by an NFA is necessarily regular.

Along these lines, let us suppose that

$$M = (Q, \Sigma, \delta, q_0, F) \tag{6.13}$$

is a DFA for which it holds that $A = \mathrm{L}(M)$. Define an NFA

$$N = (P, \Sigma, \eta, p_0, G) \tag{6.14}$$

as follows. First, we will define

$$P = \{0, 1\} \times Q, \tag{6.15}$$

and we will take the start state of N to be

$$p_0 = (0, q_0). \tag{6.16}$$

The accept states of N will be

$$G = \{(1, q) \,:\, q \in F\}. \tag{6.17}$$

It remains to describe the transition function η of N, which will be as follows:

(i) $\eta((0, q), \sigma) = \{(0, \delta(q, \sigma))\}$ for every $q \in Q$ and $\sigma \in \Sigma$.

(ii) $\eta((0, q), \varepsilon) = \{(1, \delta(q, 0)), (1, \delta(q, 1))\}$ for every $q \in Q$.

(iii) $\eta((1, q), \sigma) = \{(1, \delta(q, \sigma))\}$ for every $q \in Q$ and $\sigma \in \Sigma$.

(iv) $\eta((1, q), \varepsilon) = \varnothing$ for every $q \in Q$.

The idea behind the way that N operates is as follows. The NFA N starts in the state $(0, q_0)$ and simulates M for some number of steps. This is the effect of the transitions listed as (i) above. At some point, which is nondeterministically chosen, N follows an ε-transition from a state of the form $(0, q)$ to either the state $(1, \delta(q, 0))$ or $(1, \delta(q, 1))$. Intuitively speaking, N is reading nothing from its input while "hypothesizing" that M has read some symbol σ (which is either 0 or 1). This is the effect of the transitions listed as (ii). Then N simply continues simulating M on the remainder of the input string, which is the effect of the transitions listed as (iii). There are no ε-transitions leading out of the states of the form $(1, q)$, which is why we have the values for η listed as (iv).

If you think about the NFA N for a moment or two, it should become evident that it recognizes B.

Alternative Solution. Here is a somewhat different solution that may appeal to some of you. Part of its appeal is that it illustrates a method that may be useful in other cases. In this case we will also discuss a somewhat more detailed proof of correctness (partly because it happens to be a bit easier for this solution).

Again, let

$$M = (Q, \Sigma, \delta, q_0, F) \tag{6.18}$$

be a DFA for which $\mathrm{L}(M) = A$. For each choice of $p, q \in Q$, define a new DFA

$$M_{p,q} = (Q, \Sigma, \delta, p, \{q\}), \tag{6.19}$$

and let $A_{p,q} = \mathrm{L}(M_{p,q})$. In words, $A_{p,q}$ is the regular language consisting of all strings that cause M to transition to the state q when started in the state p. For any choice of p, q, and r, we must surely have $A_{p,r}A_{r,q} \subseteq A_{p,q}$. Indeed, $A_{p,r}A_{r,q}$ represents all of the strings that cause M to transition from p to q, touching r somewhere along the way.

Now consider the language

$$\bigcup_{(p,\sigma,r)\in Q\times\Sigma\times F} A_{q_0,p}A_{\delta(p,\sigma),r}. \tag{6.20}$$

This is a regular language because each $A_{p,q}$ is regular and the regular languages are closed under finite unions and concatenations. To complete the solution, let us observe that the language above is none other than B:

$$B = \bigcup_{(p,\sigma,r)\in Q\times\Sigma\times F} A_{q_0,p}A_{\delta(p,\sigma),r}. \tag{6.21}$$

To prove this equality, we do the natural thing, which is to separate it into two separate set inclusions. First let us prove that

$$B \subseteq \bigcup_{(p,\sigma,r)\in Q\times\Sigma\times F} A_{q_0,p}A_{\delta(p,\sigma),r}. \tag{6.22}$$

Every string in B takes the form uv, for some choice of $u, v \in \Sigma^*$ and $\sigma \in \Sigma$ for which $u\sigma v \in A$. Let $p \in Q$ be the unique state for which $u \in A_{q_0,p}$, which we could alternatively describe as the state of M reached from the start state on input u, and let $r \in F$ be the unique state (which is necessarily an accepting state) for which $u\sigma v \in A_{q_0,r}$. As σ causes M to transition from p to $\delta(p,\sigma)$, it follows that v must cause M to transition from $\delta(p,\sigma)$ to r, i.e., $v \in A_{\delta(p,\sigma),r}$. It therefore holds that $uv \in A_{q_0,p}A_{\delta(p,\sigma),r}$, which implies the required inclusion.

Next we will prove that

$$\bigcup_{(p,\sigma,r)\in Q\times\Sigma\times F} A_{q_0,p}A_{\delta(p,\sigma),r} \subseteq B. \tag{6.23}$$

The argument is quite similar to the other inclusion just considered. Pick any choice of $p \in Q$, $\sigma \in \Sigma$, and $r \in F$. An element of $A_{q_0,p}A_{\delta(p,\sigma),r}$ must take the form uv for $u \in A_{q_0,p}$ and $v \in A_{\delta(p,\sigma),r}$. One finds that $u\sigma v \in A_{p_0,r} \subseteq A$, and therefore $uv \in B$, as required.

Problem 6.2. Let $\Sigma = \{0,1\}$ and let $A \subseteq \Sigma^*$ be an arbitrarily chosen regular language. Prove that the language

$$C = \{vu \,:\, u, v \in \Sigma^* \text{ and } uv \in A\} \tag{6.24}$$

is regular.

Solution. Again, a natural way to solve this problem is to give an NFA for C. Let us assume

$$M = (Q, \Sigma, \delta, q_0, F) \tag{6.25}$$

is a DFA for which $\mathrm{L}(M) = A$, like we did above. This time our NFA will be slightly more complicated. In particular, let us define

$$N = (P, \Sigma, \eta, p_0, G) \tag{6.26}$$

as follows. First, we will define

$$P = \bigl(\{0,1\} \times Q \times Q\bigr) \cup \{p_0\}, \tag{6.27}$$

for p_0 being a special start state of N that is not contained in $\{0,1\} \times Q \times Q$. The accept states of N will be

$$G = \{(1, q, q) \,:\, q \in Q\}. \tag{6.28}$$

It remains to describe the transition function η of N, which will be as follows:

(i) $\eta(p_0, \varepsilon) = \{(0, q, q) \,:\, q \in Q\}$.

(ii) $\eta((0, r, q), \sigma) = \{(0, \delta(r, \sigma), q)\}$ for all $q, r \in Q$ and $\sigma \in \Sigma$.

(iii) $\eta((0, r, q), \varepsilon) = \{(1, q_0, q)\}$ for every $r \in F$ and $q \in Q$.

(iv) $\eta((1, r, q), \sigma) = \{(1, \delta(r, \sigma), q)\}$ for all $q, r \in Q$ and $\sigma \in \Sigma$.

All other values of η that have not been listed are to be understood as $\varnothing$.

You might look at this definition and have no idea how N works, so let's consider it in more detail. N starts out in the start state p_0, and the only thing it can do is to make a guess for some state of the form $(0, q, q)$ to jump to. The idea is that the 0 indicates that N is entering the first phase of its computation, in which it will read a portion of its input string corresponding to v in the definition of C. It jumps to any state q of M, but it also *remembers* which state it jumped to. Every state N ever moves to from this point on will have the form (a, r, q) for some $a \in \{0,1\}$ and $r \in Q$, but for the same q that it first jumped to; the third coordinate q represents the memory of where it first jumped, and it will never forget or change this part of its state. Intuitively speaking, the state q is a guess made by N for the state that M would be on after reading u (which N hasn't seen yet, so it's just a guess).

Then, N starts reading symbols and essentially mimicking M on those input symbols—this is the point of the transitions listed as (ii). At some point, nondeterministically chosen, N decides that it's time to move to the second phase of its computation, reading the second part of its input, which corresponds to the string

u in the definition of C. It can only make this nondeterministic move, from a state of the form $(0, r, q)$ to $(1, q_0, q)$, when r is an accepting state of M. The reason is that N only wants to accept vu when M accepts uv, so M should be in the initial state at the start of u and in an accepting state at the end of v. This is the point of the transitions listed as (iii). Finally, in the second phase of its computation, N simulates M on the second part of its input, which corresponds to the string u. It accepts only for states of the form $(1, q, q)$, because those are the states that indicate that N made the correct guess on its first step for the state that M would be in after reading u.

This is just an intuitive description, not a formal proof. It is the case, however, that $\mathrm{L}(N) = C$, as a low-level, formal proof would reveal, which implies that C is regular.

Alternative Solution. Again, there is another solution along the same lines as the alternative solution to the previous problem. This time it's actually a much easier solution. Let M be a DFA for A, precisely as above, and define $A_{p,q}$ for each $p, q \in Q$ as in the alternative solution to the previous problem. The language

$$\bigcup_{(p,r)\in Q\times F} A_{p,r}A_{q_0,p} \tag{6.29}$$

is regular, again by the closure of the regular languages under finite unions and concatenations. It therefore suffices to prove

$$C = \bigcup_{(p,r)\in Q\times F} A_{p,r}A_{q_0,p}. \tag{6.30}$$

By definition, every element of C may be expressed as vu for $u, v \in \Sigma^*$ satisfying $uv \in A$. Let $p \in Q$ and $r \in F$ be the unique states for which $u \in A_{q_0,p}$ and $uv \in A_{q_0,r}$. It follows that $v \in A_{p,r}$, and therefore $vu \in A_{p,r}A_{q_0,p}$, implying

$$C \subseteq \bigcup_{(p,r)\in Q\times F} A_{p,r}A_{q_0,p}. \tag{6.31}$$

Along similar lines, for any choice of $p \in Q$, $r \in F$, $u \in A_{q_0,p}$, and $v \in A_{p,r}$ it holds that $uv \in A_{q_0,p}A_{p,r} \subseteq A$, and therefore $vu \in C$, from which the inclusion

$$\bigcup_{(p,r)\in Q\times F} A_{p,r}A_{q_0,p} \subseteq C \tag{6.32}$$

follows.

The final problem demonstrates that closure properties holding for all regular languages may fail for nonregular languages. In particular, the nonregular languages are not closed under the regular operations.

Problem 6.3. For each of the following statements, give specific examples of languages over some alphabet Σ for which the statements are satisfied.

(a) There exist nonregular languages $A, B \subseteq \Sigma^*$ such that $A \cup B$ is regular.

(b) There exist nonregular languages $A, B \subseteq \Sigma^*$ such that AB is regular.

(c) There exists a nonregular language $A \subseteq \Sigma^*$ such that A^* is regular.

Solution. For statement (a), let us let $\Sigma = \{0\}$, let $A \subseteq \Sigma^*$ be any nonregular language whatsoever, such as $A = \{0^n : n \text{ is a perfect square}\}$, and let $B = \overline{A}$. We know that B is also nonregular (because if it were regular, then its complement would also be regular, but its complement is A which we know is nonregular). On the other hand, $A \cup B = \Sigma^*$, which is regular.

For statement (b), let us let $\Sigma = \{0\}$, and let us start by taking $C \subseteq \Sigma^*$ to be any nonregular language (such as $C = \{0^n : n \text{ is a perfect square}\}$). Then let us take

$$A = C \cup \{\varepsilon\} \quad \text{and} \quad B = \overline{C} \cup \{\varepsilon\}. \tag{6.33}$$

The languages A and B are nonregular, by virtue of the fact that C is nonregular (and therefore $\overline{C}$ is nonregular as well). On the other hand, $AB = \Sigma^*$, which is regular.

Finally, for statement (c), let us again take $\Sigma = \{0\}$, and let $A \subseteq \Sigma^*$ be any nonregular language that contains the single-symbol string 0. (Again, the language $A = \{0^n : n \text{ is a perfect square}\}$ will work.) We have that A is nonregular, but $A^* = \Sigma^*$, which is regular.

Lecture 7

Context-free grammars and languages

The next class of languages we will study in the course is the class of *context-free languages*. They are defined by the notion of a *context-free grammar*, or a CFG for short, which you will have encountered previously in your studies (such as in CS 241).

7.1 Definitions of CFGs and CFLs

We will start with the following definition for context-free grammars.

Definition 7.1. A *context-free grammar* (or *CFG* for short) is a 4-tuple

$$G = (V, \Sigma, R, S), \tag{7.1}$$

where V is a finite and non-empty set (whose elements we will call *variables*), Σ is an *alphabet* (disjoint from V), R is a finite and nonempty set of *rules*, each of which takes the form

$$A \rightarrow w \tag{7.2}$$

for some choice of $A \in V$ and $w \in (V \cup \Sigma)^*$, and $S \in V$ is a variable called the *start variable*.

Example 7.2. For our first example of a CFG, we may consider $G = (V, \Sigma, R, S)$, where $V = \{S\}$ (so that there is just one variable in this grammar), $\Sigma = \{0, 1\}$, S is the start variable, and R contains these two rules:

$$\begin{aligned} S &\rightarrow 0\,S\,1 \\ S &\rightarrow \varepsilon. \end{aligned} \tag{7.3}$$

It is often convenient to describe a CFG just by listing the rules, as we have in (7.3). When we do this, it is to be understood that the set of variables V and the alphabet Σ are determined implicitly: the variables are the capital letters and the alphabet contains the symbols on the right-hand side of the rules that are left over. Moreover, the start variable is understood to be the variable appearing on the left-hand side of the first rule that is listed.

Note that these are just conventions that allow us to save time, and you could simply list each of the elements V, Σ, R, and S if it was likely that the conventions would cause confusion (such as a case in which you might prefer a capital letter to be treated as an alphabet symbol, or to have a variable that doesn't appear in any rules).

Every context-free grammar $G = (V, \Sigma, R, S)$ *generates* a language $\mathrm{L}(G) \subseteq \Sigma^*$. Informally speaking, this is the language consisting of *all* strings that can be obtained by the following process:

1. Write down the start variable S.
2. Repeat the following steps any number of times:
   a. Choose any rule $A \rightarrow w$ from R.
   b. Within the string of variables and alphabet symbols you currently have written down, replace any instance of the variable A with the string w.
3. If you are eventually left with a string of the form $x \in \Sigma^*$, so that no variables remain, then stop. The string x has been obtained by the process, and is therefore among the strings generated by G.

Example 7.3. The CFG G described in Example 7.2 generates the language

$$\mathrm{L}(G) = \{0^n 1^n \,:\, n \in \mathbb{N}\}. \tag{7.4}$$

This is because we begin by writing down the start variable S, then we choose one of the two rules and perform the replacement in the only way possble: there will always be a single variable S in the middle of the string, and we replace it either by $0S1$ or by ε. The process ends precisely when we choose the rule $S \rightarrow \varepsilon$, and depending on how many times we chose the rule $S \rightarrow 0S1$ we obtain one of the strings

$$\varepsilon,\ 01,\ 0011,\ 000111,\ \ldots \tag{7.5}$$

and so on. The set of all strings that can possibly be obtained is therefore given by (7.4).

The description of the language generated by a CFG suggested above provides an intuitive, human-readable way to explain this concept, but it is not very satisfying from a mathematical viewpoint—we would prefer a definition based on sets, functions, and so on (rather than one that refers to "writing down" variables, for instance). One way to define this notion mathematically begins with the specification of the *yields relation* of a grammar that captures the notion of performing a substitution.

Definition 7.4. Let $G = (V, \Sigma, R, S)$ be a context-free grammar. The *yields relation* defined by G is a relation defined for pairs of strings over the alphabet $V \cup \Sigma$ as follows:

$$uAv \Rightarrow_G uwv \tag{7.6}$$

for every choice of strings $u, v, w \in (V \cup \Sigma)^*$ and a variable $A \in V$, provided that the rule $A \rightarrow w$ is included in R.[1]

The interpretation of this relation is that $x \Rightarrow_G y$, for $x, y \in (V \cup \Sigma)^*$, when it is possible to replace one of the variables appearing in x according to one of the rules of G in order to obtain y.

It will also be convenient to consider the reflexive transitive closure of this relation, which is defined as follows.

Definition 7.5. Let $G = (V, \Sigma, R, S)$ be a context-free grammar. For any two strings $x, y \in (V \cup \Sigma)^*$ it holds that

$$x \overset{*}{\Rightarrow}_G y \tag{7.7}$$

if there exists a positive integer m and strings $z_1, \ldots, z_m \in (V \cup \Sigma)^*$ such that $x = z_1$, $y = z_m$, and $z_k \Rightarrow_G z_{k+1}$ for all $k \in \{1, \ldots, m-1\}$.

In this case the interpretation of this relation is that $x \overset{*}{\Rightarrow}_G y$ holds when it is possible to transform x into y by performing zero or more substitutions according to the rules of G.

When a CFG G is fixed or can be safely taken as implicit, we will sometimes write $\Rightarrow$ rather than $\Rightarrow_G$, and likewise for the starred version.

We can now use the relation just defined to formally define the language generated by a given context-free grammar.

Definition 7.6. Let $G = (V, \Sigma, R, S)$ be a context-free grammar. The *language generated* by G is

$$\mathrm{L}(G) = \bigl\{x \in \Sigma^* \,:\, S \overset{*}{\Rightarrow}_G x\bigr\}. \tag{7.8}$$

[1] Recall that a relation is a subset of a Cartesian product of two sets. In this case, the relation is the subset $\{(uAv,\, uwv) \,:\, u, v, w \in (V \cup \Sigma)^*,\ A \in V, \text{ and } A \rightarrow w \text{ is a rule in } R\}$. The notation $uAv \Rightarrow_G uwv$ is a more readable way of indicating that the pair (uAv, uwv) is an element of the relation.

If it is the case that $x \in \mathrm{L}(G)$ for a context-free grammar $G = (V, \Sigma, R, S)$, and $z_1, \ldots, z_m \in (V \cup \Sigma)^*$ is a sequence of strings for which it holds that $z_1 = S$, $z_m = x$, and $z_k \Rightarrow_G z_{k+1}$ for all $k \in \{1, \ldots, m-1\}$, then the sequence $z_1, \ldots, z_m$ is said to be a *derivation* of x. If you unravel the definitions above, it becomes clear that there must of course exist at least one derivation for every string $x \in \mathrm{L}(G)$, but in general there may be more than one.

Finally, we define the class of *context-free languages* to be those languages that are generated by context-free grammars.

Definition 7.7. Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language A is *context-free* if there exists a context-free grammar G such that $\mathrm{L}(G) = A$.

Example 7.8. The language $\{0^n 1^n : n \in \mathbb{N}\}$ is a context-free language, as has been established in Example 7.3.

7.2 Basic examples

We've seen one example of a context-free language so far: $\{0^n 1^n : n \in \mathbb{N}\}$. Let us now consider a few more examples.

Example 7.9. The language

$$\mathrm{PAL} = \left\{ w \in \Sigma^* : w = w^{\mathrm{R}} \right\} \tag{7.9}$$

over the alphabet $\Sigma = \{0, 1\}$ is context-free. (In fact this is true for any choice of an alphabet Σ, but we'll stick to the binary alphabet for now for simplicity). To verify that this language is context-free, it suffices to exhibit a context-free grammar that generates it. Here is one that works:

$$\begin{aligned} S &\rightarrow 0\,S\,0 \\ S &\rightarrow 1\,S\,1 \\ S &\rightarrow 0 \\ S &\rightarrow 1 \\ S &\rightarrow \varepsilon \end{aligned} \tag{7.10}$$

We've named the language in the above example PAL because it is short for *palindrome*, which is something that reads the same forwards and backwards (like "Never odd or even" or "Yo, banana boy," so long as you ignore punctuation, spaces, and letter case—although in this example we're restricting our attention to binary string palindromes).

We often use a short-hand notation for describing grammars in which the same variable appears on the left-hand side of multiple rules, as is the case for the grammar described in the previous example. The short-hand notation is to write the variable on the left-hand side and the arrow just once, and to draw a vertical bar (which can be read as "or") among the possible alternatives for the right-hand side like this:

$$S \rightarrow 0\,S\,0 \;\big|\; 1\,S\,1 \;\big|\; 0 \;\big|\; 1 \;\big|\; \varepsilon \tag{7.11}$$

(If you use this short-hand notation when you are writing by hand, such as on an exam, be sure to make your bars tall enough so that they are easily distinguished from 1s.)

Sometimes it is easy to see that a particular CFG generates a given language—for instance, I would consider this to be obvious in the case of the previous example. In other cases it can be more challenging, or even impossibly difficult, to verify that a particular grammar generates a particular language. The next example illustrates a case in which such a verification is nontrivial.

Example 7.10. Let $\Sigma = \{0, 1\}$ be the binary alphabet, and define a language $A \subseteq \Sigma^*$ as follows:

$$A = \big\{ w \in \Sigma^* \,:\, |w|_0 = |w|_1 \big\}. \tag{7.12}$$

Here we are using a convenient notation: $|w|_0$ denotes the number of times the symbol 0 appears in w, and similarly $|w|_1$ denotes the number of times the symbol 1 appears in w. The language A therefore contains all binary strings having the same number of 0s and 1s. This is a context-free language, as it is generated by this context-free grammar:

$$S \rightarrow 0\,S\,1\,S \;\big|\; 1\,S\,0\,S \;\big|\; \varepsilon. \tag{7.13}$$

Now, it is pretty clear that every string generated by the grammar (which we will call G) described in the above example is contained in A; we begin a derivation with just the variable S, so there are an equal number of 0s and 1s written down at the start (zero of each, to be precise), and every rule maintains this property as an invariant.

On the other hand, it is not immediately obvious that every element of A can be generated by G. Let us prove that this is indeed the case.

Claim 7.11. $A \subseteq \mathrm{L}(G)$.

Proof. Let $w \in A$ be a string contained in A and let $n = |w|$. We will prove that $w \in \mathrm{L}(G)$ by (strong) induction on n.

The base case is $n = 0$, which means that $w = \varepsilon$. We have that $S \Rightarrow_G \varepsilon$ represents a derivation of ε, and therefore $w \in \mathrm{L}(G)$.

For the induction step, we assume that $n \geq 1$, and we assume that for all strings $x \in A$ with $|x| < n$ it holds that $x \in \mathrm{L}(G)$. Our goal is to prove that G generates w. Let us write

$$w = \sigma_1 \cdots \sigma_n \tag{7.14}$$

for $\sigma_1, \ldots, \sigma_n \in \Sigma$. We have assumed that $w \in A$, and therefore

$$|\sigma_1 \cdots \sigma_n|_0 = |\sigma_1 \cdots \sigma_n|_1. \tag{7.15}$$

Next, let $m \in \{1, \ldots, n\}$ be the *minimum* value for which it holds that

$$|\sigma_1 \cdots \sigma_m|_0 = |\sigma_1 \cdots \sigma_m|_1; \tag{7.16}$$

we know that this equation is satisfied when $m = n$, and there might be a smaller value of m that works—but in any case we know that m is a well-defined number. We will prove that $\sigma_1 \neq \sigma_m$.

We can reason that $\sigma_1 \neq \sigma_m$ using proof by contradiction. Toward this goal, assume $\sigma_1 = \sigma_m$, and define

$$d_k = |\sigma_1 \cdots \sigma_k|_1 - |\sigma_1 \cdots \sigma_k|_0 \tag{7.17}$$

for every $k \in \{1, \ldots, m\}$. We know that $d_m = 0$ because (7.16) holds. Moreover, because the equations

$$\begin{aligned} |\sigma_1 \cdots \sigma_m|_1 &= |\sigma_1 \cdots \sigma_{m-1}|_1 + |\sigma_m|_1 = |\sigma_1 \cdots \sigma_{m-1}|_1 + |\sigma_1|_1 \\ |\sigma_1 \cdots \sigma_m|_0 &= |\sigma_1 \cdots \sigma_{m-1}|_0 + |\sigma_m|_0 = |\sigma_1 \cdots \sigma_{m-1}|_0 + |\sigma_1|_0 \end{aligned} \tag{7.18}$$

hold, we conclude that

$$d_m = d_{m-1} + d_1 \tag{7.19}$$

by subtracting the second equation from the first. Therefore, because $d_m = 0$ and d_1 is nonzero, it must be that d_{m-1} is also nonzero, and more importantly d_1 and d_{m-1} must have opposite sign. However, because consecutive values of d_k must always differ by 1 and can only take integer values, we conclude that there must exist a choice of k in the range $\{2, \ldots, m-2\}$ for which $d_k = 0$, for otherwise it would not be possible for d_1 and d_{m-1} to have opposite sign. This, however, is in contradiction with m being the minimum value for which (7.16) holds. We have therefore concluded that $\sigma_1 \neq \sigma_m$.

At this point it is possible to describe a derivation for w. We have $w = \sigma_1 \cdots \sigma_n$, and we have that

$$|\sigma_1 \cdots \sigma_m|_0 = |\sigma_1 \cdots \sigma_m|_1 \quad \text{and} \quad \sigma_1 \neq \sigma_m \tag{7.20}$$

for some choice of $m \in \{1, \dots, n\}$. We conclude that

$$|\sigma_2 \cdots \sigma_{m-1}|_0 = |\sigma_2 \cdots \sigma_{m-1}|_1 \quad \text{and} \quad |\sigma_{m+1} \cdots \sigma_n|_0 = |\sigma_{m+1} \cdots \sigma_n|_1. \tag{7.21}$$

By the hypothesis of induction it follows that

$$S \overset{*}{\Rightarrow}_G \sigma_2 \cdots \sigma_{m-1} \quad \text{and} \quad S \overset{*}{\Rightarrow}_G \sigma_{m+1} \cdots \sigma_n. \tag{7.22}$$

Therefore the string w satisfies

$$S \Rightarrow_G 0\,S\,1\,S \overset{*}{\Rightarrow}_G 0\,\sigma_2 \cdots \sigma_{m-1}\,1\,\sigma_{m+1} \cdots \sigma_n = w \tag{7.23}$$

(in case $\sigma_1 = 0$ and $\sigma_m = 1$) or

$$S \Rightarrow_G 1\,S\,0\,S \overset{*}{\Rightarrow}_G 1\,\sigma_2 \cdots \sigma_{m-1}\,0\,\sigma_{m+1} \cdots \sigma_n = w \tag{7.24}$$

(in case $\sigma_1 = 1$ and $\sigma_m = 0$). We have proved that $w \in \mathrm{L}(G)$ as required. □

Here is another example that is related to the previous one. It is an important example and we'll refer to it from time to time throughout the course.

Example 7.12. Consider the alphabet $\Sigma = \{\,(\,,\,)\,\}$. That is, we have two symbols in this alphabet: left-parenthesis and right-parenthesis.

To say that a string w over the alphabet Σ is *properly balanced* means that by repeatedly removing the substring ( ), you can eventually reach ε. More intuitively speaking, a string over Σ is properly balanced if it would make sense to use this pattern of parentheses in an ordinary arithmetic expression (ignoring everything besides the parentheses). These are examples of properly balanced strings:

$$(\,(\,)\,(\,)\,)\,(\,), \quad (\,(\,(\,)\,(\,)\,)\,), \quad (\,(\,)\,), \quad \text{and} \quad \varepsilon. \tag{7.25}$$

These are examples of strings that are not properly balanced:

$$(\,(\,(\,)\,)\,(\,), \quad \text{and} \quad (\,)\,)\,(. \tag{7.26}$$

Now define a language

$$\mathrm{BAL} = \{w \in \Sigma^* : w \text{ is properly balanced}\}. \tag{7.27}$$

It is the case that the language BAL is context-free—here is a very simple context-free grammar that generates it:

$$S \rightarrow (\,S\,)\,S \mid \varepsilon. \tag{7.28}$$

Lecture 8

Parse trees, ambiguity, and Chomsky normal form

In this lecture we will discuss a few important notions connected with context-free grammars, including *parse trees*, *ambiguity*, and a special form for context-free grammars known as the *Chomsky normal form*.

8.1 Left-most derivations and parse trees

In the previous lecture we covered the definition of *context-free grammars* as well as *derivations* of strings by context-free grammars. Let us consider one of the context-free grammars from the previous lecture:

$$S \rightarrow 0\,S\,1\,S \;\big|\; 1\,S\,0\,S \;\big|\; \varepsilon. \tag{8.1}$$

Again we'll call this CFG G, and as we proved last time we have

$$\mathrm{L}(G) = \big\{ w \in \Sigma^* \,:\, |w|_0 = |w|_1 \big\}, \tag{8.2}$$

where $\Sigma = \{0,1\}$ is the binary alphabet and $|w|_0$ and $|w|_1$ denote the number of times the symbols 0 and 1 appear in w, respectively.

Left-most derivations

Here is an example of a derivation of the string 0101:

$$S \Rightarrow 0\,S\,1\,S \Rightarrow 0\,1\,S\,0\,S\,1\,S \Rightarrow 0\,1\,0\,S\,1\,S \Rightarrow 0\,1\,0\,1\,S \Rightarrow 0\,1\,0\,1. \tag{8.3}$$

This is an example of a *left-most derivation*, which means that it is always the left-most variable that gets replaced at each step. For the first step there is only one variable that can possibly be replaced—this is true both in this example and in general. For the second step, however, one could choose to replace either of the occurrences of the variable S, and in the derivation above it is the left-most occurrence that gets replaced. That is, if we underline the variable that gets replaced and the symbols and variables that replace it, we see that this step replaces the left-most occurrence of the variable S:

$$0\,\underline{S}\,1\,S \Rightarrow 0\,\underline{1\,S\,0\,S}\,1\,S. \tag{8.4}$$

The same is true for every other step—always we choose the left-most variable occurrence to replace, and that is why we call this a left-most derivation. The same terminology is used in general, for any context-free grammar.

If you think about it for a moment, you will quickly realize that every string that can be generated by a particular context-free grammar can also be generated by that same grammar using a left-most derivation. This is because there is no "interaction" among multiple variables and/or symbols in any context-free grammar derivation; if we know which rule is used to substitute each variable, then it doesn't matter what order the variable occurrences are substituted, so you might as well always take care of the left-most variable during each step.

We could also define the notion of a *right-most derivation*, in which the right-most variable occurrence is always evaluated first—but there isn't really anything important about right-most derivations that isn't already represented by the notion of a left-most derivation, at least from the viewpoint of this course. For this reason, we won't have any reason to discuss right-most derivations further.

Parse trees

With any derivation of a string by a context-free grammar we may associate a tree, called a *parse tree*, according to the following rules:

- We have one node of the tree for each new occurrence of either a variable, a symbol, or an ε in the derivation, with the root node of the tree corresponding to the start variable. (We only have nodes labelled ε when rules of the form $V \to \varepsilon$ are applied.)
- Each node corresponding to a symbol or an ε is a leaf node (having no children), while each node corresponding to a variable has one child for each symbol or variable with which it is replaced. The children of each (variable) node are ordered in the same way as the symbols and variables in the rule used to replace that variable.

For example, the derivation (8.3) yields the parse tree illustrated in Figure 8.1.

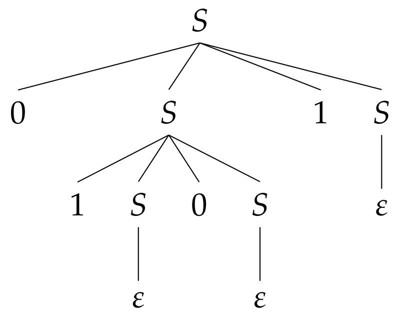


Figure 8.1: The parse tree corresponding to the derivation (8.3) of the string 0101.

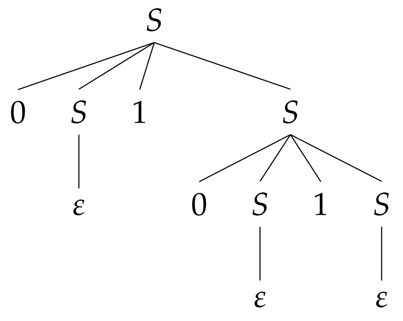


Figure 8.2: A parse tree corresponding to the derivation (8.5) of the string 0101.

There is a one-to-one and onto correspondence between parse trees and left-most derivations, meaning that every parse tree uniquely determines a left-most derivation and each left-most derivation uniquely determines a parse tree.

8.2 Ambiguity

Sometimes a context-free grammar will allow multiple parse trees (or, equivalently, multiple left-most derivations) for some strings in the language that it generates. For example, a different left-most derivation of the string 0101 by the CFG (8.1) from the derivation (8.3) is given by

$$S \Rightarrow 0\,S\,1\,S \Rightarrow 0\,1\,S \Rightarrow 0\,1\,0\,S\,1\,S \Rightarrow 0\,1\,0\,1\,S \Rightarrow 0\,1\,0\,1. \tag{8.5}$$

The parse tree corresponding to this derivation is illustrated in Figure 8.2.

When it is the case, for a given context-free grammar G, that there exists at least one string $w \in \mathrm{L}(G)$ having at least two different parse trees, then the CFG G is said to be *ambiguous*. Note that this is so even if there is just a single string having multiple parse trees—in order to be *unambiguous*, a CFG must have just a single, unique parse tree for each string it generates.

Being unambiguous is generally considered to be a positive attribute of a CFG, and indeed it is a requirement for some applications of context-free grammars.

Designing unambiguous CFGs

In some cases it is possible to come up with an unambiguous context-free grammar that generates the same language as an ambiguous context-free grammar. For example, we can come up with a different context-free grammar for the language

$$\{w \in \{0,1\}^* : |w|_0 = |w|_1\} \tag{8.6}$$

that, unlike the CFG (8.1), is unambiguous. Here is such a CFG:

$$\begin{aligned} S &\rightarrow 0\,X\,1\,S \;\big|\; 1\,Y\,0\,S \;\big|\; \varepsilon \\ X &\rightarrow 0\,X\,1\,X \;\big|\; \varepsilon \\ Y &\rightarrow 1\,Y\,0\,Y \;\big|\; \varepsilon \end{aligned} \tag{8.7}$$

We won't take the time to go through a proof that this CFG is unambiguous—but if you think about it for a few moments, it shouldn't be too hard to convince yourself that it is unambiguous. The variable X generates strings having the same number of 0s and 1s, where the number of 1s never exceeds the number of 0s, and the variable Y is similar except the role of the 0s and 1s is reversed. If you try to generate a particular string, you'll never have more than one option as to which rule to apply, assuming you restrict your attention to left-most derivations.

Here is another example of how an ambiguous CFG can be modified to make it unambiguous. Let us define an alphabet

$$\Sigma = \{a, b, +, *, (\,,\,)\} \tag{8.8}$$

along with a CFG

$$S \rightarrow S + S \;\big|\; S * S \;\big|\; (\,S\,) \;\big|\; a \;\big|\; b \tag{8.9}$$

This grammar generates strings that look like arithmetic expressions in variables a and b, where we allow the operations $*$ and $+$, along with parentheses. For instance, the string

$$(a + b) * a + b \tag{8.10}$$

is such an expression, and we can generate it (for instance) as follows:

$$\begin{aligned} S &\Rightarrow S * S \Rightarrow (S) * S \Rightarrow (S + S) * S \Rightarrow (a + S) * S \Rightarrow (a + b) * S \\ &\Rightarrow (a + b) * S + S \Rightarrow (a + b) * a + S \Rightarrow (a + b) * a + b. \end{aligned} \tag{8.11}$$

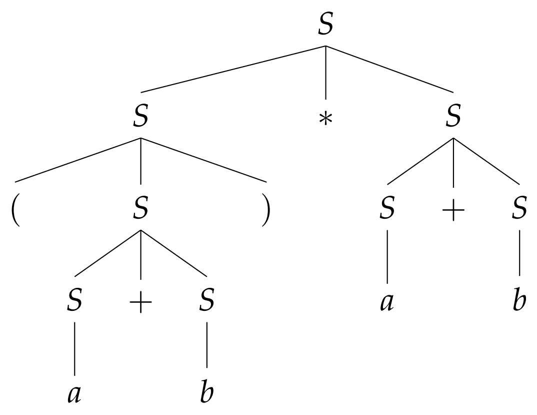


Figure 8.3: Parse tree for $(a + b) * a + b$ corresponding to the derivation (8.11).

This happens to be a left-most derivation, as it is always the left-most variable that is substituted. The parse tree corresponding to this derivation is shown in Figure 8.3.

You can imagine a more complex version of this grammar allowing for other arithmetic operations, variables, and so on, but we will stick to the grammar in (8.9) for the sake of simplicity.

Now, the CFG (8.9) is certainly ambiguous. For instance, a different (left-most) derivation for the same string $(a + b) * a + b$ as before is

$$\begin{aligned} S &\Rightarrow S + S \Rightarrow S * S + S \Rightarrow (S) * S + S \Rightarrow (S + S) * S + S \\ &\Rightarrow (a + S) * S + S \Rightarrow (a + b) * S + S \Rightarrow (a + b) * a + S \\ &\Rightarrow (a + b) * a + b, \end{aligned} \tag{8.12}$$

and the parse tree for this derivation is shown in Figure 8.4. Notice that there is something appealing about the parse tree illustrated in Figure 8.4, which is that it actually carries the meaning of the expression $(a + b) * a + b$, in the sense that the tree structure properly captures the order in which the operations should be applied. In contrast, the first parse tree seems to represent what the expression $(a + b) * a + b$ would evaluate to if we lived in a society where addition was given higher precedence than multiplication.

The ambiguity of the grammar (8.9), along with the fact that parse trees may not represent the meaning of an arithmetic expression in the sense just described, is a problem in some settings. For example, if we were designing a compiler and wanted a part of it to represent arithmetic expressions (presumably allowing much more complicated ones than our grammar from above allows), a CFG along the lines of (8.9) would be completely inadequate.

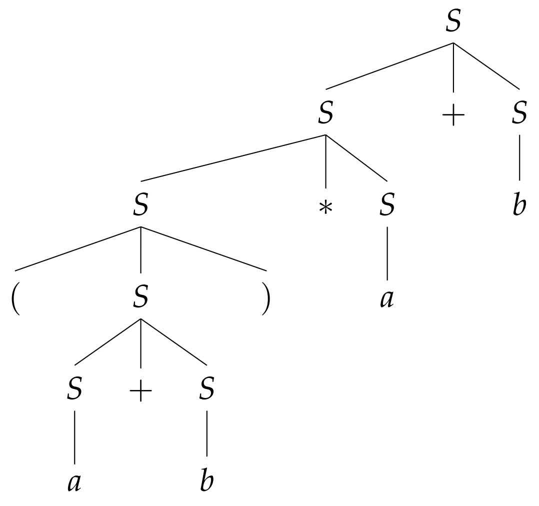


Figure 8.4: Parse tree for $(a + b) * a + b$ corresponding to the derivation (8.12).

We can, however, come up with a new CFG for the same language that is much better—it is unambiguous and it properly captures the meaning of arithmetic expressions. Here it is:

$$\begin{aligned} S &\rightarrow T \mid S + T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow I \mid (\,S\,) \\ I &\rightarrow a \mid b \end{aligned} \tag{8.13}$$

For example, the unique parse tree corresponding to the string $(a + b) * a + b$ is as shown in Figure 8.5.

In order to better understand the CFG (8.13), it may help to associate meanings with the different variables. In this CFG, the variable T generates *terms*, the variable F generates *factors*, and the variable I generates *identifiers*. An expression is either a term or a sum of terms, a term is either a factor or a product of factors, and a factor is either an identifier or an entire expression inside of parentheses.

Inherently ambiguous languages

While we have seen that it is sometime possible to come up with an unambiguous CFG that generates the same language as an ambiguous CFG, it is not always possible. There are some context-free languages that can only be generated by ambiguous CFGs. Such languages are called *inherently ambiguous* context-free languages.

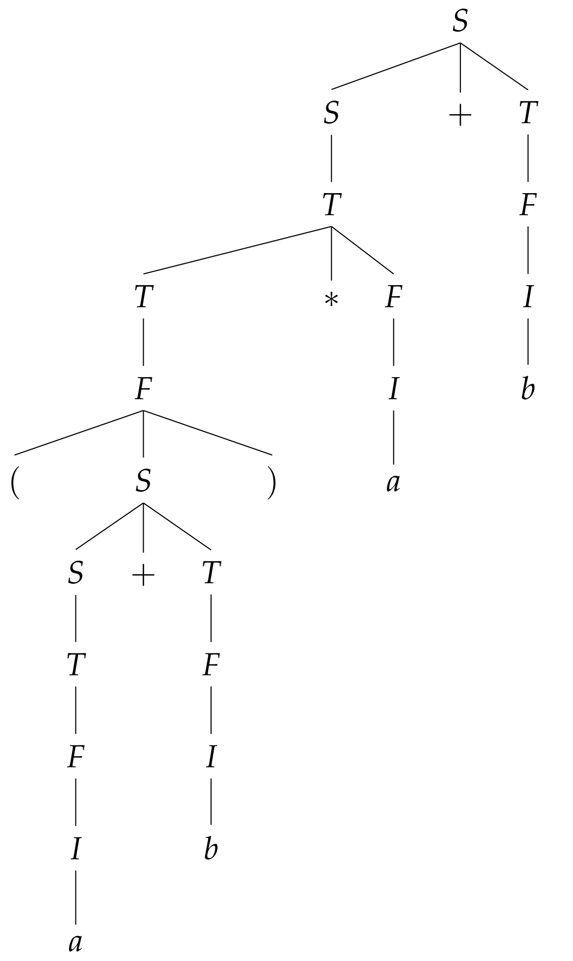


Figure 8.5: Unique parse tree for $(a + b) * a + b$ for the CFG (8.13).

An example of an inherently ambiguous context-free language is this one:

$$\{0^n 1^m 2^k : n = m \text{ or } m = k\}. \tag{8.14}$$

We will not discuss a proof that this language is inherently ambiguous, but the intuition is that the string $0^n 1^n 2^n$ will always have multiple parse trees for some sufficiently large natural number n.

8.3 Chomsky normal form

Some context-free grammars are strange. For example, the CFG

$$S \to S\,S\,S\,S \mid \varepsilon \tag{8.15}$$

simply generates the language $\{\varepsilon\}$; but it is obviously ambiguous, and even worse it has infinitely many parse trees (which of course can be arbitrarily large) for the only string ε it generates. While we know we cannot always eliminate ambiguity from CFGs—as some context-free languages are inherently ambiguous—we can at least eliminate the possibility to have infinitely many parse trees for a given string. Perhaps more importantly, for any given CFG G, we can always come up with a new CFG H for which it holds that $\mathrm{L}(H) = \mathrm{L}(G)$, and for which we are guaranteed that every parse tree for a given string $w \in \mathrm{L}(H)$ has the same size and a very simple, binary-tree-like structure.

To be more precise about the specific sort of CFGs and parse trees we're talking about, it is appropriate at this point to define what is called the *Chomsky normal form* for context-free grammars.

Definition 8.1. A context-free grammar G is in *Chomsky normal form* if every rule of G has one of the following three forms:

1. $X \rightarrow YZ$, for variables X, Y, and Z, and where neither Y nor Z is the start variable,
2. $X \rightarrow \sigma$, for a variable X and a symbol σ, or
3. $S \rightarrow \varepsilon$, for S the start variable.

Now, the reason why a CFG in Chomsky normal form is nice is that every parse tree for such a grammar has a simple form: the variable nodes form a binary tree, and for each variable node that doesn't have any variable node children, a single symbol node hangs off. A hypothetical example meant to illustrate the structure we are talking about is given in Figure 8.6. Notice that the start variable always appears exactly once at the root of the tree because it is never allowed on the right-hand side of any rule.

If the rule $S \rightarrow \varepsilon$ is present in a CFG in Chomsky normal form, then we have a special case that doesn't fit exactly into the structure described above. In this case we can have the very simple parse tree shown in Figure 8.7 for ε, and this is the only possible parse tree for this string.

Because of the very special form that a parse tree must take for a CFG G in Chomsky normal form, we have that *every* parse tree for a given string $w \in \mathrm{L}(G)$ must have exactly $2|w| - 1$ variable nodes and $|w|$ leaf nodes (except for the special case $w = \varepsilon$, in which we have one variable node and 1 leaf node). An equivalent statement is that every derivation of a (nonempty) string w by a CFG in Chomsky normal form requires exactly $2|w| - 1$ substitutions.

The following theorem establishes that every context-free language is generated by a CFG in Chomsky normal form.

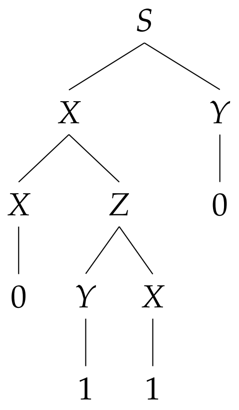


Figure 8.6: A hypothetical example of a parse tree for a CFG in Chomsky normal form.

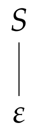


Figure 8.7: The unique parse tree for ε for a CFG in Chomsky normal form, assuming it includes the rule $S \rightarrow \varepsilon$.

Theorem 8.2. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a context-free language. There exists a CFG G in Chomsky normal form such that $A = \mathrm{L}(G)$.*

The usual way to prove this theorem is through a construction that converts an arbitrary CFG G into a CFG H in Chomsky normal form for which it holds that $\mathrm{L}(H) = \mathrm{L}(G)$. The conversion is, in fact, quite straightforward—a summary of the steps one may perform to do this conversion for an arbitrary CFG $G = (V, \Sigma, R, S)$ is as follows:

1. Add a new start variable S_0 along with the rule $S_0 \rightarrow S$.

   Doing this will ensure that the start variable S_0 never appears on the right-hand side of any rule.

2. Eliminate ε-rules of the form $X \rightarrow \varepsilon$ and "repair the damage."

   Aside from the special case $S_0 \rightarrow \varepsilon$, there is never any need for rules of the form $X \rightarrow \varepsilon$; you can get the same effect by simply duplicating rules in which X appears on the right-hand side, and directly replacing or not replacing X

with ε in all possible combinations. For example, if we have the CFG

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow (\, S\,)\, S \mid \varepsilon \end{aligned} \tag{8.16}$$

we can easily eliminate the ε-rule $S \rightarrow \varepsilon$ in this way:

$$\begin{aligned} S_0 &\rightarrow S \mid \varepsilon \\ S &\rightarrow (\, S\,)\, S \mid (\,)\, S \mid (\, S\,) \mid (\,) \end{aligned} \tag{8.17}$$

It can get messy for larger grammars, and when ε-rules for multiple variables are involved one needs to take care in making the process terminate correctly, but it is always possible to remove all ε-rules (aside from $S_0 \rightarrow \varepsilon$) in this way.

3. Eliminate unit rules, which are rules of the form $X \rightarrow Y$.

   Rules like this are never necessary, and they can be eliminated by simply "hard coding" the substitution $X \rightarrow Y$ into every other (non-unit) rule where X appears on the right-hand side. For instance, carrying on the example from above, we can eliminate the unit rule $S_0 \rightarrow S$ like this:

$$\begin{aligned} S_0 &\rightarrow (\, S\,)\, S \mid (\,)\, S \mid (\, S\,) \mid (\,) \mid \varepsilon \\ S &\rightarrow (\, S\,)\, S \mid (\,)\, S \mid (\, S\,) \mid (\,) \end{aligned} \tag{8.18}$$

4. Introduce a new variable X_σ for each symbol σ.

   Include the rule $X_\sigma \rightarrow \sigma$, and replace every instance of σ appearing on the right-hand side of a rule by X_σ (except when σ appears all by itself on the right-hand side of a rule, because you don't want to introduce new unit rules). For the example above, we may use L and R (for left-parenthesis and right-parenthesis) to obtain

$$\begin{aligned} S_0 &\rightarrow L\, S\, R\, S \mid L\, R\, S \mid L\, S\, R \mid L\, R \mid \varepsilon \\ S &\rightarrow L\, S\, R\, S \mid L\, R\, S \mid L\, S\, R \mid L\, R \\ L &\rightarrow (\\ R &\rightarrow) \end{aligned} \tag{8.19}$$

5. Finally, we can split up rules of the form $X \rightarrow Y_1 \cdots Y_m$ using auxiliary variables in a straightforward way.

   For instance, $X \rightarrow Y_1 \cdots Y_m$ can be broken up as

$$\begin{aligned} X &\rightarrow Y_1 X_2 \\ X_2 &\rightarrow Y_2 X_3 \\ &\vdots \\ X_{m-1} &\rightarrow Y_{m-1} Y_m. \end{aligned} \tag{8.20}$$

> We need to be sure to use separate auxiliary variables for each rule so that there is no “cross talk” between separate rules. Let’s not write this out explicitly for our example because it will be lengthy and hopefully the idea is clear.

The description above is only meant to give you the basic idea of how the construction works and does not constitute a proof of Theorem 8.2. It is possible, however, to be more formal and precise in describing this construction in order to obtain a proper proof of Theorem 8.2.

As you may by now suspect, this conversion of a CFG to Chomsky normal form often produces very large CFGs. The steps are routine but things get messy and it is easy to make a mistake when doing it by hand. I will never ask you to perform this conversion on an actual CFG, but we will make use of the theorem from time to time—when we are proving things about context-free languages it is sometimes extremely helpful to know that we can always assume that a given context-free language is generated by a CFG in Chomsky normal form.

Finally, it must be stressed that the Chomsky normal form says nothing about ambiguity in general—a CFG in Chomsky normal form may or may not be ambiguous, just like we have for arbitrary CFGs. On the other hand, if you start with an unambiguous CFG and perform the conversion described above, the resulting CFG in Chomsky normal form will still be unambiguous.

Lecture 9

Closure properties for context-free languages

In this lecture we will examine various properties of the context-free languages, including the fact that they are closed under the regular operations, that every regular language is context-free, and (more generally) the intersection of a context-free language and a regular language is always context-free.

9.1 Closure under the regular operations

We will begin by proving that the context-free languages are closed under each of the regular operations.

Theorem 9.1. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be context-free languages. The languages $A \cup B$, AB, and A^* are context-free.*

Proof. Because A and B are context-free languages, there must exist CFGs

$$G_A = (V_A, \Sigma, R_A, S_A) \quad \text{and} \quad G_B = (V_B, \Sigma, R_B, S_B) \tag{9.1}$$

such that $\mathrm{L}(G_A) = A$ and $\mathrm{L}(G_B) = B$. Because the specific names we choose for the variables in a context-free grammar have no effect on the language it generates, there is no loss of generality in assuming V_A and V_B are disjoint sets.

First let us construct a CFG G for the language $A \cup B$. This CFG will include all of the variables and rules of G_A and G_B together, along with a new variable S (which we assume is not already contained in V_A or V_B, and which we will take to be the start variable of G) and two new rules:

$$S \rightarrow S_A \mid S_B. \tag{9.2}$$

Formally speaking we may write

$$G = (V, \Sigma, R, S) \tag{9.3}$$

where $V = V_A \cup V_B \cup \{S\}$ and $R = R_A \cup R_B \cup \{S \rightarrow S_A,\ S \rightarrow S_B\}$. In the typical style in which we write CFGs, the grammar G looks like this:

$$S \rightarrow S_A \mid S_B$$

$$\boxed{\text{all rules of } G_A}$$

$$\boxed{\text{all rules of } G_B}$$

It is evident that $\mathrm{L}(G) = A \cup B$; each derivation may begin with $S \Rightarrow S_A$ or $S \Rightarrow S_B$, after which either S_A generates any string in A or S_B generates any string in B. As the language $A \cup B$ is generated by the CFG G, we have that it is context-free.

Next we will construct a CFG H for the language AB. The construction of H is very similar to the construction of G above. The CFG H will include all of the variables and rules of G_A and G_B, along with a new start variable S and one new rule:

$$S \rightarrow S_A S_B. \tag{9.4}$$

Formally speaking we may write

$$H = (V, \Sigma, R, S) \tag{9.5}$$

where $V = V_A \cup V_B \cup \{S\}$ and $R = R_A \cup R_B \cup \{S \rightarrow S_A S_B\}$. In the typical style in which we write CFGs, the grammar G looks like this:

$$S \rightarrow S_A S_B$$

$$\boxed{\text{all rules of } G_A}$$

$$\boxed{\text{all rules of } G_B}$$

It is evident that $\mathrm{L}(G) = AB$; each derivation must begin with $S \Rightarrow S_A S_B$, after which either S_A generates any string in A and S_B generates any string in B. As the language AB is generated by the CFG H, we have that it is context-free.

Finally we will construct a CFG K for A^*. This time the CFG K will include just the rules and variables of G_A, along with a new start variable S and two new rules:

$$S \rightarrow S\, S_A \mid \varepsilon. \tag{9.6}$$

Formally speaking we may write

$$K = (V, \Sigma, R, S) \tag{9.7}$$

where $V = V_A \cup \{S\}$ and $R = R_A \cup \{S \rightarrow S\, S_A, S \rightarrow \varepsilon\}$. In the typical style in which we write CFGs, the grammar K looks like this:

$$S \rightarrow S\, S_A \mid \varepsilon$$

$$\boxed{\text{all rules of } G_A}$$

Every possible left-most derivation of a string by K must begin with zero or more applications of the rule $S \rightarrow S\, S_A$ followed by the rule $S \rightarrow \varepsilon$. This means that every left-most derivation begins with a sequence of rule applications that is consistent with one of the following relationships:

$$\begin{aligned} & S \overset{*}{\Rightarrow} \varepsilon \\ & S \overset{*}{\Rightarrow} S_A \\ & S \overset{*}{\Rightarrow} S_A\, S_A \\ & S \overset{*}{\Rightarrow} S_A\, S_A\, S_A \\ & \quad \vdots \end{aligned} \tag{9.8}$$

and so on. After this, each occurrence of S_A generates any string in A. It therefore holds that $\mathrm{L}(K) = A^*$, so that A^* is context-free. □

9.2 Every regular language is context-free

Next we will prove that every regular language is also context-free. In fact, we will see two different ways to prove this fact.

Theorem 9.2. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language. It holds that A is context-free.*

First proof. With every regular expression R over the alphabet Σ, one may associate a CFG G by recursively applying these simple constructions:

1. If $R = \varnothing$, then G is the CFG
$$S \rightarrow S, \tag{9.9}$$
which generates the empty language $\varnothing$.
2. If $R = \varepsilon$, then G is the CFG
$$S \rightarrow \varepsilon, \tag{9.10}$$
which generates the language $\{\varepsilon\}$.
3. If $R = \sigma$ for $\sigma \in \Sigma$, then G is the CFG
$$S \rightarrow \sigma, \tag{9.11}$$
which generates the language $\{\sigma\}$.
4. If $R = (R_1 \cup R_2)$, then G is the CFG generating the language $\mathrm{L}(G_1) \cup \mathrm{L}(G_2)$, as described in the proof of Theorem 9.1, where G_1 and G_2 are CFGs associated with the regular expressions R_1 and R_2, respectively.
5. If $R = (R_1 R_2)$, then G is the CFG generating the language $\mathrm{L}(G_1)\,\mathrm{L}(G_2)$, as described in the proof of Theorem 9.1, where G_1 and G_2 are CFGs associated with the regular expressions R_1 and R_2, respectively.
6. If $R = (R_1^*)$, then G is the CFG generating the language $\mathrm{L}(G_1)^*$, as described in the proof of Theorem 9.1, where G_1 is the CFG associated with the regular expression R_1.

It holds that $\mathrm{L}(G) = \mathrm{L}(R)$.

Now, by the assumption that A is regular, there must exist a regular expression R such that $\mathrm{L}(R) = A$. For the CFG G obtained from R as described above, it holds that $\mathrm{L}(G) = A$, and therefore A is context-free. □

Second proof. Because A is regular, there must exist a DFA
$$M = (Q, \Sigma, \delta, q_0, F) \tag{9.12}$$
such that $\mathrm{L}(M) = A$. Because it doesn't matter what names we assign to the states of a DFA, there is no loss of generality in assuming that $Q = \{q_0, \ldots, q_{n-1}\}$ for some choice of a positive integer n.

We will define a CFG G that effectively simulates M, generating exactly those strings that are accepted by M. In particular, we will define

$$G = (V, \Sigma, R, X_0) \tag{9.13}$$

where the variables are $V = \{X_0, \ldots, X_{n-1}\}$ (i.e., one variable for each state of M) and the following rules are to be included in R:

1. For each choice of $k, m \in \{0, \ldots, n-1\}$ and $\sigma \in \Sigma$ satisfying $\delta(q_k, \sigma) = q_m$, the rule
$$X_k \rightarrow \sigma \, X_m \tag{9.14}$$
is included in R.
2. For each $m \in \{0, \ldots, n-1\}$ satisfying $q_m \in F$, the rule
$$X_m \rightarrow \varepsilon \tag{9.15}$$
is included in R.

Now, by examining the rules suggested above, we see that every derivation of a string by G starts with X_0 (of course), involves zero or more applications of rules of the first type listed above, and then ends when a rule of the second type is applied. There will always be a single variable appearing after each step of the derivation, until the very last step in which this variable is eliminated. It is important that this final step is only possible when the variable X_k corresponds to an accept state $q_k \in F$. By considering the rules of the first type, it is evident that

$$\big[X_0 \overset{*}{\Rightarrow} wX_m\big] \;\Leftrightarrow\; \big[\delta^*(q_0, w) = q_m\big]. \tag{9.16}$$

We therefore have $X_0 \overset{*}{\Rightarrow} w$ if and only if there exists a choice of $m \in \{0, \ldots, n-1\}$ for which $\delta^*(q_0, w) = q_m$ and $q_m \in F$. This is equivalent to the statement that $\mathrm{L}(G) = \mathrm{L}(M)$, which completes the proof. □

9.3 Intersections of regular and context-free languages

The context-free languages are not closed under some operations for which the regular languages are closed. For example, the complement of a context-free language may fail to be context-free, and the intersection of two context-free languages may fail to be context-free. (We will observe both of these facts in the next lecture.) It is the case, however, that the intersection of a context-free language and a regular language is always context-free.

Theorem 9.3. *Let Σ be an alphabet, let $A, B \subseteq \Sigma^*$ be languages, and assume A is context-free and B is regular. The language $A \cap B$ is context-free.*

Remark 9.4. Before discussing the proof of this theorem, let us note that it implies Theorem 9.2; one is free to choose $A = \Sigma^*$ (which is context-free) and B to be any regular language, and the implication is that $\Sigma^* \cap B = B$ is context-free. Because the proof is quite a bit more involved than the two proofs of Theorem 9.2 that we already discussed, however, it is worthwhile that we considered them first.

Incidentally, you should not feel obliged to understand all of the details of the proof that follows—it is a bit long and technical, and you will not be tested on it or required to reproduce these sorts of details. It would be a good idea, however, to try to understand the main ideas, as the ideas could be useful for solving different problems regarding context-free languages.

Proof. The language A is context-free, so there exists a CFG that generates it. As discussed in the previous lecture, we may in fact assume that there exists a CFG in Chomsky normal form that generates A—having this CFG be in Chomsky normal form will greatly simplify the proof. Hereafter we will assume

$$G = (V, \Sigma, R, S) \tag{9.17}$$

is a CFG in Chomsky normal form such that $\mathrm{L}(G) = A$. Because the language B is regular, there must also exist a DFA

$$M = (Q, \Sigma, \delta, q_0, F) \tag{9.18}$$

such that $\mathrm{L}(M) = B$.

The main idea of the proof is to define a new CFG H such that $\mathrm{L}(H) = A \cap B$. The CFG H will have $|Q|^2$ variables for *each* variable of G, which may be a lot but that's not a problem—it is a finite number, and that is all we require of a set of variables of a context-free grammar. In particular, for each variable $X \in V$, we will include a variable $X_{p,q}$ in H for every choice of $p, q \in Q$. In addition, we will add a new start variable S_0 to H.

The intended meaning of each variable $X_{p,q}$ is that it should generate all strings that (i) are generated by X with respect to the grammar G, and (ii) cause M to move from state p to state q. We will accomplish this by adding a collection of rules to H for each rule of G. Because the grammar G is assumed to be in Chomsky normal form, there are just three possible forms for its rules, and they can be handled one at a time as follows:

1. For each rule of the form $X \to \sigma$ in G, include the rule

$$X_{p,q} \to \sigma \tag{9.19}$$

in H for every pair of states $p, q \in Q$ for which $\delta(p, \sigma) = q$.

2. For each rule of the form $X \rightarrow Y\,Z$ in G, include the rules

$$X_{p,q} \rightarrow Y_{p,r}Z_{r,q} \tag{9.20}$$

in H for every choice of states $p, q, r \in Q$.

3. If the rule $S \rightarrow \varepsilon$ is included in G and it holds that $q_0 \in F$ (i.e., $\varepsilon \in A \cap B$), then include the rule

$$S_0 \rightarrow \varepsilon \tag{9.21}$$

in H, where S_0 is the new start variable for H mentioned above.

Once we have added all of these rules in H, we also include the rule

$$S_0 \rightarrow S_{q_0,p} \tag{9.22}$$

in H for every accepting state $p \in F$.

The intended meaning of each variable $X_{p,q}$ in H has been suggested above. More formally speaking, we wish to prove that the following equivalence holds for every nonempty string $w \in \Sigma^*$, every variable $X \in V$, and every choice of states $p, q \in Q$:

$$\big[X_{p,q} \overset{*}{\Rightarrow}_H w\big] \Leftrightarrow \big[\big(X \overset{*}{\Rightarrow}_G w\big) \wedge \big(\delta^*(p, w) = q\big)\big]. \tag{9.23}$$

The two implications can naturally be handled separately, and one of the two implications naturally splits into two parts.

First, it is almost immediate that the implication

$$\big[X_{p,q} \overset{*}{\Rightarrow}_H w\big] \Rightarrow \big[X \overset{*}{\Rightarrow}_G w\big] \tag{9.24}$$

holds, as a derivation of w starting from $X_{p,q}$ in H gives a derivation of w starting from X in G if we simply remove all of the subscripts on all of the variables.

Next, we can prove the implication

$$\big[X_{p,q} \overset{*}{\Rightarrow}_H w\big] \Rightarrow \big[\delta^*(p, w) = q\big] \tag{9.25}$$

by induction on the length of w. The base case is $|w| = 1$, and in this case we must have $X_{p,q} \Rightarrow_H \sigma$ for some $\sigma \in \Sigma$. The only rules that allow such a derivation are of the first type above, which require $\delta(p, \sigma) = q$. In the general case in which $|w| \geq 2$, it must hold that

$$X_{p,q} \Rightarrow Y_{p,r}Z_{r,q} \tag{9.26}$$

for variables $Y_{p,r}$ and $Z_{r,q}$ satisfying

$$Y_{p,r} \overset{*}{\Rightarrow}_H y \quad \text{and} \quad Z_{r,q} \overset{*}{\Rightarrow}_H z \tag{9.27}$$

for strings $y, z \in \Sigma^*$ satisfying $w = yz$. By the hypothesis of induction we conclude that $\delta^*(p, y) = r$ and $\delta^*(r, z) = q$, so that $\delta^*(p, w) = q$.

Finally one can prove

$$\big[(X \overset{*}{\Rightarrow}_G w) \wedge (\delta^*(p, w) = q) \big] \Rightarrow \big[X_{p,q} \overset{*}{\Rightarrow}_H w \big], \tag{9.28}$$

again by induction on the length of w. The base case is $|w| = 1$, which is straightforward: if $X \Rightarrow_G \sigma$ and $\delta(p, \sigma) = q$, then $X_{p,q} \Rightarrow_H \sigma$ because the rule that allows for this derivation has been included among the rules of H. In the general case in which $|w| \geq 2$, the relation $X \overset{*}{\Rightarrow}_G w$ implies that $X \Rightarrow_G YZ$ for variables $Y, Z \in V$ such that $Y \overset{*}{\Rightarrow}_G y$ and $Z \overset{*}{\Rightarrow}_G z$, for strings $y, z \in \Sigma^*$ satisfying $w = yz$. Choosing $r \in Q$ so that $\delta^*(p, y) = r$ (and therefore $\delta^*(r, z) = q$), we have that $Y_{p,r} \overset{*}{\Rightarrow} y$ and $Z_{r,q} \overset{*}{\Rightarrow} z$ by the hypothesis of induction, and therefore $X_{p,q} \Rightarrow_H Y_{p,r} Z_{r,q} \overset{*}{\Rightarrow}_H yz = w$.

Because every derivation of a nonempty string by H must begin with

$$S_0 \Rightarrow_H S_{q_0,p} \tag{9.29}$$

for some $p \in F$, we find that the nonempty strings w generated by H are precisely those strings that are generated by G and satisfy $\delta^*(q_0, w) = p$ for some $p \in F$. Equivalently, for $w \neq \varepsilon$ it holds that $w \in \mathrm{L}(H) \Leftrightarrow w \in A \cap B$. The empty string has been handled as a special case, so it follows that $\mathrm{L}(H) = A \cap B$. The language $A \cap B$ is therefore context-free. □

9.4 Prefixes, suffixes, and substrings

Let us finish off the lecture with just a few quick examples. Recall from Lecture 6 that for any language $A \subseteq \Sigma^*$ we define

$$\text{Prefix}(A) = \big\{ x \in \Sigma^* \,:\, \text{there exists } v \in \Sigma^* \text{ such that } xv \in A \big\}, \tag{9.30}$$

$$\text{Suffix}(A) = \big\{ x \in \Sigma^* \,:\, \text{there exists } u \in \Sigma^* \text{ such that } ux \in A \big\}, \tag{9.31}$$

$$\text{Substring}(A) = \big\{ x \in \Sigma^* \,:\, \text{there exist } u, v \in \Sigma^* \text{ such that } uxv \in A \big\}. \tag{9.32}$$

Let us prove that if A is context-free, then each of these languages is also context-free. In the interest of time, we will just explain how to come up with context-free grammars for these languages and not go into details regarding the proofs that these CFGs are correct. In all three cases, we will assume that $G = (V, \Sigma, R, S)$ is a CFG in Chomsky normal form such that $L(G) = A$.

For the language Prefix(A), we will design a CFG H as follows. First, for every variable $X \in V$ used by G we will include this variable in H, and in addition we will also include a variable X_0. The idea is that X will generate exactly the same strings in H that it does in G, while X_0 will generate all the prefixes of the strings generated by X in G. We include rules in H as follows:

1. For every rule of the form $X \rightarrow Y\,Z$ in G, include these rules in H:

$$\begin{aligned} X &\rightarrow Y\,Z \\ X_0 &\rightarrow Y\,Z_0 \mid Y_0 \end{aligned} \tag{9.33}$$

2. For every rule of the form $X \rightarrow \sigma$ in G, include these rules in H:

$$\begin{aligned} X &\rightarrow \sigma \\ X_0 &\rightarrow \sigma \mid \varepsilon \end{aligned} \tag{9.34}$$

Finally, we take S_0 to be the start variable of H.

The idea is similar for the language Suffix(A), for which we will construct a CFG K. This time, for every variable $X \in V$ used by G we will include this variable in K, and in addition we will also include a variable X_1. This time the idea is that X will generate exactly the same strings in K that it does in G, while X_1 will generate all the suffixes of the strings generated by X in G. We include rules in K as follows:

1. For every rule of the form $X \rightarrow Y\,Z$ in G, include these rules in K:

$$\begin{aligned} X &\rightarrow Y\,Z \\ X_1 &\rightarrow Y_1\,Z \mid Z_1 \end{aligned} \tag{9.35}$$

2. For every rule of the form $X \rightarrow \sigma$ in G, include these rules in K:

$$\begin{aligned} X &\rightarrow \sigma \\ X_1 &\rightarrow \sigma \mid \varepsilon \end{aligned} \tag{9.36}$$

Finally, we take S_1 to be the start variable of K.

To obtain a CFG J for Substring(A), we can simply combine the two constructions above (i.e., apply either one to G, then apply the other to the resulting CFG). Equivalently, we can include variables X, X_0, X_1, and X_2 in J for every $X \in V$ and include rules as follows:

1. For every rule of the form $X \rightarrow Y\,Z$ in G, include these rules in J:

$$\begin{aligned} X &\rightarrow Y\,Z \\ X_0 &\rightarrow YZ_0 \mid Y_0 \\ X_1 &\rightarrow Y_1\,Z \mid Z_1 \\ X_2 &\rightarrow Y_1Z_0 \mid Y_2 \mid Z_2 \end{aligned} \tag{9.37}$$

2. For every rule of the form $X \rightarrow \sigma$ in G, include these rules in J:

$$\begin{aligned} X &\rightarrow \sigma \\ X_0 &\rightarrow \sigma \mid \varepsilon \\ X_1 &\rightarrow \sigma \mid \varepsilon \\ X_2 &\rightarrow \sigma \mid \varepsilon. \end{aligned} \tag{9.38}$$

Finally, we take S_2 to be the start variable of J. The meaning of the variables X, X_0, X_1, and X_2 in J is that they generate precisely the strings generated by X in G, the prefixes, the suffixes, and the substrings of these strings, respectively.

Lecture 10

Proving languages to be non-context-free

In this lecture we will study a method through which certain languages can be proved to be non-context-free. The method will appear to be quite familiar, because it closely resembles the one we discussed in Lecture 5 for proving certain languages to be nonregular.

10.1 The pumping lemma (for context-free languages)

Along the same lines as the method we discussed in Lecture 5 for proving some languages to be nonregular, we will start with a variant of the pumping lemma that holds for context-free languages.

The proof of this lemma is, naturally, different from the proof of the pumping lemma for regular languages, but there are similar underlying ideas. In particular, if you have a parse tree for the derivation of a particular string by some context-free grammar, and the parse tree is sufficiently deep, then there must be a variable that appears multiple times on some path from the root to a leaf—and by modifying the parse tree in certain ways, one obtains a similar type of pumping effect that we had in the case of the pumping lemma for regular languages.

Lemma 10.1 (Pumping lemma for context-free languages)**.** *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a context-free language. There exists a positive integer n (called a* pumping length *of A) that possesses the following property. For every string $w \in A$ with $|w| \geq n$, it is possible to write $w = uvxyz$ for some choice of strings $u, v, x, y, z \in \Sigma^*$ such that*

1. *$vy \neq \varepsilon$,*
2. *$|vxy| \leq n$, and*
3. *$uv^ixy^iz \in A$ for all $i \in \mathbb{N}$.*

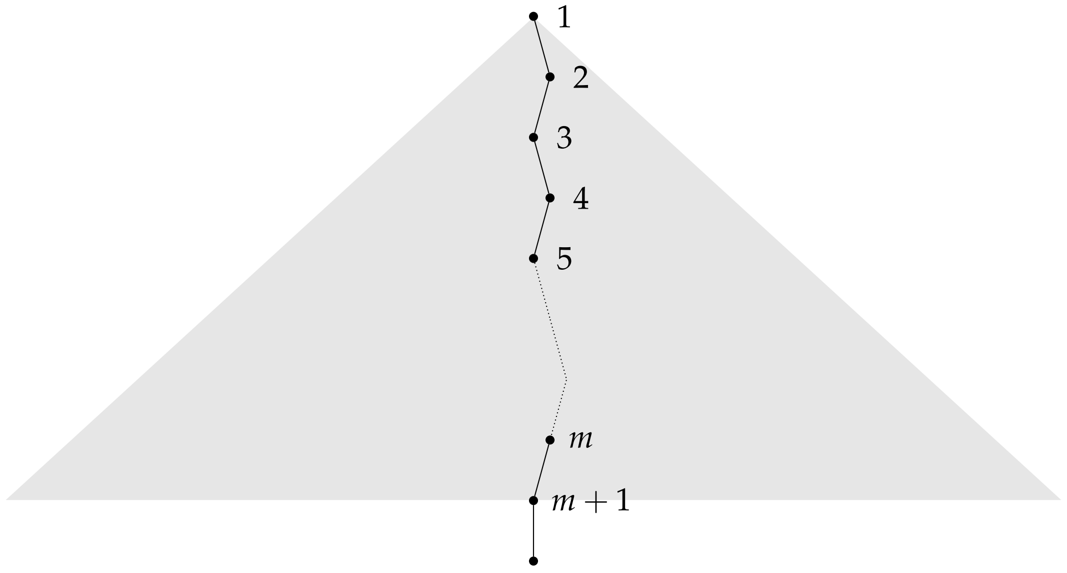


Figure 10.1: At least one path from the root to a leaf in a CNF parse tree for a string of length 2^m or more must have $m+1$ or more variable nodes. If this were not so, the total number of variable nodes (represented by the shaded region) would be at most $2^m - 1$, contradicting the fact that there must be at least 2^m variable nodes.

Proof. Given that A is context-free, we know that there must exist a CFG G in Chomsky normal form such that $A = \mathrm{L}(G)$. Let m be the number of variables in G. We will prove that the property stated in the lemma holds for $n = 2^m$.

Suppose that a string $w \in A$ satisfies $|w| \geq n = 2^m$. As G is in Chomsky normal form, every parse tree for w has exactly $2|w| - 1$ variable nodes and $|w|$ leaf nodes. Hereafter let us fix any one of these parse trees, and let us call this tree T. For the sake of this proof, what is important about the size of T is that the number of variable nodes is at least 2^m. (This is true because $2|w| - 1 \geq 2 \cdot 2^m - 1 \geq 2^m$. In fact, the last inequality must be strict because $m \geq 1$, but this makes no difference to the proof.) Because the number of variable nodes in T is at least 2^m, there must exist at least one path in T from the root to a leaf along which there are at least $m+1$ variable nodes—for if all such paths had m or fewer variable nodes, there could be at most $2^m - 1$ variable nodes in the entire tree.

Next, choose any path in T from the root to a leaf having the maximum possible length. (There may be multiple choices, but any one of them is fine.) We know that at least $m+1$ variable nodes must appear in this path, as argued above—and because there are only m different variables in total, there must be at least one variable that appears multiple times along this path. In fact, we know that some variable (let us call it X) must appear at least twice within the $m+1$ variable nodes

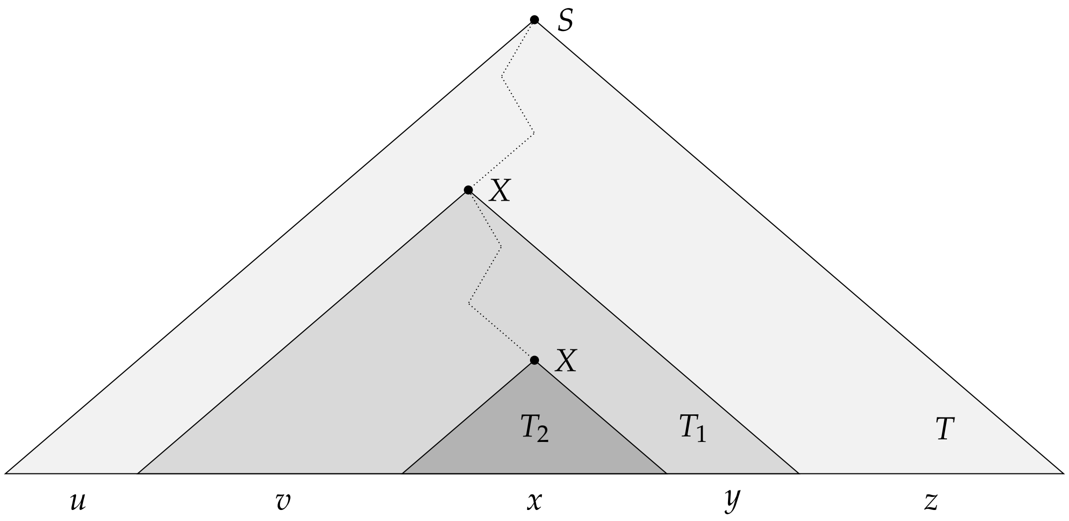


Figure 10.2: An illustration of the subtrees T_1 and T_2 of T.

closest to the leaf on the path we have selected. Let T_1 and T_2 be the subtrees of T rooted at these two bottom-most occurrences of this variable X. By the way we have chosen these subtrees, we know that T_2 is a proper subtree of T_1, and T_1 is not very large: every path from the root of the subtree T_1 to one of its leaves can have at most $m + 1$ variable nodes, and therefore T_1 has no more than $2^m = n$ leaf nodes.

Now, let x be the string for which T_2 is a parse tree (starting from the variable X) and let v and y be the strings formed by the leaves of T_1 to the left and right, respectively, of the subtree T_2, so that vxy is the string for which T_1 is a parse tree (also starting from the variable X). Finally, let u and z be the strings represented by the leaves of T to the left and right, respectively, of the subtree T_1, so that $w = uvxyz$. Figure 10.2 provides an illustration of the strings u, v, x, y, and z and how they related to the trees T, T_1, and T_2.

It remains to prove that u, v, x, y, and z have the properties required by the statement of the lemma. Let us first prove that $uv^ixy^iz \in A$ for all $i \in \mathbb{N}$. To see that $uxz = uv^0xy^0z \in A$, we observe that we can obtain a valid parse tree for uxz by replacing the subtree T_1 with the subtree T_2, as illustrated in Figure 10.3. This replacement is possible because both T_1 and T_2 have root nodes corresponding to the variable X. Along similar lines, we have that $uv^2xy^2z \in A$ because we can obtain a valid parse tree for this string by replacing the subtree T_2 with a copy of T_1, as suggested by Figure 10.4. By repeatedly replacing T_2 with a copy of T_1, a valid parse tree for any string of the form uv^ixy^iz is obtained.

Next, the fact that $vy \neq \varepsilon$ follows from the fact that every parse tree for a string

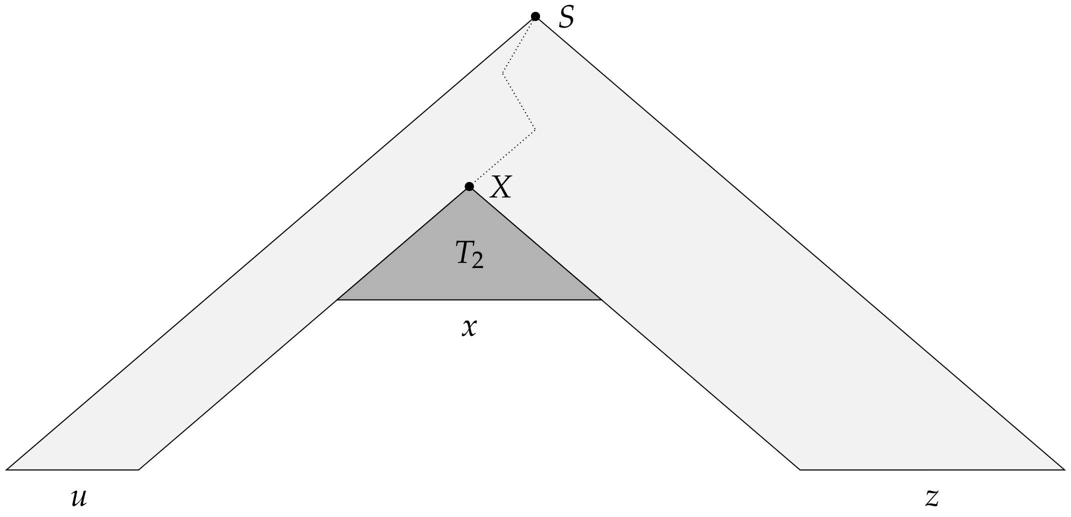


Figure 10.3: By replacing the subtree T_1 by the subtree T_2 in T, a parse tree for the string $uxz = uv^0xy^0z$ is obtained.

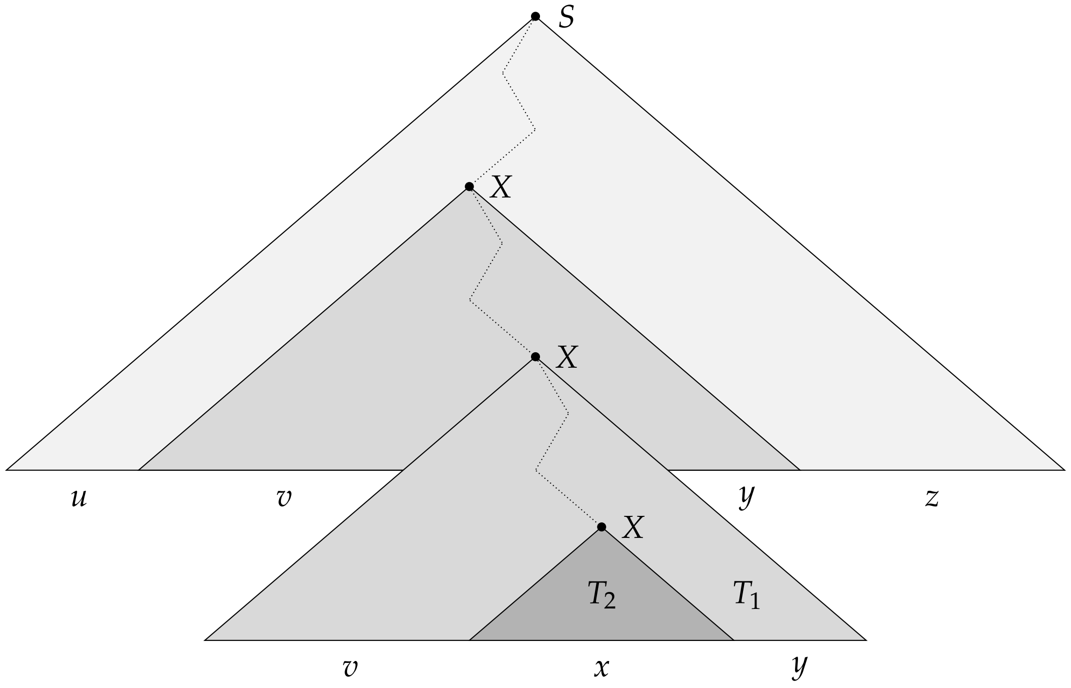


Figure 10.4: By replacing the subtree T_2 by the subtree T_1 in T, a parse tree for the string uv^2xy^2z is obtained. By repeatedly replacing T_2 with T_1 in this way, a parse tree for the string uv^ixy^iz is obtained for any positive integer $i \geq 2$.

corresponding to a CFG in Chomsky normal form has the same size. It therefore cannot be that the parse tree suggested by Figure 10.3 generates the same string as the one suggested by Figure 10.2, as the two trees have differing numbers of variable nodes. This implies that $uvxyz \neq uxz$, so $vy \neq \varepsilon$.

Finally, it holds that $|vxy| \leq n$ because the subtree T_1 has at most $2^m = n$ leaf nodes, as argued above. □

10.2 Using the context-free pumping lemma

Now that we have the pumping lemma for context-free languages in hand, we can prove that certain languages are not context-free. The methodology is very similar to what we used in Lecture 5 to prove some languages to be nonregular. Some examples, stated as propositions, follow.

Proposition 10.2. *Let $\Sigma = \{0,1,2\}$ and let A be a language defined as follows:*

$$A = \{0^m 1^m 2^m \,:\, m \in \mathbb{N}\}. \tag{10.1}$$

The language A is not context-free.

Proof. Assume toward contradiction that A is context-free. By the pumping lemma for context-free languages, there must exist a pumping length $n \geq 1$ for A.

Let $w = 0^n 1^n 2^n$. We have that $w \in A$ and $|w| = 3n \geq n$, so the pumping lemma guarantees that there must exist strings $u, v, x, y, z \in \Sigma^*$ so that $w = uvxyz$ and the three properties in the statement of that lemma hold: (i) $vy \neq \varepsilon$, (ii) $|vxy| \leq n$, and (iii) $uv^i xy^i z \in A$ for all $i \in \mathbb{N}$.

Now, given that $|vxy| \leq n$, it cannot be that the symbols 0 and 2 both appear in the string vy—the 0s and 2s are too far apart for this to happen. On the other hand, at least one of the symbols of Σ must appear within vy, because this string is nonempty. This implies that the string

$$uv^0 xy^0 z = uxz \tag{10.2}$$

must have strictly fewer occurrences of either 1 or 2 than 0, or strictly fewer occurrences of either 0 or 1 than 2. That is, if the symbol 0 does not appear in vy, then it must be that either

$$|uxz|_1 < |uxz|_0 \quad \text{or} \quad |uxz|_2 < |uxz|_0, \tag{10.3}$$

and if the symbol 2 does not appear in vy, then it must be that either

$$|uxz|_0 < |uxz|_2 \quad \text{or} \quad |uxz|_1 < |uxz|_2. \tag{10.4}$$

This, however, is in contradiction with the fact that $uv^0xy^0z = uxz$ is guaranteed to be in A by the third property.

Having obtained a contradiction, we conclude that A is not context-free, as claimed. $\square$

In some cases, such as the following one, a language can be proved to be non-context-free in almost exactly the same way that it can be proved to be nonregular.

Proposition 10.3. *Let $\Sigma = \{0\}$ and define a language over Σ as follows:*

$$B = \big\{0^m \;:\; m \text{ is a perfect square}\big\}. \tag{10.5}$$

The language B is not context-free.

Proof. Assume toward contradiction that B is context-free. By the pumping lemma for context-free languages, there must exist a pumping length $n \geq 1$ for B for which the property stated by that lemma holds. We will fix such a pumping length n for the remainder of the proof.

Define $w = 0^{n^2}$. It holds that $w \in B$ and $|w| = n^2 \geq n$, so the pumping lemma tells us that there exist strings $u, v, x, y, z \in \Sigma^*$ so that $w = uvxyz$ and the following conditions hold:

1. $vy \neq \varepsilon$,
2. $|vxy| \leq n$, and
3. $uv^ixy^iz \in B$ for all $i \in \mathbb{N}$.

There is only one symbol in the alphabet Σ, so it is immediate that $vy = 0^k$ for some choice of $k \in \mathbb{N}$. Because $vy \neq \varepsilon$ and $|vy| \leq |vxy| \leq n$ it must be the case that $1 \leq k \leq n$. It holds that

$$uv^ixy^iz = 0^{n^2+(i-1)k} \tag{10.6}$$

for each $i \in \mathbb{N}$. In particular, if we choose $i = 2$, then we have

$$uv^2xy^2z = 0^{n^2+k}. \tag{10.7}$$

However, because $1 \leq k \leq n$, it cannot be that $n^2 + k$ is a perfect square—this is because $n^2 + k$ is larger than n^2, but the next perfect square after n^2 is

$$(n+1)^2 = n^2 + 2n + 1, \tag{10.8}$$

which is strictly larger than $n^2 + k$ because $k \leq n$. The string uv^2xy^2z is therefore *not* contained in B, which contradicts the third condition stated by the pumping lemma, which guarantees us that $uv^ixy^iz \in B$ for all $i \in \mathbb{N}$.

Having obtained a contradiction, we conclude that B is not context-free, as claimed. $\square$

Remark 10.4. We will not discuss the proof, but it turns out that every context-free language over a single-symbol alphabet must be regular. By combining this fact with the fact that B is nonregular, we obtain a different proof that B is not context-free.

Here is one more example of a proof that a particular language is not context-free using the pumping lemma for context-free languages. For this one things get a bit messy because there are multiple cases to worry about as we try to get a contradiction, which turns out to be fairly common when using this method. Of course, one has to be sure to get a contradiction in *all* of the cases in order to have a valid proof by contradiction, so be sure to keep this in mind.

Proposition 10.5. *Let* $\Sigma = \{0, 1, \#\}$ *and define a language* C *over* Σ *as follows:*

$$C = \{r\#s \,:\, r, s \in \{0,1\}^*,\ r \text{ is a substring of } s\}. \tag{10.9}$$

The language C *is not context-free.*

Proof. Assume toward contradiction that C is context-free. By the pumping lemma for context-free languages, there exists a pumping length $n \geq 1$ for C.

Let

$$w = 0^n 1^n \# 0^n 1^n. \tag{10.10}$$

It is the case that $w \in C$ (because $0^n 1^n$ is a substring of itself) and $|w| = 4n + 1 \geq n$. The pumping lemma therefore guarantees that there must exist strings $u, v, x, y, z \in \Sigma^*$ so that $w = uvxyz$ and the three properties in the statement of that lemma hold: (i) $vy \neq \varepsilon$, (ii) $|vxy| \leq n$, and (iii) $uv^i xy^i z \in C$ for all $i \in \mathbb{N}$.

There is just one occurrence of the symbol # in w, so it must appear in one of the strings u, v, x, y, or z. We will consider each case separately:

Case 1: the # lies within u. In this case we have that all of the symbols in v and y appear to the right of the symbol # in w. It follows that

$$uv^0xy^0z = 0^n 1^n \# 0^{n-j} 1^{n-k} \tag{10.11}$$

for some choice of integers j and k with $j + k \geq 1$, because by removing v and y from w we must have removed at least one symbol to the right of the symbol # (and none from the left of that symbol). The string (10.11) is not contained in C, even though the third property guarantees it is, and so we have a contradiction in this case.

Case 2: the # lies within v. This is an easy case: because the # symbol lies in v, the string $uv^0xy^0z = uxz$ does not contain the symbol # at all, so it cannot be in C. This is again in contradiction with the third property, which guarantees that $uv^0xy^0z \in C$, and so we have a contradiction in this case.

Case 3: the # lies within x. In this case, we know that $vxy = 1^j\#0^k$ for some choice of integers j and k for which $j + k \geq 1$. The reason why vxy must take this form is that $|vxy| \leq n$, so this substring cannot both contain the symbol # and reach either the first block of 0s or the last block of 1s, and the reason why $j + k \geq 1$ is that $vy \neq \varepsilon$. If it happens that $j \geq 1$, then we may choose $i = 2$ to obtain a contradiction, as

$$uv^2xy^2z = 0^n1^{n+j}\#0^{n+k}1^n, \tag{10.12}$$

which is not in C because the string to the left of the # symbol has more 1s than the string to the right of the # symbol. If it happens that $k \geq 1$, then we may choose $i = 0$ to obtain a contradiction: we have

$$uv^0xy^0z = 0^n1^{n-j}\#0^{n-k}1^n \tag{10.13}$$

in this case, which is not contained in C because the string to the left of the # symbol has more 0s than the string to the right of the # symbol.

Case 4: the # lies within y. This case is identical to case 2—the string uv^0xy^0z cannot be in C because it does not contain the symbol #.

Case 5: the # lies within z. In this case we have that all of the symbols in v and y appear to the left of the symbol # in w. Because $vy \neq \varepsilon$, it follows that

$$uv^2xy^2z = r\#\,0^n1^n \tag{10.14}$$

for some string r that has length strictly larger than $2n$. The string (10.14) is not contained in C, even though the third property guarantees it is, and so we have a contradiction in this case.

Having obtained a contradiction in all of the cases, we conclude that there must really be a contradiction—so C is not context-free, as claimed. □

10.3 Non-context-free languages and closure properties

In the previous lecture it was stated that the context-free languages are not closed under either intersection or complementation. That is, there exist context-free languages A and B such that neither $A \cap B$ nor $\overline{A}$ are context-free. We can now verify these claims.

First, let us consider the case of intersection. Suppose we define languages A and B as follows:

$$\begin{aligned} A &= \{0^n1^n2^m \,:\, n, m \in \mathbb{N}\}, \\ B &= \{0^n1^m2^m \,:\, n, m \in \mathbb{N}\}. \end{aligned} \tag{10.15}$$

These are certainly context-free languages—a CFG generating A is given by

$$\begin{aligned} S &\rightarrow X\,Y \\ X &\rightarrow 0\,X\,1 \;\big|\; \varepsilon \\ Y &\rightarrow 2\,Y \;\big|\; \varepsilon \end{aligned} \tag{10.16}$$

and a CFG generating B is given by

$$\begin{aligned} S &\rightarrow X\,Y \\ X &\rightarrow 0\,X \;\big|\; \varepsilon \\ Y &\rightarrow 1\,Y\,2 \;\big|\; \varepsilon \end{aligned} \tag{10.17}$$

On the other hand, the intersection $A \cap B$ is not context-free, as our first proposition from the previous section established.

Having proved that the context-free languages are not closed under intersection, it follows immediately that the context-free languages are not closed under complementation. This is because we already know that the context-free languages are closed under union, and if they were also closed under complementation we would conclude that they must also be closed under intersection by De Morgan's laws.

Finally, let us observe that one can sometimes use closure properties to prove that certain languages are not context-free. For example, consider the language

$$D = \big\{ w \in \{0,1,2\}^* \,:\, |w|_0 = |w|_1 = |w|_2 \big\}. \tag{10.18}$$

It would be possible to prove that D is not context-free using the pumping lemma in a similar way to the first proposition from the previous section. A simpler way to conclude this fact is as follows. We assume toward contradiction that D is context-free. Because the intersection of a context-free language and a regular language must always be context-free, it follows that $D \cap \mathrm{L}(0^*1^*2^*)$ is context-free (because $\mathrm{L}(0^*1^*2^*)$ is the language matched by a regular expression and is therefore regular). However,

$$D \cap \mathrm{L}(0^*1^*2^*) = \big\{ 0^m 1^m 2^m \,:\, m \in \mathbb{N} \big\}, \tag{10.19}$$

which we already know is not context-free. Having obtained a contradiction, we conclude that D is not context-free, as required.

Lecture 11

Further discussion of context-free languages

This is the last lecture of the course that is devoted to context-free languages. As for regular languages, however, we will refer to context-free languages from time to time throughout the remainder of the course.

The first part of the lecture will focus on the pushdown automata model of computation, which gives an alternative characterization of context-free languages to the definition based on CFGs, and the second part of the lecture will be devoted to some properties of context-free languages that we have not discussed thus far.

11.1 Pushdown automata

The *pushdown automaton* (or PDA) model of computation is essentially what you get if you equip NFAs each with a single stack. It turns out that the class of languages recognized by PDAs is precisely the class of context-free languages, which provides a useful tool for reasoning about these languages.

In this course we will treat the PDA model as being *optional*—you will not be asked questions that are directly about this model or that require you to use it, but you are free to make use of it if you choose. It is arguably sometimes easier to reason about context-free languages using PDAs than it is using CFGs, or you may find that you have a personal preference for this model, so familiarizing yourself with it may be to you advantage.

A few simple examples

Let us begin with an example of a PDA, expressed in the form of a state diagram in Figure 11.1. The state diagram naturally looks a bit different from the state diagram

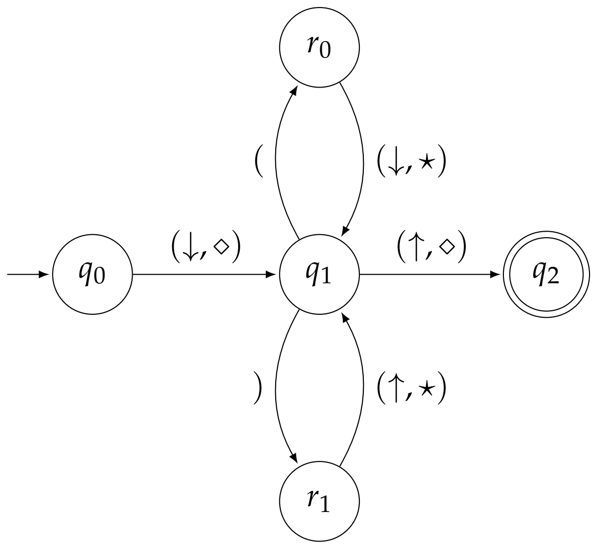


Figure 11.1: The state diagram of a PDA P

of an NFA or DFA, because it includes instructions for operating with the stack, but the basic idea is the same. A transition labeled by an input symbol or ε means that we read a symbol or take an ε-transition, just like an NFA; a transition labeled $(\downarrow, a)$ means that we *push* the symbol a onto the stack; and a transition labeled $(\uparrow, a)$ means that we *pop* the symbol a off of the stack.

Thus, the way the PDA P illustrated in Figure 11.1 works is that it first pushes the stack symbol $\diamond$ onto the stack (which we assume is initially empty) and enters state q_1 (without reading anything from the input). From state q_1 it is possible to either read the left-parenthesis symbol "(" and move to r_0 or read the right-parenthesis symbol ")" and move to r_1. To get back to q_1 we must either push the symbol $*$ onto the stack (in the case that we just read a left-parenthesis) or pop the symbol $*$ off of the stack (in the case that we just read a right-parenthesis). Finally, to get to the accept state q_2 from q_1, we must pop the symbol $\diamond$ off of the stack. Note that a transition requiring a pop operation can only be followed if that symbol is actually there on the top of the stack to be popped. It is not too hard to see that the language recognized by this PDA is the language BAL of balanced parentheses—these are precisely the input strings for which it will be possible to perform the required pops to land on the accept state q_2 after the entire input string is read.

A second example is given in Figure 11.2. In this case the PDA accepts every string in the language

$$\{0^n 1^n : n \in \mathbb{N}\}. \tag{11.1}$$

In this case the stack is essentially used as a counter: we push a star for every 0,

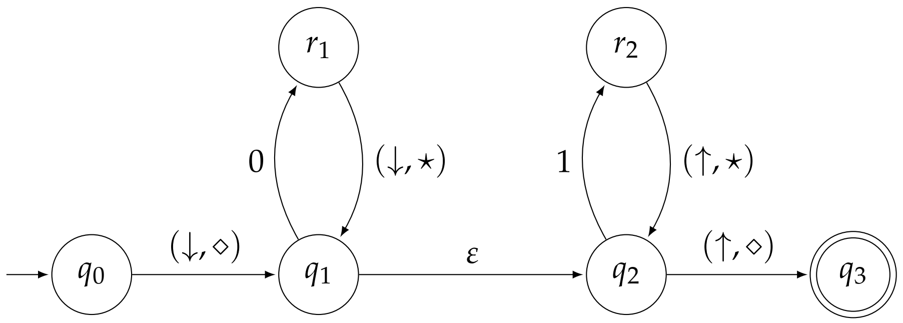


Figure 11.2: A PDA recognizing the language $\{0^n 1^n : n \in \mathbb{N}\}$.

pop a star for every 1, and by using the "bottom of the stack marker" $\diamond$ we check that an equal number of the two symbols have been read.

Definition of pushdown automata

The formal definition of the pushdown automata model is similar to that of nondeterministic finite automata, except that one must also specify the alphabet of stack symbols and take the transition function to have a form that includes a description of how the stack operates.

Definition 11.1. A *pushdown automaton* (or PDA for short) is 6-tuple

$$P = (Q, \Sigma, \Gamma, \delta, q_0, F) \tag{11.2}$$

where Q is a finite and nonempty *set of states*, Σ is an alphabet (called the *input alphabet*), Γ is an alphabet (called the *stack alphabet*), δ is a function of the form

$$\delta : Q \times \big(\Sigma \cup \mathrm{Stack}(\Gamma) \cup \{\varepsilon\}\big) \to \mathcal{P}(Q), \tag{11.3}$$

where $\mathrm{Stack}(\Gamma) = \{\downarrow, \uparrow\} \times \Gamma$, $q_0 \in Q$ is the *start state*, and $F \subseteq Q$ is a *set of accept states*. It is required that $\Sigma \cap \mathrm{Stack}(\Gamma) = \varnothing$.

The way to interpret the transition function having the above form is that the set of possible labels on transitions is

$$\Sigma \cup \mathrm{Stack}(\Gamma) \cup \{\varepsilon\}; \tag{11.4}$$

we can either read a symbol σ, push a symbol from Γ onto the stack, pop a symbol from Γ off of the stack, or take an ε-transition.

Strings of valid stack operations

Before we discuss the formal definition of acceptance for PDAs, it will be helpful to think about stacks and valid sequences of stack operations. Consider an alphabet Γ that we will think of as representing a stack alphabet, and define an alphabet

$$\text{Stack}(\Gamma) = \{\downarrow, \uparrow\} \times \Gamma \tag{11.5}$$

as we have done in Definition 11.1. The alphabet $\text{Stack}(\Gamma)$ represents the possible stack operations for a stack that uses the alphabet Γ; for each $a \in \Gamma$ we imagine that the symbol $(\downarrow, a)$ represents pushing a onto the stack, and that the symbol $(\uparrow, a)$ represents popping a off of the stack.

Now, we can view a string $v \in \text{Stack}(\Gamma)^*$ as either representing or failing to represent a valid sequence of stack operations, assuming we read it from left to right and imagine that we started with an empty stack. If a string does represent a valid sequence of stack operations, we will say that it is a *valid stack string*; and if a string fails to represent a valid sequence of stack operations, we will say that it is an *invalid stack string*.

For example, if $\Gamma = \{0, 1\}$, then these strings are valid stack strings:

$$\begin{gathered} (\downarrow,0)(\downarrow,1)(\uparrow,1)(\downarrow,0)(\uparrow,0)(\uparrow,0), \\ (\downarrow,0)(\downarrow,1)(\uparrow,1)(\downarrow,0)(\uparrow,0). \end{gathered} \tag{11.6}$$

In the first case the stack is transformed like this (where the left-most symbol represents the top of the stack):

$$\varepsilon \rightarrow 0 \rightarrow 10 \rightarrow 0 \rightarrow 00 \rightarrow 0 \rightarrow \varepsilon. \tag{11.7}$$

The second case is similar, except that we don't leave the stack empty at the end:

$$\varepsilon \rightarrow 0 \rightarrow 10 \rightarrow 0 \rightarrow 00 \rightarrow 0. \tag{11.8}$$

On the other hand, these strings are invalid stack strings:

$$\begin{gathered} (\downarrow,0)(\downarrow,1)(\uparrow,0)(\downarrow,0)(\uparrow,1)(\uparrow,0), \\ (\downarrow,0)(\downarrow,1)(\uparrow,1)(\downarrow,0)(\uparrow,0)(\uparrow,0)(\uparrow,1). \end{gathered} \tag{11.9}$$

For the first case we start by pushing 0 and then 1, which is fine, but then we try to pop 0 even though 1 is on the top of the stack. In the second case the very last symbol is the problem: we try to pop 0 even through the stack is empty.

It is the case that the language over the alphabet $\text{Stack}(\Gamma)$ consisting of all valid stack strings is a context-free language. To see that this is so, let us first consider the language of all valid stack strings that also leave the stack empty after the last operation. For instance, the first sequence in (11.6) has this property while the second does not. We can obtain a CFG for this language by mimicking the CFG for the balanced parentheses language, but imagining a different parenthesis type for each symbol. To be more precise, let us define a CFG G so that it includes the rule

$$S \rightarrow (\downarrow, a)\, S\, (\uparrow, a)\, S \tag{11.10}$$

for each symbol $a \in \Gamma$, as well as the rule $S \rightarrow \varepsilon$. This CFG generates the language of valid stack strings for the stack alphabet Γ that leave the stack empty at the end.

If we drop the requirement that the stack be left empty after the last operation, then we still have a context-free language. This is because this is the language of all prefixes of the language generated by the CFG in the previous paragraph, and the context-free languages are closed under taking prefixes.

Definition of acceptance for PDAs

Next let us consider a formal definition of what it means for a PDA P to accept or reject a string w.

Definition 11.2. Let $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$ be a PDA and let $w \in \Sigma^*$ be a string. The PDA P *accepts* the string w if there exists a natural number $m \in \mathbb{N}$, a sequence of states $r_0, \ldots, r_m$, and a sequence

$$a_1, \ldots, a_m \in \Sigma \cup \text{Stack}(\Gamma) \cup \{\varepsilon\} \tag{11.11}$$

for which these properties hold:

1. $r_0 = q_0$ and $r_m \in F$,
2. $r_{k+1} \in \delta(r_k, a_{k+1})$ for every $k \in \{0, \ldots, m-1\}$,
3. by removing every symbol from the alphabet $\text{Stack}(\Gamma)$ from $a_1 \cdots a_m$, the input string w is obtained, and
4. by removing every symbol from the alphabet Σ from $a_1 \cdots a_m$, a valid stack string is obtained.

If P does not accept w, then P *rejects* w.

For the most part the definition is straightforward—in order for P to accept w, there must exist a sequence of states, along with moves between these states, that agree with the input string and the transition function. In addition, the usage of the stack must be consistent with our understanding of what a stack is, and this is represented by the fourth property.

As you would expect, for a given PDA P, we let $\text{L}(P)$ denote the *language recognized* by P, which is the language of all strings accepted by P.

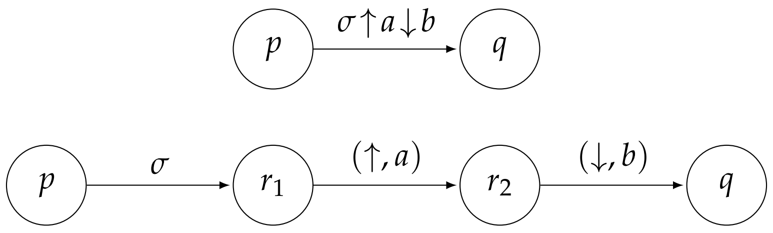


Figure 11.3: The shorthand notation for PDAs appears on the top, and the actual PDA represented by this shorthand notation appears on the bottom.

Some useful shorthand notation for PDA state diagrams

There is a shorthand notation for PDA state diagrams that is sometimes useful, which is essentially to represent a sequence of transitions as if it were a single transition. In particular, if a transition is labeled

$$\sigma \uparrow a \downarrow b, \tag{11.12}$$

the meaning is that the symbol σ is read, a is popped off of the stack, and then b is pushed onto the stack. Figure 11.3 illustrates how this shorthand is to be interpreted. It is to be understood that the "implicit" states in a PDA represented by this shorthand are unique to each edge. For instance, the states r_1 and r_2 in Figure 11.3 are only used to implement this one transition from p to q, and are not reachable from any other states or used to implement other transitions.

This sort of shorthand notation can also be used in case multiple symbols are to be pushed or popped. For instance, an edge labeled

$$\sigma \uparrow a_1 a_2 a_3 \downarrow b_1 b_2 b_3 b_4 \tag{11.13}$$

means that σ is read, $a_1a_2a_3$ is popped off the top of the stack, and $b_1b_2b_3b_4$ is pushed onto the stack. We will always follow the convention that the top of the stack corresponds to the left-hand side of any string of stack symbols, so such a transition requires a_1 on the top of the stack, a_2 next on the stack, and a_3 third on the stack—and when the entire operation is done, b_1 is on top of the stack, b_2 is next, and so on. One can follow a similar pattern to what is shown in Figure 11.3 to implement such a transition using the ordinary types of transitions from the definition of PDAs, along with intermediate states to perform the operations in the right order. Finally, we can simply omit parts of a transition of the above form if those parts are not used. For instance, the transition label

$$\sigma \uparrow a \tag{11.14}$$

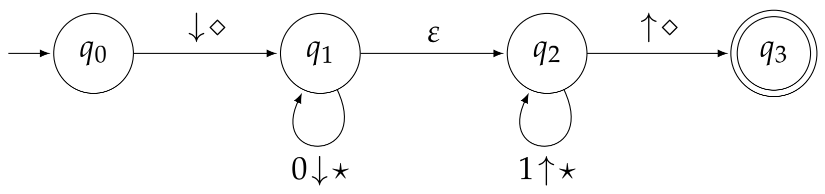


Figure 11.4: The state diagram of a PDA for $\{0^n 1^n : n \in \mathbb{N}\}$ using the shorthand notation for PDA transitions.

means "read σ, pop a, and push nothing," the transition label

$$\uparrow a \downarrow b_1 b_2 \tag{11.15}$$

means "read nothing, pop a, and push $b_1 b_2$," and so on. Figure 11.4 illustrates the same PDA as in Figure 11.2 using this shorthand.

A remark on deterministic pushdown automata

It must be stressed that pushdown automata are, by default, considered to be nondeterministic. It is possible to define a deterministic version of the PDA model, but if we do this we end up with a strictly weaker computational model. That is, every deterministic PDA will recognize a context-free language, but some context-free languages cannot be recognized by a deterministic PDA.

An example is the language PAL of palindromes over the alphabet $\Sigma = \{0, 1\}$; this language is recognized by the PDA in Figure 11.5, but no deterministic PDA can recognize it. We will not prove this—and indeed we have not even discussed a formal definition for deterministic PDAs—but the intuition is clear enough. Deterministic PDAs cannot detect when they have reached the middle of a string, and for this reason the use of a stack is not enough to recognize palindromes—no matter how you do it, the machine will never know when to stop pushing and start popping. A nondeterministic machine, on the other hand, can simply guess when to do this.

11.2 Further examples

Next we will consider a few additional operations under which the context-free languages are closed. These include string reversals, symmetric differences with finite languages, and a couple of operations that involve inserting and deleting certain alphabet symbols from strings.

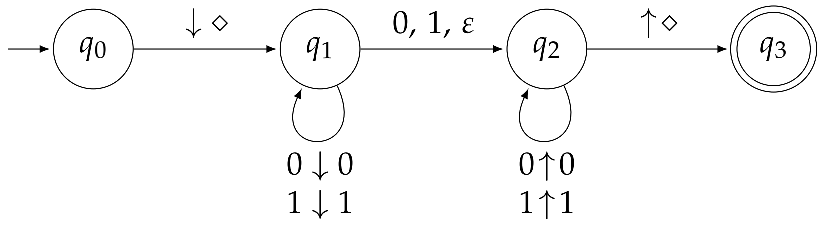


Figure 11.5: A PDA recognizing the language PAL.

Reverse

We already discussed string reversals in Lecture 6, where we observed that the reverse of a regular language is always regular. The same thing is true of context-free languages, as the following simple proposition establishes.

Proposition 11.3. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a context-free language. The language A^{R} is context-free.*

Proof. Because A is context-free, there must exists a CFG G such that $A = \mathrm{L}(G)$. Define a new CFG H as follows: H contains exactly the same variables as G, and for each rule $X \rightarrow w$ of G we include the rule $X \rightarrow w^{\mathrm{R}}$ in H. In words, H is the CFG obtained by reversing the right-hand side of every rule in G. It is evident that $\mathrm{L}(H) = \mathrm{L}(G)^{\mathrm{R}} = A^{\mathrm{R}}$, and therefore A^{R} is context-free. □

Symmetric difference with a finite language

Next we will consider symmetric differences, which were also defined in Lecture 6. It is certainly not the case that the symmetric difference between two context-free languages is always context-free, or even that the symmetric difference between a context-free language and a regular language is context-free.

For example, if $A \subseteq \Sigma^*$ is context-free but $\overline{A}$ is not, then the symmetric difference between A and the regular language Σ^* is not context-free, as

$$A \vartriangle \Sigma^* = \overline{A}. \tag{11.16}$$

On the other hand, the symmetric difference between a context-free language and a finite language must always be context-free, as the following proposition shows. This is interesting because the symmetric difference between a given language and a finite language carries an intuitive meaning: it means we modify that language on a finite number of strings, by either including or excluding them.

The proposition therefore shows that the property of being context-free does not change when a language is modified on a finite number of strings.

Proposition 11.4. *Let Σ be an alphabet, let $A \subseteq \Sigma^*$ be a context-free language, and let and $B \subseteq \Sigma^*$ be a finite language. The the language $A \vartriangle B$ is context-free.*

Proof. First, given that B is finite, we have that B is regular, and therefore $\overline{B}$ is regular as well, because the regular languages are closed under complementation. This implies that $A \cap \overline{B}$ is context-free, because the intersection of a context-free language and a regular language is context-free.

Next, we observe that $\overline{A} \cap B$ is contained in B, and is therefore finite. Every finite language is context-free, and therefore $\overline{A} \cap B$ context-free.

Finally, given that we have proved that both $A \cap \overline{B}$ and $\overline{A} \cap B$ are context-free, it holds that $A \vartriangle B = \bigl(A \cap \overline{B}\bigr) \cup \bigl(\overline{A} \cap B\bigr)$ is context-free because the union of two context-free languages is necessarily context-free. □

Closure under string projections

Suppose that Σ and Γ are disjoint alphabets, and we have a string $w \in (\Sigma \cup \Gamma)^*$ that may contain symbols from either or both of these alphabets. We can imagine *deleting* all of the symbols in w that are contained in the alphabet Γ, which leaves us with a string over Σ. Sometimes we call such an operation a *projection* of a string over the alphabet $\Sigma \cup \Gamma$ onto the alphabet Σ.

We will prove two simple closure properties of the context-free languages that concern this notion. The first one says that if you have a context-free language over the alphabet $\Sigma \cup \Gamma$, and you delete all of the symbols in Γ from all of the strings in A, you're left with a context-free language.

Proposition 11.5. *Let Σ and Γ be disjoint alphabets, let $A \subseteq (\Sigma \cup \Gamma)^*$ be a context-free language, and define*

$$B = \left\{ w \in \Sigma^* \; : \; \begin{array}{l} \text{there exists a string } x \in A \text{ such that } w \text{ is} \\ \text{obtained from } x \text{ by deleting all symbols in } \Gamma \end{array} \right\}. \tag{11.17}$$

It holds that B is context-free.

Proof. Because A is context-free, there exists a CFG G in Chomsky normal form such that $\mathrm{L}(G) = A$. We will create a new CFG H as follows:

1. For every rule of the form $X \rightarrow Y\,Z$ appearing in G, include the same rule in H. Also, if the rule $S \rightarrow \varepsilon$ appears in G, include this rule in H as well.
2. For every rule of the form $X \rightarrow \sigma$ in G, where $\sigma \in \Sigma$, include the same rule $X \rightarrow \sigma$ in H.

3. For every rule of the form $X \rightarrow \tau$ in G, where $\tau \in \Gamma$, include the rule $X \rightarrow \varepsilon$ in H.

It is apparent that $\mathrm{L}(H) = B$, and therefore B is context-free. $\square$

We can also go the other way, so to speak: if A is a context-free language over the alphabet Σ, and we consider the language consisting of all strings over the alphabet $\Sigma \cup \Gamma$ that result in a string in A once all of the symbols in Γ are deleted, then this new language over $\Sigma \cup \Gamma$ will also be context-free. In essence, this is the language you get by picking any string in A, and then inserting any number of symbols from Γ anywhere into the string.

Proposition 11.6. *Let Σ and Γ be disjoint alphabets, let $A \subseteq \Sigma^*$ be a context-free language, and define*

$$B = \left\{ x \in (\Sigma \cup \Gamma)^* \; : \; \begin{array}{l} \textit{the string } w \textit{ obtained from } x \textit{ by deleting} \\ \textit{all symbols in } \Gamma \textit{ satisfies } w \in A \end{array} \right\}. \tag{11.18}$$

It holds that B is context-free.

Proof. Because A is context-free, there exists a CFG G in Chomsky normal for such that $\mathrm{L}(G) = A$. Define a new CFG H as follows:

1. Include the rule
$$W \rightarrow \sigma W \tag{11.19}$$
in H for each $\sigma \in \Gamma$, as well as the rule $W \rightarrow \varepsilon$, for W being a variable that is not already used in G. The variable W generates any string of symbols from Γ, including the empty string.
2. For each rule of the form $X \rightarrow Y Z$ in G, include the same rule in H without modifying it.
3. For each rule of the form $X \rightarrow \tau$ in G, include this rule in H:
$$X \rightarrow W\tau W \tag{11.20}$$
4. If the rule $S \rightarrow \varepsilon$ is contained in G, then include this rule in H:
$$S \rightarrow W \tag{11.21}$$

Intuitively speaking, H operates in much the same way as G, except that any time G generates a symbol or the empty string, H is free to generate the same string with any number of symbols from Γ inserted. It holds that $\mathrm{L}(H) = B$, and therefore B is context-free. $\square$

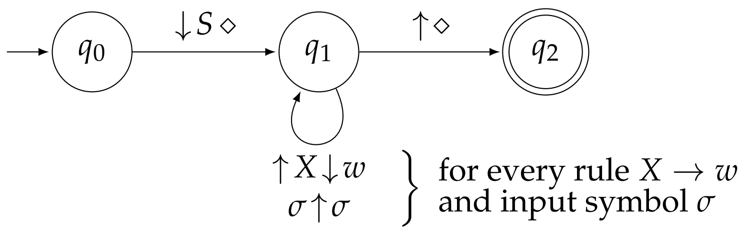


Figure 11.6: A PDA recognizing the language of an arbitrary CFG.

11.3 Equivalence of PDAs and CFGs

As suggested earlier in the lecture, it is the case that a language is context-free if and only if it is recognized by a PDA. You will not be tested on any of the details of how this is proved—but in case you are interested, this section gives a high-level description of one way to prove this equivalence.

Every context-free language is recognized by a PDA

To prove that every context-free language is recognized by some PDA, we can define a PDA that corresponds directly to a given CFG. That is, if $G = (V, \Sigma, R, S)$ is a CFG, then we can obtain a PDA P such that $\mathrm{L}(P) = \mathrm{L}(G)$ in the manner suggested by Figure 11.6. The stack symbols of P are taken to be $V \cup \Sigma$, along with a special bottom of the stack marker $\diamond$ (which we assume is not contained in $V \cup \Sigma$), and during the computation the stack will provide a way to store the symbols and variables needed to carry out a derivation with respect to the grammar G.

If you consider how derivations of strings by a grammar G and the operation of the corresponding PDA P work, it will be evident that P accepts precisely those strings that can be generated by G. We start with just the start variable on the stack (in addition to the bottom of the stack marker). In general, if a variable appears on the top of the stack, we can pop it off and replace it with any string of symbols and variables appearing on the right-hand side of a rule for the variable that was popped; and if a symbol appears on the top of the stack we essentially just match it up with an input symbol—so long as the input symbol matches the symbol on the top of the stack we can pop it off, move to the next input symbol, and process whatever is left on the stack. We can move to the accept state whenever the stack is empty (meaning that just the bottom of the stack marker is present), and if all of the input symbols have been read we accept. This situation is representative of the input string having been derived by the grammar.

Every language recognized by a PDA is context-free

We will now argue that every language recognized by a PDA is context-free. There is a method through which a given PDA can actually be converted into an equivalent CFG, but it is messy and the intuition tends to get lost in the details. Here we will summarize a different way to prove that every language recognized by a PDA is context-free that is pretty simple (given the tools that we've already collected in our study of context-free languages). If you wanted to, you could turn this proof into an explicit construction of a CFG for a given PDA, and it wouldn't be all that different from the method just mentioned—but we'll focus just on the proof and not on turning it into an explicit construction.

Suppose we have a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, F)$. The transition function δ takes the form

$$\delta : Q \times \bigl(\Sigma \cup \text{Stack}(\Gamma) \cup \{\varepsilon\}\bigr) \rightarrow \mathcal{P}(Q), \tag{11.22}$$

so if we wanted to, we could think of P as being an NFA for some language over the alphabet $\Sigma \cup \text{Stack}(\Gamma)$. Slightly more formally, let N be the NFA defined as

$$N = \bigl(Q, \Sigma \cup \text{Stack}(\Gamma), \delta, q_0, F\bigr); \tag{11.23}$$

we don't even need to change the transition function because it already has the right form of a transition function for an NFA over the alphabet $\Sigma \cup \text{Stack}(\Gamma)$. Also define $B = \mathrm{L}(N) \subseteq (\Sigma \cup \text{Stack}(\Gamma))^*$ to be the language recognized by N. In general, the strings in B include symbols in both Σ and $\text{Stack}(\Gamma)$. Even though symbols in $\text{Stack}(\Gamma)$ may be present in the strings accepted by N, there is no requirement on these strings to actually represent a valid use of a stack—because N doesn't have a stack with which to check this condition.

Now let us consider a second language $C \subseteq (\Sigma \cup \text{Stack}(\Gamma))^*$. This will be the language consisting of all strings over the alphabet $\Sigma \cup \text{Stack}(\Gamma)$ having the property that by deleting every symbol in Σ, a valid stack string is obtained. We already discussed the fact that the language consisting of all valid stack strings is context-free, and so it follows from Proposition 11.6 that the language C is also context-free.

Next, we consider the intersection $D = B \cap C$. Because D is the intersection of a regular language and a context-free language, it is context-free. The strings in D actually correspond to valid computations of the PDA P that lead to an accept state; but in addition to the input symbols in Σ that are read by P, these strings also include symbols in $\text{Stack}(\Gamma)$ that represent transitions of P that involve stack operations. The language D is therefore not the same as the language A, but it is closely related—A is the language that is obtained from D by deleting all of the symbols in $\text{Stack}(\Gamma)$ and leaving the symbols in Σ alone. Because we know that D is context-free, it therefore follows that A is context-free by Proposition 11.5, which is what we wanted to prove.

Lecture 12

The Turing machine model of computation

For most of the remainder of the course we will study the *Turing machine* model of computation, named after Alan Turing (1912–1954) who proposed the model in 1936. Turing machines are intended to provide a simple mathematical abstraction of general computations. As we study this model, it may help you to keep this thought in the back of your mind.

The idea that Turing machine computations are representative of a fully general computational model is called the *Church–Turing thesis*. Here is one statement of this thesis (although it is the idea rather than the exact choice of words that is important):

Church–Turing thesis: Any function that can be computed by a mechanical process can be computed by a Turing machine.

This is not a mathematical statement that can be proved or disproved—if you wanted to try to prove a statement along these lines, the first thing you would most likely do is look for a mathematical definition of what it means for a function to be "computed by a mechanical process," and this is precisely what the Turing machine model was intended to provide.

While people have actually built machines that behave like Turing machines, this is mostly a recreational activity—the Turing machine model was never intended to serve as a practical device for performing computations. Rather, it was intended to provide a rigorous mathematical foundation for reasoning about computation, and it has served this purpose very well since its invention.

It is worth mentioning that there are other mathematical models that play a similar role to Turing machines, as a way of formalizing the notion of computation. For instance, Alonzo Church proposed λ-calculus a short time before Turing proposed what we now call the Turing machine, and a sketch of a proof of the

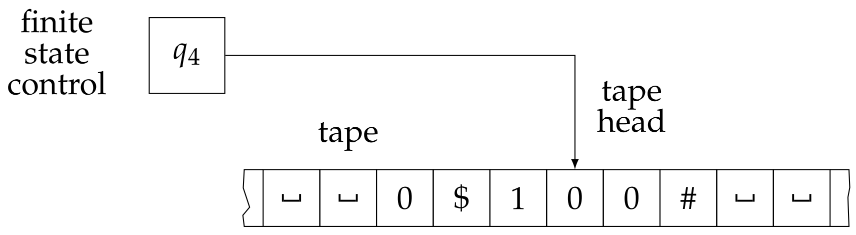


Figure 12.1: An illustration of the three components of a Turing machine: the finite state control, the tape head, and the tape. The tape is infinite in both directions (although it appears that this Turing machine's tape was torn at both ends to fit it into the figure).

equivalence of the two models appears in Turing's 1936 paper. Others, including Kurt Gödel, Stephen Kleene, and Emil Post, also contributed important ideas in the development of these notions. It is the case, however, that the Turing machine model provides a particularly clean and appealing way of formalizing computation, which is a reasons why this model is often the one that is preferred.

12.1 Turing machine components and definitions

We will begin with an informal description of the Turing machine model before stating the formal definition. There are three components of a Turing machine:

1. The *finite state control*. This component is in one of a finite number of states at each instant, and is connected to the tape head component.
2. The *tape head*. This component scans one of the tape squares of the tape at each instant, and is connected to the finite state control. It can read and write symbols from/onto the tape, and it can move left and right along the tape.
3. The *tape*. This component consists of an infinite number of tape squares, each of which can store one of a finite number of tape symbols at each instant. The tape is infinite both to the left and to the right.

Figure 12.1 illustrates these three components and the way they are connected.

The idea is that the action of a Turing machine at each instant is determined by the state of the finite state control together with just the one symbol that is stored by the tape square that the tape head is currently reading. Thus, the action is determined by a finite number of possible alternatives: one action for each state/symbol pair. Depending on the state and the symbol being scanned, the action that the machine is to perform may involve changing the state of the finite state control, changing the symbol on the tape square being scanned, and/or moving the tape head to the left or right. Once this action is performed, the machine will again have some state for its finite state control and will be reading some symbol on its tape, and the process continues. One may consider both deterministic and nondeterministic variants of the Turing machine model, but our main focus will be on the deterministic version of this model.

We also give Turing machines an opportunity to stop the computational process and produce an output. The possibility of producing a string as an output will be discussed later, but for now we'll concentrate just on *accepting* or *rejecting*, as we did for DFAs (for instance). To do this, we require that there are two special states of the finite state control: an *accept* state q_{acc} and a *reject* state q_{rej}. If the machine enters one of these two states, the computation immediately stops and *accepts* or *rejects* accordingly. We do not allow the possibility that q_{acc} and q_{rej} are the same state—these must be distinct states.

When a Turing machine begins a computation, an input string is written on its tape, and every other tape square is initialized to a special *blank* symbol (which may not be included in the input alphabet). Naturally, we need an actual symbol to represent the blank symbol in these notes, and we will use the symbol $\llcorner\!\lrcorner$ for this purpose. More generally, we will allow the possible symbols written on the tape to include other non-input symbols in addition to the blank symbol, as it is sometimes convenient to allow this possibility.

Formal definition of DTMs

With the informal description of Turing machines from above in mind, we will now proceed to the formal definition.

Definition 12.1. A *deterministic Turing machine* (or DTM, for short) is a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}), \tag{12.1}$$

where Q is a finite and nonempty set of *states*; Σ is an alphabet, called the *input alphabet*, which may not include the blank symbol $\llcorner\!\lrcorner$; Γ is an alphabet, called the *tape alphabet*, which must satisfy $\Sigma \cup \{\llcorner\!\lrcorner\} \subseteq \Gamma$; δ is a transition function having the form

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}; \tag{12.2}$$

$q_0 \in Q$ is the *initial state*; and $q_{\text{acc}}, q_{\text{rej}} \in Q$ are the *accept* and *reject* states, which satisfy $q_{\text{acc}} \neq q_{\text{rej}}$.

The interpretation of the transition function is as follows. Suppose the DTM is currently in a state $q \in Q$, the symbol stored in the tape square being scanned by

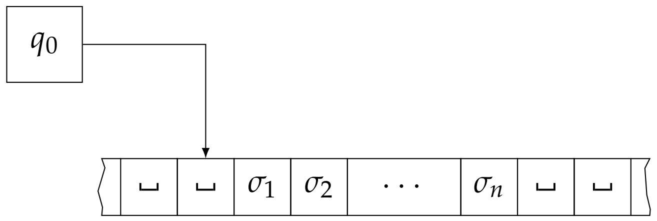


Figure 12.2: The initial configuration of a DTM when run on input $w = \sigma_1\sigma_2 \cdots \sigma_n$.

the tape head is σ, and it holds that $\delta(q, \sigma) = (r, \tau, D)$ for $D \in \{\leftarrow, \rightarrow\}$. The action performed by the DTM is then to (i) change state to r, (ii) overwrite the contents of the tape square being scanned by the tape head with τ, and (iii) move the tape head in direction D (either left or right). In the case that the state is q_{acc} or q_{rej}, the transition function does not specify an action, because we assume that the DTM stops once it reaches one of these two states.

12.2 Turing machine computations

If we have a DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, and we wish to consider its operation on some input string $w \in \Sigma^*$, we assume that it is started with its components initialized as illustrated in Figure 12.2. That is, the input string is written on the tape, one symbol per square, with each other tape square containing the blank symbol, and with the tape head scanning the tape square immediately to the left of the first input symbol. (In the case that the input string is ε, all of the tape squares start out storing blanks.)

Once the initial arrangement of the DTM is set up, the DTM begins taking *steps*, as determined by the transition function δ in the manner suggested above. So long as the DTM does not enter one of the two states q_{acc} or q_{rej}, the computation continues. If the DTM eventually enters the state q_{acc}, it *accepts* the input string, and if it eventually enters the state q_{rej}, it *rejects* the input string. Notice that there are three possible alternatives for a DTM M on a given input string w:

1. M accepts w.
2. M rejects w.
3. M runs forever on input w.

In some cases one can design a particular DTM so that the third alternative does not occur, but in general this could be what happens (assuming that nothing external to the DTM interrupts the computation).

Representing configurations of DTMs

In order to speak more precisely about Turing machines and state a formal definition concerning their behavior, we will require a bit more terminology. When we speak of a *configuration* of a DTM, we are speaking of a description of all of the Turing machine's components at some instant:

1. the *state* of the finite state control,
2. the *contents* of the entire tape, and
3. the *tape head position* on the tape.

Rather than drawing pictures depicting the different parts of Turing machines, such as was done in Figure 12.2, we use the following compact notation to represent configurations. If we have a DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$, and we wish to refer to a configuration of this DTM, we express it in the form

$$u(q, a)v \tag{12.3}$$

for some state $q \in Q$, a tape symbol $a \in \Gamma$, and (possibly empty) strings of tape symbols u and v such that

$$u \in \Gamma^* \backslash \{\sqcup\}\Gamma^* \quad \text{and} \quad v \in \Gamma^* \backslash \Gamma^*\{\sqcup\} \tag{12.4}$$

(i.e., u and v are strings of tape symbols such that u does not start with a blank and v does not end with a blank). What the expression (12.3) means is that the string uav is written on the tape in consecutive squares, with all other tape squares containing the blank symbol; the state of M is q; and the tape head of M is positioned over the symbol a that occurs between u and v.

For example, the configuration of the DTM in Figure 12.1 is expressed as

$$0\$1(q_4, 0)0\# \tag{12.5}$$

while the configuration of the DTM in Figure 12.2 is

$$(q_0, \sqcup)w \tag{12.6}$$

(for $w = \sigma_1 \cdots \sigma_n$).

When working with descriptions of configurations, it is convenient to define a few functions as follows. We define $\alpha : \Gamma^* \to \Gamma^* \backslash \{\sqcup\}\Gamma^*$ and $\beta : \Gamma^* \to \Gamma^* \backslash \Gamma^*\{\sqcup\}$ recursively as

$$\begin{aligned} \alpha(w) &= w && (\text{for } w \in \Gamma^* \backslash \{\sqcup\}\Gamma^*) \\ \alpha(\sqcup w) &= \alpha(w) && (\text{for } w \in \Gamma^*) \end{aligned} \tag{12.7}$$

and

$$\begin{aligned} \beta(w) &= w && (\text{for } w \in \Gamma^* \backslash \Gamma^* \{␣\}) \\ \beta(w␣) &= \beta(w) && (\text{for } w \in \Gamma^*), \end{aligned} \tag{12.8}$$

and we define

$$\gamma : \Gamma^*(Q \times \Gamma)\Gamma^* \rightarrow \big(\Gamma^* \backslash \{␣\}\Gamma^*\big)(Q \times \Gamma)\big(\Gamma^* \backslash \Gamma^* \{␣\}\big) \tag{12.9}$$

as

$$\gamma(u(q,a)v) = \alpha(u)(q,a)\beta(v) \tag{12.10}$$

for all $u, v \in \Gamma^*$, $q \in Q$, and $a \in \Gamma$. This may look more complicated than it really is—the function γ just throws away all blank symbols on the left-most end of u and the right-most end of v, so that a proper expression of a configuration remains.

A yields relation for DTMs

Now we will define a *yields* relation, in a similar way to what we did for context-free grammars. This will in turn allow us to formally define acceptance and rejection for DTMs.

Definition 12.2. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a DTM. We define a *yields* relation $\vdash_M$ on pairs of expressions representing configurations as follows:

1. For every choice of $p \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

$$\delta(p, a) = (q, b, \rightarrow), \tag{12.11}$$

   the yields relation includes these pairs for all $u \in \Gamma^* \backslash \{␣\}\Gamma^*$, $v \in \Gamma^* \backslash \Gamma^* \{␣\}$, and $c \in \Gamma$:

$$\begin{aligned} u(p,a)cv &\vdash_M \gamma\big(ub(q,c)v\big) \\ u(p,a) &\vdash_M \gamma\big(ub(q,␣)\big) \end{aligned} \tag{12.12}$$

2. For every choice of $p \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

$$\delta(p, a) = (q, b, \leftarrow), \tag{12.13}$$

   the yields relation includes these pairs for all $u \in \Gamma^* \backslash \{␣\}\Gamma^*$, $v \in \Gamma^* \backslash \Gamma^* \{␣\}$, and $c \in \Gamma$:

$$\begin{aligned} uc(p,a)v &\vdash_M \gamma\big(u(q,c)bv\big) \\ (p,a)v &\vdash_M \gamma\big((q,␣)bv\big) \end{aligned} \tag{12.14}$$

In addition, we let $\vdash_M^*$ denote the reflexive, transitive closure of $\vdash_M$. That is, we have

$$u(p,a)v \vdash_M^* y(q,b)z \tag{12.15}$$

if and only if there exists an integer $m \geq 1$, strings $w_1,\ldots,w_m,x_1,\ldots,x_m \in \Gamma^*$, symbols $c_1,\ldots,c_m \in \Gamma$, and states $r_1,\ldots,r_m \in Q$ such that $u(p,a)v = w_1(r_1,c_1)x_1$, $y(q,b)v = w_m(r_m,c_m)x_m$, and

$$w_k(r_k,c_k)x_k \vdash_M w_{k+1}(r_{k+1},c_{k+1})x_{k+1} \tag{12.16}$$

for all $k \in \{1,\ldots,m-1\}$.

That definition looks a bit technical, but it is actually very simple—whenever we have

$$u(p,a)v \vdash_M y(q,b)z \tag{12.17}$$

it means that by running M for one step we move from the configuration represented by $u(p,a)v$ to the configuration represented by $y(q,b)z$; and whenever we have

$$u(p,a)v \vdash_M^* y(q,b)z \tag{12.18}$$

it means that by running M for some number of steps, possibly zero steps, we will move from the configuration represented by $u(p,a)v$ to the configuration represented by $y(q,b)z$.

Acceptance and rejection for DTMs

Finally, we can write down a definition for acceptance and rejection by a DTM, using the relation $\vdash_M^*$ we just defined.

Definition 12.3. Let $M = (Q,\Sigma,\Gamma,\delta,q_0,q_{\text{acc}},q_{\text{rej}})$ be a DTM and let $w \in \Sigma^*$ be a string. If there exist strings $u,v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_{\text{init}},\sqcup)w \vdash_M^* u(q_{\text{acc}},a)v, \tag{12.19}$$

then M *accepts* w. If there exist strings $u,v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_{\text{init}},\sqcup)w \vdash_M^* u(q_{\text{rej}},a)v, \tag{12.20}$$

then M *rejects* w. If neither of these conditions hold, then M *runs forever* on input w.

In words, if we start out in the initial configuration of a DTM M on an input w and start it running, we *accept* if we eventually hit the accept state, we *reject* if we eventually hit the reject state, and we *run forever* if neither of these two possibilities holds.

Similar to what we have done for DFAs, NFAs, CFGs, and PDAs, we write $\mathrm{L}(M)$ to denote the language of all strings that are accepted by a DTM M. In the case of DTMs, the language $\mathrm{L}(M)$ doesn't really tell the whole story—a string $w \notin \mathrm{L}(M)$ might either be rejected or it may cause M to run forever—but the notation is useful nevertheless.

Decidable and Turing-recognizable languages

It has been a lecture filled with definitions, but we still need to discuss two more. All of the definitions are important, but these are particularly important because they introduce concepts that we will be discussing for the next several lectures.

Definition 12.4. Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language.

1. The language A is *Turing recognizable* if there exists a DTM M for which it holds that $A = \mathrm{L}(M)$.
2. The language A is *decidable* if there exists a DTM M that satisfies two conditions: (i) $A = \mathrm{L}(M)$, and (ii) M never runs forever (i.e., it either accepts or rejects each input string $w \in \Sigma^*$).

It is evident from the definition that every decidable language is also Turing recognizable—we've added an additional condition on the DTM M for the case of decidability. We will see later that the other containment does not hold: there are languages that are Turing recognizable but not decidable. Understanding which languages are decidable, which are Turing recognizable but not decidable, and what relationships hold among the two classes will occupy us for several lectures, and we'll see many interesting results and ideas about computation along the way.

From time to time you may hear about these concepts by different names. Sometimes the term *recursive* is used in place of *decidable* and *recursively enumerable* is used in place of *Turing recognizable*. These are historical terms based on different, but equivalent, ways of defining these classes of languages.

12.3 A simple example of a Turing machine

Let us now see an example of a DTM. For this first example, we will describe the DTM using a state diagram. The syntax is a bit different, but the idea is similar to state diagrams for DFAs. In the DTM case, we represent the property that the transition function satisfies $\delta(p, a) = (q, b, \rightarrow)$ with a transition of the form

$$\begin{tikzcd} p \arrow[r, "{a, b \rightarrow}"] & q \end{tikzcd}$$

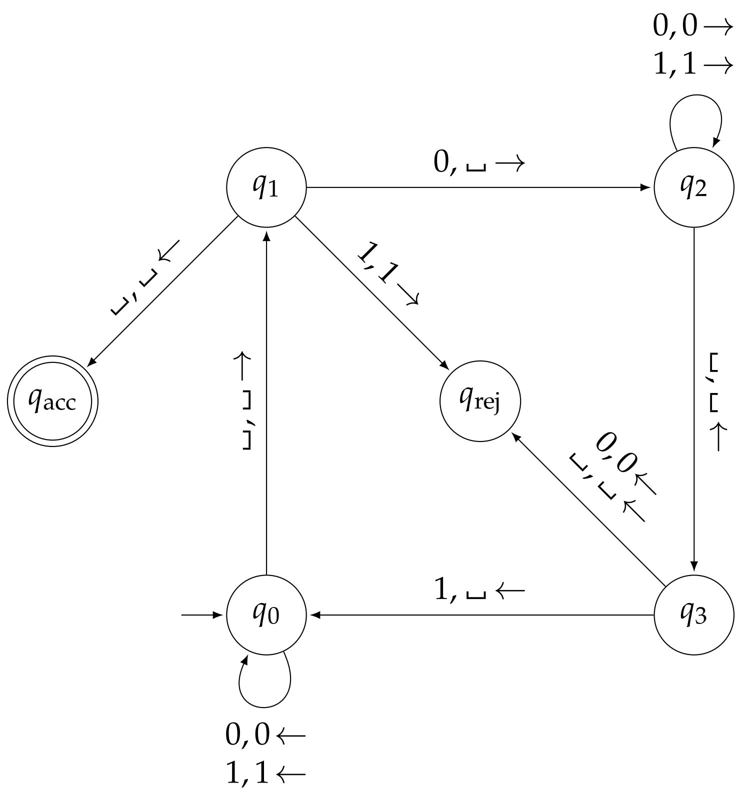


Figure 12.3: A DTM M for the language $\{0^n1^n : n \in \mathbb{N}\}$.

and similarly we represent the property that $\delta(p,a) = (q,b,\leftarrow)$ with a transition of the form

$$\begin{tikzcd} p \arrow[r, "{a,b\leftarrow}"] & q \end{tikzcd}$$

The state diagram for the example is given in Figure 12.3. The DTM M described by this diagram is for the language

$$A = \{0^n1^n : n \in \mathbb{N}\}. \tag{12.21}$$

To be more precise, M accepts every string in A and rejects every string in $\overline{A}$, and therefore the language A is *decidable*.

The specific way that the DTM M works can be summarized as follows. The DTM M starts out with its tape head scanning the blank symbol immediately to the left of its input. It moves the tape head right, and if it sees a 1 it rejects: the input string must not be of the form 0^n1^n if this happens. On the other hand, if it sees another blank symbol, it accepts: the input must be the empty string, which corresponds to the $n = 0$ case in the description of A. Otherwise, it must have seen the symbol 0, and in this case the 0 is erased (meaning that it replaces it with the blank symbol), the tape head repeatedly moves right until a blank is found, and then it moves one square back to the left. If a 1 is not found at this location the DTM rejects: there weren't enough 1s at the right end of the string. Otherwise, if a 1 is found, it is erased, and the tape head moves all the way back to the left, where we essentially recurse on a slightly shorter string.

Of course, the summary just suggested doesn't tell you precisely how the DTM works—but if you didn't already have the state diagram from Figure 12.3, the summary would probably be enough to give you a good idea for how to come up with the state diagram (or perhaps a slightly different one operating in a similar way).

In fact, an even higher-level summary would probably be enough. For instance, we could describe the functioning of the DTM M as follows:

1. Accept if the input is the empty string.
2. Check that the left-most non-blank symbol on the tape is a 0 and that the right-most non-blank symbol is a 1. Reject if this is not the case, and otherwise erase these symbols and goto 1.

There will, of course, be several specific ways to implement this algorithm with a DTM, with the DTM M from Figure 12.3 being one of them.

Back to the Church–Turing thesis

The example of a DTM above was very simple, which makes it atypical. The DTMs we will be most interested in will almost always be much more complicated—so complicated, in fact, that the idea of representing them by state diagrams would be absurd. The reality is that state diagrams turn out to be almost totally useless for describing DTMs, and so we will almost never refer to them; this would be quite similar to describing complex programs using machine code.

The more typical way to describe DTMs will be in terms of *algorithms*, typically expressed in the form of pseudo-code or high-level descriptions like we did at the end of our first example above. As you build your understanding of how DTMs work and what they can do, you'll gain more confidence that these sorts of high-level descriptions actually could be implemented by DTMs. It may take some time for you to convince yourself of this, but perhaps your experience as programmers will help—you already know that very complex algorithms can be implemented through very primitive low-level instructions carried out by a computer, and the situation is similar for Turing machines.

This is connected to the idea that the Church–Turing thesis suggests. If you can implement an algorithm using your favorite programming language, then with enough motivation and stamina you could also implement the same algorithm on a Turing machine. (Of course we are talking about algorithms that take an input string and produce an output, not interactive, multi-media applications or things like this.) We will continue discussing this idea in the next couple of lectures.

Lecture 13

Variants of Turing machines and encoding objects as strings

In this lecture we will continue to discuss the Turing machine model, beginning with some ways in which the model can be changed without affecting its power. We will also start discussing various *encoding schemes* that allow one to represent mathematical objects as strings of symbols, so that they can be viewed as inputs to Turing machines.

13.1 Variants of Turing machines

There is nothing sacred about the specific definition of DTMs that we covered in the previous lecture. In fact, if you look at two different books on the theory of computation, you're pretty likely to see two definitions of Turing machines that differ in one or more respects.

For example, the definition we discussed specifies that a Turing machine's tape is infinite in both directions, but sometimes people choose to define the model so that the tape is only infinite to the right. Naturally, if there is a left-most tape square on the tape, the definition must clearly specify how the Turing machine is to behave if it tries to move its tape head left from this point. Perhaps the Turing machine immediately rejects if its tape head tries to move off the left edge of the tape, or the tape head might remain on the left-most tape square in this situation.

Another example concerns tape head movements. Our definition states that the tape head must move left or right at every step, while some alternative Turing machine definitions allow the possibility for the tape head to remain stationary. It is also common that Turing machines with multiple tapes are considered, and we will indeed consider this Turing machine variant shortly.

DTMs allowing stationary tape heads

Let us begin with a very simple Turing machine variant already mentioned above, where the tape head is permitted to remain stationary on a given step if the DTM designer wishes. This is an extremely minor change to the Turing machine definition, but because it is our first example of a Turing machine variant we will go through it in detail (and perhaps in more detail than it actually deserves).

If the tape head of a DTM is allowed to remain stationary, we would naturally expect that instead of the transition function taking the form

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \rightarrow\}, \tag{13.1}$$

it would instead take the form

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow Q \times \Gamma \times \{\leftarrow, \downarrow, \rightarrow\}, \tag{13.2}$$

where the arrow pointing down indicates that the tape head does not move. Specifically, if it is the case that $\delta(p, a) = (q, b, \downarrow)$, then whenever the machine is in the state p and its tape head is positioned over a square that contains the symbol a, it overwrites a with b, changes state to q, and *leaves the position of the tape head unchanged.*

For the sake of clarity let us give this new model a different name, to distinguish it from the ordinary DTM model we already defined in the previous lecture. In particular, we will define a *stationary-head-DTM* to be a 7-tuple

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}), \tag{13.3}$$

where each part of this tuple is just like an ordinary DTM except that the transition function δ takes the form (13.2).

Now, if we wanted to give a formal definition of what it means for a stationary-head-DTM to accept or reject, we could of course do that. This would require that we extend the yields relation that we discussed last time for ordinary DTMs to account for the possibility that $\delta(p, a) = (q, b, \downarrow)$ for some choices of $p \in Q$ and $a \in \Gamma$. This is actually quite easy—we simply include the following third rule to the two rules that define the yields relation for ordinary DTMs:

3. For every choice of $p \in Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

$$\delta(p, a) = (q, b, \downarrow), \tag{13.4}$$

the yields relation includes these pairs for all $u \in \Gamma^*\backslash\{\sqcup\}\Gamma^*$ and $v \in \Gamma^*\backslash\Gamma^*\{\sqcup\}$:

$$u(p, a)v \vdash_M u(q, b)v \tag{13.5}$$

As suggested before, allowing the tape head to remain stationary doesn't actually change the computational power of the Turing machine model. The standard way to argue that this is so is through the technique of *simulation*. A standard DTM cannot leave its tape head stationary, so it cannot behave *precisely* like a stationary-head-DTM, but it is straightforward to *simulate* a stationary-head-DTM with an ordinary one—by simply moving the tape head to the left and back to the right (for instance), we can obtain the same outcome as we would have if the tape head had remained stationary. Naturally, this requires that we remember what state we are supposed to be in after moving left and back to the right, but it can be done without difficulty.

To be more precise, if

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \tag{13.6}$$

is a stationary-head-DTM, then we can simulate this machine with an ordinary DTM

$$K = (R, \Sigma, \Gamma, \eta, q_0, q_{\text{acc}}, q_{\text{rej}}) \tag{13.7}$$

as follows:

1. For each state $q \in Q$ of M, the state set R of K will include q, as well as a distinct copy of this state that we will denote q'. The intuitive meaning of the state q' is that it indicates that K needs to move its tape head one square to the right and enter the state q.
2. The transition function η of K is defined as

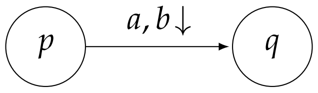
$$\eta(p, a) = \begin{cases} (q, b, \leftarrow) & \text{if } \delta(p, a) = (q, b, \leftarrow) \\ (q, b, \rightarrow) & \text{if } \delta(p, a) = (q, b, \rightarrow) \\ (q', b, \leftarrow) & \text{if } \delta(p, a) = (q, b, \downarrow) \end{cases} \tag{13.8}$$

for each $p \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}$ and $a \in \Gamma$, as well as

$$\eta(q', c) = (q, c, \rightarrow) \tag{13.9}$$

for each $q \in Q$ and $c \in \Gamma$.

Written in terms of state diagrams, one can describe this simulation as follows. Suppose that the state diagram of a stationary-head-DTM M contains a transition that looks like this:



The state diagram for K replaces this transition as follows:

$$\begin{tikzcd}
\boxed{p} \arrow[r, "{a, b \leftarrow}"] & \boxed{q'} \arrow[r, "{c, c \rightarrow}"] & \boxed{q}
\end{tikzcd}$$

(Here, the transition from q' to q is to be included for every tape symbol $c \in \Gamma$. The same state q' can safely be used for every stationary tape head transition into q.)

It is not hard to see that the computation of K will directly mimic the computation of M. The DTM K might take longer to run, because it sometimes requires two steps to simulate one step of M, but this does not concern us. The bottom line is that every language that is either recognized or decided by a stationary-head-DTM is also recognized or decided by an ordinary DTM.

The other direction is trivial: a stationary-head-DTM can easily simulate an ordinary DTM by simply not making use of its ability to leave the tape head stationary. Consequently, the two models are equivalent.

Thus, if you were to decide at some point that it would be more convenient to work with the stationary-head-DTM model, you could switch to this model—and by observing the equivalence we just proved, you would be able to conclude interesting facts concerning the original DTM model. In reality, however, the stationary-head-DTM model just discussed is not a significant enough departure from the ordinary DTM model to really be concerned with. We've gone through the equivalence in detail because it is our first example, but in comparison to some of the other variants of Turing machines we will discuss shortly, it's not worthy of too much thought. For this reason there will not likely be a need for us to refer specifically to the stationary-head-DTM model again.

DTMs with multi-track tapes

Another useful variant of the DTM model is one in which the tape has multiple tracks, as suggested by Figure 13.1. More specifically, we may suppose that the tape has k tracks for some positive integer k, and for each tape head position the tape has k separate tape squares that can each store a symbol. It is useful to allow the different tracks to store different symbols, so we may imagine k different tape alphabets $\Gamma_1, \ldots, \Gamma_k$, with Γ_j being the tape alphabet for track number j.

For example, based on the picture in Figure 13.1 it appears as though the first tape track of this DTM stores symbols from the tape alphabet $\Gamma_1 = \{0, 1, \text{␣}\}$, the second track stores symbols from the tape alphabet $\Gamma_2 = \{\#, \text{␣}\}$, and the third track stores symbols from the tape alphabet $\Gamma_3 = \{\clubsuit, \heartsuit, \diamondsuit, \spadesuit, \text{␣}\}$.

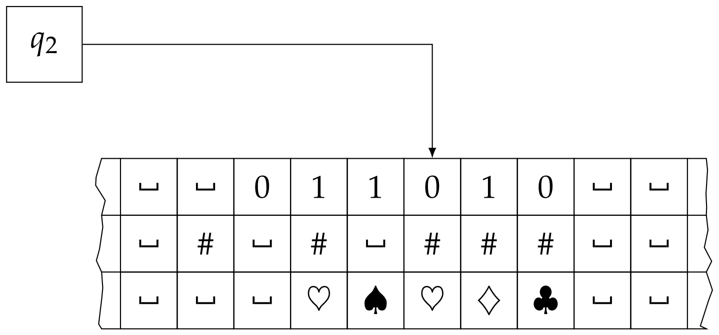


Figure 13.1: A DTM with a three-track tape.

When the tape head scans a particular location on the tape, it can effectively see and modify all of the symbols stored on the tape tracks for this tape head location simultaneously.

It turns out that this isn't really even a variant of the DTM definition at all—it's just an ordinary DTM whose tape alphabet Γ is equal to the Cartesian product

$$\Gamma = \Gamma_1 \times \cdots \times \Gamma_k. \tag{13.10}$$

Given that DTMs are supposed to have tape alphabets that contain all of the possible input symbols, meaning that $\Sigma \subset \Gamma$, and also contain a blank symbol, we should perhaps make clear that we identify each alphabet symbol $\sigma \in \Sigma$ with the tape symbol $(\sigma, \sqcup, \ldots, \sqcup)$, and also that we consider the symbol $(\sqcup, \ldots, \sqcup)$ to be the blank symbol of the multi-track DTM.

DTMs with one-way infinite tapes

DTMs with one-way infinite tapes were mentioned before as a common alternative to DTMs with two-way infinite tapes. Figure 13.2 illustrates such a DTM. Let us say that if the DTM ever tries to move its tape head left when it is on the leftmost tape square, its head simply remains on this square and the computation continues—maybe it makes an unpleasant crunching sound in this situation.

It is easy to simulate a DTM with a one-way infinite tape using an ordinary DTM (with a two-way infinite tape). For instance, we could drop a special symbol, such as ✂, on the two-way infinite tape at the beginning of the computation, to the left of the input. The DTM with the two-way infinite tape will exactly mimic the behavior of the one-way infinite tape, but if the tape head ever scans the special ✂ symbol during the computation, it moves one square right without changing state. This exactly mimics the behavior of the DTM with a one-way infinite tape that was suggested above.

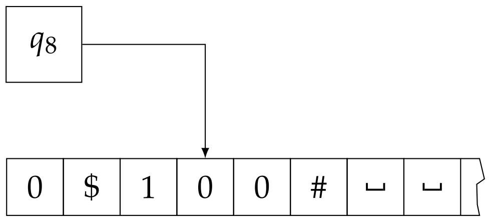


Figure 13.2: A DTM with a one-way infinite tape.

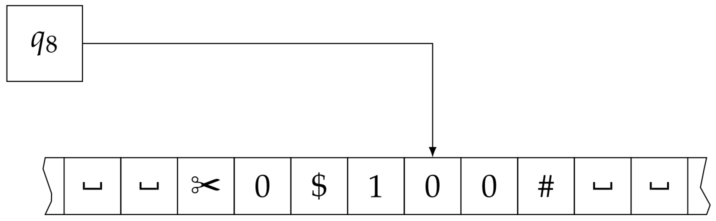


Figure 13.3: A DTM with a two-way infinite tape can easily simulate a DTM with a one-way infinite tape, like the one pictured in Figure 13.2, by writing a special symbol on the tape (in this case the symbol is ✂) that indicates where we should imagine the tape has been cut. When the tape head scans this symbol, the DTM adjusts its behavior accordingly.

Simulating an ordinary DTM having a two-way infinite tape with one having just a one-way infinite tape is slightly more challenging, but not difficult. Two natural ways to do it come to mind.

The first way is suggested by Figure 13.4. In essence, the one-way infinite, two-track tape of the DTM suggested by the figure may be viewed as the tape of the original DTM being simulated, but folded in half. The finite state control keeps track of the state of the DTM being simulated and which track of the tape stores the symbol being scanned. A special tape symbol, such as ➥, could be placed on the first square of the bottom track to assist in the simulation.

The second way to perform the simulation of a two-way infinite tape with a one-way infinite tape does not require two tracks, but will result in a simulation that is somewhat less efficient with respect to the number of steps required. A special symbol could be placed in the left-most square of the one-way infinite tape, and anytime this symbol is scanned the DTM can transition into a subroutine in which every other symbol on the tape is shifted one square to the right in order to "make room" for a new square to the left. This would presumably require that we also use a special symbol marking the right-most non-blank symbol on the tape, so

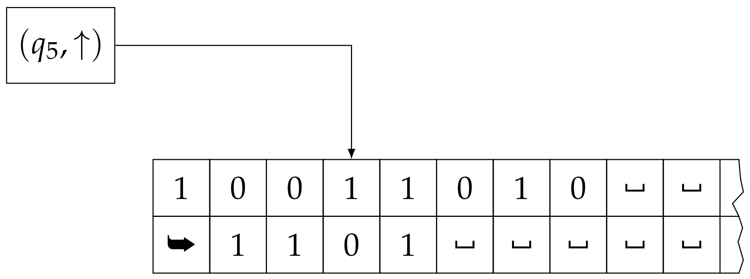


Figure 13.4: A DTM with a one-way infinite tape that simulates an ordinary DTM having a two-way infinite tape. The top track represents the portion of the two-way infinite tape that extends to the right and the bottom track represents the portion extending to the left.

that the shifting subroutine can be completed—for otherwise we might not know when every (non-blank) symbol on the tape had been shifted one square to the right.

Multi-tape DTMs

The last variant of the Turing machine model that we will consider will also be the most useful variant for our purposes. A *multi-tape DTM* works in a similar way to an ordinary, single-tape DTM, except that it has k tape heads that operate independently on k tapes, for some fixed positive integer k. For example, Figure 13.5 illustrates a multi-tap DTM with three tapes.

In general, a k-tape DTM is defined in a similar way to an ordinary DTM, except that the transition function has a slightly more complicated form. In particular, if the tape alphabets of a k-tape DTM are $\Gamma_1, \ldots, \Gamma_k$, then the transition function might take this form:

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma_1 \times \cdots \times \Gamma_k \rightarrow Q \times \Gamma_1 \times \cdots \times \Gamma_k \times \{\leftarrow, \downarrow, \rightarrow\}^k. \tag{13.11}$$

If we make the simplifying assumption that the same alphabet is used for each tape (which of course does not restrict the model, as we could always take this single tape alphabet to be the union $\Gamma = \Gamma_1 \cup \cdots \cup \Gamma_k$ of multiple alphabets), the transition function takes the form

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{\leftarrow, \downarrow, \rightarrow\}^k. \tag{13.12}$$

(In both of these cases it is evident that the tape heads are allowed to remain stationary. Naturally you could also consider a variant in which every one of the tape heads must move at each step, but we may as well allow for stationary tape heads

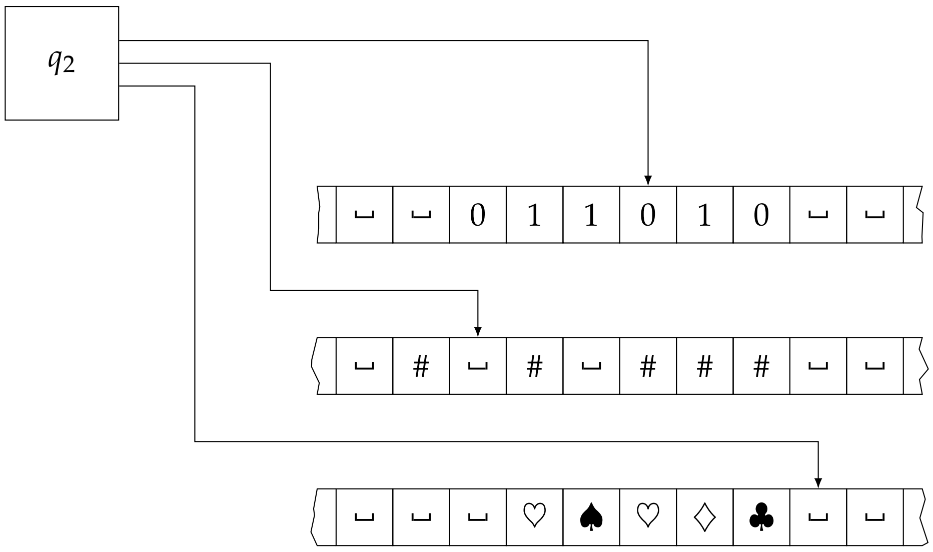


Figure 13.5: A DTM with three tapes.

when considering the multi-tape DTM model—it is meant to be flexible and general, so as to make it easier to perform complex computations.) The interpretation of the transition function taking the form (13.12) is as follows. If it holds that

$$\delta(p, a_1, \ldots, a_k) = (q, b_1, \ldots, b_k, h_1, \ldots, h_k), \tag{13.13}$$

then if the DTM is in the state p and is reading the symbols $a_1, \ldots, a_k$ on its k tapes, then (i) the new state becomes q, (ii) the symbols $b_1, \ldots, b_k$ are written onto the k tapes (overwriting the symbols $a_1, \ldots, a_k$), and (iii) the j-th tape head either moves or remains stationary depending on the value $h_j \in \{\leftarrow, \downarrow, \rightarrow\}$, for each $j = 1, \ldots, k$.

One way to simulate a multi-tape DTM with a single-tape DTM is to store the contents of the k tapes, as well as the positions of the k tape heads, on separate tracks of a single-tape DTM whose tape has multiple tracks. For example, the configuration of the multi-tape DTM pictured in Figure 13.5 could be represented by a single-tape DTM as suggested by Figure 13.6.

Naturally, a simulation of this sort will require many steps of the single-tape DTM to simulate a single step of the multi-tape DTM. Let us refer to the multi-tape DTM as K and the single-tape DTM as M. To simulate one step of K, the DTM M needs many steps: it must first scan through the tape in order to determine which symbols are being scanned by the k tape heads of K, and store these symbols within its finite state control. Once it knows these symbols, it can decide what action K is supposed to take, and then implement this action—which means again scanning through the tape in order to update the symbols stored on the tracks that represent

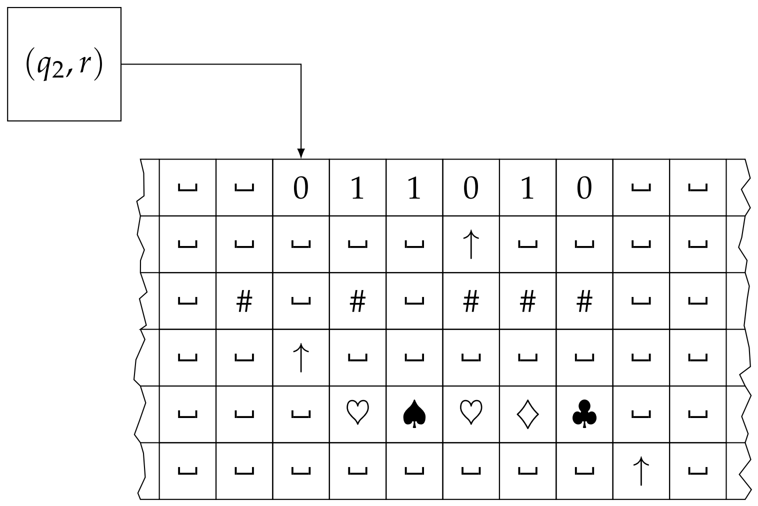


Figure 13.6: A single-tape DTM with a multi-track tape can simulate a multi-tape DTM. Here, the odd numbered tracks represent the contents of the tapes of the DTM illustrated in Figure 13.5, while the even numbered tracks store the locations of the tape heads of the multi-tape DTM. The finite state control of this single-tape DTM stores the state of the multi-tape DTM it simulates, but it would also need to store other information (represented by the component r in the picture) in order to carry out the simulation.

the tape contents of K and the positions of the tape heads, which may have to move. It would be complicated to write this all down carefully, and there are many specific ways in which this general idea could be carried out—but with enough time and motivation it would certainly be possible to give a formal definition for a single-tape DTM M that simulates a given multi-tape DTM K in this way.

13.2 Encoding various objects as strings

As we continue to discuss Turing machines and the languages they recognize or decide, it will be necessary for us to consider ways that interesting mathematical objects can be represented as strings. For example, we may wish to consider a DTM that takes as input a number, a graph, a DFA, a CFG, another DTM (maybe even a description of itself), or a list of such objects of multiple types.

In the remainder of this lecture we will begin this discussion with some simple examples, and the discussion will continue next lecture as we discuss the decidability of various languages connected with concepts we've seen previously in the course.

Encoding strings over one alphabet by strings over another

Suppose we have two alphabets: $\Sigma = \{0,1\}$ and $\Gamma = \{0,\ldots,n-1\}$, for n being some positive integer that could be very large. (Here, we are imagining that each integer between 0 and $n-1$ is a single symbol.)

There are a number of ways that we could encode symbols from the alphabet Γ by strings over the alphabet Σ, by selecting a different binary strings for each symbol in Γ. For example, if $n = 4$, we could choose to use the following simple encoding:

$$0 \to 00 \qquad 1 \to 01 \qquad 2 \to 10 \qquad 3 \to 11. \tag{13.14}$$

Or, we could choose this encoding:

$$0 \to 1 \qquad 1 \to 10 \qquad 2 \to 100 \qquad 3 \to 1000. \tag{13.15}$$

Now, if we have a *string* w over the alphabet Γ, we could aim to encode this string by simply concatenating together the strings over the alphabet Σ that encode the individual symbols of w. For example, with respect to the first example of an encoding above, the string $221302 \in \Gamma^*$ would be encoded as

$$221302 \to 101001110010, \tag{13.16}$$

and with respect to the second example of an encoding we would obtain

$$221302 \to 1001001010001100. \tag{13.17}$$

This will work so long as we've chosen the encodings of the individual symbols in Γ well.

When we say we've"chosen well," we mean that there won't be any ambiguity in the encodings of strings over the alphabet Γ obtained in this way. For instance, the encoding

$$0 \to 0 \qquad 1 \to 1 \qquad 2 \to 01 \qquad 3 \to 10 \tag{13.18}$$

is not chosen well: by concatenating these encodings together as suggested above, we would have (for example) that 01 and 2 have the same encoding 01. This is unacceptable, because we should be able to recover the original string from its encoding. If the encoding of each symbol of Γ has the same length, however, but different symbols are encoded by different strings, such as in the first example encoding above, we will never have this sort of ambiguity. More generally, if there is no encoding of a symbol in Γ that happens to be a prefix of an encoding of a different symbol in Γ, no ambiguity will arise.

There are good reasons to prefer encodings for which different symbols in Γ may have encodings of different lengths. For instance, the second example of an encoding suggested above can obviously be generalized for any alphabet $\Gamma = \{0, \ldots, n-1\}$, and it could be handy to use this encoding in a situation in which n is not known beforehand. Another example of a variable-length encoding is this one:

$$0 \rightarrow 0 \qquad 1 \rightarrow 10 \qquad 2 \rightarrow 110 \qquad 3 \rightarrow 111. \tag{13.19}$$

This encoding never causes any ambiguity of the sort described above, and it might result in shorter encodings in a situation where 0 is much more likely to appear than the other three symbols for some reason. (It is an example of a so-called *Huffman code*, which is useful for data compression.)

Encoding multiple strings into one

Next, suppose that we wish to represent *two or more* strings over some alphabet Σ by a single string over the same alphabet. Of course you could also consider encoding multiple strings over one alphabet by a single string over another, but it will be evident that this situation could be handled in a similar way.

This is easily done. One (of many) ways to encode multiple strings into a single string is to first introduce a new symbol that is not in our alphabet and use it as a marker to separate distinct strings. For example, if we have strings $x = \sigma_1 \cdots \sigma_n$ and $y = \tau_1 \cdots \tau_m$ over the alphabet $\Sigma = \{0, 1\}$, for instance, then perhaps we could introduce the symbol # to indicate a separation between the strings. The pair (x, y) could then be represented by the string

$$\sigma_1 \cdots \sigma_n \# \tau_1 \cdots \tau_m \tag{13.20}$$

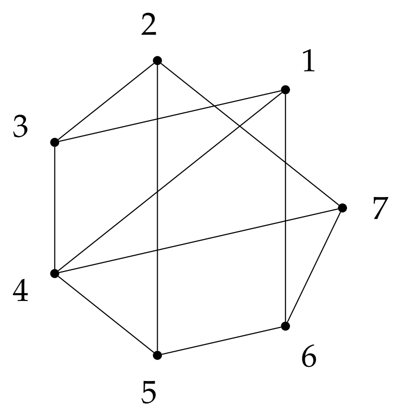
over the alphabet $\{0, 1, \#\}$. More generally, any tuple of strings $(x_1, \ldots, x_m)$ can be encoded as

$$x_1 \# x_2 \# \cdots \# x_m. \tag{13.21}$$

Once we have such a string over the alphabet $\{0, 1, \#\}$, we can then encode this string as a string over Σ as suggested in the previous subsection.

Numbers, vectors, and matrices

It probably goes without saying that natural numbers can be represented by binary strings using binary notation, as can arbitrary integers if we interpret the first bit of the encoding to be a sign bit. Rational numbers can be encoded as pairs of integers (representing the numerator and denominator), by first expressing the individual integers in binary, and then encoding the two strings into one as suggested above. One could also consider floating point representations, which are of course very



$$\begin{pmatrix} 0 & 0 & 1 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 1 & 0 & 1 & 0 \end{pmatrix}$$

Figure 13.7: A graph together with its adjacency matrix.

common in practice, but also have the disadvantage that they only represent rational numbers for which the denominator is a power of two.

Of course there are alternatives. For instance, we could use a *unary encoding*, where the nonnegative integer n is represented as 1^n (i.e., a string of length n consisting only of 1s), or you could invent new ways of representing numbers. Representing a nonnegative integer in unary notation may seem inefficient, but there are situations in which we do not care at all about efficiency, and the simplicity of this notation might be appealing for some reason.

With a method for encoding numbers in mind, one can then represent vectors by encoding the entries as strings, then encoding these multiple strings into a single string; and matrices can be encoded by combining the encodings of multiple vectors together. Alternatively, you might decide to encode a matrix by (for instance) introducing one special symbol that separate entries within a row and another special symbol that separates the columns.

Encoding graphs as binary strings

The last example of an encoding for today is for graphs. If we have a graph G with n nodes, one simple way to represent G as a binary string is by specifying the adjacency matrix. An example of a graph and its adjacency matrix appear in Figure 13.7. Because adjacency matrices contain only 0 and 1 entries, we could simply concatenate the entries of this matrix without doing anything further. For instance, the graph in Figure 13.7 would then be encoded by the binary string

$$0011010001010111010001010101010101010001010101010. \tag{13.22}$$

Alternatively, we could just concatenate the bits in the upper-triangular part of the adjacency matrix, as the diagonal entries are all 0 and the matrix is symmetric (for an undirected graph). For the same graph as above, this would yield the encoding

$$011010101011000101101. \tag{13.23}$$

Another alternative is not to use the adjacency matrix at all, but instead to encode a graph by describing a list of its edges, in some arbitrary order. Each edge could be represented as a pairs of integers, with the integers representing names for the vertices. For instance, the graph from Figure 13.7 has the following set of edges:

$$\begin{multlined}\{\{1,3\},\{1,4\},\{1,6\},\{2,3\},\{2,5\},\{2,7\},\\ \{3,4\},\{4,5\},\{4,7\},\{5,6\},\{6,7\}\}.\end{multlined} \tag{13.24}$$

Such a list of edges could be encoded as a string in a variety of straightforward ways.

Lecture 14

Decidable languages

In the previous lecture we discussed some examples of encoding schemes, through which various objects can be represented by strings over a given alphabet. We will begin this lecture by considering a few more encoding schemes, particularly for the models of computation we have discussed thus far in the course. We will then discuss several interesting examples of decidable languages.

14.1 Encoding for different models of computation

In this section we will discuss ways of encoding DFAs, NFAs, CFGs, and DTMs as strings. As it turns out, the specifics of these encoding schemes won't actually matter all that much to us, so we'll focus more on the general ideas than on the specific details.

Encoding DFAs

For the sake of simplicity, let us fix the alphabet $\Sigma = \{0,1\}$, and consider one way that an arbitrary DFA $D = (Q, \Gamma, \delta, q_0, F)$ can be encoded by a string over the alphabet Σ. Let us first make some observations and clarify our expectations about what we actually mean by encoding D in this way.

1. The result of our encoding should be a binary string, which we will denote by $\langle D \rangle$. Given the string $\langle D \rangle$, it should be possible to recover a description of each component of D. Stated in more intuitive terms, you can imagine that this means that if you were given the string $\langle D \rangle$ and had enough paper, pencils, and time to devote to the task, you could draw a state diagram for D without using any creativity during the process.
2. The state set Q of D is a finite and nonempty set, but beyond this there are no restrictions on it. Of course there is no way to devise a unique binary string en-

coding for every possible state name that someone could choose—for instance, we could imagine a different state name corresponding to every snowflake that could ever exist in nature. For this reason, we will assume that the state set of Q necessarily takes the form $Q = \{q_0, \ldots, q_{n-1}\}$ for some positive integer n. Because the specific names we choose for states are irrelevant to the way a DFA functions, this assumption does not create any difficulties.

3. It is important to notice that the alphabet Γ of the DFA D does not have to be the same as Σ; the alphabet Γ could be different from Σ, and moreover we would like to use a single encoding scheme that can handle any choice of Γ. On the other hand, the same comments we just made regarding the names of the states in Q also hold for the names of the symbols in Γ, and for this reason we will assume that $\Gamma = \{0, \ldots, m-1\}$ for some positive integer m (and where we are viewing each integer in the range $0, \ldots, m-1$ as a single symbol).

   With those points in mind, we could first encode D over the alphabet

$$\Delta = \{0, 1, \#, /\} \tag{14.1}$$

along the lines suggested by this hypothetical example:

$$\underbrace{1011}_{m}\,\#\,\underbrace{01100\cdots011}_{F}\,\#\,\underbrace{0/0/110\,\#\,0/1/1001\,\#\,\cdots\,\#\,11001/1010/011}_{\delta} \tag{14.2}$$

Here, m is the number of alphabet symbols, encoded in binary (so we appear to have $|\Gamma| = 11$ in this example). The string labeled F indicates which states are accept states and which are reject states, and its length indicates the total number of states (so, it appears that q_1, q_2, q_{n-2}, and q_{n-1}, for whatever value n happens to take, are among the accept states in the example). If we agree that q_0 is always the start state, there is no need to indicate this in the encoding. Finally, we assume that δ is encoded by a string in some fashion. The suggestion in this example is that a string of the form $a/b/c$ (for a, b, and c being natural numbers represented in binary notation) indicates that $\delta(q_a, b) = q_c$; and we concatenate together a list of such strings separated by # symbols in order to fully describe δ.

Once we have an encoding of the form suggested above, we could then convert it to a binary string using the method we discussed last time for encoding strings over one alphabet (in this case Δ) by strings in another (in this case the binary alphabet).

Of course this is just one way to encode a DFA as a binary string. There is nothing particularly special about this encoding scheme—it was invented for the sake of the lecture, and one could easily come up with other schemes that are just as effective.

It is worth noting that if we have chosen an encoding scheme for DFAs, such as the one just suggested, then every DFA (subject to the assumptions we placed on the state set and alphabet of that DFA) will have an encoding. Nothing says, however, that the encoding is unique—there could be multiple strings that encode the same DFA—and it may be that some strings will not correspond to the encoding of any DFA at all. Neither of these things will cause us any difficulties.

Encoding NFAs

If you use the encoding scheme suggested above for DFAs as a model, it should not be hard to come up with an encoding scheme for NFAs. So long as you have a way to represent the transition function of an NFA as a string, as opposed to the transition function of a DFA, then the rest could go unchanged. For the particular scheme that was suggested above, it would be pretty straightforward to extend it so that nondeterministic transition functions can be handled, and I will leave it to your imagination exactly how it might be done.

Encoding CFGs and DTMs

In the interest of time, we will not discuss any specific schemes for encoding CFGs and DTMs, but it is appropriate for us to observe that we could do this in a variety of different ways, perhaps using a similar style of encoding to the one suggested above for DFAs (but of course allowing for descriptions of the different components of CFGs and DTMs).

In the case of CFGs, an encoding scheme allows one to represent each CFG G by a binary string $\langle G \rangle$, from which one can uniquely identify G, obtain a description of the different components of G, and so on. Similar to our assumptions concerning encodings of DFAs, an assumption on the possible variable names and symbol names is required. For this reason, we might assume that the variable set of G takes the form $\{X_0, \ldots, X_{n-1}\}$ for some positive integer n, and the alphabet of G takes the form $\{0, \ldots, m-1\}$ for a positive integer m.

The situation is similar for DTMs, whereby any given DTM M can be encoded as a binary string $\langle M \rangle$ that allows one to recover M, describe its parts, an so on. Once again, similar assumptions should be made on the state names and alphabet symbol names (in this case including both the input alphabet and tape alphabet).

A general remark on alphabets used for encodings

In the discussion above, we described encodings over the binary alphabet $\Sigma = \{0, 1\}$ for simplicity—but any other alphabet could be used instead. You could even use an alphabet with a single symbol, such as $\Delta = \{0\}$, to encode objects like DFAs, NFAs, or anything else you might care to encode. One simple way to do this is to first come up with an encoding that uses the binary alphabet, and then translate this encoding into one that uses the alphabet Δ by placing strings in lexicographic correspondence:

$$\begin{aligned} \varepsilon &\rightarrow \varepsilon \\ 0 &\rightarrow 0 \\ 1 &\rightarrow 00 \\ 00 &\rightarrow 000 \end{aligned} \tag{14.3}$$

and so on.

14.2 Simple examples of decidable languages

Now we will see some examples of decidable languages, beginning with some simple examples. One goal of this discussion is to develop the connections between DTMs and the intuitive notion of an algorithm, along the lines that the Church–Turing thesis suggests, and also to introduce a general style through which DTMs can be expressed at a high level.

Comparing natural numbers

As a first example, let us suppose that we have agreed upon a scheme for encoding any pair of natural numbers (n, m) into a string that we will denote[1] by $\langle n, m \rangle$. Consider the following language:

$$A = \{ \langle n, m \rangle : n \text{ and } m \text{ are natural numbers that satisfy } n > m \}. \tag{14.4}$$

We may then ask whether or not this language is decidable.

Now, it is certainly the case that the language A depends on the particular encoding scheme we selected. It may therefore seem as though the answer to the question of whether or not A is decidable also depends on the encoding scheme that is selected—but the reality is that the problem is decidable for *any reasonable encoding scheme whatsoever*.

[1] In general, whenever we wish to consider the encoding of some mathematical object X as a string, with respect to some encoding scheme that has been selected beforehand, we will denote this string by $\langle X \rangle$; if we have two objects X and Y, the string encoding both objects together is denoted $\langle X, Y \rangle$; and so on for any finite number of objects.

If the specific encoding scheme we chose when defining A was particularly simple, we could probably write down a state diagram of a DTM that decides A—but while this might be feasible, it would also be tedious and not very enlightening.

An alternative is to design a two-tape DTM that decides A. Perhaps this DTM would process the encoding $\langle n, m\rangle$ in such a way that we are left with the binary representation of n on one tape and the binary representation of m on the other. The precise manner in which this is done would necessarily depend on the encoding scheme, but it is safe to say that any encoding of a pair of natural numbers that doesn't allow a DTM to extract binary representations of the two numbers is not a reasonable one. Once this is done, the lengths of these binary representations could be compared. If one string is shorter than the other, then it becomes clear which of n and m is greater; and if the binary representations are the same length, then by comparing bits from most significant to least significant we can decide if $n > m$.

This is just one way to do it, and you can imagine that even if we agreed on this general strategy, there could be variations in how a two-tape DTM implementing this strategy might behave. In any case, once we know that a two-tape DTM decides the language A, then based on the discussion from the previous lecture we conclude that A is decidable by an ordinary DTM, and is therefore decidable.

Other languages based on numbers and arithmetic

One can imagine many other languages based on the natural numbers, arithmetic, and so on. Here are just a few examples:

$$\begin{aligned}
C &= \bigl\{\langle a,b,c\rangle \,:\, a, b, \text{ and } c \text{ are natural numbers satisfying } a+b=c\bigr\},\\
D &= \bigl\{\langle a,b,c\rangle \,:\, a, b, \text{ and } c \text{ are natural numbers satisfying } ab=c\bigr\},\\
E &= \bigl\{\langle n\rangle \,:\, n \text{ is a prime number}\bigr\}.
\end{aligned} \tag{14.5}$$

For the first two languages, we assume that $\langle a,b,c\rangle$ is an encoding of three natural numbers a, b, and c, and for the third language $\langle n\rangle$ is the encoding of a natural number n. These are all decidable languages (once again, for any reasonable encoding scheme). Obtaining an actual description of a DTM that decides the languages would be increasingly difficult—but if you think about the fact that these languages could easily be decided by a computer program in your favorite programming language, which can be broken down into a sequence of primitive computation steps, you might convince yourself that it would be possible to come up DTMs that decide these languages.

Generally speaking, when we want to describe a DTM that performs a particular task, we describe it in high-level terms—usually not bothering to say very much (or perhaps nothing at all) about tapes or tape head movements. For example, we could describe a DTM M deciding the language E in following style:

```
On input $\langle n \rangle$, where $n$ is a natural number:
1.  If $n \leq 1$ then *reject*. (We don't consider 0 and 1 to be prime.)
2.  Set $t \leftarrow 2$.
3.  If $t = n$, then *accept*.
4.  If $t$ divides $n$ evenly, then *reject*.
5.  Set $t \leftarrow t + 1$ and goto step 3.
```

Note that when we write "On input $\langle n \rangle$, where n is a natural number" we are implicitly defining M so that it immediately rejects if the input does not have this form. A similar interpretation should be made for DTMs in general in these notes.

Of course, if you wanted to turn this high-level Turing machine description into a formal specification of a DTM, you would have a lot of work ahead of you—but the description nevertheless explains key algorithmic ideas for deciding E. The most complicated step is step 4, and of course one could explain it in more detail. However, it is probably safe to assume that everyone reading these notes knows how to compute the remainder of one natural number divided by another, assuming the numbers are expressed in binary notation, and one could implement that method on a Turing machine.

Graph reachability

Here is one final example of a simple decidable language, which this time concerns graph reachability.

$$\text{REACHABLE} = \left\{ \langle G, u, v \rangle \; : \; \begin{array}{l} \text{there exists a path from vertex } u \text{ to} \\ \text{vertex } v \text{ in the undirected graph } G \end{array} \right\}. \tag{14.6}$$

Once again, we are not specifying the specific encoding method that is used to represent a graph G and two vertices u, v of G as a string $\langle G, u, v \rangle$—but so long as the encoding is reasonable it does not change whether or not the language is decidable. Here is a high-level description of a DTM that decides this language:

```
On input $\langle G, u, v \rangle$, where $G$ is an undirected graph and $u$ and $v$ are vertices of $G$:
1.  Set $S \leftarrow \{u\}$.
2.  Set $F \leftarrow 1$. ($F$ is a flag indicating we are finished adding vertices to $S$.)
3.  For each vertex $w$ of $G$ do the following:
4.      If $w$ is not contained in $S$ and $w$ is adjacent to a vertex in $S$, then
        set $S \leftarrow S \cup \{w\}$ and $F \leftarrow 0$.
5.  If $F = 0$ then goto 2.
6.  *Accept* if $v \in S$, otherwise *reject*.
```

14.3 Languages concerning DFAs, NFAs, and CFGs

Now let us turn our attention toward languages that concern the models of computation we have studied previously in the course. The first language we will consider is this one:

$$\mathrm{A}_{\mathrm{DFA}} = \big\{ \langle D, w \rangle \,:\, D \text{ is a DFA and } w \in \mathrm{L}(D) \big\}. \tag{14.7}$$

Along the same lines as the previous examples in this lecture, we understand $\langle D, w \rangle$ to be the encoding of a DFA D together with a string w. The alphabet of D and w is not assumed to be fixed ahead of time, but as discussed earlier in the lecture we will assume that for each particular choice of D and w, the alphabet symbols appearing are drawn from the set $\{0, \ldots, n-1\}$ for some positive integer n.

Here is a high-level description of a DTM M that decides $\mathrm{A}_{\mathrm{DFA}}$:

On input $\langle D, w \rangle$, where D is a DFA and w is a string:

1. If w contains symbols not in the alphabet of D, then *reject*.
2. Simulate the computation of D on input w. If this simulation ends on an accept state of D, then *accept*, and otherwise *reject*.

By specifying that this DTM should "simulate the computation of D on input w," it might seem like we are cheating somehow—but it really is quite simple and direct to simulate a DFA on a given input string, so stating that this process should be performed by a computer is reasonable. It follows from the fact that the DTM M decides $\mathrm{A}_{\mathrm{DFA}}$ that this language is decidable.

We may also consider a variant of this language for NFAs in place of DFAs:

$$\mathrm{A}_{\mathrm{NFA}} = \big\{ \langle N, w \rangle \,:\, N \text{ is an NFA and } w \in \mathrm{L}(N) \big\}. \tag{14.8}$$

This language is also decidable. However, it would not be reasonable to simply define a DTM that "simulates the computation of N on input w," because it really isn't clear how one simulates a nondeterministic computation with a DTM. Here is an alternative description of a DTM that decides $\mathrm{A}_{\mathrm{NFA}}$:

On input $\langle N, w \rangle$, where N is an NFA and w is a string:

1. Convert N into an equivalent DFA D using the subset construction described in Lecture 3.
2. If w contains symbols not in the alphabet of D, then *reject*.
3. Simulate the computation of D on input w. If this simulation ends on an accept state of D, then *accept*, and otherwise *reject*.

One could also define a similar language

$$A_{\text{REX}} = \{\langle R, w\rangle : R \text{ is a regular expression and } w \in L(R)\}, \tag{14.9}$$

and conclude that it is decidable through a similar argument (referring to a construction that converts a regular expression into a DFA).

Here is a different example of a language, which we will also prove is decidable:

$$E_{\text{DFA}} = \{\langle D\rangle : D \text{ is a DFA and } L(D) = \varnothing\}. \tag{14.10}$$

In this case, one cannot decide this language simply by "simulating the DFA D," because we don't necessarily know what string to simulate it on. Here is a Turing machine that decides this language based on a different approach, which essentially is to test to see if there is an accept state that is reachable from the start state. The approach is similar to the graph reachability example from before.

```
On input $\langle D\rangle$, where $D$ is a DFA:
 1.  Set $S \leftarrow \{q_0\}$, for $q_0$ the start state of $D$.
 2.  Set $F \leftarrow 1$.
 3.  For each state $q$ of $D$ do the following:
 4.      If $q$ is not contained in $S$ and there exists a transition from any
         state in $S$ to $q$, then set $S \leftarrow S \cup \{q\}$ and $F \leftarrow 0$.
 5.  If $F = 0$ then goto 2.
 6.  *Reject* if there exists an accept state in $S$, otherwise *accept*.
```

By combining this DTM with ideas from the previous examples, one can prove that analogously defined languages E_{NFA} and E_{REX} are decidable.

One more example of a decidable language concerning DFAs is this language:

$$EQ_{\text{DFA}} = \{\langle D, E\rangle : D \text{ and } E \text{ are DFAs and } L(D) = L(E)\}. \tag{14.11}$$

Here is a high-level description of a DFA that decides it:

```
On input $\langle D, E\rangle$, where $D$ and $E$ are DFAs:
 1.  Construct a DFA $M$ for which it holds that $L(M) = L(D) \triangle L(E)$.
 2.  If $\langle M\rangle \in E_{\text{DFA}}$, then *accept*, otherwise *reject*.
```

One natural way to perform the construction in step 1 of the previous DTM description is to use the so-called Cartesian product construction, which we discussed in lecture earlier in the course.

Next let us turn to a couple of examples of decidable languages concerning context-free grammars. Following along the same lines as the examples discussed above, we may consider these languages:

$$\begin{aligned} \mathrm{A}_{\mathrm{CFG}} &= \bigl\{ \langle G, w \rangle \,:\, G \text{ is a CFG and } w \in \mathrm{L}(G) \bigr\}, \\ \mathrm{E}_{\mathrm{CFG}} &= \bigl\{ \langle G \rangle \,:\, G \text{ is a CFG and } \mathrm{L}(G) = \varnothing \bigr\}. \end{aligned} \tag{14.12}$$

Here is a DTM that decides the first language:

```
On input ⟨G, w⟩, where G is a CFG and w is a string:
 1.  Convert G into an equivalent CFG H in Chomsky normal form.
 2.  If $w = \varepsilon$ then *accept* if $S \rightarrow \varepsilon$ is a rule in H and *reject* otherwise.
 3.  Search over all possible derivations by H having $2|w| - 1$ steps
     (of which there are finitely many). *Accept* if a valid derivation of
     w is found, and *reject* otherwise.
```

It is worth noting that this is a ridiculously inefficient way to decide the language $\mathrm{A}_{\mathrm{CFG}}$, but right now we don't care! We're just trying to prove it is decidable. There are, in fact, much more efficient ways to decide this language, but we will not discuss them now.

Finally, the language $\mathrm{E}_{\mathrm{CFG}}$ can be decided using a variation on the reachability technique. In essence, we keep track of a set containing variables that generate at least one string, and then test to see if the start variable is contained in this set.

```
On input ⟨G⟩, where G is a CFG:
 1.  Set $T \leftarrow \Sigma$, for $\Sigma$ the alphabet of G.
 2.  Set $F \leftarrow 1$.
 3.  For each rule $X \rightarrow w$ of G do the following:
 4.      If X is not contained in T, and every variable and every symbol of w
         is contained in T, then set $T \leftarrow T \cup \{X\}$ and $F \leftarrow 0$.
 5.  If $F = 0$ then goto 2.
 6.  *Reject* if the start variable of G is contained in T, otherwise *accept*.
```

Now, you may be wondering about this next language, as it is analogous to one concerning DFAs from above:

$$\mathrm{EQ}_{\mathrm{CFG}} = \bigl\{ \langle G, H \rangle \,:\, G \text{ and } H \text{ are CFGs and } \mathrm{L}(G) = \mathrm{L}(H) \bigr\}. \tag{14.13}$$

As it turns out, this language is not decidable. (We won't go through the proof, because it would take us a bit too far off the path of the rest of the course, but it would not be too difficult to prove sometime after the lecture following this one.)

Some other examples of undecidable languages concerning context-free grammars are as follows:

$$\begin{aligned}
&\bigl\{\langle G\rangle \,:\, G \text{ is a CFG that generates all strings over its alphabet}\bigr\},\\
&\bigl\{\langle G\rangle \,:\, G \text{ is an ambiguous CFG}\bigr\},\\
&\bigl\{\langle G\rangle \,:\, G \text{ is a CFG and } \mathrm{L}(G) \text{ is inherently ambiguous}\bigr\}.
\end{aligned} \tag{14.14}$$

Lecture 15

Undecidable languages

In the previous lecture we discussed several examples of decidable languages relating to finite automata and context-free grammars. In this lecture we will discuss analogous languages relating to Turing machines, and we will find that these languages are *undecidable*.

15.1 Simulating one Turing machine with another

Let us begin by defining a language $\mathrm{A}_{\mathrm{DTM}}$ that is analogous to ones we considered in the previous lecture for DFAs, CFGs, and so on.

$$\mathrm{A}_{\mathrm{DTM}} = \big\{ \langle M, w \rangle \,:\, M \text{ is a DTM and } w \in \mathrm{L}(M) \big\}. \tag{15.1}$$

All of the same assumptions on encoding schemes that we have made previously are in place here. For instance, we assume $\langle M, w \rangle$ is a string over a fixed alphabet (such as $\{0,1\}$), the state set of M has the form $\{q_0, \ldots, q_{n-1}\}$ for some positive integer n, the alphabet of M takes the form $\{0, \ldots, m-1\}$ for some positive integer m, and so on. This language is not decidable, as we will prove shortly, but it is Turing recognizable.

The idea is pretty straightforward, and similar to what we discussed for the DFA variant of this language: given a DTM M and a string w as input, we can *simulate* M on input w and see what happens. In particular we can define a DTM U having the following high-level description:

```
On input ⟨M, w⟩, where M is a DTM and w is a string:
1.  If w is not a string over the alphabet of M, then reject.
2.  Simulate M on input w. If at any point in the simulation M accepts w,
        then accept; and if M rejects w, then reject.
```

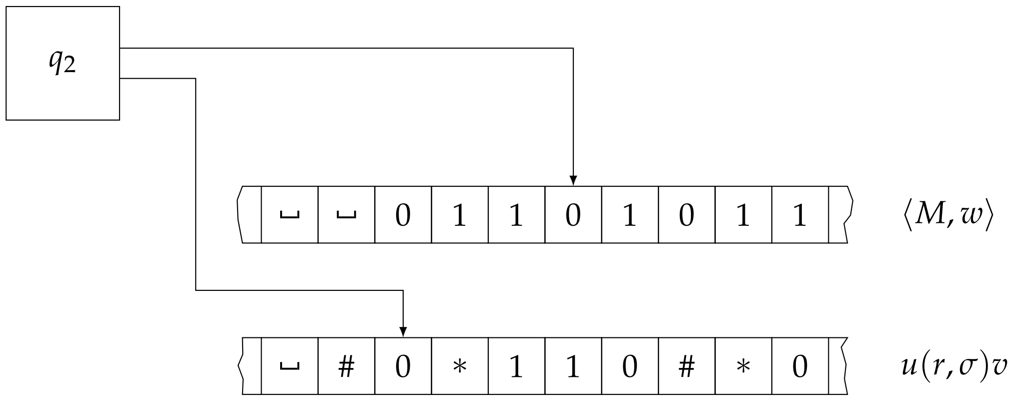


Figure 15.1: A two-tape DTM for A_{DTM} could leave the input $\langle M, w\rangle$ on the first tape and store a configuration of M on the second tape. Through multiple steps of this DTM, it could transform the configuration into the next configuration after one step, and by repeating this process simulate M on input w.

We've chosen the name U for this DTM because it is a so-called *universal Turing machine*—meaning that it is a single Turing machine capable of simulating *every* DTM (even itself).

If you are interested, you can find detailed descriptions of universal Turing machines in some books. It is a critical step in the study of computability theory to recognize that such DTMs exist, so it is indeed worthy of careful scrutiny. On the other hand, it is fairly intuitive—if you have a description of a DTM M and an input string w, you can simulate M on input w by keeping track of the configuration of M and repeatedly update it one step at a time. Figure 15.1 suggests one way that this might be done using a two-tape DTM, which could then be converted into a one-tape DTM matching the description of U above.

It is important to note that the DTM U suggested above keeps on simulating M on input w until it has a reason to stop. If it so happens that M either accepts or rejects w, then U will accept or reject $\langle M, w\rangle$; but if M runs forever on input w, then U will also run forever on input $\langle M, w\rangle$. (Of course, if U receives an input that does not encode a DTM and a string, or if the string is not over the alphabet of the DTM, then it will reject this input.)

For the DTM U as defined above, it holds that $\text{L}(U) = \text{A}_{\text{DTM}}$, and therefore A_{DTM} is Turing recognizable. On the other hand, U does not decide A_{DTM} because it might run forever on some inputs. Naturally, this does not imply that A_{DTM} is undecidable—we have not yet argued that there cannot exist a different DTM that decides it, but we will do that shortly.

Before moving on, let us observe that if we included a limitation on the number of steps a DTM is allowed to run, as part of the input, we obtain a variant of the language defined above that is decidable. To be more precise, the following language is decidable:

$$\mathrm{S}_{\mathrm{DTM}} = \left\{ \langle M, w, t \rangle \, : \, \begin{array}{l} M \text{ is a DTM, } w \text{ is a string, } t \in \mathbb{N}, \\ \text{and } M \text{ accepts } w \text{ within } t \text{ steps} \end{array} \right\}. \tag{15.2}$$

This language could be decided by a DTM similar to U defined above, but where it cuts the simulation off after t steps if M has not accepted w.

15.2 A non-Turing-recognizable language

We will now define a language and prove it is undecidable. (In fact, this language will not even be Turing recognizable.) Here is the language:

$$\mathrm{DIAG} = \bigl\{ \langle M \rangle \, : \, M \text{ is a DTM and } \langle M \rangle \notin \mathrm{L}(M) \bigr\}. \tag{15.3}$$

Along the same lines as the other languages we have considered in this lecture and the previous one, the language DIAG is defined with respect to some encoding scheme for DTMs, so the language itself is over a fixed alphabet (such as the binary alphabet). The language DIAG contains all strings that, with respect to the chosen encoding scheme, encode a DTM that does not accept this encoding of itself.

If it so happens that a string $\langle M \rangle$ encodes a DTM whose input alphabet does not include every symbol in this encoding (such as in the case that M has input alphabet $\{0\}$ but we have chosen an encoding scheme over the alphabet $\{0,1\}$), then it is indeed the case that $\langle M \rangle \notin \mathrm{L}(M)$.

Theorem 15.1. *The language* DIAG *is not Turing recognizable.*

Proof. Assume toward contradiction that DIAG is Turing recognizable. There must therefore exist a DTM T such that $\mathrm{L}(T) = \mathrm{DIAG}$.

Now, consider any encoding $\langle T \rangle$ of T. By the definition of the language DIAG one has

$$\langle T \rangle \in \mathrm{DIAG} \;\Leftrightarrow\; \langle T \rangle \notin \mathrm{L}(T). \tag{15.4}$$

On the other hand, because T recognizes DIAG it holds that

$$\langle T \rangle \in \mathrm{DIAG} \;\Leftrightarrow\; \langle T \rangle \in \mathrm{L}(T). \tag{15.5}$$

Consequently,

$$\langle T \rangle \notin \mathrm{L}(T) \;\Leftrightarrow\; \langle T \rangle \in \mathrm{L}(T), \tag{15.6}$$

which is a contradiction. We conclude that DIAG is not Turing recognizable. □

Remark 15.2. Note that this proof is very similar to the proof that $\mathcal{P}(\mathbb{N})$ is not countable from the very first lecture of the course.

15.3 Some undecidable languages

Now that we know DIAG is not Turing recognizable, we can go back and prove that $\mathrm{A}_{\mathrm{DTM}}$ is not decidable.

Proposition 15.3. *The language* A_{DTM} *is undecidable.*

Proof. Assume toward contradiction that $\mathrm{A}_{\mathrm{DTM}}$ is decidable. There must therefore exist a DTM T that decides $\mathrm{A}_{\mathrm{DTM}}$. Define a new DTM K as follows.

On input $\langle M \rangle$, where M is a DTM:

    Run T on input $\langle M, \langle M \rangle \rangle$. If T accepts, then *reject*, otherwise *accept*.

For a given DTM M, we may now ask ourselves what K does on the input $\langle M \rangle$.

If it is the case that $\langle M \rangle \in \mathrm{DIAG}$, then by the definition of DIAG it holds that $\langle M \rangle \notin \mathrm{L}(M)$, and therefore $\langle M, \langle M \rangle \rangle \notin \mathrm{A}_{\mathrm{DTM}}$ (because M does not accept $\langle M \rangle$). This implies that T rejects the input $\langle M, \langle M \rangle \rangle$, and so K must accept the input $\langle M \rangle$. If, on the other hand, it is the case that $\langle M \rangle \notin \mathrm{DIAG}$, then $\langle M \rangle \in \mathrm{L}(M)$, and therefore $\langle M, \langle M \rangle \rangle \in \mathrm{A}_{\mathrm{DTM}}$. This implies that T accepts the input $\langle M, \langle M \rangle \rangle$, and so K must reject the input $\langle M \rangle$.

One final possibility is that that K is run on an input string that does not encode a DTM at all, and in this case it rejects.

Considering these possibilities, we find that K decides DIAG. This, however, is in contradiction with the fact that DIAG is not Turing recognizable (and is therefore undecidable). Having obtained a contradiction, we conclude that $\mathrm{A}_{\mathrm{DTM}}$ is undecidable, as required. □

Here is another example, which is a famous relative of $\mathrm{A}_{\mathrm{DTM}}$.

$$\mathrm{HALT} = \big\{ \langle M, w \rangle \;:\; M \text{ is a DTM that halts on input } w \big\}. \tag{15.7}$$

To say that M *halts* on input w means that it stops, either by accepting or rejecting. Let us agree that the statement "M halts on input w" is false in case w contains symbols not in the input alphabet of M—purely as a matter of terminology.

It is easy to prove that HALT is Turing recognizable—we just run a modified version of our universal Turing machine U on input $\langle M, w \rangle$, except that we *accept* in case the simulation results in either accept or reject—and when it is the case that M does not halt on input w this modified version of U will run forever on input $\langle M, w \rangle$.

Proposition 15.4. *The language* HALT *is undecidable.*

Proof. Assume toward contradiction that HALT is decidable, so that there exists a DTM T that decides it. Define a new DTM K as follows:

On input $\langle M, w\rangle$, where M is a DTM and w is a string:

1. Run T on input $\langle M, w\rangle$ and *reject* if T rejects.
2. Simulate M on input w; *accept* if M accepts and *reject* if M rejects.

The DTM K decides A_{DTM}, as a case analysis reveals:

- If it is the case that M accepts w, then T will accept $\langle M, w\rangle$ (because M halts on w), and the simulation of M on input w will result in acceptance.
- If it is the case that M rejects w, then T will accept $\langle M, w\rangle$ (because M halts on w), and the simulation of M on input w will result in rejection.
- If it is the case that M runs forever on w, then T will reject $\langle M, w\rangle$, and therefore K rejects without running the simulation of M on input w.

This, however, is in contradiction with the fact that A_{DTM} is undecidable. Having obtained a contradiction, we conclude that HALT is undecidable. □

15.4 A couple of basic Turing machine tricks

There are many specific techniques through which certain languages can be proved to be either decidable or not, or Turing recognizable or not. Here we will illustrate a couple of these techniques, mainly through examples.

Limiting infinite search spaces

Sometimes we would like a Turing machine to effectively search over an infinitely large search space, but it is not immediately clear how to do this in a straightforward way. It can sometimes be helpful to use a single positive integer to serve as a bound on the search space, and then allow this positive integer to grow. (Something similar is useful for proving certain sets to be countable.) The proofs of the proposition and theorem that follow illustrate this method.

Proposition 15.5. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be Turing-recognizable languages. The language $A \cup B$ is Turing recognizable.*

Proof. Because A and B are Turing-recognizable languages, there must exist DTMs M_A and M_B such that $A = \text{L}(M_A)$ and $B = \text{L}(M_B)$. Define a new DTM M as follows:

On input w:

```
1. Set $t \leftarrow 1$.
2. Run $M_A$ for $t$ steps on input $w$. If $M_A$ has accepted within $t$ steps, then *accept*.
3. Run $M_B$ for $t$ steps on input $w$. If $M_B$ has accepted within $t$ steps, then *accept*.
4. Set $t \leftarrow t + 1$ and goto 2.
```

It is evident that $\mathrm{L}(M) = A \cup B$, and therefore $A \cup B$ is Turing recognizable. □

Remark 15.6. Of course we cannot replace the DTM M in the previous proof by one that first runs M_A on w (without any bound on its running time) and then runs M_B on w, because it could be that M_A runs forever on w but M_B accepts this string. It could also be the other way around, so that M_B cannot be run first without limiting its running time. Using t to bound the number of steps of both simulations is an effective way to handle this situation.

Theorem 15.7. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language such that both A and $\overline{A}$ are Turing recognizable. The language A is decidable.*

Proof. Because A and $\overline{A}$ are Turing-recognizable languages, there must exist DTMs M_0 and M_1 such that $A = \mathrm{L}(M_0)$ and $\overline{A} = \mathrm{L}(M_1)$. Define a new DTM M as follows:

On input w:

```
1. Set $t \leftarrow 1$.
2. Run $M_0$ for $t$ steps on input $w$. If $M_0$ has accepted within $t$ steps, then *accept*.
3. Run $M_1$ for $t$ steps on input $w$. If $M_1$ has accepted within $t$ steps, then *reject*.
4. Set $t \leftarrow t + 1$ and goto 2.
```

Now let us consider the behavior of the DTM M on a given input string w. If it is the case that $w \in A$, then M_0 eventually accepts w, while M_1 does not. (It could be that M_1 either rejects or runs forever, but it cannot accept w.) It is therefore the case that M accepts w. On the other hand, if $w \notin A$, then M_1 eventually accepts w while M_0 does not, and therefore M rejects w. Consequently, M decides A, so A is decidable. □

Hard-coding input strings

Suppose that we have a DTM M along with a string x over the input alphabet of M. Define a new DTM M_x as follows:

On input w:

```
    Ignore the input string $w$ and run $M$ on input $x$.
```

This may seem like a curious thing to do—the DTM M_x runs the same way regardless of its input string, which is to run M on the string x that has been "hard-coded" directly into its description. We will see, however, that it is sometimes very useful to consider a DTM defined like this.

Let us also note that if you had an encoding $\langle M, x\rangle$ of a DTM M along with a string x over the input alphabet of M, it would be possible to compute an encoding $\langle M_x\rangle$ of the DTM M_x without any difficulties—the DTM M_x would have some initial phase of its computation that erases its input and writes x in its place, and this simple computation phase could be composed with the DTM M.

Here is an example that illustrates the usefulness of this construction. Define a language

$$\mathrm{E}_{\mathrm{DTM}} = \bigl\{\langle M\rangle \,:\, M \text{ is a DTM with } \mathrm{L}(M) = \varnothing\bigr\}. \tag{15.8}$$

Proposition 15.8. *The language* E_{DTM} *is not decidable.*

Proof. Assume toward contradiction that $\mathrm{E}_{\mathrm{DTM}}$ is decidable, so that there exists a DTM T that decides this language. Define a new DTM K as follows:

On input $\langle M, w\rangle$ where M is a DTM and w is a string:

1. If w is not a string over the input alphabet of M, then *reject*.
2. Compute an encoding $\langle M_w\rangle$ of the DTM M_w described previously.
3. Run T on input $\langle M_w\rangle$. If T accepts $\langle M_w\rangle$, then *reject*, and otherwise *accept*.

Now, suppose that M is a DTM and $w \in \mathrm{L}(M)$, and consider the behavior of K on input $\langle M, w\rangle$. Because M accepts w, it holds that M_w accepts *every* string over its alphabet—because whatever string you give it as input, it erases this string and runs M on w, leading to acceptance. It is therefore certainly not the case that $\mathrm{L}(M_w) = \varnothing$, so T must reject $\langle M_w\rangle$, and therefore K accepts $\langle M, w\rangle$.

On the other hand, if M is a DTM and $w \notin \mathrm{L}(M)$ then K will reject the input $\langle M, w\rangle$. Either w is not a string over the alphabet of M, which immediately leads to rejection, or M either rejects or runs forever on input w. In this second case, M_w either rejects or runs forever on every string, and therefore $\mathrm{L}(M_w) = \varnothing$. The DTM T therefore accepts $\langle M_w\rangle$, causing K to reject the input $\langle M, w\rangle$.

Thus, K decides $\mathrm{A}_{\mathrm{DTM}}$, which contradicts the fact that this language is undecidable. We conclude that $\mathrm{E}_{\mathrm{DTM}}$ is undecidable, as required. □

Lecture 16

Computable functions and mapping reductions

In this lecture we will discuss the notion of a *reduction*, which is useful for proving that certain languages are undecidable (or that they are non-Turing-recognizable).

16.1 Computable functions and reductions

Before discussing reductions, we must define what it means for a function to be *computable*.

Definition 16.1. Let Σ be an alphabet and let $f : \Sigma^* \rightarrow \Sigma^*$ be a function. It is said that f is *computable* if there exists a DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ whose yields relation satisfies

$$(q_0, \sqcup)\, w \vdash_M^* (q_{\text{acc}}, \sqcup)\, f(w) \tag{16.1}$$

for every string $w \in \Sigma^*$.

In words, a function $f : \Sigma^* \rightarrow \Sigma^*$ is computable if there exists a DTM M such that, if we run M on input w, it will eventually accept, with just the output $f(w)$ written on the tape (and the tape head scanning the square to the left of this output string). It is not really important that this DTM accepts, as opposed to rejecting—the fact that it always halts with the correct output of the function written on the tape is what is important.

Now that we have introduced the definition of computable functions, we can move on to reductions. There are, in fact, many different types of reductions—the specific type of reduction we will consider is sometimes called a *mapping reduction*—but because this is the only type of reduction we will be concerned with, we'll stick with the simpler term *reduction*.

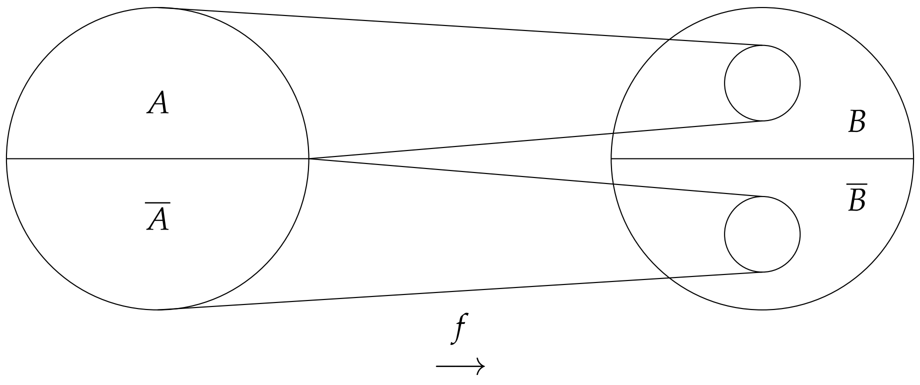


Figure 16.1: An illustration of a reduction f from A to B.

Definition 16.2. Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be languages. It is said that A *reduces* to B if there exists a computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that

$$w \in A \Leftrightarrow f(w) \in B \tag{16.2}$$

for all $w \in \Sigma^*$. One writes

$$A \leq_m B \tag{16.3}$$

to indicate that A reduces to B, and any function f that establishes that this is so may be called a *reduction* from A to B.

Figure 16.1 illustrates the action of a reduction. Intuitively speaking, a reduction is a way of transforming one computational decision problem into another. Imagine that you receive an input string $w \in \Sigma^*$, and you wish to determine whether or not w is contained in some language A. Perhaps you do not know how to make this determination, but you happen to have a friend who is able to tell you whether or not a particular string $y \in \Sigma^*$ is contained in a different language B. If you have a reduction f from A to B, then you can determine whether or not $w \in A$ using your friend's help: you compute $y = f(w)$, ask your friend whether or not $y \in B$, and take their answer as your answer to whether or not $w \in A$.

The following theorem has a simple and direct proof, but it will nevertheless have central importance with respect to the way that we use reductions to reason about decidability and Turing recognizability.

Theorem 16.3. *Let Σ be an alphabet, let $A, B \subseteq \Sigma^*$ be languages, and assume $A \leq_m B$. The following two implications hold:*

1. *If B is decidable, then A is decidable.*
2. *If B is Turing recognizable, then A is Turing recognizable.*

Proof. Let $f : \Sigma^* \to \Sigma^*$ be a reduction from A to B. We know that such a function exists by the assumption $A \leq_m B$.

We will first prove the second implication. Because B is Turing recognizable, there must exist a DTM M_B such that $B = \mathrm{L}(M_B)$. Define a new DTM M_A as follows:

```
On input $w \in \Sigma^*$:
1.  Compute $y = f(w)$.
2.  Run $M_B$ on input $y$.
```

It is possible to define a DTM in this way because f is a computable function.

For a given input string $w \in A$, we have that $y = f(w) \in B$, because this property is guaranteed by the reduction f. When M_A is run on input w, it will therefore accept because M_B accept y. Along similar lines, if $w \notin A$, then $y = f(w) \notin B$. When M_A is run on input w, it will therefore not accept because M_B does not accepts y. (It may be that these machines reject or run forever, but we do not care which.) It has been established that $A = \mathrm{L}(M_A)$, and therefore A is Turing recognizable.

The proof for the first implication is almost identical, except that we take M_B to be a DTM that decides B. The DTM M_A defined above then decides A, and therefore A is decidable. □

We will soon find some applications of the previous theorem, but let us first observe two simple but nevertheless useful facts about reductions.

Proposition 16.4. *Let Σ be an alphabet and let $A, B, C \subseteq \Sigma^*$ be languages. If it holds that $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$. (In other words, $\leq_m$ is a transitive relation among languages.)*

Proof. As $A \leq_m B$ and $B \leq_m C$, there must exist computable functions $f : \Sigma^* \to \Sigma^*$ and $g : \Sigma^* \to \Sigma^*$ such that

$$w \in A \Leftrightarrow f(w) \in B \quad \text{and} \quad w \in B \Leftrightarrow g(w) \in C \tag{16.4}$$

for all $w \in \Sigma^*$.

Define a function $h : \Sigma^* \to \Sigma^*$ as $h(w) = g(f(w))$ for all $w \in \Sigma^*$. It is evident that h is a computable function: if we have DTMs M_f and M_g that compute f and g, respectively, then we can easily obtain a DTM M_h that computes h by first running M_f and then running M_g.

It remains to observe that h is a reduction from A to C. If $w \in A$, then $f(w) \in B$, and therefore $h(w) = g(f(w)) \in C$; and if $w \notin A$, then $f(w) \notin B$, and therefore $h(w) = g(f(w)) \notin C$. □

Proposition 16.5. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be languages. It holds that $A \leq_m B$ if and only if $\overline{A} \leq_m \overline{B}$.*

Proof. This proposition is almost trivial—for a given function $f : \Sigma^* \rightarrow \Sigma^*$ and a string $w \in \Sigma^*$, the statements

$$w \in A \Leftrightarrow f(w) \in B \qquad \text{and} \qquad w \in \overline{A} \Leftrightarrow f(w) \in \overline{B} \tag{16.5}$$

are logically equivalent. If we have a reduction f from A to B, then the same function also serves as a reduction from $\overline{A}$ to $\overline{B}$, and vice versa. □

16.2 Proving undecidability through reductions

It is possible to use Theorem 16.3 to prove that certain languages are either decidable or Turing recognizable, but we will focus mainly on using it to prove that languages are either *not* decidable or *not* Turing recognizable. When using the theorem in this way, we consider the two implications in the contrapositive form. That is, if two languages $A, B \subseteq \Sigma^*$ satisfy $A \leq_m B$, then the following two implications hold:

1. If A is undecidable, then B is undecidable.
2. If A is non-Turing-recognizable, then B is non-Turing-recognizable.

So, if we want to prove that a particular language B is undecidable, then it suffices to pick any language A that we already know to be undecidable, and then prove $A \leq_m B$. The situation is similar for proving languages to be non-Turing-recognizable. The examples that follow illustrate how this is done.

Example 16.6 ($\text{A}_{\text{DTM}} \leq_m \text{HALT}$)**.** Recall the following two languages from the previous lecture:

$$\text{A}_{\text{DTM}} = \{\langle M, w\rangle \,:\, M \text{ is a DTM and } w \in \text{L}(M)\}, \tag{16.6}$$
$$\text{HALT} = \{\langle M, w\rangle \,:\, M \text{ is a DTM that halts on input } w\}. \tag{16.7}$$

We will prove that the first one reduces to the second, meaning $\text{A}_{\text{DTM}} \leq_m \text{HALT}$. (As this is our first example, the discussion will include more detail than we would typically include when proving reductions—when you prove a reduction on an assignment or exam, you would not be expected to provide this much detail.)

The first thing we will need to consider is a simple way of modifying an arbitrary DTM M to obtain a slightly different one. In particular, for an arbitrary DTM M, let us define a new DTM K_M as follows:

```
On input $w$:
1. Run $M$ on input $w$.
2.     If $M$ accepts $w$ then accept.
3.     If $M$ rejects $w$, then run forever.
```

Of course, if it is the case that M runs forever on some input string w, then K_M also runs forever on w—the description above is meant to suggest that K_M makes a decision to either accept or purposely run forever in the event that M halts.

If you are given a description of a DTM M, it is very easy to come up with a description of a DTM K_M that operates as suggested above: just replace the reject state of M with a new state that purposely causes an infinite loop (by repeatedly moving the tape head to the right, for instance).

Now let us define a function $f : \Sigma^* \to \Sigma^*$, where Σ is the alphabet over which A_{DTM} and HALT are defined, as follows:

$$f(x) = \begin{cases} \langle K_M, w\rangle & \text{if } x = \langle M, w\rangle \text{ for a DTM } M \text{ and a string } w \\ x & \text{otherwise.} \end{cases} \tag{16.8}$$

The most important property of this function is that

$$f\big(\langle M, w\rangle\big) = \langle K_M, w\rangle \tag{16.9}$$

whenever M is a DTM and x is a string. The other part is not so interesting—but we should nevertheless indicate how this function is defined on an input that happens to not take the form $\langle M, w\rangle$ for a DTM M and a string w. We have opted to define $f(x) = x$ in this situation so that $f(x) \notin \text{HALT}$.

The function f is computable: all it does is that it essentially looks at an input string, determines whether or not this string is an encoding $\langle M, w\rangle$ of a DTM M and a string w, and if so it replaces M with K_M as described above. Because it is straightforward to modify M to obtain K_M, there is no question that this modification could be performed by a DTM.

Now let us check to see that f is a reduction from A_{DTM} to HALT. Suppose first that we have an input $\langle M, w\rangle \in \text{A}_{\text{DTM}}$. These implications hold:

$$\begin{aligned} &\langle M, w\rangle \in \text{A}_{\text{DTM}} \;\Rightarrow\; M \text{ accepts } w \;\Rightarrow\; K_M \text{ accepts } w \\ &\quad\Rightarrow\; K_M \text{ halts on } w \;\Rightarrow\; \langle K_M, w\rangle \in \text{HALT} \;\Rightarrow\; f(\langle M, w\rangle) \in \text{HALT}. \end{aligned} \tag{16.10}$$

We therefore have

$$\langle M, w\rangle \in \text{A}_{\text{DTM}} \;\Rightarrow\; f(\langle M, w\rangle) \in \text{HALT}, \tag{16.11}$$

which is half of what we need to verify that f is indeed a reduction from A_{DTM} to HALT. It remains to consider the output of the function f on inputs that are not contained in A_{DTM}, and here there are two cases: one is that the input takes the form $\langle M, w\rangle$ for a DTM M and a string w, and the other is that it does not. For the first case, we have these implications:

$$\begin{aligned} \langle M, w\rangle \notin \text{A}_{\text{DTM}} \;\Rightarrow\; M \text{ does not accept } w \;\Rightarrow\; K_M \text{ runs forever on } w \\ \Rightarrow\; \langle K_M, w\rangle \notin \text{HALT} \;\Rightarrow\; f(\langle M, w\rangle) \notin \text{HALT}. \end{aligned} \tag{16.12}$$

The key here is that K_M is defined so that it will definitely run forever in case M does not accept (regardless of whether that happens by M rejecting or running forever). The remaining case is that we have a string $x \in \Sigma^*$ that does not take the form $\langle M, w\rangle$ for a DTM M and a string w, and in this case it trivially holds that $f(x) = x \notin \text{HALT}$. We have therefore proved that

$$x \in \text{A}_{\text{DTM}} \Leftrightarrow f(x) \in \text{HALT}, \tag{16.13}$$

and therefore $\text{A}_{\text{DTM}} \leq_m \text{HALT}$.

We already proved that HALT is undecidable, but the fact that $\text{A}_{\text{DTM}} \leq_m \text{HALT}$ provides an alternative proof: because we already know that A_{DTM} is undecidable, it follows that HALT is also undecidable.

It might not seem that there is any advantage to this proof over the proof we saw in the previous lecture that HALT is undecidable (which wasn't particularly difficult). We have, however, established a closer relationship between A_{DTM} and HALT than we did previously. In general, using a reduction is sometimes an easy shortcut to proving that a language is undecidable (or non-Turing-recognizable).

Example 16.7 (DIAG $\leq_m$ E_{DTM})**.** Our first example of a non-Turing-recognizable language, from the previous lecture, was this language:

$$\text{DIAG} = \big\{\langle M\rangle \,:\, M \text{ is a DTM and } \langle M\rangle \notin \text{L}(M)\big\}. \tag{16.14}$$

We also defined the following language, and proved it is undecidable:

$$\text{E}_{\text{DTM}} = \big\{\langle M\rangle \,:\, M \text{ is a DTM and } \text{L}(M) = \varnothing\big\}. \tag{16.15}$$

We will now prove that DIAG $\leq_m$ E_{DTM}. Because we already know that DIAG is non-Turing-recognizable, we will conclude from this reduction that E_{DTM} is not just undecidable, but in fact it is also non-Turing-recognizable.

Let us use a similar trick to one we used in the previous lecture: for a given DTM M, let us define a new DTM M_{self} as follows.

```
On input $w$:
    Ignore the input string $w$ and run $M$ on input $\langle M\rangle$.
```

This description really only makes sense if the input alphabet of M includes the symbols in the encoding $\langle M \rangle$, so let us agree that M_{self} immediately rejects if this is not the case.

Now let us define a function $f : \Sigma^* \rightarrow \Sigma^*$, for Σ being the alphabet over which DIAG and E_{DTM} are defined, as follows:

$$f(x) = \begin{cases} \langle M_{\text{self}} \rangle & \text{if } x = \langle M \rangle \text{ for a DTM } M \\ x & \text{otherwise.} \end{cases} \tag{16.16}$$

If you think about it for a few moments, it should not be hard to convince yourself that f is computable.

Now let us verify that f is a reduction from DIAG to E_{DTM}. For any string $x \in \text{DIAG}$ we have that $x = \langle M \rangle$ for some DTM M that satisfies $\langle M \rangle \notin \text{L}(M)$. In this case we have that $f(x) = \langle M_{\text{self}} \rangle$, and because $\langle M \rangle \notin \text{L}(M)$ it must be that M_{self} never accepts. It is therefore the case that $f(x) = \langle M_{\text{self}} \rangle \in \text{E}_{\text{DTM}}$.

Now suppose that $x \notin \text{DIAG}$. There are two cases: either $x = \langle M \rangle$ for a DTM M such that $\langle M \rangle \in \text{L}(M)$, or x does not encode a DTM at all. If it is the case that $x = \langle M \rangle$ for a DTM M such that $\langle M \rangle \in \text{L}(M)$, we have that M_{self} accepts *every* string over its alphabet, and therefore $f(x) = \langle M_{\text{self}} \rangle \notin \text{E}_{\text{DTM}}$. If it is the case that x does not encode a DTM, then it trivially holds that $f(x) = x \notin \text{E}_{\text{DTM}}$.

We have proved that

$$x \in \text{DIAG} \Leftrightarrow f(x) \in \text{E}_{\text{DTM}}, \tag{16.17}$$

so the proof that $\text{DIAG} \leq_m \text{E}_{\text{DTM}}$ is complete.

Example 16.8 ($\text{A}_{\text{DTM}} \leq_m \text{AE}$)**.** Define a language

$$\text{AE} = \big\{ \langle M \rangle \;:\; M \text{ is a DTM that accepts } \varepsilon \big\}. \tag{16.18}$$

The name AE stands for "accepts the empty string."

It is very easy to prove that $\text{AE} \leq_m \text{A}_{\text{DTM}}$. (The main point of the example is to prove the other reduction, $\text{A}_{\text{DTM}} \leq_m \text{AE}$, so this is just a warm-up.) Define a function $f : \Sigma^* \rightarrow \Sigma^*$ as follows:

$$f(w) = \begin{cases} \langle M, \varepsilon \rangle & \text{if } w = \langle M \rangle \text{ for some DTM } M \\ \langle M_0, \varepsilon \rangle & \text{otherwise,} \end{cases} \tag{16.19}$$

where M_0 is some fixed DTM that always rejects. This function is computable. Now let us check that f is a valid reduction from AE to A_{DTM}.

First, for any string $w \in \mathrm{AE}$, we must have $w = \langle M \rangle$ for M being a DTM that accepts ε. In this case $f(w) = \langle M, \varepsilon \rangle$, which is contained in $\mathrm{A}_{\mathrm{DTM}}$ (because M accepts ε).

Now consider any string $w \notin \mathrm{AE}$. There are two cases: either $w = \langle M \rangle$ for some DTM M, or this is not the case. If $w = \langle M \rangle$ for a DTM M, then $w \notin \mathrm{AE}$ implies that M does not accept ε. In this case we have $f(w) = \langle M, \varepsilon \rangle \notin \mathrm{A}_{\mathrm{DTM}}$ (because M does not accept ε). If $w \neq \langle M \rangle$ for a DTM M, then $f(w) = \langle M_0, \varepsilon \rangle \notin \mathrm{A}_{\mathrm{DTM}}$ (because M_0 does not accept any strings at all, including ε).

We have shown that $w \in \mathrm{AE} \Leftrightarrow f(w) \in \mathrm{A}_{\mathrm{DTM}}$ holds for every string $w \in \Sigma^*$, and therefore $\mathrm{AE} \leq_m \mathrm{A}_{\mathrm{DTM}}$, as required.

That was indeed easy, but now let's prove the reverse reduction: $\mathrm{A}_{\mathrm{DTM}} \leq_m \mathrm{AE}$. We'll use a similar trick to the one from the previous example. For every DTM M and every string w over the alphabet of M, define a new DTM M_w as follows:

```
On input $x$:
    Erase $x$ and run $M$ on input $w$.
```

Now define a function $f : \Sigma^* \to \Sigma^*$ as follows:

$$f(y) = \begin{cases} \langle M_w \rangle & \text{if } y = \langle M, w \rangle \text{ for some DTM } M \text{ and string } w \\ \langle M_0 \rangle & \text{otherwise.} \end{cases} \tag{16.20}$$

(Just like for the easier reduction above, M_0 is a DTM that always rejects.) Now let us check that f is a valid reduction from $\mathrm{A}_{\mathrm{DTM}}$ to AE.

First, for any string $y \in \mathrm{A}_{\mathrm{DTM}}$ we have $y = \langle M, w \rangle$ for a DTM M that accepts the string w. In this case, $g(y) = \langle M_w \rangle$. We have that M_w accepts every string, including the empty string, because M accepts w. Therefore $f(y) = \langle M_w \rangle \in \mathrm{AE}$.

Now consider any string $y \notin \mathrm{A}_{\mathrm{DTM}}$. Again there are two cases: either $y = \langle M, w \rangle$ for some DTM M and input string w, or this is not the case. If $y = \langle M, w \rangle$ for a DTM M and a string w, then $y \notin \mathrm{A}_{\mathrm{DTM}}$ implies that M does not accept w. In this case we have $f(y) = \langle M_w \rangle \notin \mathrm{AE}$, because M_w does not accept any strings at all (including the empty string). If $y \neq \langle M, w \rangle$ for a DTM M and string w, then $f(y) = \langle M_0 \rangle \notin \mathrm{AE}$ (again because M_0 does not accept any strings, including ε).

We have shown that $y \in \mathrm{A}_{\mathrm{DTM}} \Leftrightarrow f(y) \in \mathrm{AE}$ holds for every string $y \in \Sigma^*$, and therefore $\mathrm{A}_{\mathrm{DTM}} \leq_m \mathrm{AE}$, as required.

Example 16.9 ($\mathrm{E}_{\mathrm{DTM}} \leq_m \mathrm{REG}$)**.** Define a language as follows:

$$\mathrm{REG} = \big\{ \langle M \rangle \;:\; M \text{ is a DTM such that } \mathrm{L}(M) \text{ is regular} \big\}. \tag{16.21}$$

We will prove $\mathrm{E}_{\mathrm{DTM}} \leq_m \mathrm{REG}$.

We'll need a strange way to modify DTMs in order to do this one. Given an arbitrary DTM M, let us define a new DTM K_M as follows:

```
On input $x \in \{0,1\}^*$:
1. Set $t \leftarrow 1$.
2. For every input string $w$ over the input alphabet of $M$ satisfying $|w| \leq t$:
3.     Run $M$ for $t$ steps on input $w$.
4.     If $M$ accepts $w$ within $t$ steps, goto 6.
5. Set $t \leftarrow t + 1$ and goto 2.
6. *Accept* if $x \in \{0^n 1^n : n \in \mathbb{N}\}$, *reject* otherwise.
```

This is indeed a strange way to define a DTM, but it's alright if it's strange—we're just proving a reduction.

Now let us define a function $f : \Sigma^* \to \Sigma^*$ as

$$f(x) = \begin{cases} \langle K_M \rangle & \text{if } x = \langle M \rangle \text{ for a DTM } M \\ x & \text{otherwise.} \end{cases} \tag{16.22}$$

This is a computable function, and it remains to verify that it is a reduction from E_{DTM} to REG.

Suppose $\langle M \rangle \in \text{E}_{\text{DTM}}$. We therefore have that $\text{L}(M) = \varnothing$; and by considering the way that K_M behaves we see that $\text{L}(K_M) = \varnothing$ as well (because we never get to step 6 if M never accepts). The empty language is regular, and therefore $f(\langle M \rangle) = \langle K_M \rangle \in \text{REG}$.

On the other hand, if M is a DTM and $\langle M \rangle \notin \text{E}_{\text{DTM}}$, then M must accept at least one string. This means that $\text{L}(K_M) = \{0^n 1^n : n \in \mathbb{N}\}$, because K_M will eventually find a string accepted by M, reach step 6, and then accept or reject based on whether the input string x is contained in $\{0^n 1^n : n \in \mathbb{N}\}$. Therefore $f(\langle M \rangle) = \langle K_M \rangle \notin \text{REG}$. The remaining case, in which x does not encode a DTM, is straightforward as usual: we have $f(x) = x \notin \text{REG}$ in this case.

We have shown that $x \in \text{E}_{\text{DTM}} \Leftrightarrow f(x) \in \text{REG}$ holds for every string $x \in \Sigma^*$, and therefore $\text{E}_{\text{DTM}} \leq_m \text{REG}$, as required. We conclude that REG is not Turing recognizable, as we already know that E_{DTM} is not Turing recognizable.

Lecture 17

Further discussion of Turing machines

In this lecture we will discuss various aspects of decidable and Turing-recognizable languages that were not mentioned in previous lectures. In particular, we will discuss closure properties of these classes of languages, briefly discuss nondeterministic Turing machines, and prove a useful alternative characterization of Turing-recognizable languages.

17.1 Decidable language closure properties

The decidable languages are closed under many of the operations on languages that we've considered thus far in the course (although not all). While we won't go through every operation we've discussed, it is worthwhile to mention some basic examples.

First let us observe that the decidable languages are closed under the regular operations as well as complementation. In short, if A and B are decidable, then there is no difficulty in deciding the languages $A \cup B$, AB, A^*, and $\overline{A}$ in a straightforward way.

Proposition 17.1. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be decidable languages. The languages $A \cup B$, AB, and A^* are decidable.*

Proof. Because the languages A and B are decidable, there must exist a DTM M_A that decides A and a DTM M_B that decides B.

The following DTM decides $A \cup B$, which implies that $A \cup B$ is decidable.

```
On input $w$:
1.  Run $M_A$ and $M_B$ on input $w$.
2.  If either $M_A$ or $M_B$ accepts, then *accept*, otherwise *reject*.
```

The following DTM decides AB, which implies that AB is decidable.

```
On input $w$:
1.  For every choice of strings $u, v$ satisfying $w = uv$:
2.          Run $M_A$ on input $u$ and run $M_B$ on input $v$.
3.          If both $M_A$ and $M_B$ accept, then accept.
4.  Reject.
```

Finally, the following DTM decides A^*, which implies that A^* is decidable.

```
On input $w$:
1.  If $w = \varepsilon$, then accept.
2.  For every way of writing $w = u_1 \cdots u_m$ for nonempty strings $u_1, \dots, u_m$:
2.          Run $M_A$ on each of the strings $u_1, \dots, u_m$.
3.          If $M_A$ accepts all of the strings $u_1, \dots, u_m$, then accept.
4.  Reject.
```

This completes the proof. □

Proposition 17.2. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a decidable language. The language $\overline{A}$ is decidable.*

Perhaps we don't even need to bother writing a proof for this one—if a DTM M decides A, then one can obtain a DTM deciding $\overline{A}$ by simply exchanging the accept and reject states of M.

There are a variety of other operations under which the decidable languages are closed. For example, because the decidable languages are closed under union and complementation, we immediately have that they are closed under intersection and symmetric difference. Another example is string reversal: if a language A is decidable, then A^{R} is also decidable, because a DTM can decide A^{R} simply by reversing the input string, then deciding whether the string that is obtained is contained in A.

There are, however, some natural operations under which the decidable languages are not closed. The following example shows that this is the case for the prefix operation.

Example 17.3. The language $\operatorname{Prefix}(A)$ might not be decidable, even if A is decidable. To construct an example that illustrates that this is so, let us first take $B \subseteq \{0,1\}^*$ to be any language that is Turing recognizable but not decidable (such as HALT, assuming we define this language with respect to a binary string encoding of DTMs).

Let M_B be a DTM such that $\mathrm{L}(M_B) = B$, and define a language $A \subseteq \{0,1,2\}^*$ as follows:

$$A = \{w2^t \,:\, M_B \text{ accepts } w \text{ within } t \text{ steps}\}. \tag{17.1}$$

This is a decidable language, but $\mathrm{Prefix}(A)$ is not—for if $\mathrm{Prefix}(A)$ were decidable, then one could easily decide B by using the fact that a string $w \in \{0,1\}^*$ is contained in B if and only if $w2 \in \mathrm{Prefix}(A)$. (That is, $w \in B$ and $w2 \in \mathrm{Prefix}(A)$ are both equivalent to the existence of a positive integer t such that $w2^t \in A$.)

17.2 Turing-recognizable language closure properties

The Turing-recognizable languages are also closed under a variety of operations, although not precisely the same operations under which the decidable languages are closed.

Let us begin with the regular operations, under which the Turing-recognizable languages are indeed closed. In this case, one needs to be a bit more careful than was sufficient when proving the analogous property for decidable languages, as the Turing machines that recognize these languages might run forever.

Proposition 17.4. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be Turing-recognizable languages. The languages $A \cup B$, AB, and A^* are Turing recognizable.*

Proof. Because the languages A and B are Turing recognizable, there must exist DTMs M_A and M_B such that $\mathrm{L}(M_A) = A$ and $\mathrm{L}(M_B) = B$.

The following DTM recognizes the language $A \cup B$, which implies that $A \cup B$ is Turing recognizable.

```
On input $w$:
1.  Set $t \leftarrow 1$.
2.  Run $M_A$ and $M_B$ on input $w$ for $t$ steps.
3.  If either $M_A$ or $M_B$ accepts within $t$ steps, then accept.
4.  Set $t \leftarrow t + 1$ and goto 2.
```

The following DTM recognizes AB, which implies that AB is Turing recognizable.

```
On input $w$:
1.  Set $t \leftarrow 1$.
2.  For every choice of strings $u, v$ satisfying $w = uv$:
3.      Run $M_A$ on input $u$ and run $M_B$ on input $v$, both for $t$ steps.
4.      If both $M_A$ and $M_B$ accept within $t$ steps, then accept.
5.  Set $t \leftarrow t + 1$ and goto 2.
```

Finally, the following DTM recognizes A^*, which implies that A^* is Turing recognizable.

```
On input $w$:
1.  If $w = \varepsilon$, then accept.
2.  Set $t \leftarrow 1$.
3.  For every way of writing $w = u_1 \cdots u_m$ for nonempty strings $u_1, \ldots, u_m$:
4.          Run $M_A$ on each of the strings $u_1, \ldots, u_m$ for $t$ steps.
5.          If $M_A$ accepts all of the strings $u_1, \ldots, u_m$ within $t$ steps, then accept.
6.  Set $t \leftarrow t + 1$ and goto 3.
```

This completes the proof. □

The Turing-recognizable languages are also closed under intersection. This can be proved through a similar method to closure under union, but in fact this is a situation in which we don't actually need to be as careful about running forever.

Proposition 17.5. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be Turing-recognizable languages. The language $A \cap B$ is Turing recognizable.*

Proof. Because the languages A and B are Turing recognizable, there must exist DTMs M_A and M_B such that $\mathrm{L}(M_A) = A$ and $\mathrm{L}(M_B) = B$.

The following DTM recognizes the language $A \cap B$, which implies that $A \cap B$ is Turing recognizable.

```
On input $w$:
1.  Run $M_A$ on input $w$.
2.  Run $M_B$ on input $w$.
3.  If both $M_A$ and $M_B$ accept, then accept, otherwise reject.
```

If it so happens that either M_A or M_B runs forever on input w, then the DTM just described also runs forever, which is fine—what is relevant is that the DTM accepts if and only if both M_A and M_B accept. This completes the proof. □

The Turing-recognizable languages are not, however, closed under complementation. We already proved in Lecture 15 that if A and $\overline{A}$ are Turing recognizable, then A is decidable. We also know that there exist languages, such as HALT, that are Turing recognizable but not decidable—so it cannot be that the Turing-recognizable languages are closed under complementation.

Finally, there are some operations under which the Turing-recognizable languages are closed, but under which the decidable languages are not. For example, if A is Turing recognizable, then so too are the languages $\mathrm{Prefix}(A)$, $\mathrm{Suffix}(A)$, and $\mathrm{Substring}(A)$, but this is not necessarily so for decidable languages.

17.3 Nondeterministic Turing machines

One can define a nondeterministic variant of the Turing machine model along similar lines to the definition of nondeterministic finite automata, as compared with deterministic finite automata.

Definition 17.6. A *nondeterministic Turing machine* (or NTM, for short) is a 7-tuple

$$N = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}), \tag{17.2}$$

where Q is a finite and nonempty set of *states*; Σ is an alphabet, called the *input alphabet*, which may not include the blank symbol $\sqcup$; Γ is an alphabet, called the *tape alphabet*, which must satisfy $\Sigma \cup \{\sqcup\} \subseteq \Gamma$; δ is a transition function having the form

$$\delta : Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow \mathcal{P}\big(Q \times \Gamma \times \{\leftarrow, \rightarrow\}\big); \tag{17.3}$$

$q_0 \in Q$ is the *initial state*; and $q_{\text{acc}}, q_{\text{rej}} \in Q$ are the *accept* and *reject* states, which satisfy $q_{\text{acc}} \neq q_{\text{rej}}$.

For a given state $q \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}$ and tape symbol a, the elements in the set $\delta(q, a)$ represent the possible moves the NTM can make. The yields relation of an NTM is then defined in a similar manner to DTMs, except this time there may be multiple choices of configurations that are reachable from a given configuration.

When we think about the yields relation of an NTM N, and the way that it computes on a given input string w, it is natural to envision a *computation tree*. This is a possibly infinite tree in which each node corresponds to a configuration. The root node is the initial configuration, and the children of each node are all of the configurations that can be reached in one step from the configuration represented by that node. If we thought about such a tree for a deterministic computation, it would not be very interesting—it would essentially be a stick, where each node has either one child (representing the next configuration, one step later) or no children (in the case of an accepting or rejecting configuration). For a nondeterministic computation, however, the tree can branch, as each configuration could have multiple next configurations. Note that we allow multiple nodes in the tree to represent the same configuration, as there could be different *computation paths* (or paths starting from the root node and going down the tree) that reach a particular configuration.

Acceptance for an NTM is defined in a similar way to NFAs (and PDAs as well). That is, if there *exists* a sequence of computation steps that reach an accepting configuration, then the NTM accepts, and otherwise it does not. In terms of the computation tree, this simply means that somewhere in the tree there exists an accepting configuration. Note that there might or might not be rejecting configurations in the computation tree, and there might also be computation paths starting from the root that are infinite, but all that matters when it comes to defining acceptance is whether or not there exists an accepting configuration that is reachable from the initial configuration. Formally speaking, the definition of acceptance for an NTM is the same as for a DTM, albeit for a yields relation that comes from an NTM. That is, N accepts w if there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_0, \sqcup)w \vdash_N^* u(q_{\text{acc}}, a)v. \tag{17.4}$$

As usual, we write $\mathrm{L}(N)$ to denote the language of all strings w accepted by an NTM N, which we refer to as the language recognized by N.

The following theorem states that the languages recognized by NTMs are the same as for DTMs.

Theorem 17.7. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. There exists an NTM N such that $\mathrm{L}(N) = A$ if and only if A is Turing recognizable.*

It is not difficult to prove this theorem, but I will leave it to you to contemplate if you are interested. The key idea for showing that the language $\mathrm{L}(N)$ for an NTM N is Turing recognizable is to perform a breadth-first search of the computation tree of N on a given input. If there is an accepting configuration anywhere in the tree, a breadth-first search will find it. Of course, because computation trees may be infinite, the search might never terminate—this is unavoidable, but it is not an obstacle for proving the theorem. Note that an alternative approach of performing a depth-first search of the computation tree would not work: it might happen that such a search descends down an infinite path of the tree, potentially missing an accepting configuration elsewhere in the tree.

17.4 The range of a computable function

We will now consider an alternative characterization of Turing-recognizable languages (with the exception of the empty language), which is that they are precisely the languages that are equal to the *range* of a computable function. Recall that the range of a function $f : \Sigma^* \rightarrow \Sigma^*$ is defined as follows:

$$\mathrm{range}(f) = \{f(w) : w \in \Sigma^*\}. \tag{17.5}$$

Theorem 17.8. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a nonempty language. The following two statements are equivalent:*

1. *A is Turing recognizable.*
2. *There exists a computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that $A = \mathrm{range}(f)$.*

Proof. Let us first suppose that the second statement holds: $A = \text{range}(f)$ for some computable function $f : \Sigma^* \to \Sigma^*$. Let us define a DTM M as follows:

```
On input $w$:
1. Set $x \leftarrow \varepsilon$.
2. Compute $y = f(x)$, and accept if $w = y$.
3. Increment $x$ with respect to the lexicographic ordering of $\Sigma^*$ and goto 2.
```

In essence, this DTM searches over Σ^* to find a string that f maps to a given input string w. If it is the case that $w \in \text{range}(f)$, then M will eventually find $x \in \Sigma^*$ such that $f(x) = w$ and accept, while M will certainly not accept if $w \notin \text{range}(f)$. Thus, we have $\text{L}(M) = \text{range}(f) = A$, which implies that A is Turing recognizable.

Now suppose that A is Turing recognizable, so that there exists a DTM M such that $\text{L}(M) = A$. We will also make use of the assumption that A is nonempty—there exists at least one string in A, so we may take w_0 to be such a string. (If you like, you may define w_0 more concretely as the first string in A with respect to the lexicographic ordering of Σ^*, but it is not important for the proof that we make this particular choice.)

We will define a computable function f for which $A = \text{range}(f)$. Here is one such function:

$$f(x) = \begin{cases} w & \text{if } x = \langle w, t \rangle \text{ for } w \in \Sigma^* \text{ and } t \in \mathbb{N}, \text{ and} \\ & M \text{ accepts } w \text{ within } t \text{ steps} \\ w_0 & \text{otherwise.} \end{cases} \tag{17.6}$$

To be a bit more precise, the condition "$x = \langle w, t \rangle$ for $w \in \Sigma^*$ and $t \in \mathbb{N}$" refers to an encoding scheme through which a string $w \in \Sigma^*$ and a natural number $t \in \mathbb{N}$ are encoded as a string $\langle w, t \rangle$ over the alphabet Σ.

It is evident that the function f is computable: a DTM M_f can compute f by checking to see if the input has the form $\langle w, t \rangle$, simulating M for t steps on input w if so, and then outputting either w or w_0 depending on the outcome. If M accepts a particular string w, then it must hold that $w = f(\langle w, t \rangle)$ for some sufficiently large natural number t, so $A \subset \text{range}(f)$. On the other hand, every output of f is either a string w accepted by M or the string w_0, and therefore $\text{range}(f) \subseteq A$. It therefore holds that $A = \text{range}(f)$, which completes the proof. $\square$

Remark 17.9. The assumption that A is nonempty is essential in the previous theorem because it cannot be that $\text{range}(f) = \varnothing$ for a computable function f. Indeed, it cannot be that $\text{range}(f) = \varnothing$ for any function whatsoever. (If we were to consider *partial functions*, which are functions that may be undefined for some inputs, the situation would be different—but the standard interpretation of the word *function*, at least as far as this course is concerned, means *total function* and not *partial function*.)

Theorem 17.8 provides a useful characterization of the Turing-recognizable languages. For instance, you can use this theorem to come up with alternative proofs for all of the closure properties of the Turing-recognizable languages stated in the previous section.

For example, suppose that $A \subseteq \Sigma^*$ is a nonempty Turing-recognizable language, so that there must exist a computable function $f : \Sigma^* \to \Sigma^*$ such that $\text{range}(f) = A$. Define a new function

$$g(x) = \begin{cases} f(u_1)\cdots f(u_m) & \text{if } x = \langle u_1, \ldots, u_m \rangle \text{ for } m \geq 0 \text{ and} \\ & u_1, \ldots, u_m \in \Sigma^* \\ \varepsilon & \text{otherwise,} \end{cases} \tag{17.7}$$

where the statement "$x = \langle u_1, \ldots, u_m \rangle$ for $m \geq 0$ and $u_1, \ldots, u_m \in \Sigma^*$" refers to an encoding scheme in which zero or more strings $u_1, \ldots, u_m \in \Sigma^*$ are encoded as a single string over Σ. One sees that g is computable, and $\text{range}(g) = A^*$, which implies that A^* is Turing recognizable.

Here is another example of an application of Theorem 17.8.

Corollary 17.10. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be any infinite Turing-recognizable language. There exists an infinite decidable language $B \subseteq A$.*

Proof. Because A is infinite (and therefore nonempty), there exists a computable function $f : \Sigma^* \to \Sigma^*$ such that $A = \text{range}(f)$.

We will define a language B that satisfies the requirements of the corollary by first defining a DTM M and then arguing that $B = \mathrm{L}(M)$ works. This will require proving that M never runs forever (so that B is decidable), M only accepts strings that are contained in A (so that $B \subseteq A$), and M accepts infinitely many different strings (so that B is infinite). We'll prove each of those things, but first here is the DTM M:

```
On input $w$:
1.  Set $x \leftarrow \varepsilon$.
2.  Compute $y = f(x)$.
3.  If $y = w$ then *accept*.
4.  If $y > w$ (with respect to the lexicographic ordering of $\Sigma^*$) then *reject*.
5.  Increment $x$ with respect to the lexicographic ordering of $\Sigma^*$ and goto 2.
```

The fact that M never runs forever follows from the assumption that A is infinite. That is, because A is infinite, the function f must output infinitely many different strings, so regardless of what input string w is input into M, the loop will eventually reach a string x so that $f(x) = w$ or $f(x) > w$, either of which causes M to halt.

The fact that M only accepts strings in A follows from the fact that the condition for acceptance is that the input string w is equal to y, which is contained in $\text{range}(f) = A$.

Finally, let us observe that M accepts precisely the strings in this set:

$$\left\{ w \in \Sigma^* : \begin{array}{l} \text{there exists } x \in \Sigma^* \text{ such that } w = f(x) \\ \text{and } w > f(z) \text{ for all } z < x \end{array} \right\}. \tag{17.8}$$

The fact that this set is infinite follows from the assumption that $A = \text{range}(f)$ is infinite—for if the set were finite, there would necessarily be a maximal output of f with respect to the lexicographic ordering of Σ^*, contradicting the assumption that $\text{range}(f)$ is infinite.

The language $B = \text{L}(M)$ therefore satisfies the requirements of the corollary, which completes the proof. □

Lecture 18

Time-bounded computations

We now begin the final part of the course, which is on *complexity theory*. We'll have time to only scratch the surface—complexity theory is a rich subject, and many scientists around the world are engaged in a study of this field. Unlike formal language theory and computability theory, many of the central questions of complexity theory remain unanswered to this day.

The motivation for the subject of this lecture, which is on time-bounded computation, is simple: it is usually not enough to know that a language is decidable (or that a function is computable)—we need computations to be *efficient*. For instance, if we have a particular computational task that we would like to perform, and someone gives us a computational device that will complete this task only after running for one million years, it is practically useless.

It is typical that efficiency is equated with running time, and this is why we begin by considering the amount of time needed to perform computations. It is, however, possible to consider other notions of efficiency, by referring to things like memory usage, communication (in a distributed setting), power consumption, or a variety of other notions concerning resource usage.

18.1 Time complexity

We will start with a definition of the running time of a DTM, assuming that this DTM never runs forever.

Definition 18.1. Let M be a DTM with input alphabet Σ that halts on every input string $w \in \Sigma^*$. The *running time* of M is the function $t : \mathbb{N} \rightarrow \mathbb{N}$ defined as follows for each $n \in \mathbb{N}$:

$$t(n) = \begin{cases} \text{the maximum number of steps required for } M \\ \text{to halt, over all inputs } w \in \Sigma^* \text{ with } |w| = n. \end{cases} \tag{18.1}$$

One can define the running time for any Turing machine variant, but we will focus primarily on ordinary (one-tape) DTMs.

Deterministic time complexity classes

Next, for every function $f : \mathbb{N} \rightarrow \mathbb{N}$, we define a class of languages $\mathrm{DTIME}(f)$ representing those languages decidable in time $O(f(n))$.

Definition 18.2. Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a function. A language A is contained in the class $\mathrm{DTIME}(f)$ if there exists a DTM M that decides A and whose running time t satisfies $t(n) = O(f(n))$.

We define $\mathrm{DTIME}(f)$ in this way, using $O(f(n))$ rather than $f(n)$, because we are generally not interested in constant factors or in what might happen in finitely many special cases. One fact that motivates this choice is that it is usually possible to "speed up" a DTM by defining a new DTM, having a larger tape alphabet than the original, that succeeds in simulating multiple computation steps of the original DTM with each step it performs.

When it is reasonable to do so, we use the variable name n to refer to the input length for whatever language or DTM we are considering. So, for example, we may refer to a DTM that runs in time $O(n^2)$ or refer to the class of languages $\mathrm{DTIME}(n^2)$ with the understanding that we are speaking of the function $f(n) = n^2$, without explicitly saying that n is the input length.

We also sometimes refer to classes such as

$$\mathrm{DTIME}\big(n\sqrt{n}\big) \quad \text{or} \quad \mathrm{DTIME}\big(n^2\log(n)\big), \tag{18.2}$$

where the function f that we are implicitly referring to appears to take non-integer values for some choices of n. This is done in an attempt to keep the expressions of these classes simple and intuitive, and you can interpret these things as referring to functions of the form $f : \mathbb{N} \rightarrow \mathbb{N}$ obtained by rounding up to the next positive integer. For instance, $\mathrm{DTIME}(n^2\log(n))$ means $\mathrm{DTIME}(f)$ for

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \lceil n^2\log(n) \rceil & \text{otherwise.} \end{cases} \tag{18.3}$$

Because the definition of $\mathrm{DTIME}(f)$ refers only to $O(f(n))$ and not $f(n)$, you could just as easily modify Definition 18.2 to allow f to be a real-valued function—the resulting complexity class $\mathrm{DTIME}(f)$ would be the same for the examples mentioned above.

Example 18.3. The language $A = \{0^m 1^m : m \in \mathbb{N}\}$ is contained in DTIME(n^2). The example of a DTM for deciding this language from Lecture 12, for instance, runs in time $O(n^2)$ on inputs of length n.

It is, in fact, possible to do better: it is the case that $A \in \text{DTIME}(n\log(n))$. To see that A is contained in DTIME$(n\log(n))$, consider a DTM that repeatedly "crosses out" every other symbol on the tape, and compares the parity of the number of 0s and 1s crossed out after each pass over the input. Through this method, A can be decided in time $O(n\log(n))$ by making just a logarithmic number of passes over the portion of the tape initially containing the input.

After considering the previous example, it is natural to ask if one can do even better than $O(n\log(n))$ for the running time of a DTM deciding the language $A = \{0^m 1^m : m \in \mathbb{N}\}$. The answer is that this is not possible. This is a consequence of the following theorem (which we will not prove).

Theorem 18.4. *Let A be a language. If there exists a DTM M that decides A in time $o(n\log(n))$, meaning that the running time t of M satisfies*

$$\lim_{n\to\infty} \frac{t(n)}{n\log(n)} = 0, \tag{18.4}$$

then A is regular.

It is, of course, critical that we understand the previous theorem to be referring to ordinary, one-tape DTMs. With a two-tape DTM, for instance, it is easy to decide some nonregular languages, including $\{0^m 1^m : m \in \mathbb{N}\}$, in time $O(n)$.

Time-constructible functions

The complexity class DTIME(f) has been defined for an arbitrary function of the form $f : \mathbb{N} \to \mathbb{N}$, but there is a sense in which most functions of this form are uninteresting—because they have absolutely nothing to do with the running time of any DTM.

There are, in fact, some choices of functions $f : \mathbb{N} \to \mathbb{N}$ that are so strange that they lead to highly counter-intuitive results. For example, there exists a function f such that

$$\text{DTIME}(f) = \text{DTIME}(g), \qquad \text{for } g(n) = 2^{f(n)}; \tag{18.5}$$

even though g is exponentially larger than f, they both result in exactly the same deterministic time complexity classes. This doesn't necessarily imply anything important about time complexity, it's more a statement about the strangeness of the function f.

For this reason we define a collection of functions, called *time-constructible functions*, that represent well-behaved upper bounds on the possible running times of DTMs. Here is a precise definition.

Definition 18.5. Let $f : \mathbb{N} \to \mathbb{N}$ be a function satisfying $f(n) = \Omega(n \log(n))$. The function f is said to be *time constructible* if there exists a DTM M that operates as follows:

1. On each input 0^n the DTM M computes the binary representation of $f(n)$, for every $n \in \mathbb{N}$.
2. M runs in time $O(f(n))$.

It might not be clear why we would define a class of functions in this particular way, but the essence is that these are functions that can serve as upper bounds for DTM computations. That is, a DTM can compute $f(n)$ on any input of length n, and doing this doesn't take more than $O(f(n))$ steps—and then it has the number $f(n)$ written in a convenient form so that it could use this number to limit some subsequent part of its computation (perhaps the number of steps for which it runs during a second phase of its computation).

As it turns out, just about any reasonable function f with $f(n) = \Omega(n \log(n))$ that you are likely to care about as a bound on running time is time constructible. Examples include the following:

1. For any choice of an integer $k \geq 2$, the functions
$$f(n) = n^k \qquad \text{and} \qquad f(n) = k^n \tag{18.6}$$
are time constructible.
2. For any choice of an integer $k \geq 1$, the functions
$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \lceil n^k \log(n) \rceil & \text{otherwise} \end{cases} \tag{18.7}$$
and
$$f(n) = \lceil n^k \sqrt{n} \rceil \tag{18.8}$$
are time constructible.
3. If f and g are time-constructible functions, then the functions
$$h_1(n) = f(n) + g(n), \quad h_2(n) = f(n)g(n), \quad \text{and} \quad h_3(n) = f(g(n)) \tag{18.9}$$
are also time constructible.

18.2 The time-hierarchy theorem

What we will do next is to discuss a fairly intuitive theorem concerning time complexity. A highly informal statement of the theorem is this: more languages can be decided with more time. While this is indeed an intuitive idea, it is not obvious how a formal version of this statement is to be proved. We will begin with a somewhat high-level discussion of how the theorem is proved, and then state the strongest-known form of the theorem (without going through the low-level details needed to obtain the stronger form).

We will restrict our attention to ordinary one-tape DTMs whose input alphabet is $\Sigma = \{0, 1\}$, and let us suppose that we have fixed an encoding scheme over Σ for DTMs having just this particular input alphabet. That is, for each DTM M having input alphabet Σ, the encoding $\langle M \rangle$ is a string over the alphabet Σ. (We're free to use an encoding scheme for all DTMs if we want, but we will regard encodings of DTMs having input alphabets differing from Σ as if they are invalid encodings.)

Now, suppose that a time-constructible function $f : \mathbb{N} \to \mathbb{N}$ has been selected, and define a DTM K as follows:

```
On input $w = \langle M \rangle 01^k$, where $M$ is a DTM with input alphabet $\Sigma$ and $k \in \mathbb{N}$:
1.  Compute $t = f(|w|)$.
2.  Simulate $M$ on input $w$ for $t$ steps.
3.  If $M$ has rejected during this simulation, then *accept*, otherwise *reject*.
```

It is not immediately clear what the running time is for K, because that depends on precisely how the simulation of M is done—different ways of performing the simulation could of course lead to different running times. For the time being, let us take $g : \mathbb{N} \to \mathbb{N}$ to be the running time of K, and we'll worry later about how specifically g relates to f.

Next, let us think about the language $\mathrm{L}(K)$ decided by K. It is obvious that $\mathrm{L}(K) \in \mathrm{DTIME}(g)$, because K itself is a DTM that decides $\mathrm{L}(K)$ in time $g(n)$. What we will show is that $\mathrm{L}(K)$ cannot possibly be decided by a DTM that runs in time $o(f(n))$.

Assume toward contradiction that there does exist a DTM M that decides $\mathrm{L}(K)$ in time $o(f(n))$. Because the running time of M is $o(f(n))$, we know that there must exist a natural number n_0 such that, for all $n \geq n_0$, the DTM M halts on all inputs of length n in strictly fewer than $f(n)$ steps. Choose k to be large enough so that the string $w = \langle M \rangle 01^k$ satisfies $|w| \geq n_0$, and (as always) let $n = |w|$. Because M halts on input w after fewer than $f(n)$ steps, we find that

$$w \in \mathrm{L}(K) \;\Leftrightarrow\; w \notin \mathrm{L}(M). \tag{18.10}$$

The reason is that K simulates M on input w, it completes the simulation because M runs for fewer than $f(n)$ step, and it answers *opposite* to the way M answers (i.e., if M accepts, then K rejects; and if M rejects, then K accepts). This contradicts the assumption that M decides $\mathrm{L}(K)$. We conclude that no DTM whose running time is $o(f(n))$ can decide $\mathrm{L}(K)$.

It is natural to wonder what the purpose is for taking the input to K to have the form $\langle M\rangle 01^k$, as opposed to just $\langle M\rangle$ (for instance). The reason is pretty simple: it's just a way of letting the length of the input string grow, so that the asymptotic behavior of the function f and the running time of M take over (even though we're really interested in fixed choices of M). If we were to change the language, so that the input takes the form $w = \langle M\rangle$ rather than $\langle M\rangle 01^k$, we would have no way to guarantee that K is capable of finishing the simulation of M on input $\langle M\rangle$ within $f(|\langle M\rangle|)$ steps—for it could be that the running time of M on input $\langle M\rangle$ exceeds $f(|\langle M\rangle|)$ steps, even though the running time of M is small compared with f for significantly longer input strings.

What we have proved is that for any choice of a time-constructible function $f : \mathbb{N} \rightarrow \mathbb{N}$, it holds that

$$\mathrm{DTIME}(h) \subsetneq \mathrm{DTIME}(g) \tag{18.11}$$

whenever $h(n) = o(f(n))$, where g is the running time of K (which depends somehow on f). If you work very hard to make K run as efficiently as possible, the following theorem can be obtained.

Theorem 18.6 (Time-hierarchy theorem). *If $f, g : \mathbb{N} \rightarrow \mathbb{N}$ are time-constructible functions for which it holds that $f(n) = o(g(n)/\log(g(n)))$, then*

$$\mathrm{DTIME}(f) \subsetneq \mathrm{DTIME}(g). \tag{18.12}$$

The main reason that we will not go through the details required to prove this theorem is that optimizing K to simulate a given DTM as efficiently as possible gets very technical. For the sake of this course, it is enough that you understand the basic idea of proving this theorem through the diagonalization technique (along similar lines to the proof that DIAG is not Turing-recognizable, and to the proof that $\mathcal{P}(\mathbb{N})$ is uncountable).

From the time-hierarchy theorem, one can conclude the following down-to-earth corollary.

Corollary 18.7. *For all $k \in \mathbb{N}$ with $k \geq 1$, it holds that*

$$\mathrm{DTIME}(n^k) \subsetneq \mathrm{DTIME}(n^{k+1}). \tag{18.13}$$

18.3 Polynomial and exponential time

We'll finish off the lecture by introducing a few important notions based on deterministic time complexity.

First, let us define two complexity classes, known as P and EXP, as follows:

$$\mathrm{P} = \bigcup_{k\geq 1} \mathrm{DTIME}(n^k) \quad \text{and} \quad \mathrm{EXP} = \bigcup_{k\geq 1} \mathrm{DTIME}\big(2^{n^k}\big). \tag{18.14}$$

In words, a language A is contained in the complexity class P if there exists a DTM M that decides A and has *polynomial running time*, meaning a running time that is $O(n^k)$ for some choice of $k \geq 1$; and a language A is contained in the complexity class EXP if there exists a DTM M that decides A and has *exponential running time*, meaning a running time that is $O\big(2^{n^k}\big)$ for some choice of $k \geq 1$.

As a very rough but nevertheless useful simplification, we often view the class P as representing languages that can be *efficiently* decided by a DTM, while EXP contains languages that are decidable by a *brute force* approach. These are undoubtedly over-simplifications in some respects, but for languages that correspond to "natural" computational problems that arise in practical settings, this is a reasonable picture to keep in mind. We'll have more to say about these (and other) complexity classes in the next couple of lectures.

By the time-hierarchy theorem, it holds that P $\subsetneq$ EXP. In particular, if we take $f(n) = 2^n$ and $g(n) = 2^{2n}$, then the time hierarchy theorem establishes the middle (proper) inclusion in this expression:

$$\mathrm{P} \subseteq \mathrm{DTIME}(2^n) \subsetneq \mathrm{DTIME}(2^{2n}) \subseteq \mathrm{EXP}. \tag{18.15}$$

A simple example of a language in the class P is the graph reachability problem from Lecture 14:

$$\left\{ \langle G, u, v\rangle \; : \; \begin{array}{l}\text{there exists a path from vertex } u \text{ to}\\ \text{vertex } v \text{ in the undirected graph } G\end{array} \right\}. \tag{18.16}$$

The DTM for this language that we discussed in that lecture decides the language in polynomial time (assuming we use any of the graph encoding schemes mentioned in Lecture 13, or something similar). Another example is this language:

$$\left\{ \langle M, w, 0^t\rangle \; : \; \begin{array}{l} M \text{ is a DTM, } w \text{ is a string, } t \in \mathbb{N},\\ \text{and } M \text{ accepts } w \text{ within } t \text{ steps}\end{array} \right\}. \tag{18.17}$$

This language is similar to one described in Lecture 15, except that the number of steps t for which the DTM M is to run is input in unary. (If t was given in binary, we would have a language in EXP but not in P.) Finally, it is interesting to note that every context-free language is in P, although we will not have time to discuss a method through which to prove this. There are of course many, many other interesting examples of languages contained in P—the ones just mentioned represent only a few examples.

Finally, let us observe that one may consider not only languages that are decided by DTMs having bounded running times, but also functions that can be computed by time-bounded DTMs. It will be enough for the purposes of the remaining lectures of this course to consider the class of *polynomial-time computable functions*, which are functions that can be computed by a DTM with running time $O(n^k)$ for some fixed positive integer k.

It will be convenient for the purposes of the next few lectures that we extend the notion of computable functions slightly by allowing for functions having different input and output alphabets. In particular, if Σ and Γ are (possibly different) alphabets, then we say that a function of the form

$$f : \Sigma^* \rightarrow \Gamma^* \tag{18.18}$$

is computable if there exists a DTM $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ whose yields relation satisfies

$$(q_0, \sqcup)\, w \vdash_M^* (q_{\text{acc}}, \sqcup)\, f(w) \tag{18.19}$$

for every string $w \in \Sigma^*$. The only change in the definition from the one in Lecture 16 is that we do not require that $f(w)$ is a string over the input alphabet of M. (Naturally, the tape alphabet Δ of M must include the symbols in Γ that appear in any string in the range of f in order for this relation to be satisfied.)

Lecture 19

NP, polynomial-time mapping reductions, and NP-completeness

In the previous lecture we discussed deterministic time complexity, along with the time-hierarchy theorem, and introduced two complexity classes: P and EXP. In this lecture we will introduce another complexity class, called NP, and study its relationship to P and EXP. In addition, we will define a polynomial-time variant of mapping reductions along with the very important notion of *completeness* for the class NP.

19.1 The complexity class NP

There are two, equivalent ways to define the complexity class NP that we will cover in this lecture. The first way is arguably more intuitive and more closely connected with the way in which the complexity class NP is typically viewed. The second way directly connects the class with the notion of nondeterminism, and leads to a more general notion (which we will not have time to explore further).

A few conventions about alphabets and encodings

Before getting to the definitions, it will be helpful for us to spend a couple of moments establishing some conventions that will be used throughout the lecture. First, let us agree that for this lecture (and Lectures 20 and 21 as well) we will always take $\Sigma = \{0, 1\}$ to be the binary alphabet, and if we wish to consider other alphabets, which may or may not be equal to Σ, we will use the names Γ and Δ.

Second, when we consider an encoding $\langle x, y \rangle$ of two strings x and y, we will make the assumption that the encoding $\langle x, y \rangle$ itself is always a string over the binary alphabet Σ, and we assume that this encoding scheme is *efficient*.[1] In particular, it is possible for a DTM that stores x and y on its tape to compute the encoding $\langle x, y \rangle$ in time polynomial in the combined length of x and y, which implies that the length of the encoding $\langle x, y \rangle$ is necessarily polynomial in the combined length of x and y. Furthermore, it must be possible for a DTM that stores the encoding $\langle x, y \rangle$ on its tape to extract each of the strings x and y from that encoding in polynomial time. Finally, assuming that the alphabets from which x and y are drawn have been fixed in advance, we will demand that the length $|\langle x, y \rangle|$ of the encoding depends only on the lengths $|x|$ and $|y|$ of the two strings, and not on the particular symbols that appear in x and y. This particular assumption regarding the length of an encoding $\langle x, y \rangle$ doesn't really have fundamental importance, but it is easily met and will be convenient in the lectures to follow.

As an aside, we may observe that the sort of encoding scheme described in Lecture 13, through which a pair of strings can be encoded as a single string, has all of the properties we require, provided we use fixed-length encodings to encode symbols in a given alphabet Γ as binary strings. In more detail, if it is the case that either of the strings x and y is not a string over the binary alphabet, we first use a fixed-length encoding scheme to encode that string as a binary string, whose length is therefore determined by the length of the string being encoded. Once this is done (possibly to both strings), then we have reduced our task to encoding a pair of binary strings (x, y) into a single binary string $\langle x, y \rangle$. One may first encode the pair (x, y) into the string $x\#y$ over the alphabet $\Sigma \cup \{\#\}$, and then individually encode the elements of $\Sigma \cup \{\#\}$ as length-two strings over Σ. The resulting scheme evidently satisfies all of our requirements.

NP as certificate verification

With the assumptions concerning alphabets and encoding schemes for pairs of strings described above in mind, we define the complexity class NP as follows.

Definition 19.1. Let Γ be an alphabet and let $A \subseteq \Gamma^*$ be a language. The language A is contained in NP if there exists a positive integer k, a time-constructible function $f(n) = O(n^k)$, and a language $B \in \mathrm{P}$ (over the binary alphabet Σ) such that

$$A = \big\{ x \in \Gamma^* \;:\; \text{there exists } y \in \Sigma^* \text{ such that } |y| = f(|x|) \text{ and } \langle x, y \rangle \in B \big\}. \tag{19.1}$$

[1] In general, when we discuss the computational complexity of deciding languages and computing functions that concern various mathematical objects (such as strings, numbers, matrices, graphs, DFAs, DTMs, and any number of other objects), we naturally assume that the encoding schemes used to describe these objects are efficient—polynomial-time computations should suffice to perform standard, low-level manipulations of the encodings.

The essential idea that this definition expresses is that $A \in \mathrm{NP}$ means that membership in A is *efficiently verifiable*. The string y in the definition plays the role of a *proof* that a string x is contained A, while the language B represents an efficient *verification procedure* that checks the validity of this proof of membership for x. The terms *certificate* and *witness* are alternatives (to the term *proof*) that are often used to describe the string y.

Example 19.2 (Graph 3-colorability)**.** A classic example of a language in NP is the *graph 3-colorability* language. We define that an undirected graph $G = (V, E)$ is *3-colorable* if there exists a function $c : V \rightarrow \{0, 1, 2\}$ from the vertices of G to a three element set $\{0, 1, 2\}$ (whose elements we view as representing three different colors) such that $c(u) \neq c(v)$ for every edge $\{u, v\} \in E$. Such a function is called a *3-coloring* of G. For some graphs there does indeed exists a 3-coloring, and for others there does not—and a natural computational problem associated with this notion is to determine whether or not a given graph has a 3-coloring.

Now, in order to help to clarify the example, let us choose a couple of concrete encoding schemes for graphs and colorings. We will choose the encoding scheme for graphs suggested in Lecture 13, in which we suppose that each graph G has a vertex set of the form $V = \{1, \ldots, m\}$ for some positive integer m, and the encoding $\langle G \rangle$ is a binary string of length m^2 that specifies the adjacency matrix of G. For a function of the form $c : \{1, \ldots, m\} \rightarrow \{0, 1, 2\}$, which might or might not be a valid 3-coloring of a graph G with vertex set $\{1, \ldots, m\}$, we may simply encode the colors $\{0, 1, 2\}$ as binary strings (as $0 \rightarrow 00$, $1 \rightarrow 01$, and $2 \rightarrow 10$, let us say) and then concatenate these encodings together in the order $c(1), c(2), \ldots, c(m)$ to obtain a binary string encoding $\langle c \rangle$ of length $2m$.

With respect to the encoding scheme for graphs just described, define a language as follows:

$$\mathrm{3COL} = \big\{ \langle G \rangle \,:\, G \text{ is a 3-colorable graph} \big\}. \tag{19.2}$$

The language 3COL is in NP. To see that this is so, we first define a language B as follows:

$$B = \left\{ \langle \langle G \rangle, y \rangle \,:\, \begin{array}{l} G \text{ is an undirected graph on } m \text{ vertices, and the} \\ \text{first } 2m \text{ bits of } y \text{ encode a valid 3-coloring } c \text{ of } G \end{array} \right\}. \tag{19.3}$$

Observe that the language B is contained in P. For a given input string w, we can first check that $w = \langle \langle G \rangle, y \rangle$ for G being a graph and y being a binary string whose first $2m$ bits (for m being the number of vertices of G) encode a function of the form $c : \{1, \ldots, m\} \rightarrow \{0, 1, 2\}$. We can then go through the edges of G one at a time and check that $c(u) \neq c(v)$ for each edge $\{u, v\}$. The entire process can be performed in polynomial time.

Finally, we observe that the requirements of Definition 19.1 are satisfied for the language 3COL in place of A. For the function f it suffices to take $f(n) = n^2 + 1$, which is unnecessarily large in most cases, but it is a polynomially bounded time constructible function, and we always have enough bits to encode the candidate colorings. For any graph G we have that $\langle G \rangle \in$ 3COL if and only if there exists a 3-coloring c of G, which is equivalent to the existence of a string y with $|y| = f(|\langle G \rangle|)$ such that $\langle \langle G \rangle, y \rangle \in B$; for such a string is contained in B if and only if the first $2m$ bits of y actually encode a valid 3-coloring of G.

Before we consider this example to be finished, we should observe that the particular encoding schemes we selected were not really critical—you could of course choose different schemes, so long as a polynomial-time conversion between the selected schemes and the ones described above is possible.

Remark 19.3. Graph 3-coloring is just one of thousands of interesting examples of languages in NP. It also happens to be an NP-complete language, the meaning of which will be made clear by the end of the lecture. A study of the natural computational problems represented by NP-complete languages is a part of CS 341, and so we will not venture in this direction in this course.

NP as polynomial-time nondeterministic computations

As was already suggested before, there is a second way to define the class NP that is equivalent to the one above. The way that this definition works is that we first define a complexity class NTIME(f), for every function $f : \mathbb{N} \rightarrow \mathbb{N}$, to be the class of all languages that are decided by a nondeterministic Turing machine running in time $O(f(n))$.

In more detail, to say that a nondeterministic Turing machine (or NTM) N runs in time $t(n)$ means that, for every $n \in \mathbb{N}$ and every input string w of length n, it is the case that *all* computation paths of N on input w lead to either acceptance or rejection after $t(n)$ or fewer steps. To say that such an NTM decides a language A means that, in addition to satisfying the condition relating to running time just mentioned, N has an accepting computation path on each input w if and only if $w \in A$.

Once we have defined NTIME(f) for every function $f : \mathbb{N} \rightarrow \mathbb{N}$, we define NP in a similar way to what we did for P:

$$\mathrm{NP} = \bigcup_{k \geq 1} \mathrm{NTIME}(n^k). \tag{19.4}$$

This definition is where NP gets its name: NP is short for *nondeterministic polynomial time*.

The equivalence of the two definitions is not too difficult to establish, but we won't go through it in detail. The basic ideas needed to prove the equivalence are (i) a nondeterministic Turing machine can first guess a polynomial-length binary string certificate and then verify it deterministically, and (ii) a polynomial-length binary string certificate can encode the nondeterministic moves of a polynomial-time nondeterministic Turing machine, and it could then be verified deterministically in polynomial time that this sequence of nondeterministic moves would lead the original NTM to acceptance.

Relationships among P, NP, and EXP

Let us now observe the following inclusions:

$$\mathrm{P} \subseteq \mathrm{NP} \subseteq \mathrm{EXP}. \tag{19.5}$$

The first of these inclusions, P $\subseteq$ NP, is straightforward. Suppose Γ is an alphabet and $A \subseteq \Gamma^*$ is a language, and assume that $A \in \mathrm{P}$. We may then define a language $B \subseteq \Sigma^*$ as follows:

$$B = \bigl\{ \langle x, y \rangle \,:\, x \in A,\; y \in \Sigma^* \bigr\}. \tag{19.6}$$

It is evident that $B \in \mathrm{P}$; if we have a DTM M_A that decides A in polynomial time, we can easily decide B in polynomial time by first decoding x from $\langle x, y\rangle$ (and completely ignoring y), and then running M_A on x. For $f(n) = n^2$, or any other polynomially bounded, time-constructible function, it holds that

$$A = \bigl\{ x \in \Gamma^* : \text{there exists } y \in \Sigma^* \text{ such that } |y| = f(|x|) \text{ and } \langle x, y\rangle \in B \bigr\}, \tag{19.7}$$

and therefore $A \in \mathrm{NP}$.

With respect to the second definition of NP, as the union of NTIME(n^k) over all $k \geq 1$, the inclusion P $\subseteq$ NP is perhaps obvious. Every DTM is equivalent to an NTM that happens never to allow multiple nondeterministic moves during its computation, so

$$\mathrm{DTIME}(f) \subseteq \mathrm{NTIME}(f) \tag{19.8}$$

holds for all functions $f : \mathbb{N} \rightarrow \mathbb{N}$, and therefore

$$\mathrm{DTIME}(n^k) \subseteq \mathrm{NTIME}(n^k) \tag{19.9}$$

for all $k \geq 1$, which implies P $\subseteq$ NP.

Now let us observe that NP $\subseteq$ EXP. Suppose Γ is an alphabet and $A \subseteq \Gamma^*$ is language such that $A \in \mathrm{NP}$. This implies that there exists a positive integer k, a time-constructible function f satisfying $f(n) = O(n^k)$, and a language $B \in \mathrm{P}$ such that

$$A = \bigl\{ x \in \Sigma^* \,:\, \text{there exists } y \in \Sigma^* \text{ such that } |y| = f(|x|) \text{ and } \langle x, y \rangle \in B \bigr\}. \tag{19.10}$$

Define a DTM M as follows:

```
On input $x \in \Gamma^*$:
1.  For every string $y \in \Sigma^*$ satisfying $|y| = f(|x|)$, do the following:
2.      Accept if it holds that $\langle x, y \rangle \in B$.
3.  Reject.
```

It is evident that M decides A, as it simply searches over the set of all strings $y \in \Sigma^*$ with $|y| = f(|x|)$ to find if there exists one such that $\langle x, y \rangle \in B$. It remains to consider the running time of M.

Let us first consider step 2, in which M tests whether $\langle x, y \rangle \in B$ for an input string $x \in \Gamma^*$ and a binary string y satisfying $|y| = f(|x|)$. This test takes a number of steps that is polynomial in $|x|$, and the reason why is as follows. First, we have $|y| = f(|x|)$ for $f(n) = O(n^k)$, and therefore the length of the string $|\langle x, y \rangle|$ is polynomial in $|x|$. Now, because $B \in \mathrm{P}$, we have that membership in B can be tested in polynomial time. Because the input in this case is $\langle x, y \rangle$, this means that the time required to test membership in B is polynomial in $|\langle x, y \rangle|$. However, because the composition of two polynomials is another polynomial, we have that the time required to test whether $\langle x, y \rangle \in B$ is polynomial in $|x|$.

Next, on any input of length n, the number of times $T(n)$ the for-loop is iterated is given by

$$T(n) = 2^{f(n)}. \tag{19.11}$$

Because we have $f(n) = O(n^k)$, it therefore holds that

$$T(n) = O\Bigl(2^{n^{k+1}}\Bigr). \tag{19.12}$$

Finally, there are some other manipulations that M must perform, such as computing $f(|x|)$, managing how the loop is to range over strings y with $|y| = f(|x|)$, and computing $\langle x, y \rangle$ for each choice of y. For each iteration of the loop, these manipulations can all be performed in time polynomial in $|x|$.

Putting everything together, the entire computation certainly runs in time

$$O\Bigl(2^{n^{k+2}}\Bigr), \tag{19.13}$$

which is a rather coarse upper bound that is nevertheless sufficient for our needs. We have established that M runs in exponential time, so $A \in \mathrm{EXP}$.

Now we know that P $\subseteq$ NP $\subseteq$ EXP, and we also know that P $\subsetneq$ EXP by the time hierarchy theorem. Of course this means that one (or both) of the following proper containments must hold: (i) P $\subsetneq$ NP, or (ii) NP $\subsetneq$ EXP. Neither one has yet been proved, and a correct proof of either one would be a major breakthrough in complexity theory. (Determining whether or not P $=$ NP is, in fact, widely viewed as being among the greatest mathematical challenges of our time.)

19.2 Polynomial-time reductions and NP-completeness

We discussed reductions in Lecture 16 and used them to prove that certain languages are undecidable or non-Turing-recognizable. *Polynomial-time reductions* are defined similarly, except that we add the condition that the reductions themselves must be given by polynomial-time computable functions.

Definition 19.4. Let Γ and Δ be alphabets and let $A \subseteq \Gamma^*$ and $B \subseteq \Delta^*$ be languages. It is said that *A polynomial-time reduces* to B if there exists a polynomial-time computable function $f : \Gamma^* \rightarrow \Delta^*$ such that

$$w \in A \Leftrightarrow f(w) \in B \tag{19.14}$$

for all $w \in \Gamma^*$. One writes

$$A \leq_m^p B \tag{19.15}$$

to indicate that A polynomial-time reduces to B, and any function f that establishes that this is so may be called a *polynomial-time reduction* from A to B.

Polynomial-time reductions of this form are sometimes called *polynomial-time mapping reductions* (and also *polynomial-time many-to-one reductions*) to differentiate them from other types of reductions that we will not consider—but we will stick with the term *polynomial-time reductions* for simplicity. They are also sometimes called *Karp reductions*, named after Richard Karp, one of the pioneers of the theory of NP-completeness.

With the definition of polynomial-time reductions in hand, we can now define NP-completeness.

Definition 19.5. Let Γ be an alphabet and let $B \subseteq \Gamma^*$ be a language.

1. It is said that B is NP-hard if, for every language $A \in$ NP, it holds that $A \leq_m^p B$.
2. It is said that B is NP-complete if B is NP-hard and $B \in$ NP.

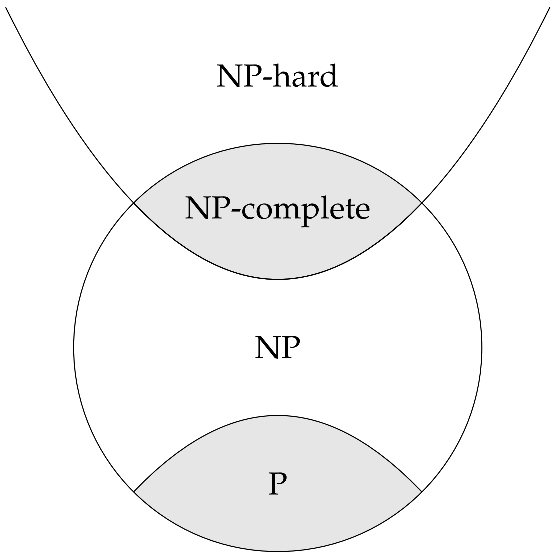


Figure 19.1: The relationship among the classes P and NP, and the NP-hard and NP-complete languages. The figure assumes P $\neq$ NP.

The idea behind this definition is that the NP-complete languages represent the hardest languages to decide in NP—*every* language in NP can be polynomial-time reduced to an NP-complete language, so if we view the difficulty of performing a polynomial-time reduction as being negligible, the ability to decide any one NP-complete language would give you a key to unlocking the computational difficulty of the class NP in its entirety. Figure 19.1 illustrates the relationship among the classes P and NP, and the NP-hard and NP-complete languages, under the assumption that P $\neq$ NP.

Now, it is not at all obvious from the definition that there should exist any NP-complete languages at all, for it is a strong condition that *every* language in NP must polynomial-time reduce to such a language. We will, however, prove that NP-complete languages do exist. There are, in fact, thousands of known NP-complete languages that correspond to natural computational problems of interest.

We will finish off the lecture by listing several properties of polynomial-time reductions, NP-completeness, and related concepts. Each of these facts has a simple proof, so if you are interested in some practice problems, try proving all of these facts. For all of these facts, it is to be assumed that A, B, and C are languages.

1. If $A \leq_m^p B$ and $B \leq_m^p C$, then $A \leq_m^p C$.
2. If $A \leq_m^p B$ and $B \in \text{P}$, then $A \in \text{P}$.
3. If $A \leq_m^p B$ and $B \in \text{NP}$, then $A \in \text{NP}$.

4. If A is NP-hard and $A \leq_m^p B$, then B is NP-hard.
5. If A is NP-complete, $B \in \text{NP}$, and $A \leq_m^p B$, then B is NP-complete.
6. If A is NP-hard and $A \in \text{P}$, then $\text{P} = \text{NP}$.

We typically use statement 5 when we wish to prove that a certain language B is NP-complete: we first prove that $B \in \text{NP}$ (which is often easy) and then look for a known NP-complete language A for which we can prove $A \leq_m^p B$. This type of problem is done in CS 341. Of course, to get things started, we need to find a "first" NP-complete language that can be reduced to others—effectively priming the NP-completeness pump. This is the content of the *Cook–Levin theorem*, which we will discuss in Lecture 21.

Lecture 20

Boolean circuits

In this lecture we will discuss the *Boolean circuit* model of computation and its connection to the Turing machine model. Although the Boolean circuit model is fundamentally important in theoretical computer science, we will not have time to explore it in depth—our study of Boolean circuits will mainly be aimed toward obtaining a "primordial" NP-complete language through which many other languages can be proved NP-complete.

20.1 Definitions

You probably already have something in mind when the term *Boolean circuit* is suggested—presumably a circuit consisting of AND, OR, and NOT gates connected by wires. Indeed this is essentially what the model represents, with the additional constraint that Boolean circuits must be *acyclic*. Formally speaking, we represent Boolean circuits with *directed acyclic graphs*, as the following definition suggests.

Definition 20.1. An n-input, m-output Boolean circuit C is a directed acyclic graph, along with various labels associated with its nodes, satisfying the following constraints:

1. The nodes of C are partitioned into three disjoint sets: the set of *input nodes*, the set of *constant nodes*, and the set of *gate nodes*.
2. The set of input nodes has precisely n members, labeled $X_1, \ldots, X_n$. Each input node and must have in-degree 0.
3. Each constant node must have in-degree 0, and must be labeled either 0 or 1.
4. Each gate node of C is labeled by an element of the set $\{\wedge, \vee, \neg\}$. Nodes labeled $\wedge$ are called AND gates and must have in-degree 2, nodes labeled $\vee$ are called

OR gates and must have in-degree 2, and nodes labeled $\neg$ are called NOT gates and must have in-degree 1.

5. Finally, m nodes of C are identified as output nodes, and are labeled $Y_1, \ldots, Y_m$. An output node can be an input node, a constant node, or a gate node.

When it is said that a directed graph is *acyclic*, it is meant that there is no path in the graph from a node to itself. Because this restriction is in place for Boolean circuits, we cannot have circuits with feedback loops, so it is not possible to implement latches, flip-flops, and so on. This is not a problem because our interest is in describing computations, not hardware. You can reasonably view each gate in a Boolean circuit as representing one computational step rather than one piece of hardware.

A Boolean circuit C having n inputs and m outputs computes a function of the form

$$f : \{0,1\}^n \rightarrow \{0,1\}^m \tag{20.1}$$

in the most natural way. In particular, for an input $(x_1, \ldots, x_n) \in \{0,1\}^n$ we first assign a Boolean value to each node as follows:

1. The value of each input node X_k is x_k, for $k = 1, \ldots, n$, and the value of each constant node is either 0 or 1, according to its label.
2. The value of each gate node is determined by the gate type and the value of the node or nodes from which there exist incoming edges. (For instance, if u is an AND gate node, and v_1 and v_2 are the nodes from which there exist edges to u, then the value of u is given by the AND of the values of v_1 and v_2. The situation is similar for OR and NOT gate nodes.)

Once the value of each of the nodes has been defined, we define the output of the function f by setting $f(x_1, \ldots, x_n) = (y_1, \ldots, y_m)$, where y_k is the value assigned to output node Y_k, for $k = 1, \ldots, m$. For example, Figure 20.1 illustrates a Boolean circuit that computes the exclusive-OR of two input Boolean values. (In this case there are no constant nodes.)

It is standard to use whatever name has been assigned to a Boolean circuit to refer also to the function it computes—so for an n-input Boolean circuit C, we might write $C(x_1, \ldots, x_n)$ rather than introducing a new name $f(x_1, \ldots, x_n)$ to refer to the function C computes. This is purely a shorthand convention used for convenience, and there is no harm in using distinct names for a Boolean circuit and the function it computes if there is a reason to do this.

The *size* of a Boolean circuit C is defined to be its number of nodes, and we write $\text{size}(C)$ to represent this number. The circuit illustrated in Figure 20.1, for instance, has size 7.

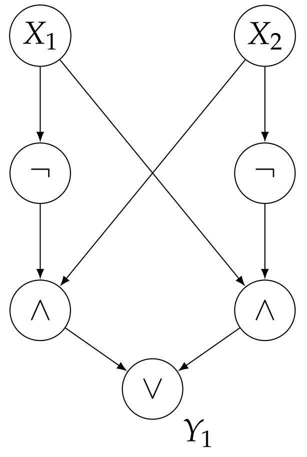


Figure 20.1: A Boolean circuit that computes the function $f(x_1, x_2) = x_1 \oplus x_2$ (where $\oplus$ denotes the exclusive-OR).

There are a variety of ways that one could choose to encode Boolean circuits as strings. By assigning a unique number to each node, and then listing each of the nodes one at a time (including their numbers, labels, and the numbers of the vertices from which incoming edges exist), it is not difficult to come up with an encoding scheme whereby each Boolean circuit C has an encoding $\langle C \rangle$ such that

$$|\langle C \rangle| = O(N \log(N)) \tag{20.2}$$

for $N = \text{size}(C)$.

20.2 Universality of Boolean circuits

We will now observe that Boolean circuits are universal, in the sense that *every* Boolean function of the form

$$f : \{0,1\}^n \rightarrow \{0,1\}^m, \tag{20.3}$$

for positive integers n and m, can computed by some Boolean circuit C.

Let us begin by observing that if we can build a Boolean circuit for every function of the form

$$g : \{0,1\}^n \rightarrow \{0,1\} \tag{20.4}$$

(i.e., the $m = 1$ case of the more general fact we're aiming for), then we can easily handle the general case. This is because every Boolean function of the form (20.3) can always be expressed as

$$f(x_1, \ldots, x_n) = (g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n)) \tag{20.5}$$

for some choice of functions $g_1, \ldots, g_m$, each of the form

$$g_k : \{0,1\}^n \rightarrow \{0,1\}; \tag{20.6}$$

and once we have a Boolean circuit C_k that computes g_k, for each $k = 1, \ldots, m$, we can combine these circuits in a straightforward way to get a circuit C computing f. In particular, with the exception of the input nodes $X_1, \ldots, X_n$, which are to be shared among the circuits $C_1, \ldots, C_m$, we allow the circuits $C_1, \ldots, C_m$ to operate independently, and label their output nodes appropriately.

It therefore remains to show that for any given function g of the form (20.4), there is a Boolean circuit C that computes g. This can actually be done using a rather simple *brute force* type of approach: we list all of the possible settings for the input variables that cause the function to evaluate to 1, represent each individual setting by an AND of input variables or their negations, and then take the OR of the results.

For example, consider the case that $n = 3$ and $g : \{0,1\}^3 \rightarrow \{0,1\}$ is a function such that

$$g(x_1, x_2, x_3) = \begin{cases} 1 & \text{if } x_1x_2x_3 \in \{001, 010, 111\} \\ 0 & \text{otherwise.} \end{cases} \tag{20.7}$$

The condition that $x_1x_2x_3 = 001$ is represented by the formula

$$(\neg X_1) \wedge (\neg X_2) \wedge X_3, \tag{20.8}$$

the condition that $x_1x_2x_3 = 010$ is represented by the formula

$$(\neg X_1) \wedge X_2 \wedge (\neg X_3), \tag{20.9}$$

and the condition that $x_1x_2x_3 = 111$ is represented by the formula

$$X_1 \wedge X_2 \wedge X_3. \tag{20.10}$$

The function g is therefore represented by the formula

$$\bigl((\neg X_1) \wedge (\neg X_2) \wedge X_3\bigr) \vee \bigl((\neg X_1) \wedge X_2 \wedge (\neg X_3)\bigr) \vee \bigl(X_1 \wedge X_2 \wedge X_3\bigr). \tag{20.11}$$

Once we have this formula, it is easy to come up with a Boolean circuit that computes the formula, which is the same as computing the function. (Because AND and OR gates are supposed to have just two inputs, we must combine two such gates to compute the AND or OR of three inputs, and of course this can be generalized to ANDs and ORs of more than three inputs. Aside from this, there is little work to do in turning a formula such as the one above into a Boolean circuit.)

There is one special case that we should take note of, which is the function that takes the value 0 on all inputs. There is nothing to take the OR of in this case, so it doesn't fit the pattern described above—but we can instead make use of a constant node whose value is 0 to implement a circuit whose output value is always 0. Alternatively, we could simply implement the formula

$$X_1 \wedge (\neg X_1) \tag{20.12}$$

as a Boolean circuit to deal with this function.

The type of Boolean formula of which (20.11) is an example, where we have an OR of some number of expressions, each of which is an AND of a collection of variables or their negations, is said to be a formula in *disjunctive normal form*. If we switch the roles of AND and OR, so that we have an AND of some number of expressions, each being an OR of a collection of variables or their negations, we have a formula in *conjunctive normal form*. What we've argued above is that every Boolean formula in variables $X_1, \ldots, X_n$ can be expressed in disjunctive normal form (or as a *DNF formula*, for short). One can also show that every Boolean formula can be expressed in conjunctive normal form (or as a *CNF formula*): just follow the steps above for $\neg g$ in place of g, negate the entire formula, and let De Morgan's laws do the work.

One thing you may notice about the method described above for obtaining a Boolean circuit that computes a given Boolean function is that it generally results in very large circuits. While there will sometimes exist a much smaller Boolean circuit for computing a given function, it is inevitable that some functions can only be computed by very large circuits. The reason is that the number of different functions of the form $g : \{0,1\}^n \rightarrow \{0,1\}$ is doubly exponential—there are precisely

$$2^{2^n} \tag{20.13}$$

functions of this form, as $g(x)$ can be 0 or 1 for each of the 2^n choices of $x \in \{0,1\}^n$. In contrast, the number of Boolean circuits of size N is merely exponential in N. As a consequence, one finds that some functions of the form $g : \{0,1\}^n \rightarrow \{0,1\}$ can only be computed by Boolean circuits having size exponential in n (and in fact this is true for most functions of the form $g : \{0,1\}^n \rightarrow \{0,1\}$).

20.3 Boolean circuits and Turing machines

The last thing we will do in this lecture is to connect Boolean circuits with Turing machines, in a couple of different ways.

Let us start with a simple observation concerning the following language:

$$\text{CVP} = \left\{ \langle C, x \rangle \;:\; \begin{array}{l} C \text{ is an } n\text{-input, 1-output Boolean} \\ \text{circuit, } x \in \{0,1\}^n, \text{ and } C(x) = 1 \end{array} \right\}. \tag{20.14}$$

The name CVP is short for *circuit value problem*. It is the case that this problem is contained in P; a DTM can decide CVP in polynomial time by evaluating the nodes of C and then accepting if and only if the unique output node evaluates to 1. In short, a DTM can easily simulate a Boolean circuit on a given input by simply evaluating the circuit.

The other direction—meaning the simulation of DTMs by Boolean circuits—is a bit more challenging. Moreover, it requires that we take a moment to clarify what is meant by the simulation of a DTM by a Boolean circuit. In particular, we must observe that there is a fundamental difference between Boolean circuits and DTMs, which is that DTMs are ready to handle input strings of any length, but Boolean circuits can only take inputs of a single, fixed length.[1] The most typical way to address this problem is to consider *families* of Boolean circuits, one for each possible input length, rather than individual Boolean circuits.

For example, suppose that we have a language $A \subseteq \{0,1\}^*$, and we wish to understand something about the *circuit complexity* of this language. A single Boolean circuit C cannot decide A, because C expects some fixed number of input bits, as opposed to a binary string having arbitrary length. We can, however, consider a family of circuits

$$\mathcal{F} = \{C_n \;:\; n \in \mathbb{N}\}, \tag{20.15}$$

where C_n is a Boolean circuit with n inputs and 1 output for each $n \in \mathbb{N}$. Such a family decides A if it is the case that

$$A = \big\{ x \in \{0,1\}^* \;:\; C_n(x) = 1 \text{ for } n = |x| \big\}. \tag{20.16}$$

In words, on an input string $x \in \{0,1\}^*$ of length n, we find the circuit C_n from $\mathcal{F}$ that expects an input of length n, feed x into this circuit, and interpret the outputs 0 and 1 as accept and reject, respectively.

One word of caution: *every* language $A \subseteq \{0,1\}^*$ is decided by some family of Boolean circuits in the sense just described, so you cannot conclude that a language is decidable from the existence of a family of circuits that decides it. We can, however, ask whether certain languages have families of Boolean circuits that obey various special constraints. The following theorem, which represents the main fact we're aiming for in this section, provides an example.

[1] Another difference is that DTMs can have arbitrary input alphabets, while Boolean circuits only work for Boolean values. This, however, is somewhat secondary, as we can always imagine that the strings of a language over an arbitrary alphabet have been encoded as binary strings, which will not change the properties of the language that are most relevant to complexity theory.

Theorem 20.2. *Let $f : \mathbb{N} \to \mathbb{N}$ be a time-constructible function, and let M be a DTM with input alphabet $\{0,1\}$ whose running time $t(n)$ satisfies $t(n) \leq f(n)$ for all $n \in \mathbb{N}$. There exists a family*

$$\mathcal{F} = \{C_n : n \in \mathbb{N}\} \tag{20.17}$$

of Boolean circuits, where each circuit C_n has n inputs and 1 output, such that the following two properties are satisfied:

1. *For every $n \in \mathbb{N}$ and every $x \in \{0,1\}^n$, M accepts x if and only if $C_n(x) = 1$.*
2. *It holds that* $\operatorname{size}(C_n) = O(f(n)^2)$.

Moreover, there exists a DTM K that runs in time polynomial in $f(n)$ that outputs an encoding $\langle C_n \rangle$ of the circuit C_n on input 0^n, for each $n \in \mathbb{N}$.

Remark 20.3. We will be primarily interested in the case in which $f(n) = O(n^c)$ for some fixed choice of an integer $c \geq 2$, and in this case the DTM K that outputs a description $\langle C_n \rangle$ of the circuit C_n on each input 0^n runs in polynomial time.

Proof. The proof is essentially a description of the circuit C_n for each choice of $n \in \mathbb{N}$. After the construction of C_n has been explained, it may be observed that $\operatorname{size}(C_n) = O(f(n)^2)$ and that a DTM could output a description of C_n as claimed in the statement of the theorem.

We will start with a few low-level details concerning encodings of the states and tape symbols of M. Assume that the state set of M is Q, and let $k = \lceil \log_2(|Q| + 1) \rceil$. This means that k bits is just large enough to encode each element of Q as a binary string of length k without using the string 0^k to encode a state—we're going to use this all-zero string in a few moments for a different purpose. Aside from the requirement that each state is encoded by a different string of length k, and that we haven't used the string 0^k to encode a state, the specific encoding used can be selected arbitrarily.

Let us also do something similar with the tape symbols of M. This time we'll use the all-zero string to encode the blank symbol $\sqcup$ rather than using it for a different purpose. That is, if the tape alphabet of M is Γ, then we will encode the tape symbols as binary strings of length $m = \lceil \log_2(|\Gamma|) \rceil$, using the string 0^m to encode the blank symbol $\sqcup$. The other symbols can be encoded arbitrarily, so long as each symbol is encoded by a different string of length m.

With the encodings of states and tape symbols just described in mind, we can imagine that any one single tape square of M at any moment in time could be represented by $m + k$ bits in the following manner:

$$\underbrace{\square\,\square\,\square\,\square\,\cdots\,\square\,\square\,\square}_{\substack{m \text{ bits to encode a} \\ \text{tape symbol}}}\underbrace{\square\,\square\,\square\,\square\,\cdots\,\square\,\square\,\square}_{\substack{k \text{ bits to encode a} \\ \text{state (if the tape} \\ \text{head is here)}}} \tag{20.18}$$

(In this picture, each square represents one bit.) For the k rightmost bits, we associate these meanings to the bit values:

1. The string 0^k means the tape head is not positioned over this particular tape square, and that nothing can be concluded regarding the state of the DTM.
2. Any string other than 0^k indicates that the tape head is scanning this square, and the state of the DTM is determined by these k bits with respect to our encoding scheme for states.

It is now evident why we did not want to use the string 0^k to encode a state, as this string is being used to indicate the absence of the tape head (and no information concerning the state).

Next, as we work our way toward a description of the circuit C_n, for any choice of $n \in \mathbb{N}$, we imagine an array of bits as suggested by Figure 20.2. To be more precise, the array contains $f(n) + 1$ rows, where each row represents a configuration of M at one step of its computation: the first row represents the initial configuration, the second row represents the configuration of M after one step, and so on. Within each row there are $2f(n) + 1$ blocks—each block represents one tape square, and taken together the blocks represent the tape squares numbered $-f(n), \ldots, f(n)$, relative to the starting location of the tape head (which we view as position 0). Each block consists of $m + k$ bits, and represents a tape symbol and possibly the presence of the tape head and state, as described above. We have chosen to include only blocks for the tape squares numbered $-f(n), \ldots, f(n)$, for the times $0, \ldots, f(n)$, because the computation of M on an input of length n is limited to this region of the tape: the computation lasts no more than $f(n)$ steps, and the tape head cannot possibly escape the region of the tape represented by the tape squares $-f(n), \ldots, f(n)$ within $f(n)$ steps.

What we wish to do next is to essentially hook up all of these bits to form the Boolean circuit C_n that will simulate M on a given input of length n for $f(n)$ steps. The key observation needed to do this is that all of the changes in the configurations represented by the rows of our array are "local." That is, if we want to know the contents of one of the blocks of $m + k$ bits at a particular time, we need only look at three blocks above that block; if the block in question represents tape square s at time t, the only blocks we need to examine in order to determine the correct

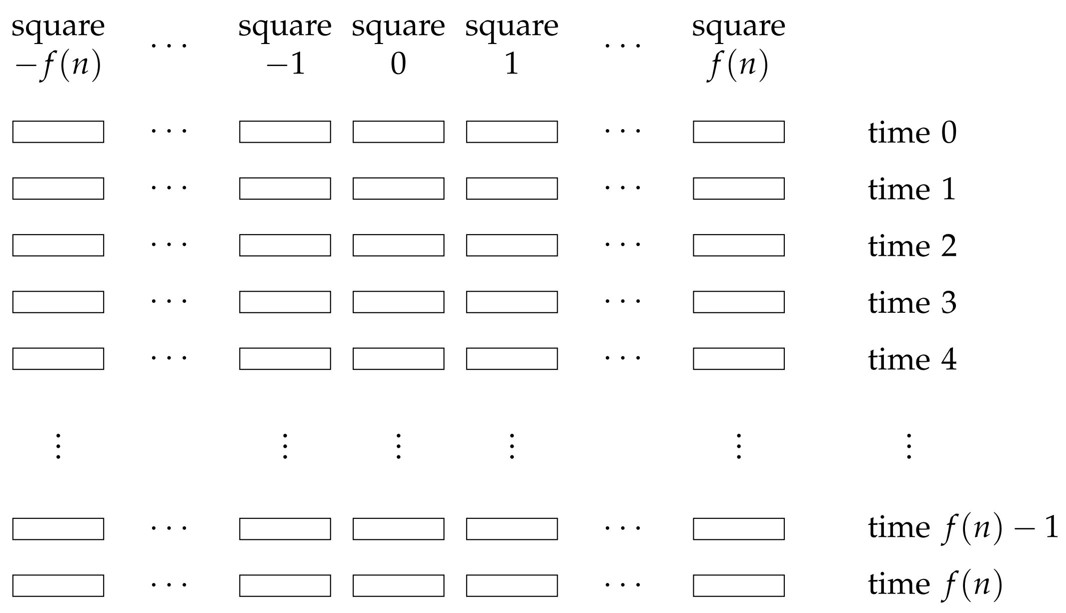


Figure 20.2: An array of bits representing configurations of the DTM M during the first $f(n)$ steps of its computation. Each rectangle correspond to one tape square at one time during the computation, and represents $m + k$ bits.

setting of the bits in this block are the ones corresponding to squares numbered $s - 1$, s, and $s + 1$, at time $t - 1$. This dependence can represented by some Boolean function of the form

$$g : \{0, 1\}^{3(m+k)} \rightarrow \{0, 1\}^{m+k}. \tag{20.19}$$

It is not necessary for us to describe precisely how this function g is defined—it is enough to observe that it is determined by the transition function of M. It may, however, be helpful to make some observations about this function in order to clarify the proof.

Each of the points that follow refer to the arrangement of blocks suggested by Figure 20.3, in which u, v, and w are binary strings of length $m + k$ and represent the tape squares $s - 1$, s, and $s + 1$, respectively (each at time $t - 1$), while $g(uvw)$ represents tape square s at time t.

1. There is only one tape head of M, so we don't need to worry about the case in which more than one of the strings u, v, and w indicates the presence of the tape head—the function g can be defined arbitrarily in any such case. Similarly, if u, v, or w includes an improper encoding of a tape symbol or state, the function g can be defined arbitrarily.

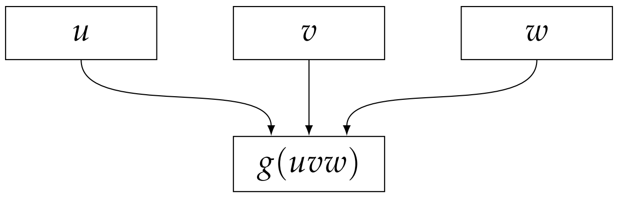


Figure 20.3: The correct values for the block of $m + k$ bits describing a tape square at a particular time depend only on the blocks of bits describing the same tape square and its adjacent tape squares one time step earlier. This dependence can be represented by some function $g : \{0,1\}^{3(m+k)} \rightarrow \{0,1\}^{m+k}$.

2. If none of u, v, or w indicates the presence of the tape head, it will be the case that $g(uvw) = v$. This reflects the fact that there's no action happening around the tape square s at this point in the computation.
3. If v indicates the presence of the tape head (and therefore specifies the state of M as well), then the symbol stored in tape square s might change, depending on the transition function of M. The first m bits of the output of g will need to be set appropriately, and it so happens that these first m output bits will be a function of v alone. Under the assumption that the state is a non-halting state, we also know that the last k bits of the output of g will be set to 0^k, as the tape head is forced to move left or right, off of square s.
4. If either u or w indicates the presence of the tape head (and therefore the state of M), then we might or might not end up with the tape head on square s, so the last k bits of the output of g will need to be set appropriately. The first m bits of the output of g will be in agreement with the first m bits of v in this case, as the symbol stored in square s does not change—the symbol can only change on the square the tape head is scanning.
5. We should be sure to set $g(uvw) = v$ in case v indicates the presence of the tape head and the state being either an accept or reject state. This accounts for the possibility that M finished its computation in fewer than $f(n)$ steps, allowing the correct answer to be propagated to the output of the circuit C_n we are constructing.

Now, because g is a Boolean function, we know that it must be implemented by some Boolean circuit (which we will call G). Because m and k are fixed—they only depend on the DTM M and not on the input length n of a string input to M—the size of G is also constant, independent of n.

Once we have in mind a description of the Boolean circuit G, we can envision a large Boolean circuit C_n in which the blocks of bits in the array described above are

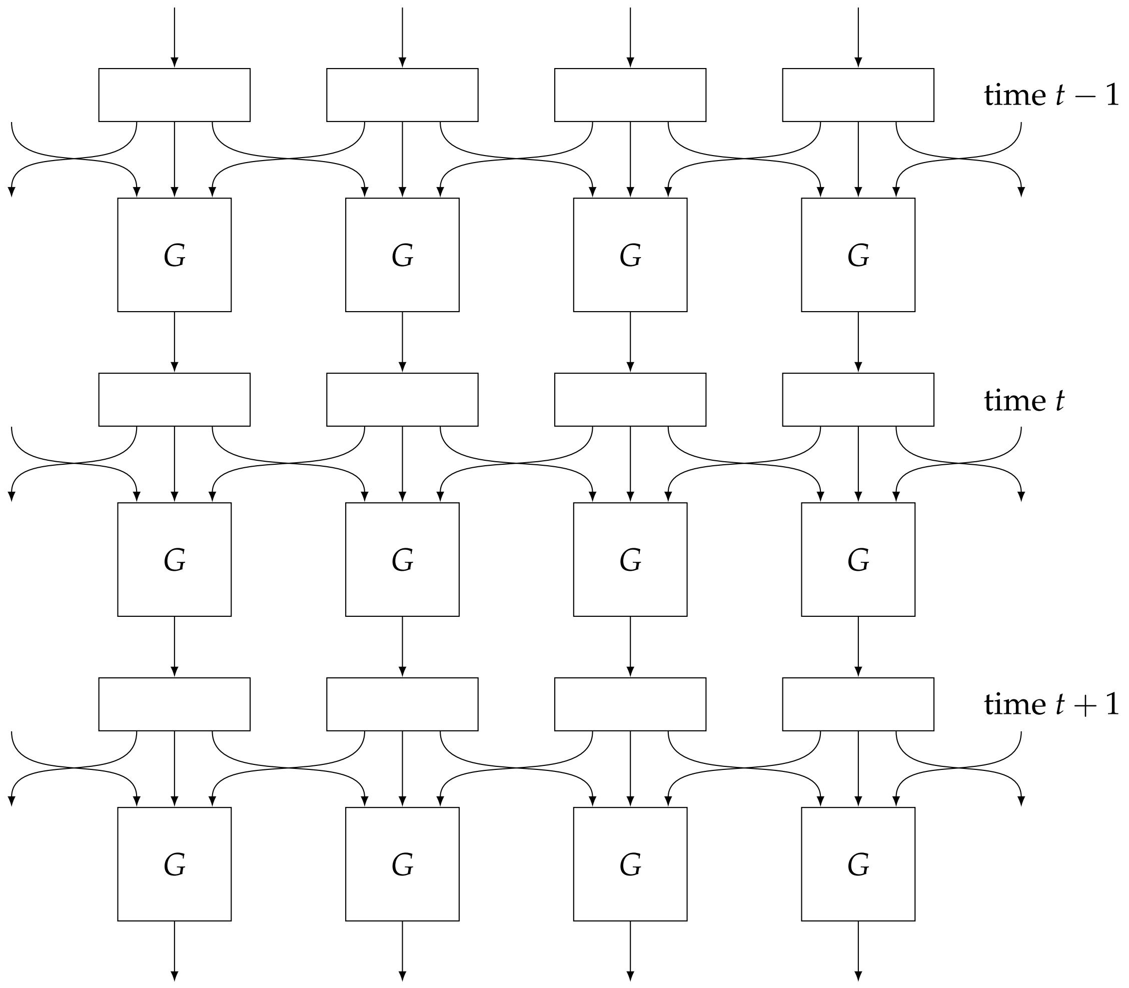


Figure 20.4: The pattern of connections among the blocks representing tape squares and the copies of the circuit G that computes the function g.

connected together, as suggested by Figure 20.4. In order to deal with the boundaries, for which hypothetical inputs correspond to blocks not included in our array, we introduce constant nodes giving the correct Boolean values (which are all-zero inputs given our encoding scheme choices).

It remains to create two additional parts of the circuit C_n: one part for an initialization stage at the beginning and one part that computes a single output bit corresponding to acceptance or rejection. The actual input string of length n upon which we wish to simulate M should correspond to n input bits to the circuit C_n. It is not difficult to make use of constant nodes and n copies of a constant-size Boolean circuit so that n input nodes $X_1, \ldots, X_n$ are connected to the blocks at time 0 numbered $1, \ldots, n$ in such a way that these blocks are initialized appropriately (so that the top row of blocks correctly represents the initial configuration of M on whatever input string is input to C_n). For all of the other blocks at time 0, constant nodes may be used for a correct initialization. Lastly, one may imagine another constant-size circuit that takes $m + k$ input bits and outputs 1 bit, which is to take the value 1 if and only if the $m + k$ input bits represent a block corresponding to a tape square in which the tape head is present and the state is the accept state. The output of the circuit C_n being constructed is then given by the OR of all of these output bits taken over all of the blocks at time $f(n)$. In other words, the output of C_n is 1 if and only if M is in an accept state at time $f(n)$ after being run on whatever input of length n has been input to C_n.

The size of the circuit C_n just described is evidently $O(f(n)^2)$. A description of this circuit can certainly be output by a DTM M in time polynomial in $f(n)$. Indeed, the structure of the circuit has a very simple, regular pattern, with the only dependence on M being the description of the constant-size circuits (including G and the circuit used to initialize the blocks in the top row). □

While there are some low-level details of the circuit C_n that have been left to the imagination in the proof above, hopefully the main idea is clear enough that you could (if you were forced to do it) come up with a more detailed description.

Lecture 21

The Cook–Levin theorem

The purpose of this lecture is to state and prove the Cook–Levin theorem, which gives us a primordial NP-complete language, from which many other languages can be proved NP-complete. There are thousands of interesting NP-complete languages known, and the proofs that they are NP-complete all rely on the Cook–Levin theorem. We won't go further down this path in this course, but this is done in CS 341; the process of proving languages (or their associated computational problems) to be NP-complete, based on reductions from other NP-complete problems, is covered in that course.

We're going to split the Cook–Levin theorem into two pieces, mainly for the sake of clarity and exposition. While the actual statement of the second piece is what people usually mean when they refer to "the Cook–Levin theorem," the proof of the first piece is an important part of it.

21.1 Circuit satisfiability

The first piece of the Cook–Levin theorem is concerned with the following language:

$$\text{CIRCUIT-SAT} = \left\{ \langle C \rangle \,:\, \begin{array}{l} C \text{ is an } m\text{-input, 1-output Boolean circuit, and} \\ \text{there exists } x \in \{0,1\}^m \text{ such that } C(x) = 1 \end{array} \right\}. \tag{21.1}$$

Here $\langle C \rangle$ is the encoding of a Boolean circuit C, which we may assume is over the binary alphabet $\Sigma = \{0,1\}$ for simplicity. As usual, it should be assumed that this is an efficient encoding scheme.

Let us first observe that CIRCUIT-SAT $\in$ NP, which turns out to be rather easy. Consider the following language:

$$B = \left\{ \langle \langle C \rangle, y \rangle \,:\, \begin{array}{l} C \text{ is an } m\text{-input, 1-output Boolean circuit,} \\ y \in \Sigma^*,\ |y| \geq m, \text{ and } C(y_1, \ldots, y_m) = 1 \end{array} \right\}. \tag{21.2}$$

The language B is contained in P—it is almost the same as the circuit value problem that we observed is in P in the previous lecture. If we set $f(n) = n^2$ for instance, which is a polynomially-bounded time constructible function, we see that

$$\text{CIRCUIT-SAT} = \left\{ x \in \Sigma^* : \begin{array}{l} \text{there exists } y \in \Sigma^* \text{ such that} \\ |y| = f(|x|) \text{ and } \langle x, y \rangle \in B \end{array} \right\}. \tag{21.3}$$

This establishes that CIRCUIT-SAT $\in$ NP.

The first piece of the Cook–Levin theorem establishes that CIRCUIT-SAT is an NP-complete language.

Theorem 21.1 (Cook–Levin theorem, part 1)**.** CIRCUIT-SAT *is NP-complete.*

Proof. We have already observed that CIRCUIT-SAT $\in$ NP, so it remains to prove that CIRCUIT-SAT is NP-hard. To this end, let $A \subseteq \Gamma^*$ be any language contained in NP. Our goal is to prove $A \leq_m^p$ CIRCUIT-SAT.

Because $A \in$ NP, there must exist a language $B \in$ P and a polynomially bounded, time-constructible function $f : \mathbb{N} \rightarrow \mathbb{N}$ such that

$$x \in A \Leftrightarrow \big(\exists y \in \Sigma^*\big)\big[|y| = f(|x|)\ \wedge \langle x, y \rangle \in B\big] \tag{21.4}$$

for every $x \in \Gamma^*$. Because B is decided by a polynomial-time DTM, we may invoke the theorem we proved at the end of the previous lecture to conclude[1] that for every $n \in \mathbb{N}$, there exists a Boolean circuit C_n as follows:

$$C_n(\langle x, y \rangle) = \begin{cases} 1 & \text{if } \langle x, y \rangle \in B \\ 0 & \text{if } \langle x, y \rangle \notin B, \end{cases} \tag{21.5}$$

for every choice of strings $x \in \Gamma^n$ and $y \in \Sigma^{f(n)}$. Moreover, it is possible to compute C_n from x in polynomial time. (In fact, C_n can be computed from 0^n, where $n = |x|$, in polynomial time, as one may conclude from the theorem.) Note that the number of input bits of C_n is not generally n, but rather is the length of the encoding $\langle x, y \rangle$ for x having length n and y having length $f(n)$.

Next, for each string $x \in \Gamma^*$, let us consider a very simple Boolean circuit D_x that computes the Boolean function

$$E_x(y) = \langle x, y \rangle \tag{21.6}$$

[1] Note that we are making use of the assumptions we stated a couple of lectures back on our choice of an encoding scheme, whereby two strings x and y are efficiently encoded as a binary string $\langle x, y \rangle$ whose length $|\langle x, y \rangle|$ only depends on the lengths of x and y, and not on the particular bits in x and y.

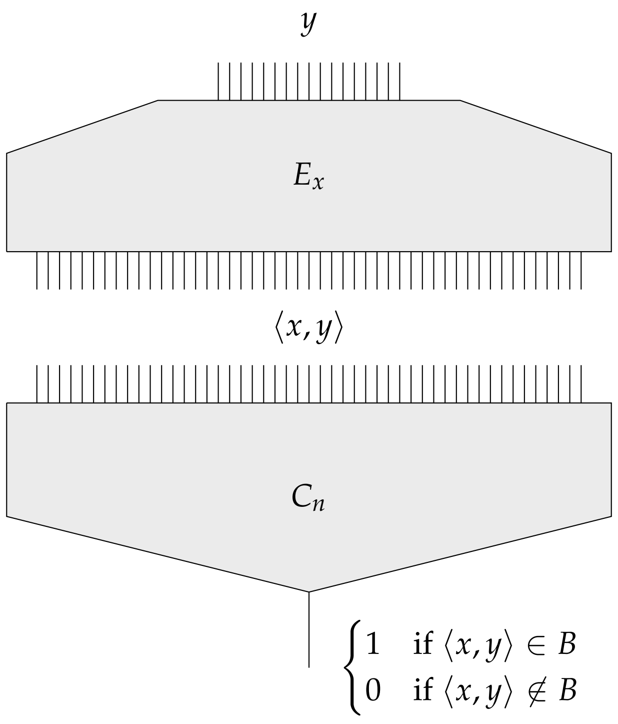

y
E_x
$\langle x, y\rangle$
C_n
$\begin{cases} 1 & \text{if } \langle x,y\rangle \in B \\ 0 & \text{if } \langle x,y\rangle \notin B \end{cases}$

Figure 21.1: The circuit D_x is obtained by composing E_x with C_n, for $n = |x|$. It holds that $D_x(y) = 1$ if and only if $\langle x, y\rangle \in B$.

for all strings $y \in \Sigma^{f(n)}$, where $n = |x|$. If x has length n, then this Boolean circuit takes a binary input string y of length $f(n)$, and effectively hard-codes the string x, obtaining the encoding $\langle x, y\rangle$. The function $g : \Gamma^* \to \Sigma^*$ defined as

$$g(x) = \langle E_x \rangle \tag{21.7}$$

is certainly a polynomial-time computable function.

Finally, for each $x \in \Gamma^*$, define a Boolean circuit D_x as suggested in Figure 21.1. That is, the circuit D_x is a composition of E_x with C_n, and it possesses the property that

$$D_x(y) = \begin{cases} 1 & \text{if } \langle x,y\rangle \in B \\ 0 & \text{if } \langle x,y\rangle \notin B. \end{cases} \tag{21.8}$$

The function $f : \Gamma^* \to \Sigma^*$ defined as

$$f(x) = \langle D_x \rangle \tag{21.9}$$

for all $x \in \Gamma^*$ is polynomial-time computable, by virtue of the fact that both C_n and E_x can be computed from x in polynomial time.

The function f is a polynomial-time reduction from A to CIRCUIT-SAT, as $x \in A$ if and only if there exists a string $y \in \Sigma^{f(n)}$ such that $\langle x, y \rangle \in B$, which is equivalent to the existence of a string $y \in \Sigma^{f(n)}$ such that $D_x(y) = 1$. As $A \subseteq \Gamma^*$ was an arbitrarily chosen language in NP, and it has been established that $A \leq_m^p$ CIRCUIT-SAT, the proof is complete. $\square$

21.2 Satisfiability of 3CNF formulas

The second part of the Cook–Levin theorem is concerned with *Boolean formulas* in place of Boolean circuits. Recall that a Boolean formula ϕ in Boolean variables $X_1, \ldots, X_n$ is said to be in *conjunctive normal form* (or CNF for short) if it is the case that

$$\phi = \psi_1 \wedge \psi_2 \wedge \cdots \wedge \psi_m \tag{21.10}$$

where each ψ_k is an OR of a collection of variables or their negations.

Hereafter, whenever we wish to refer to either a variable or the negation of a variable, we will call such a thing a *literal*. For any variable X_j, we often write $\overline{X_j}$ in place of $\neg X_j$, so that the set

$$\{X_1, \overline{X_1}, X_2, \overline{X_2}, \ldots, X_n, \overline{X_n}\} \tag{21.11}$$

represents the possible literals from which each ψ_k above may be formed. Each ψ_k, being an OR of a collection of literals, is called a *clause*. You can think of a clause in a CNF formula as being similar to a clause in a contract or legal agreement. That is, because the Boolean formula ϕ is an AND of the clauses $\psi_1, \ldots, \psi_m$, it must be that all of the clauses are true in order for ϕ to be true—just like all of the clauses in a legal document represent things that must be obeyed.

Here is an example of a CNF formula in the variables X_1, X_2, X_3, X_4:

$$\phi = (X_1 \vee X_2 \vee X_3 \vee \overline{X_4}) \wedge (\overline{X_2} \vee \overline{X_3}) \wedge (X_1 \vee X_3 \vee X_4) \wedge (\overline{X_1} \vee X_3). \tag{21.12}$$

There are four clauses in this formula:

$$\begin{aligned} \psi_1 &= (X_1 \vee X_2 \vee X_3 \vee \overline{X_4}), \\ \psi_2 &= (\overline{X_2} \vee \overline{X_3}), \\ \psi_3 &= (X_1 \vee X_3 \vee X_4), \\ \psi_4 &= (\overline{X_1} \vee X_3). \end{aligned} \tag{21.13}$$

No restrictions are placed on the number of literals in each clause of a CNF formula—you could even have a clause consisting of just one literal if you like. A formula is said to be a *3CNF formula* if it is the case that it is a CNF formula with exactly three literals in each clause. For example, the formula (21.12) is not a 3CNF formula, but the following formula is:

$$\psi = \bigl(X_2 \vee X_3 \vee \overline{X_4}\bigr) \wedge \bigl(X_1 \vee \overline{X_2} \vee \overline{X_3}\bigr) \wedge \bigl(X_1 \vee X_3 \vee X_4\bigr) \wedge \bigl(\overline{X_1} \vee X_2 \vee X_3\bigr). \tag{21.14}$$

A Boolean formula ϕ in the variables $X_1, \ldots, X_n$ is *satisfiable* if there exists a Boolean assignment for the variables that causes the formula to evaluate to 1. For example, both of the formulas ϕ and ψ given above are satisfiable: the assignment $X_1 = 1$, $X_2 = 0$, $X_3 = 1$, $X_4 = 0$ works for both of them.

Now let us define a new language:

$$\text{3SAT} = \bigl\{\langle\phi\rangle \,:\, \phi \text{ is a satisfiable 3CNF formula}\bigr\}. \tag{21.15}$$

The second part of the Cook–Levin theorem establishes that this language is NP-complete.

Theorem 21.2 (Cook–Levin theorem, part 2)**.** 3SAT *is NP-complete.*

Proof. The fact that 3SAT $\in$ NP is similar to CIRCUIT-SAT $\in$ NP, which we have already observed. It therefore remains to prove that 3SAT is NP-hard.

To this end we will prove that

$$\text{CIRCUIT-SAT} \leq_m^p \text{3SAT}. \tag{21.16}$$

In order to establish this reduction, we will devise a polynomial-time computable function f such that, for every Boolean circuit C with m inputs and 1 outputs, it holds that

$$f(\langle C\rangle) = \langle\phi_C\rangle \tag{21.17}$$

for some 3CNF formula ϕ_C with the property that ϕ_C is satisfiable if and only if C is satisfiable.

We cannot directly express a Boolean circuit directly as a 3CNF formula—it could be that the smallest 3CNF formula equivalent to a given circuit has an exponential number of clauses. So, instead of trying to represent a circuit directly as a 3CNF formula we will use a simple trick: for a given Boolean circuit C, we construct a 3CNF formula ϕ_C having a separate variable for every *node* of C (as opposed to just the nodes corresponding to inputs).

More specifically, suppose that C is a Boolean circuit, and let us define a 3CNF formula ϕ_C as follows. We introduce one Boolean variable for each node of C according to the following pattern:

1. The input nodes of C, labeled by its n inputs $X_1, \ldots, X_n$, will correspond to n variables of ϕ_C that are also denoted $X_1, \ldots, X_n$; no confusion will result from using the same names for the inputs of C and the variables of ϕ_C corresponding to these inputs.

2. The constant nodes of C will have associated variables $Y_1, \ldots, Y_k$.
3. The gate nodes of C will have associated variables $Z_1, \ldots, Z_m$.

We now include a small collection of clauses in ϕ_C for each node of C, with the exception of the nodes labeled $X_1, \ldots, X_n$. These clauses represent constraints that would necessarily be satisfied if the variable associated with each node were equal to that node's Boolean value. In addition, we will include one clause that ensures that the circuit output is 1. The way this is done depends on the type of each gate, in the following way:

1. *AND gates.* If Z is the variable corresponding to an AND gate, and the two input nodes for this gate have corresponding variables W_1 and W_2, then these four clauses (expressed as an AND of clauses) are included in ϕ_C:

   $$\left(W_1 \vee W_2 \vee \overline{Z}\right) \wedge \left(W_1 \vee \overline{W_2} \vee \overline{Z}\right) \wedge \left(\overline{W_1} \vee W_2 \vee \overline{Z}\right) \wedge \left(\overline{W_1} \vee \overline{W_2} \vee Z\right). \tag{21.18}$$

   Note that the Boolean formula corresponding to the AND of these four clauses evaluates to 1 if and only if $W_1 \wedge W_2 = Z$.

2. *OR gates.* If Z is the variable corresponding to an OR gate, and the two input nodes for this gate have corresponding variables W_1 and W_2, then these four clauses (again expressed as an AND of clauses) are included in ϕ_C:

   $$\left(W_1 \vee W_2 \vee \overline{Z}\right) \wedge \left(W_1 \vee \overline{W_2} \vee Z\right) \wedge \left(\overline{W_1} \vee W_2 \vee Z\right) \wedge \left(\overline{W_1} \vee \overline{W_2} \vee Z\right). \tag{21.19}$$

   Note that the Boolean formula corresponding to the AND of these four clauses evaluates to 1 if and only if $W_1 \vee W_2 = Z$.

3. *NOT gates.* If Z is the variable corresponding to a NOT gate, and the input node for this gate has corresponding variable W, then these two clauses[2] (again expressed as an AND of clauses) are included in ϕ_C:

   $$\left(W \vee W \vee Z\right) \wedge \left(\overline{W} \vee \overline{W} \vee \overline{Z}\right). \tag{21.20}$$

   Note that the Boolean formula corresponding to the AND of these two clauses evaluates to 1 if and only if $Z = \neg W$.

[2] The literals W and $\overline{W}$ have been repeated so that each clause contain 3 literals, as is required by a 3CNF formula. If for some reason we wish to disallow repeated literals, we can always introduce a dummy variable instead. For example, the clauses $(W \vee Z \vee D) \wedge (W \vee Z \vee \overline{D})$ can substitute $(W \vee W \vee Z)$. A similar replacement (using two dummy variables and four clauses) is possible for a clause of the form $(Z \vee Z \vee Z)$ where a single literal appears three times.

4. *Constant nodes.* If Z is the variable corresponding to a constant node that is set to 0, then this clause is included in ϕ_C:

$$\big(\overline{Z} \vee \overline{Z} \vee \overline{Z}\big). \tag{21.21}$$

   If Z is the variable corresponding to a constant node that is set to 1, then this clause is included in ϕ_C:

$$\big(Z \vee Z \vee Z\big). \tag{21.22}$$

   In each case, the clause evaluates to 1 if and only if the variable equals the constant value corresponding to the node.

5. *Output node.* If Z is the variable corresponding to the node that has been designated as the output node of C, then this clause is included in ϕ_C:

$$\big(Z \vee Z \vee Z\big). \tag{21.23}$$

   The clause evaluates to 1 if and only if the variable is set to 1.

The formula ϕ_C is the AND of all of the clauses described above. It is evident that $\langle \phi_C \rangle$ can be computed from $\langle C \rangle$ in polynomial time—it is a very simple computational procedure to examine the nodes of C, in an arbitrary order, and output the clauses described above.

Finally, we observe that ϕ_C is satisfiable if and only if C is satisfiable. The reason is that for any setting of the variables $X_1, \ldots, X_n$, there is a unique assignment to the remaining variables that satisfy all of the clauses introduced above, with the possible exception of the last clause corresponding to the output node of C. This is because the clauses constructed above force each variable to take a value that is consistent with an evaluation of each node of the circuit C. Consequently, if C is satisfiable, then a satisfying assignment to ϕ_C is obtained by first setting the input variables $X_1, \ldots, X_n$ of ϕ_C to any values that satisfy C, and then setting every other variable in a way that is consistent with an evaluation of the circuit. Conversely, if ϕ_C is satisfiable, then whatever values this assignment gives to the variables $X_1, \ldots, X_n$ must necessarily satisfy C; for an evaluation of C must be consistent with the setting of the variables that satisfied ϕ_C. $\square$