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

October 19, 2020

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 classify computational problems in terms of which models can solve them and which 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.
- Mathematical proofs.
- Certain biological processes.

The precise definition of what constitutes a computation can be debated (which is something we will not 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 low-level similarity: 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 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, such as 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 will be assumed throughout these notes that you are familiar with naive set theory and basic propositional logic.

Basic set theory

Naive set theory treats the concept of a set to be self-evident. 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:

$$S = \{T : T \not\in T\}.$$

Is it the case that S is an element of itself?

If $S \in S$, then by the condition that a set must satisfy to be included in S, it must be that $S \not\in S$. On the other hand, if $S \not\in S$, 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 $S \not\in S$, 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, something along the lines of axiomatic set theory would be needed.

On the other hand, axiomatic set theory is more complicated than naive set theory, and it is also outside of the scope of this course. Fortunately, there will be no specific situations that arise in this course for which the advantages of axiomatic set theory over naive set theory explicitly appear, and for this reason 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.

Countability

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*

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

and *rational numbers*

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

The sets of *real* and *complex numbers* are also infinite, but we will not define these sets here because they will not play a major role in this course 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 in these notes 0 is included among the natural numbers. It is not right or wrong to make such a choice, it is only a definition, and what is most important is that we make clear the precise meaning of the terms we use.

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 a 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. Notice that substituting f for the function g in statement 2 above makes that statement true, and likewise for statement 3 when f is substituted for the function h.

The function $f(n) = n$ is not the only function that works to establish that $\mathbb{N}$ is countable. For example, the function

$$f(n) = \begin{cases} n+1 & \text{if } n \text{ is even} \\ n-1 & \text{if } n \text{ is odd} \end{cases} \tag{1.5}$$

also works. The first few values of this function are

$$f(0) = 1, \quad f(1) = 0, \quad f(2) = 3, \quad f(3) = 2, \tag{1.6}$$

and it is not too hard to see that this function is both one-to-one and onto. There are (infinitely) many other choices of functions that work equally well to establish that $\mathbb{N}$ is countable.

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

$$f : \mathbb{N} \to \mathbb{Z}. \tag{1.7}$$

As in the previous example, there are many possible choices of f that work, one of which is this function:

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \frac{n+1}{2} & \text{if } n \text{ is odd} \\ -\frac{n}{2} & \text{if } n \text{ is even.} \end{cases} \tag{1.8}$$

Thus, we have

$$f(0) = 0, \quad f(1) = 1, \quad f(2) = -1, \quad f(3) = 2, \quad f(4) = -2, \tag{1.9}$$

and so on. This is a well-defined[2] function 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 evident from the pattern exhibited by (1.9).

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

First, imagine that we create a sequence of finite sequences (or lists)

$$(L_0, L_1, L_2, \ldots) \tag{1.10}$$

that starts like this:

$$L_0 = (0), \tag{1.11}$$

$$L_1 = (-1, 1), \tag{1.12}$$

$$L_2 = \left(-2, -\frac{1}{2}, \frac{1}{2}, 2\right), \tag{1.13}$$

$$L_3 = \left(-3, -\frac{3}{2}, -\frac{2}{3}, -\frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{3}{2}, 3\right), \tag{1.14}$$

$$L_4 = \left(-4, -\frac{4}{3}, -\frac{3}{4}, -\frac{1}{4}, \frac{1}{4}, \frac{3}{4}, \frac{4}{3}, 4\right), \tag{1.15}$$

$$L_5 = \left(-5, -\frac{5}{2}, -\frac{5}{3}, -\frac{5}{4}, -\frac{4}{5}, -\frac{3}{5}, -\frac{2}{5}, -\frac{1}{5}, \frac{1}{5}, \frac{2}{5}, \frac{3}{5}, \frac{4}{5}, \frac{5}{4}, \frac{5}{3}, \frac{5}{2}, 5\right). \tag{1.16}$$

In general, for any $n \geq 1$ we let L_n be the sorted list of all numbers that can be written as k/m for $k, m \in \{-n, \ldots, n\}$ satisfying $m \neq 0$, as well as not being included in any of the previous lists L_j, for $j < n$. The sequences get longer and longer, but for every natural number n it is surely the case that L_n is a *finite* sequence.

Now consider the single sequence S obtained by concatenating together the sequences L_0, L_1, L_2, etc. The beginning of S looks like this:

$$S = \left(0, -1, 1, -2, -\frac{1}{2}, \frac{1}{2}, 2, -3, -\frac{3}{2}, -\frac{2}{3}, -\frac{1}{3}, \frac{1}{3}, \frac{2}{3}, \frac{3}{2}, 3, -4, -\frac{4}{3}, -\frac{3}{4}, \ldots\right). \tag{1.17}$$

It is a well-defined sequence because each of the lists L_j is finite. (In contrast, it would *not* be clear what was meant by the concatenation of two or more *infinite* sequences.)

[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 or where they consider the definition.

Finally, let

$$f : \mathbb{N} \to \mathbb{Q} \tag{1.18}$$

be the function we obtain by setting $f(n)$ to be the number in position n of the sequence S, assuming that S begins with position 0. For example,

$$f(0) = 0, \quad f(1) = -1, \quad \text{and} \quad f(8) = -3/2. \tag{1.19}$$

Even though we did not write down an *explicit formula* for the function f, it is a well-defined function of the proper form (1.18).

Most importantly, f is an onto function—for any rational number you choose, you will eventually find that rational number in the list constructed above. It follows from the fact that f is onto that $\mathbb{Q}$ is countable.

The function f also happens to be one-to-one, but we do not need to know this to conclude that $\mathbb{Q}$ is countable.

An uncountable set

It is natural at this point to ask a question: Is every set countable? The answer is no, and we will soon see an example of an uncountable set. First, however, 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:

$$\mathcal{P}(A) = \{B : B \subseteq A\}. \tag{1.20}$$

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

$$\mathcal{P}(\{1,2,3\}) = \{\varnothing, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}. \tag{1.21}$$

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 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, the power set of the natural numbers $\mathcal{P}(\mathbb{N})$ is the set containing all subsets of $\mathbb{N}$.

The set $\mathcal{P}(\mathbb{N})$ is, 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, so that there exists an onto function of the form $f : \mathbb{N} \to \mathcal{P}(\mathbb{N})$. From this function we may define a subset of natural numbers as follows:

$$S = \{n \in \mathbb{N} : n \notin f(n)\}. \tag{1.22}$$

This definition makes sense because, for each $n \in \mathbb{N}$, $f(n)$ is an element of $\mathcal{P}(\mathbb{N})$, which is equivalent to $f(n)$ being a subset of $\mathbb{N}$.

Now, the set S is a subset of $\mathbb{N}$, or equivalently, $S \in \mathcal{P}(\mathbb{N})$. We have assumed that f is onto, and therefore there must 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.

We will now consider whether or not the number m is contained in the set S. The statement that $m \in S$ is equivalent to the statement that $m \in f(m)$ by the requirement that $S = f(m)$. The statement that $m \in f(m)$ is, however, equivalent to the statement that $m \notin S$ by the definition of the set S. That is to say, $m \in S$ if and only if $m \notin S$, which is a contradiction.

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

There is a technique at work in this proof, known as *diagonalization*. It is a fundamentally important technique in the theory of computation, and we will see instances of it later.

Using this technique, one can also prove that the sets $\mathbb{R}$ and $\mathbb{C}$ of real and complex numbers are uncountable. The central idea is the same as the proof above, but there is a small inconvenience caused by the fact that the natural approach of associating real numbers with infinite sequences of digits, as in the decimal or binary representation we are familiar with, does not define a one-to-one and onto function—there will always be real numbers having multiple representations. If you are interested, try to make it work!

1.3 Alphabets, strings, and languages

The last thing we will do for this lecture is to introduce some basic terminology that will be used throughout the course.

Alphabets

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, encoding information, 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*, *encoding*, or *calculating* 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 used for alphabets in this course are capital Greek letters such as Σ, Γ, and Δ. We refer to elements of alphabets as *symbols*, and we will often use lower-case letters appearing at the beginning of the Roman alphabet, such as a, b, c, and d, as variable names when referring to symbols.

Our favorite alphabet in these notes will be the *binary alphabet* $\Sigma = \{0, 1\}$.

Sometimes we will refer to the *unary alphabet* $\Sigma = \{0\}$ that has just one symbol. Although it is not a very efficient choice for encoding information, the unary alphabet is a valid alphabet, and we will find good uses for it.

We can also think about alphabets more abstractly. For instance, we may consider the alphabet

$$\Sigma = \{0, 1, \ldots, n-1\}, \tag{1.23}$$

where n is a large positive integer, like $n = 1{,}000{,}000$, or it may even be the case that n is a hypothetical positive integer that has not been explicitly chosen. Of course we do not need to dream up different symbols in order to contemplate such an alphabet in a mathematical sense.

Alphabets can also contain other symbols, such as

$$\Sigma = \{A, B, C, \ldots, Z\}, \quad \Sigma = \{\heartsuit, \diamondsuit, \spadesuit, \clubsuit\}, \quad \text{or} \quad \Sigma = \{\text{ZoSo}, ⟁, ⚭, ⦶\}, \tag{1.24}$$

but from the viewpoint of this course the actual symbols that appear in alphabets will not really matter all that much. From a mathematical point of view, there is nothing special about the alphabets $\{\heartsuit, \diamondsuit, \spadesuit, \clubsuit\}$ and $\{\text{ZoSo}, ⟁, ⚭, ⦶\}$ that distinguishes them from the alphabet $\{0, 1, 2, 3\}$. For this reason, when it is convenient to do so, we may assume without loss of generality that a given alphabet we are working with takes the form (1.23) for some positive integer n.

On the other hand, when it is convenient for us to choose symbols other than those suggested above, meaning 0, 1, 2, etc., we will not hesitate to do that. Sometimes it is very convenient to pick different symbols for alphabets, such as in a construction (or formal description) of a machine or algorithm that performs a complicated task, as we will see. A simple example is that we often choose the symbol # to suggest a separator between strings not containing this symbol.

Strings

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

Definition 1.8. A *string* over an alphabet Σ is a finite sequence of symbols drawn from Σ. The *length* of a string is the total number of symbols it contains, counting repetitions.

For example, 11010 is a string of length 5 over the binary alphabet $\Sigma = \{0,1\}$. It is also a string over the alphabet $\Gamma = \{0,1,2\}$; it just does not happen to include the symbol 2.

On the other hand,

$$0101010101\cdots \quad \text{(repeating forever)} \tag{1.25}$$

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, but in this course we will not refer to such things as *strings*.

There is a special string, called the *empty string*, that has length zero, meaning that it is an empty sequence with no symbols in it. We will denote this string by ε. (You may find that other writers choose a different symbol, such as λ, to represent the empty string.)

We will typically use lower-case letters appearing near the end of the Roman 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 are not thinking about u, v, w, x, y, and z as being single symbols from the Roman alphabet in this context. Because we are 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 $|w|$.

Languages

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

Definition 1.9. A *language* over an alphabet Σ is any set of strings, with each one 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 includes 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 letters near the beginning of the Roman alphabet, such as A, B, C, and D, to refer to languages. Sometimes we will also give special languages special names, such as PAL and DIAG, as you will see later.

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

$$A = \bigl\{0010, 110110, 011000010110, 111110000110100010010\bigr\}, \tag{1.26}$$
$$B = \bigl\{x \in \Sigma^* \,:\, x \text{ starts with 0 and ends with 1}\bigr\}, \tag{1.27}$$
$$C = \bigl\{x \in \Sigma^* \,:\, x \text{ is a binary representation of a prime number}\bigr\}, \tag{1.28}$$
$$D = \bigl\{x \in \Sigma^* \,:\, |x| \text{ and } |x| + 2 \text{ are prime numbers}\bigr\}. \tag{1.29}$$

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; deterministic finite automata

The main goal of this lecture is to introduce our first model of computation, 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 the notion of countability in the context of 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 are proud of, and *propositions* are also important facts, but we are embarrassed to call them theorems because they are so easy to prove. *Corollaries* are facts that follow easily from theorems, and *lemmas* (or *lemmata* for Latin purists) are boring technical facts that nobody cares about except for the fact that they are useful for proving more interesting 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 this ordering we can define a function f of the form (2.1) simply 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 calculating $f(n)$ is to write $n + 1$ in binary notation and then throw away the leading 1.

It is not hard to see that the function f 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, which is to say that the lexicographic ordering provides us with a one-to-one and onto correspondence between $\mathbb{N}$ and Σ^*.

It is easy to generalize this argument to any other alphabet. The 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 $\Gamma = \{\heartsuit, \diamondsuit, \spadesuit, \clubsuit\}$, it might not be clear how to order the symbols, and people might disagree on what ordering is best. But it does not matter to us so long as long as we pick a single ordering and remain consistent with it. Once we have 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 use this term to mean something different: dictionary ordering *without* first ordering strings according to length. They then use the term *quasi-lexicographic order* to refer to what we have called lexicographic order. There is no point in worrying too much about such discrepancies; there are many cases in science and mathematics where people disagree on terminology. What is important is that everyone is clear about what the terminology means when it is being used. With that in mind, 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. (I will leave it to you to prove this yourself. It is a good practice problem to gain familiarity with the concept of countability.)

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 $\mathcal{P}(\Sigma^*)$. The following 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 alphabet Σ, that every language $A \subseteq \Sigma^*$ is countable. It is natural to ask 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 do not need to repeat the same sort of diagonalization argument used to prove that $\mathcal{P}(\mathbb{N})$ is uncountable. Instead, 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 earlier from the lexicographic ordering of Σ^*. We can use this function f to define a function

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

as follows: for every $A \subseteq \mathbb{N}$, we define

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

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 simple one, called the *deterministic finite automata* model. Deterministic finite automata are also known as *finite state machines*.

Remark 2.4. Computer science students at the University of Waterloo have already encountered finite automata in a previous course (CS 241 *Foundations of Sequential Programs*). Regardless of one's prior exposure of the topic, however, it is natural to begin with precise definitions—we need them to proceed mathematically.

Please keep in mind the following two points as you consider the definition of the deterministic finite automata model:

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 a 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 are not trying to model general purpose computers at this stage, and the concept of finite automata is an extremely useful one.

Definition 2.5. A *deterministic finite automaton* (or *DFA*, for short) is a 5-tuple

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

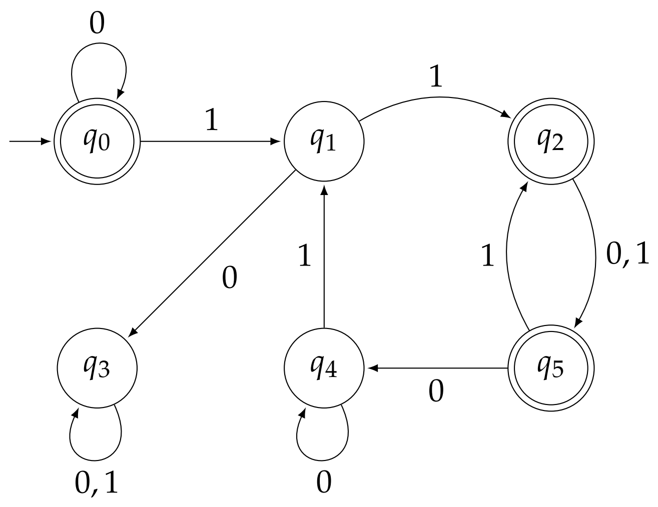


Figure 2.1: The state diagram of a DFA.

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

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

$q_0 \in Q$ 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 that appears in Figure 2.1. 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, their labels, 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 state diagram in Figure 2.1, for example, the state set is

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

the alphabet is $\Sigma = \{0, 1\}$, the start state is q_0, the set of accepts states is

$$F = \{q_0, q_2, q_5\}, \tag{2.13}$$

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

$$\begin{array}{lll} \delta(q_0, 0) = q_0, & \delta(q_1, 0) = q_3, & \delta(q_2, 0) = q_5, \\ \delta(q_0, 1) = q_1, & \delta(q_1, 1) = q_2, & \delta(q_2, 1) = q_5, \\ \delta(q_3, 0) = q_3, & \delta(q_4, 0) = q_4, & \delta(q_5, 0) = q_4, \\ \delta(q_3, 1) = q_3, & \delta(q_4, 1) = q_1, & \delta(q_5, 1) = q_2. \end{array} \tag{2.14}$$

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

Note, by the way, that when a single arrow is labeled by multiple symbols, such as in the case of the arrows labeled "0, 1" in Figure 2.1, it should be interpreted that there are actually multiple arrows, each labeled by a single symbol. This is just a way of making our diagrams a bit less cluttered by reusing the same arrow to express multiple transitions.

You can also go the other way and draw a state diagram from a formal description of a 5-tuple $(Q, \Sigma, \delta, q_0, F)$.

DFA computations

It is easy enough to say in words what it means for a DFA to *accept* or *reject* a given input string, 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 the input string (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 2.6. 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. $w = a_1 \cdots a_n$ for a positive integer n and symbols $a_1, \ldots, a_n \in \Sigma$, and there exist states $r_0, \ldots, r_n \in Q$ such that $r_0 = q_0$, $r_n \in F$, and $r_{k+1} = \delta(r_k, a_{k+1})$ 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 dictates: if we are in the state q and read the symbol a, the new state becomes $p = \delta(q, a)$. 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

$$\delta^* : Q \times \Sigma^* \to Q \tag{2.15}$$

recursively, based on a given transition function $\delta : Q \times \Sigma \to Q$, as follows:

1. $\delta^*(q, \varepsilon) = q$ for every $q \in Q$, and
2. $\delta^*(q, aw) = \delta^*(\delta(q, a), w)$ for all $q \in Q$, $a \in \Sigma$, 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. $w = a_1 \cdots a_n$ for a positive integer n and symbols $a_1, \ldots, a_n \in \Sigma$, and there exist states $r_0, \ldots, r_n \in Q$ such that $r_0 = q$, $r_n = p$, and $r_{k+1} = \delta(r_k, a_{k+1})$ 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 2.7. By now it is evident that we will not formally prove every statement we make in this course. If we did, we would not have sufficient time to cover all of the course material, 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 little material about computation in a one-term course. Moreover, our proofs would most likely be incomprehensible, and would quite possibly contain as many errors as you would expect to find in a complicated and untested program written in assembly language.

Naturally we will not take this path, but from time to time we will discuss the nature of proofs, how we would formally 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. If you are unsure at this point what actually constitutes a proof, or how much detail and formality you should aim for in your own proofs, do not worry—it is one of the aims of this course to assist in sorting this out.

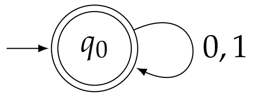
Languages recognized by DFAs and regular languages

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

$$\mathrm{L}(M) = \{w \in \Sigma^* \,:\, M \text{ accepts } w\}. \tag{2.16}$$

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 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:

$$\mathrm{L}(M) = \Sigma^*. \tag{2.17}$$

[2] Some refer to $\mathrm{L}(M)$ as the *language accepted by M*. This terminology does have the potential to cause confusion, though, as it overloads the term *accept*.

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

$$A = \{w \in \Sigma^* : |w| \text{ is a prime number}\}, \tag{2.18}$$

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 important terminology.

Definition 2.8. 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 $A = \mathrm{L}(M)$.

We have not seen many DFAs thus far, so we do not have many examples of regular languages to mention at this point, but we will see plenty of them soon enough, and throughout the course.

Let us finish off the lecture with a question: For a given alphabet Σ, is the set of all regular languages over the alphabet Σ countable or uncountable?

The answer is that this is a countable set. 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 the definition of what it means for a language to be regular, an onto function.

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 choose for different states of a DFA have 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, sometimes we do not even bother assigning names to states when drawing state diagrams of DFAs, because the state names are irrelevant to the way DFAs operates.

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.) So, we can form a finite sequence L_1 of all of the DFAs having just one state. Now consider the set of all DFAs with two states: $Q = \{q_0, q_1\}$. Again, there are only finitely many, so we may take L_2 to be any finite sequence of these DFAs—the ordering does not matter, it can be chosen arbitrarily. 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. The number of DFAs with m states happens to grow exponentially with m, but this is not important at this moment, we just need to known that the number is finite. Assuming that some way to order each of these finite lists of DFAs as been chosen, we can then concatenate the lists together starting, beginning with the 1 state DFAs, then the 2 state DFAs, and so on. We obtain a single infinite sequence containing every DFA having alphabet Σ. From such a list you can obtain an onto function from $\mathbb{N}$ to the set of all DFAs having alphabet 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, and does not give us a specific language that is not regular—it just tells us that there is one. We will 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 a critically 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 the possible outcomes that these choices can have, usually focusing on whether or not there *exists* a sequence of choices that leads to a particular outcome (such as acceptance for a finite automaton).

This may sound like a fantasy mode of computation not likely to be relevant from a practical viewpoint, because real computers do not make nondeterministic choices: each step a real computer makes is uniquely determined by its configuration at any given moment. Our interest in nondeterminism does not suggest otherwise. 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 $\delta(q,a)$ was a *state*, for any choice of a state $q \in Q$ and a symbol $a \in \Sigma$, representing the next state that the DFA would move to if it was in the state q and read the symbol a. For an NFA, each $\delta(q,a)$ is not a state, but rather a *subset of states*, which is equivalent to $\delta(q,a)$ 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 a. 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 $\delta(q,a) = \varnothing$.

We also have that the transition function of an NFA is not only defined for every pair $(q,a) \in Q \times \Sigma$, 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 a, there may be multiple arrows leading out of the circle representing the state q labeled by a, which tells us which states are contained in $\delta(q,a)$, or there may be no arrows like this when $\delta(q,a) = \varnothing$. 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 figure, we see that $Q = \{q_0, q_1, q_2, q_3\}$, q_0 is the start state, and $F = \{q_1\}$, just like we would have if this diagram represented a DFA. It is reasonable to guess from the diagram that the alphabet for the NFA it describes is $\Sigma = \{0,1\}$, although all we can be sure of is that Σ includes the symbols 0 and 1; it could be, for instance, that $\Sigma = \{0,1,2\}$, but it so happens that $\delta(q,2) = \varnothing$ for every $q \in Q$. Let us agree, however, that unless we explicitly indicate otherwise, the alphabet for an NFA described by a state diagram includes precisely those symbols (not including ε of course) that label transitions in the diagram, so that $\Sigma = \{0,1\}$ for this particular example. The transition function, which must take the form

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

is given by

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

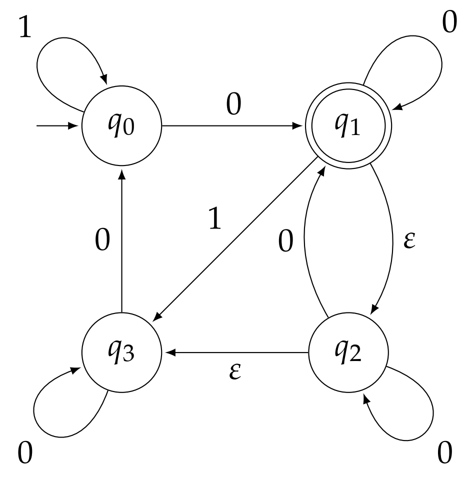


Figure 3.1: The state diagram of an NFA.

NFA computations

Next let us consider the definition of acceptance and rejection for NFAs. This time we will 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 $a_1, \ldots, a_m \in \Sigma \cup \{\varepsilon\}$ such that the following statements all hold:

1. $r_0 = q_0$.
2. $r_m \in F$.
3. $w = a_1 \cdots a_m$.
4. $r_{k+1} \in \delta(r_k, a_{k+1})$ for every $k \in \{0, \ldots, m-1\}$.

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

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 a from the input, provided that $p \in \delta(q, a)$; 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). 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 $w = a_1 \cdots a_m$. The condition $r_{k+1} \in \delta(r_k, a_{k+1})$ 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 $m = 0$. 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 or empty strings $a_1, \ldots, a_0 \in \Sigma \cup \{\varepsilon\}$, 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 $a_1, \ldots, a_0 \in \Sigma \cup \{\varepsilon\}$, on the other hand, looks like it does not make any sense—but it actually does make sense if you interpret it as an *empty* sequence having no elements in it. The condition $w = a_1 \cdots a_0$ in this case, which refers to a concatenation of an empty sequence of symbols or empty strings, is that it means $w = \varepsilon$.[1] Asking that the condition $r_{k+1} \in \delta(r_k, a_{k+1})$ should hold for every $k \in \{0, \ldots, m-1\}$ when $m = 0$ is a vacuous statement, and is 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.

Note 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 did not need to take the two possibilities separately. Alternatively, we could have added a special case to Definition 3.2, but it would make the definition longer, and the convention described above is good to know about anyway.

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

[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.

transition of an NFA, we define a new function

$$\delta^* : Q \times \Sigma^* \to \mathcal{P}(Q) \tag{3.5}$$

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

$$\varepsilon(R) = \left\{ q \in Q \,:\, \begin{array}{l} q \text{ is reachable from some } r \in R \text{ by following} \\ \text{zero or more } \varepsilon\text{-transitions} \end{array} \right\}. \tag{3.6}$$

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. $\delta^*(q, aw) = \bigcup_{p \in \varepsilon(\{q\})} \bigcup_{r \in \delta(p,a)} \delta^*(r, w)$ for every $q \in Q$, $a \in \Sigma$, 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:

$$\mathrm{L}(N) = \{ w \in \Sigma^* \,:\, N \text{ accepts } w \}. \tag{3.7}$$

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. 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 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 be wise to choose the first: it is 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, as 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 is not *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

$$\mu(q, a) = \{\delta(q, a)\} \quad \text{and} \quad \mu(q, \varepsilon) = \varnothing \tag{3.8}$$

for all $q \in Q$ and $a \in \Sigma$. It is the case that $\mathrm{L}(N) = \mathrm{L}(M) = A$, and so we are 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 the proof will be complete.

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 the 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 NFA N described in Figure 3.2. If we describe this NFA formally, according to the definition of NFAs, it is given by

$$N = (Q, \Sigma, \delta, q_0, F) \tag{3.9}$$

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:

$$\begin{aligned} &\delta(q_0, 0) = \{q_0, q_1\}, \qquad &\delta(q_0, 1) = \{q_1\}, \qquad &\delta(q_0, \varepsilon) = \varnothing, \\ &\delta(q_1, 0) = \varnothing, &\delta(q_1, 1) = \{q_0\}, &\delta(q_1, \varepsilon) = \varnothing. \end{aligned} \tag{3.10}$$

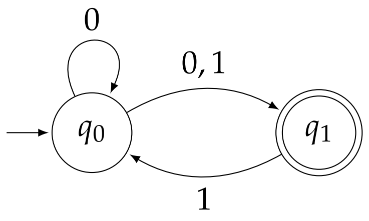


Figure 3.2: An NFA that will be converted into an equivalent DFA.

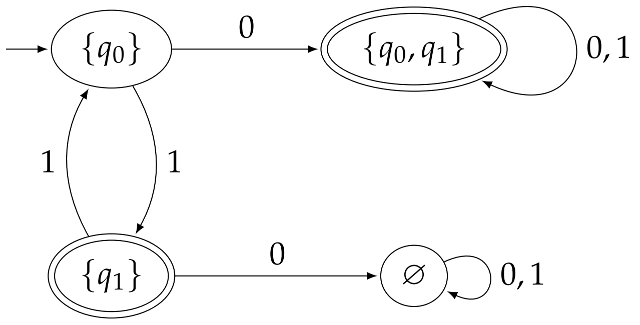


Figure 3.3: A DFA equivalent to the NFA from Figure 3.2.

We are going to define an DFA 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 $\mathcal{P}(Q)$.

Consider the state diagram in Figure 3.3. Formally speaking, this DFA is given by

$$M = (\mathcal{P}(Q), \Sigma, \mu, \{q_0\}, \{\{q_1\}, \{q_0, q_1\}\}), \tag{3.11}$$

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

$$\begin{aligned} &\mu(\{q_0\}, 0) = \{q_0, q_1\}, && \mu(\{q_0\}, 1) = \{q_1\}, \\ &\mu(\{q_1\}, 0) = \varnothing, && \mu(\{q_1\}, 1) = \{q_0\}, \\ &\mu(\{q_0, q_1\}, 0) = \{q_0, q_1\}, && \mu(\{q_0, q_1\}, 1) = \{q_0, q_1\}, \\ &\mu(\varnothing, 0) = \varnothing, && \mu(\varnothing, 1) = \varnothing. \end{aligned} \tag{3.12}$$

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 are not 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 are not 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

$$N = (Q, \Sigma, \delta, q_0, F). \tag{3.13}$$

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 is going on using state diagrams, so we will 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

$$\mu : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q) \tag{3.14}$$

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

$$\mu(R, a) \subseteq Q \tag{3.15}$$

for every subset $R \subseteq Q$ and every symbol $a \in \Sigma$. One way to do this is as follows:

$$\mu(R, a) = \bigcup_{q \in R} \varepsilon(\delta(q, a)). \tag{3.16}$$

In words, the right-hand side of (3.16) represents every state in N that you can get to by (i) starting at any state in R, then (ii) following a transition labeled a, 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

$$G = \big\{R \in \mathcal{P}(Q) \,:\, R \cap F \neq \varnothing\big\}. \tag{3.17}$$

The DFA M can now be specified formally as

$$M = (\mathcal{P}(Q), \Sigma, \mu, \varepsilon(\{q_0\}), G). \tag{3.18}$$

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 have done is to define a DFA M from a given NFA N that *seems* like it should satisfy this equality. It is, in fact, true that $\mathrm{L}(M) = \mathrm{L}(N)$, but we will not go through a formal proof that this really is the case. It is worthwhile, however, to think about how we would do this if we had to.

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 separate statements:

1. $\mathrm{L}(M) \subseteq \mathrm{L}(N)$.
2. $\mathrm{L}(N) \subseteq \mathrm{L}(M)$.

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 us 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 is 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

$$\mu^*(\varepsilon(R), w) = \bigcup_{q \in R} \delta^*(q, w) \tag{3.19}$$

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, you are not being asked to formalize and verify the proofs just suggested at this stage, but only to *think about* how it would be done.

On the process of converting NFAs to DFAs

It is a typical type of exercise in courses such as this one that students are presented with an NFA and asked to come up with an equivalent DFA using the construction described above. This is a mechanical exercise, and it will be important later in the course to observe that the construction itself can be performed by a computer. This fact may become more clear once you have gone through a few examples by hand.

When performing this construction by hand, it is worth noting you do not 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 sometimes be that many of these states are 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 then to only draw new states of M as you need them.

In the worst case, however, you might actually need exponentially many states. Indeed, 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 in comparison to the number of states of the original NFA.

Lecture 4

Regular operations and regular expressions

This lecture focuses on three fundamentally important operations on languages—*union*, *concatenation*, and *Kleene star*—which are collectively known as the *regular operations*. We will prove that the regular languages are *closed* under the regular operations, as well as some other basic operations defined on languages. We will then formally define *regular expressions*, and prove that they offer an alternative characterization of the regular languages.

4.1 Regular operations

Let us begin with a formal definition of the regular operations.

Definition 4.1. The *regular operations* are the operations *union*, *concatenation*, and *Kleene star* (or just *star*, for short), which are defined as follows for any choice of an alphabet Σ and languages $A, B \subseteq \Sigma^*$:

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.* 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 have made and not something mandated in a mathematical sense.

4.2 Closure of regular languages under regular operations

Now we will prove that when regular operations are performed on regular languages, the result must always be a regular language. That is to say, the regular languages are *closed* with respect to the regular operations.

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. Let us assume A and B are fixed regular languages for the remainder of the proof. Because these languages 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 $P \cap Q = \varnothing$).

The first regular operation is union. From the previous 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

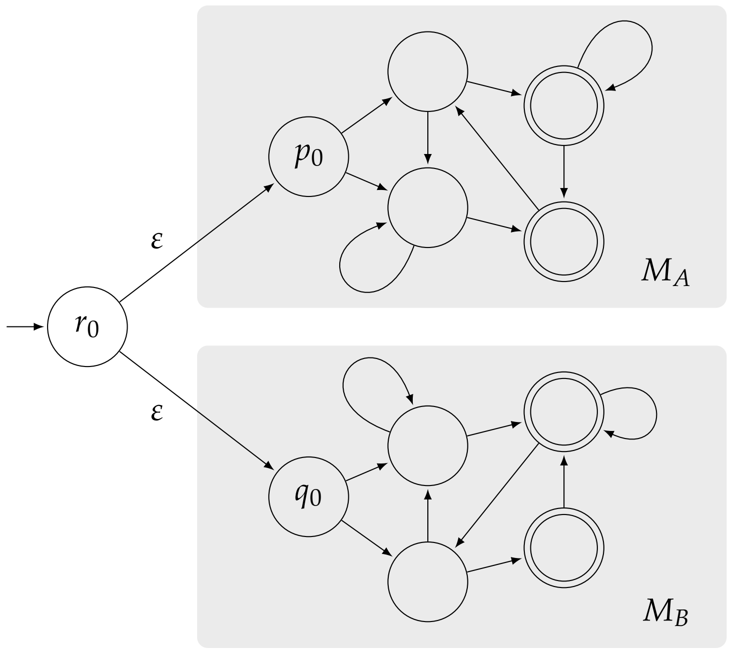


Figure 4.1: DFAs M_A and M_B are combined to form an NFA for the language $\mathrm{L}(M_A) \cup \mathrm{L}(M_B)$.

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 diagrams of M_A and M_B. (The illustrations in the figure are only meant to suggest hypothetical state diagrams for these two DFAs. The actual state diagrams for M_A and M_B can be arbitrary.)

We can specify N more formally as follows:

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

where

$$R = P \cup Q \cup \{r_0\} \tag{4.6}$$

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

$$\eta : R \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(R) \tag{4.7}$$

is defined as follows:

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

The set of accept states of N in $F \cup G$.

Every string that is accepted by M_A is also accepted by N. This is because N may first follow the ε-transition from r_0 to p_0 and then follow the same transitions that would be followed by M_A. Because the accept states of N include all of the accept states of M_A, this allows N to accept.

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

$$\mathrm{L}(N) = \mathrm{L}(M_A) \cup \mathrm{L}(M_B) = A \cup B, \tag{4.8}$$

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

The second regular operation is concatenation. 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

$$N = (P \cup Q, \Sigma, \eta, p_0, G) \tag{4.9}$$

where the transition function

$$\eta : (P \cup Q) \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(P \cup Q) \tag{4.10}$$

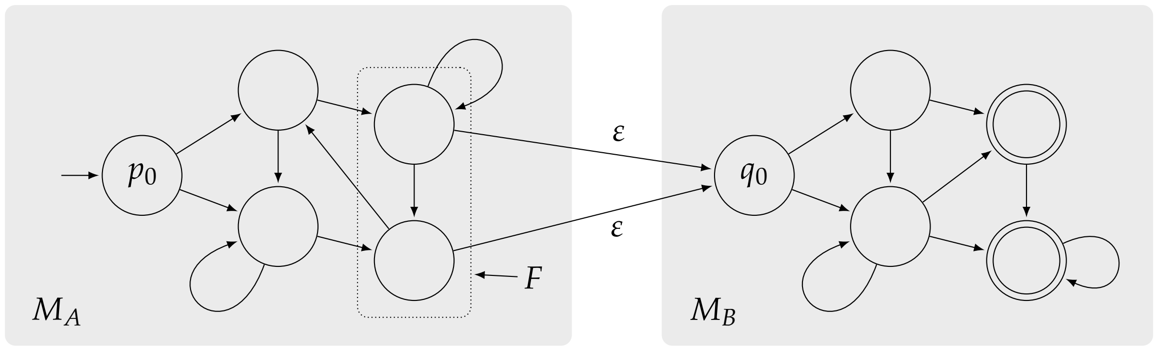


Figure 4.2: DFAs M_A and M_B are combined to form an NFA for the language $\mathrm{L}(M_A)\,\mathrm{L}(M_B)$.

is given by

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

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.

The third regular operation is star. 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; define

$$N = (R, \Sigma, \eta, r_0, \{r_0\}) \tag{4.11}$$

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

$$\eta : R \times (\Sigma \cup \{\varepsilon\}) \rightarrow \mathcal{P}(R) \tag{4.12}$$

is defined as

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

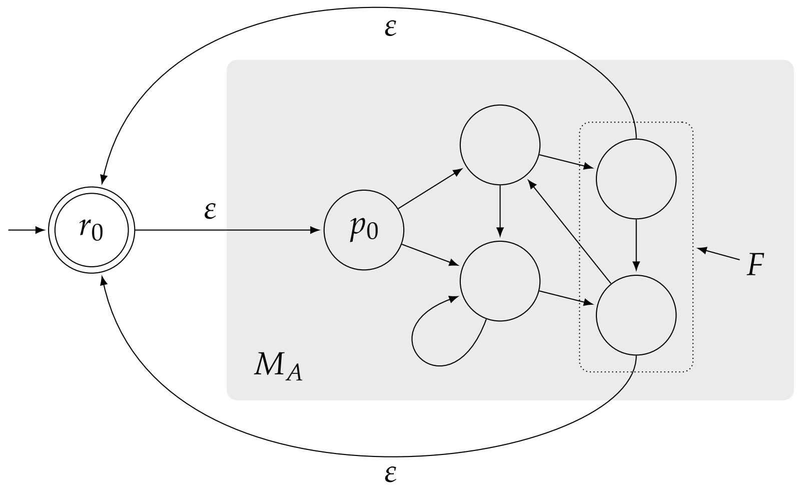


Figure 4.3: The DFA M_A is modified to form an NFA for the language $\mathrm{L}(M_A)^*$.

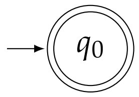
In words, we take N to be the NFA whose states are the states of M_A along with an additional state r_0, which is both the start state of N and its only accept state. The transitions of N include all of the transitions of M_A, along with an ε-transition from r_0 to the start state p_0 of M_A, and ε-transitions 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 fact that the regular languages are closed under both union and concatenation. In more detail, we have that

$$A^* = \{\varepsilon\} \cup A \cup AA \cup AAA \cup \cdots \tag{4.13}$$

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

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

is regular, the language

$$\{\varepsilon\} \cup A \cup AA \cup AAA \cup AAAA \tag{4.15}$$

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 (4.13) is regular, but only that *finite* unions such as (4.15) 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

$$A^* = \{\varepsilon, 0, 00, 000, \ldots\} \tag{4.16}$$

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.

4.3 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:

$$\overline{A} = \Sigma^* \backslash A. \tag{4.17}$$

(For the sake of clarity, we will use a backslash to denote set differences: $S\backslash T$ 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 imply that

$$A \cap B = \overline{\overline{A} \cup \overline{B}}. \tag{4.18}$$

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

$$M_A = (P, \Sigma, \delta, p_0, F) \quad \text{and} \quad M_B = (Q, \Sigma, \mu, q_0, G) \tag{4.19}$$

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

$$M = (P \times Q, \Sigma, \eta, (p_0, q_0), F \times G) \tag{4.20}$$

where

$$\eta((p, q), a) = (\delta(p, a), \mu(q, a)) \tag{4.21}$$

for every $p \in P$, $q \in Q$, and $a \in \Sigma$. 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).

4.4 Regular expressions

Regular expressions are commonly used in programming languages and other applications to specify patterns for searching and string matching. When regular expressions are used in practice, they are typically endowed with a rich set of convenient operations, but in this course we shall take a minimal definition of regular expressions allowing only the three regular operations (and no other operations like negation or special symbols marking the first or last characters of an input).

Here is the formal definition of regular expressions. The definition is an example of an *inductive definition*, and some comments on inductive definitions will follow.

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. $R = a$ for some choice of $a \in \Sigma$.
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:

$$R = (((\cdots(\cdots)^*\cdots)^*)^*)^*. \tag{4.22}$$

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 $\Sigma = \{0,1\}$:

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

When we are talking about regular expressions over an alphabet Σ, you should think of them as being strings over the alphabet

$$\Sigma \cup \{\, (\, ,\,)\, ,\, *\, ,\, \cup,\, \varepsilon,\, \varnothing\} \tag{4.23}$$

(assuming of course that Σ and $\{\, (\, ,\,)\, ,\, *\, ,\, \cup,\, \varepsilon,\, \varnothing\}$ are disjoint). Some authors will use a different font for regular expressions so that this is more obvious, but this will not be done in these notes.

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

Order of precedence for regular operations

It might appear that regular expressions arising from Definition 4.6 have a lot 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 are not changing the formal definition of regular expressions, we are just introducing a convention that allows some parentheses to be implicit, which makes for simpler-looking regular expressions. For example, we write

$$10^* \cup 1 \tag{4.24}$$

rather than

$$((1(0^*)) \cup 1). \tag{4.25}$$

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 names were chosen as they were.

There are two implications needed to establish that the regular languages coincide 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. The language $\mathrm{L}(R)$ is regular.*

The idea behind the proof of this proposition is simple enough: we can easily build DFAs for the languages $\varnothing$, $\{\varepsilon\}$, and $\{a\}$ (for each symbol $a \in \Sigma$), 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.

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 the language

$$B^k_{p,q} \subseteq \Sigma^*, \tag{4.26}$$

for every choice of states $p, q \in \{1, \ldots, n\}$ and an integer $k \in \{0, \ldots, n\}$, to be the set of all strings w that cause M to operate in the following way:

> 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 in general 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

$$\mathrm{L}(R^k_{p,q}) = B^k_{p,q}, \tag{4.27}$$

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 $R_{p,q}^{k-1}$ satisfying

$$\mathrm{L}(R_{p,q}^{k-1}) = B_{p,q}^{k-1} \tag{4.28}$$

for every $p, q \in \{1, \ldots, n\}$. It is the case that

$$B_{p,q}^{k} = B_{p,q}^{k-1} \cup B_{p,k}^{k-1}\big(B_{k,k}^{k-1}\big)^{*} B_{k,q}^{k-1}. \tag{4.29}$$

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 through the intermediate states $\{1, \ldots, k-1\}$, so that state k is not visited as an intermediate state, or (ii) cause M to move from p to q through the intermediate states $\{1, \ldots, k\}$, visiting the state k as an intermediate state one or more times. We may therefore define a regular expression $R_{p,q}^{k}$ satisfying (4.27) for every $p, q \in \{1, \ldots, n\}$ as

$$R_{p,q}^{k} = R_{p,q}^{k-1} \cup R_{p,k}^{k-1}\big(R_{k,k}^{k-1}\big)^{*} R_{k,q}^{k-1}. \tag{4.30}$$

Finally, we obtain a regular expression R satisfying $\mathrm{L}(R) = A$ by defining

$$R = \bigcup_{q \in F} R_{q_0,q}^{n}. \tag{4.31}$$

In words, R is the regular expression we obtain by forming the union over all regular expressions $R_{q_0,q}^{n}$ where q is an accept state. This completes the proof. $\square$

There is a procedure that can be used to convert a given DFA into an equivalent regular expression. The idea behind this conversion process has some similarities to the proof above. It tends to get messy, producing rather large and complicated-looking regular expressions from relatively simple DFAs, but it works—and just like the conversion of an NFA to an equivalent DFA, it can be implemented by a computer.

Lecture 5

Proving languages to be nonregular

We already know, for any alphabet Σ, that there exist languages $A \subseteq \Sigma^*$ that are nonregular. This is because there are *uncountably* many languages over Σ but only *countably* many regular languages over Σ. However, this observation does not allow us to conclude that specific nonregular languages are indeed nonregular. 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 simple fact—known as the *pumping lemma*—which establishes that a certain property must hold for all regular languages. Later in the lecture, in the section following this one, we will 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

$$w = a_1 \cdots a_m \tag{5.1}$$

for $m = |w|$ and $a_1, \ldots, a_m \in \Sigma$. 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

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

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.[1] 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:

$$x = a_1 \cdots a_s, \qquad y = a_{s+1} \cdots a_t, \qquad z = a_{t+1} \cdots a_m. \tag{5.4}$$

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

[1] 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.

length t, which is at most n because t was chosen from the set $\{0, \ldots, n\}$. 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

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

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

The pumping lemma for an example DFA

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 state diagram illustrated in Figure 5.1.

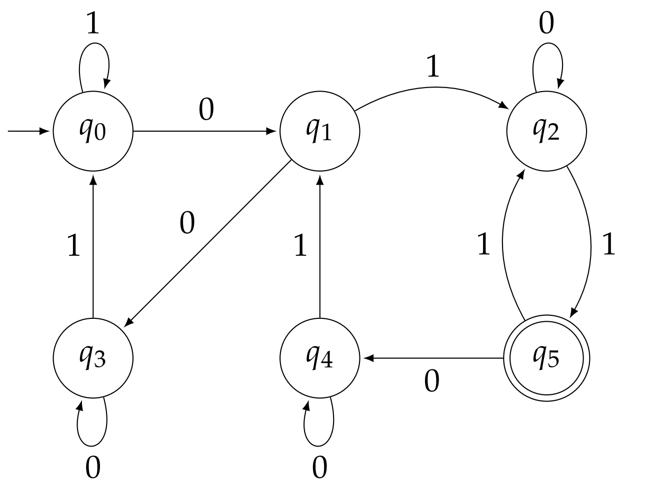


Figure 5.1: The state diagram of a DFA M, to be used to provide an example to explain the pumping lemma.

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, and in this particular case there are actually 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, we see that M must not only accept $w = xyz$, but it must also accept xz, $xyyz$, $xyyyz$, and so on.

There is nothing special about the example just described; something similar always happens. Pick any DFA whatsoever, and then pick 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. 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

It is helpful to keep in mind that the pumping lemma is a statement about regular languages: it establishes a property that must always hold for every chosen regular language.

Although the pumping lemma is a statement about regular languages, we can use it 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 apply the pumping lemma to it.
2. Using the property that the pumping lemma establishes for A, we derive a contradiction. The contradiction will almost always 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 deduce that A is nonregular.

Examples of nonregularity proved through the pumping lemma

Let us illustrate this method for a few example languages. 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:*

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

The language SAME *is not regular.*

Remark 5.3. Whenever we give a language a special name like this, it is to be understood that this language is so defined for the remainder of the course (although reminders will often appear). It is a good idea to remember the languages that are given special names—they will serve well as examples. Two additional named languages will appear in this lecture.

Proof. Assume toward contradiction that SAME is regular. By the pumping lemma for regular languages, there must exist a pumping length $n \geq 1$ for SAME 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 is the case that $w \in \text{SAME}$ 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. $xy^i z \in \text{SAME}$ 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, as the substring xy is not long enough to reach the 1s in xyz. This means that $y = 0^k$ for some choice of $k \in \mathbb{N}$, and because $y \neq \varepsilon$, we 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 do not actually need this more general formula for the sake of the current proof, but in other similar cases a formula like this can be helpful.)

However, because $k \geq 1$, we see that the string $xy^2z = 0^{n+k}1^n$ is *not* contained in SAME. This contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^iz \in \text{SAME}$ for all $i \in \mathbb{N}$.

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

Proposition 5.4. *Let $\Sigma = \{0, 1\}$ be the binary alphabet and define a language over Σ as follows:*

$$A = \{0^m1^r : m, r \in \mathbb{N},\ m > r\}. \tag{5.11}$$

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 $w = 0^{n+1}1^n$. We see that $w \in A$ 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 are satisfied:

1. $y \neq \varepsilon$,
2. $|xy| \leq n$, and
3. $xy^iz \in A$ 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

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

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

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

However, because $k \geq 1$, and therefore $n + 1 - k \leq n$, we see that the string xy^0z is *not* contained in A. This contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^iz \in A$ 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. □

Remark 5.5. 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 5.6. *Let $\Sigma = \{0\}$ and define a language over Σ as follows:*

$$\text{SQUARE} = \left\{0^{m^2} : m \in \mathbb{N}\right\}. \tag{5.14}$$

The language SQUARE *is not regular.*

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

Define $w = 0^{n^2}$. We observe that $w \in$ SQUARE 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 are satisfied:

1. $y \neq \varepsilon$,
2. $|xy| \leq n$, and
3. $xy^iz \in$ SQUARE 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 $1 \leq k \leq n$, and therefore

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

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; the number $n^2 + k$ 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 SQUARE, which contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^iz \in$ SQUARE for all $i \in \mathbb{N}$.

Having obtained a contradiction, we conclude that the assumption of SQUARE being regular was wrong. The language SQUARE is therefore nonregular. □

In advance of the next example, let us introduce some notation that will be useful 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, assuming w is a string over an alphabet Σ, we may define the string reversal operation inductively as follows:

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

Let us also define the language

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

over the binary alphabet $\Sigma = \{0,1\}$. This language is named PAL because it is short for *palindrome*, which (as you may know) is something that reads the same forward and backward.

Proposition 5.7. *The language* PAL *is not regular.*

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

Define $w = 0^n 1 0^n$. We observe that $w \in \mathrm{PAL}$ 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 are satisfied:

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

Once again, we may conclude that $y = 0^k$ for $k \geq 1$. This time it is the case 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 PAL. This contradicts the third condition stated by the pumping lemma, which guarantees us that $xy^iz \in \mathrm{PAL}$ for all $i \in \mathbb{N}$.

Having obtained a contradiction, we conclude that our assumption that PAL is regular was wrong. The language PAL 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 are 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.

If you do not know what string w to choose, take a guess and aim for a contradiction. If you do not succeed, you may find that you have gained some intuition on what a better choice might be. Of course, you should thoroughly be convinced by your own arguments and actively look for ways they might be going wrong; if you do not truly believe your own proof, it is not likely anyone else will believe it either.

5.3 Nonregularity from closure properties

Sometimes we can prove that a particular language is nonregular by combining together closure properties for regular languages our knowledge of other languages being nonregular. Here are two examples, again stated as propositions.

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

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

The language B is not regular.

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

Proposition 5.9. *Let $\Sigma = \{0,1\}$ and define a language over Σ as follows:*

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

The language C is not regular.

Proof. Assume toward contradiction that C 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 $C \cap \mathrm{L}(0^*1^*)$ is regular. However, we have that

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

the language defined in Proposition 5.4, which we already proved is nonregular. This is a contradiction, and therefore our assumption that C is regular was wrong. We conclude that C is nonregular. □

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

Lecture 6

Further discussion of regular languages

In this lecture we will discuss some additional operations under which the regular languages are closed and go over a few example problems concerning regular languages. This is the last lecture of the course 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 as the course progresses.

6.1 Other operations on languages

We have 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 will 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 $w = ax$ for $a \in \Sigma$ and $x \in \Sigma^*$, then $w^{\mathrm{R}} = x^{\mathrm{R}}a$.

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

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

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

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

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 $S = a$ for some choice of $a \in \Sigma$, then $S^{\mathrm{R}} = a$.
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 $S^{\mathrm{R}} = \left((S_1^{\mathrm{R}})^*\right)$.

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 $\mathrm{L}(S^{\mathrm{R}}) = \mathrm{L}(S)^{\mathrm{R}} = A^{\mathrm{R}}$, we have that A^{R} is regular, as required. □

Second proof (sketch). We will 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

$$M = (Q, \Sigma, \delta, q_0, F) \tag{6.3}$$

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:

$$N = (Q \cup \{r_0\}, \Sigma, \mu, r_0, \{q_0\}), \tag{6.4}$$

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:

$$\begin{aligned} \mu(r_0, \varepsilon) &= F, \\ \mu(r_0, a) &= \varnothing, \\ \mu(q, \varepsilon) &= \varnothing, \\ \mu(q, a) &= \{p \in Q : \delta(p, a) = q\}, \end{aligned} \tag{6.5}$$

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

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

$$A \mathbin{\triangle} B = (A \backslash B) \cup (B \backslash A). \tag{6.6}$$

In words, the elements of the symmetric difference $A \mathbin{\triangle} B$ 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 $A \mathbin{\triangle} B$ 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 terms of these operations. More specifically, if we assume that A and B are regular,

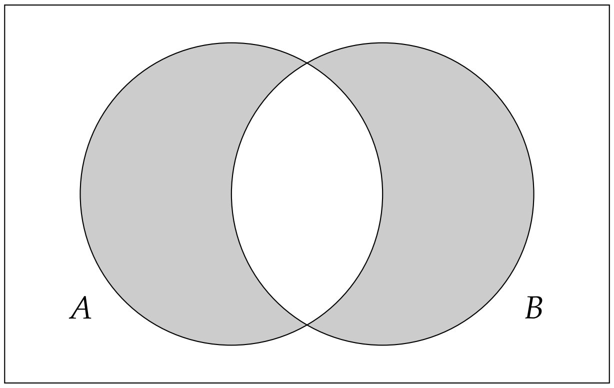


Figure 6.1: The shaded region denotes the symmetric difference $A \triangle B$ of two sets A and B.

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

$$A \triangle B = \bigl(A \cap \overline{B}\bigr) \cup \bigl(\overline{A} \cap B\bigr), \tag{6.7}$$

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 that can be obtained from w by removing zero or more symbols from the right-hand side of w; a *suffix* of w is any string that can be obtained by removing zero or more symbols from the left-hand side of w; and a *substring* of w is any string that can be obtained 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

$$\begin{aligned}
\operatorname{Prefix}(A) &= \bigl\{x \in \Sigma^* \,:\, \text{there exists } v \in \Sigma^* \text{ such that } xv \in A\bigr\}, && (6.8)\\
\operatorname{Suffix}(A) &= \bigl\{x \in \Sigma^* \,:\, \text{there exists } u \in \Sigma^* \text{ such that } ux \in A\bigr\}, && (6.9)\\
\operatorname{Substring}(A) &= \bigl\{x \in \Sigma^* \,:\, \text{there exist } u, v \in \Sigma^* \text{ such that } uxv \in A\bigr\}. && (6.10)
\end{aligned}$$

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 establishes.

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

$$R = \{q \in Q : \text{there exists } w \in \Sigma^* \text{ such that } \delta^*(q_0, w) = q\}. \tag{6.11}$$

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

$$P = \{q \in Q : \text{there exist } w \in \Sigma^* \text{ such that } \delta^*(q, w) \in F\}. \tag{6.12}$$

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

First, define a DFA $K = (Q, \Sigma, \delta, q_0, P)$. In words, K 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. We see that $\text{L}(K) = \text{Prefix}(A)$, and therefore $\text{Prefix}(A)$ is regular.

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

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

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

Finally, the fact that $\text{Substring}(A)$ is regular follows from the observation that $\text{Substring}(A) = \text{Suffix}(\text{Prefix}(A))$ (or that $\text{Substring}(A) = \text{Prefix}(\text{Suffix}(A))$). $\square$

[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.

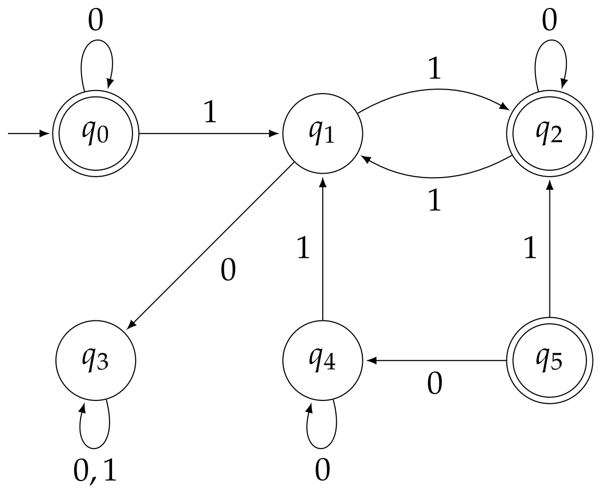


Figure 6.2: An example of a DFA M. In this case, the set R of reachable states is $R = \{q_0, q_1, q_2, q_3\}$ while the set P of states from which it is possible to reach an accepting state of M is $P = \{q_0, q_1, q_2, q_4, q_5\}$.

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 a regular language. Prove that the language

$$B = \{uv : u, v \in \Sigma^* \text{ and } uav \in A \text{ for some choice of } a \in \Sigma\} \tag{6.13}$$

is regular.

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 exactly 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

$$M = (Q, \Sigma, \delta, q_0, F) \tag{6.14}$$

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

$$N = (P, \Sigma, \eta, p_0, G) \tag{6.15}$$

as follows. First, we will define

$$P = \{0,1\} \times Q, \tag{6.16}$$

and we will take the start state of N to be

$$p_0 = (0, q_0). \tag{6.17}$$

The accept states of N will be

$$G = \{(1,q) \,:\, q \in F\}. \tag{6.18}$$

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

1. $\eta((0,q),a) = \{(0,\delta(q,a))\}$ for every $q \in Q$ and $a \in \Sigma$.
2. $\eta((0,q),\varepsilon) = \{(1,\delta(q,0)),(1,\delta(q,1))\}$ for every $q \in Q$.
3. $\eta((1,q),a) = \{(1,\delta(q,a))\}$ for every $q \in Q$ and $a \in \Sigma$.
4. $\eta((1,q),\varepsilon) = \varnothing$ 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 1 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 the state $(1,\delta(q,1))$. Intuitively speaking, N is reading nothing from its input while "hypothesizing" that M has read some symbol a (which is either 0 or 1). This is the effect of the transitions listed as 2. Then N simply continues simulating M on the remainder of the input string, which is the effect of the transitions listed as 3. 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 4.

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 to the same problem. 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

$$M = (Q, \Sigma, \delta, q_0, F) \tag{6.19}$$

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

$$M_{p,q} = (Q, \Sigma, \delta, p, \{q\}), \tag{6.20}$$

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

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

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:

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

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

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

Every string in B takes the form uv, for some choice of $u, v \in \Sigma^*$ and $a \in \Sigma$ for which $uav \in A$. 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 $uav \in A_{q_0,r}$. As a causes M to transition from p to $\delta(p,a)$, it follows that v must cause M to transition from $\delta(p,a)$ to r, i.e., $v \in A_{\delta(p,a),r}$. It therefore the case that $uv \in A_{q_0,p}A_{\delta(p,a),r}$, which implies the required inclusion.

Next we will prove that

$$\bigcup_{(p,a,r) \in Q \times \Sigma \times F} A_{q_0,p} A_{\delta(p,a),r} \subseteq B. \tag{6.24}$$

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

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

$$C = \{vu \,:\, u,v \in \Sigma^* \text{ and } uv \in A\} \tag{6.25}$$

is regular.

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

$$M = (Q, \Sigma, \delta, q_0, F) \tag{6.26}$$

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

$$N = (P, \Sigma, \eta, p_0, G) \tag{6.27}$$

as follows. First, we will define

$$P = \big(\{0,1\} \times Q \times Q\big) \cup \{p_0\}, \tag{6.28}$$

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

$$G = \{(1,q,q) \,:\, q \in Q\}. \tag{6.29}$$

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

1. $\eta(p_0, \varepsilon) = \{(0,q,q) \,:\, q \in Q\}$.
2. $\eta((0,r,q),a) = \{(0,\delta(r,a),q)\}$ for all $q, r \in Q$ and $a \in \Sigma$.
3. $\eta((0,r,q),\varepsilon) = \{(1,q_0,q)\}$ for every $r \in F$ and $q \in Q$.
4. $\eta((1,r,q),a) = \{(1,\delta(r,a),q)\}$ for all $q, r \in Q$ and $a \in \Sigma$.

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

Let us consider this definition in greater detail to understand how it works. 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 has not seen yet, so it is just a guess).

Then, N starts reading symbols and essentially mimicking M on those input symbols—this is the point of the transitions listed in item 2. At some point, nondeterministically chosen, N decides that it is 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 in item 3. 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 is 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

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

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

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

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

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

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

$$\bigcup_{(p,r)\in Q\times F} A_{p,r}A_{q_0,p} \subseteq C \tag{6.33}$$

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

$$A = C \cup \{\varepsilon\} \quad \text{and} \quad B = \overline{C} \cup \{\varepsilon\}. \tag{6.34}$$

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 this course is the class of *context-free languages*, which are defined by the notion of a *context-free grammar*, or a CFG for short.

7.1 Definitions of context-free grammars and languages

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:

$$\begin{aligned} S &\rightarrow 0S1 \\ S &\rightarrow \varepsilon. \end{aligned} \tag{7.3}$$

It is often convenient to describe a CFG just by listing the rules, like 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 appearing on the left-hand side of the rules 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.

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:
   - 2.1 Choose any rule $A \rightarrow w$ from R.
   - 2.2 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

$$\mathrm{SAME} = \{0^n 1^n \, : \, n \in \mathbb{N}\}. \tag{7.4}$$

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 possible: 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 process through which the language generated by a CFG is determined 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 $x, y \in (V \cup \Sigma)^*$ one has 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

1. $x = z_1$,
2. $y = z_m$, and
3. $z_k \Rightarrow_G z_{k+1}$ for every $k \in \{1, \ldots, m-1\}$.

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

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

If $x \in \mathrm{L}(G)$ for a CFG $G = (V, \Sigma, R, S)$, and $z_1, \ldots, z_m \in (V \cup \Sigma)^*$ is a sequence of strings for which $z_1 = S$, $z_m = x$, and $z_k \Rightarrow_G z_{k+1}$ for all $k \in \{1, \ldots, m-1\}$, then

[1] Recall that a relation is a subset of a Cartesian product of two sets. In this case, the relation is the subset $\{(uAv, uwv) \,:\, u, v, w \in (V \cup \Sigma)^*,\ A \in V,$ and $A \rightarrow w$ 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.

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 might be more than one derivation of a given string $x \in \mathrm{L}(G)$.

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 SAME is a context-free language, as has been established in Example 7.3.

7.2 Examples of context-free grammars and languages

We have seen one example of a context-free language so far: SAME. Let us now consider a few more examples.

Basic examples

Example 7.9. The language

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

over the alphabet $\Sigma = \{0, 1\}$, which we first encountered in Lecture 5, is context free. (In fact this is true for any choice of an alphabet Σ, but let us stick with 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:

$$\begin{aligned} S &\rightarrow 0S0 \\ S &\rightarrow 1S1 \\ S &\rightarrow 0 \\ S &\rightarrow 1 \\ S &\rightarrow \varepsilon \end{aligned} \tag{7.10}$$

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:

$$S \rightarrow 0S0 \mid 1S1 \mid 0 \mid 1 \mid \varepsilon \tag{7.11}$$

When 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:

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

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:

$$S \rightarrow 0S1S \mid 1S0S \mid \varepsilon. \tag{7.13}$$

Now, it is clear that every string generated by this grammar, which we will call G, is contained in A: we begin any derivation with the variable S alone, so there are an equal number of 0s and 1s 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 the hypothesis of induction is that $x \in \mathrm{L}(G)$ for every string $x \in A$ with $|x| < n$. Our goal is to prove that G generates w. Let us write

$$w = a_1 \cdots a_n \tag{7.14}$$

for $a_1, \ldots, a_n \in \Sigma$. We have assumed that $w \in A$, and therefore

$$|a_1 \cdots a_n|_0 = |a_1 \cdots a_n|_1. \tag{7.15}$$

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

$$|a_1 \cdots a_m|_0 = |a_1 \cdots a_m|_1; \tag{7.16}$$

we know that this equation is satisfied when $m = n$, and there might be a smaller value of m that works. We will now prove that $a_1 \neq a_m$.

The fact that $a_1 \neq a_m$ follows from a proof by contradiction. Toward this goal, assume $a_1 = a_m$, and define

$$N_k = |a_1 \cdots a_k|_1 - |a_1 \cdots a_k|_0 \tag{7.17}$$

for every $k \in \{1, \ldots, m\}$. We know that $N_m = 0$ because equation (7.16) is satisfied. Moreover, using the assumption $a_1 = a_m$, we observe the equations

$$\begin{aligned} |a_1 \cdots a_m|_1 &= |a_1 \cdots a_{m-1}|_1 + |a_m|_1 = |a_1 \cdots a_{m-1}|_1 + |a_1|_1 \\ |a_1 \cdots a_m|_0 &= |a_1 \cdots a_{m-1}|_0 + |a_m|_0 = |a_1 \cdots a_{m-1}|_0 + |a_1|_0, \end{aligned} \tag{7.18}$$

and conclude that

$$N_m = N_{m-1} + N_1 \tag{7.19}$$

by subtracting the second equation from the first. Therefore, because $N_m = 0$ and N_1 is nonzero, it must be that N_{m-1} is also nonzero, and more importantly N_1 and N_{m-1} must have opposite sign. However, because consecutive values of N_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 $N_k = 0$, for otherwise it would not be possible for N_1 and N_{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 proved that $a_1 \neq a_m$.

At this point it is possible to describe a derivation for w. We have $w = a_1 \cdots a_n$, and we have that

$$|a_1 \cdots a_m|_0 = |a_1 \cdots a_m|_1 \quad \text{and} \quad a_1 \neq a_m \tag{7.20}$$

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

$$|a_2 \cdots a_{m-1}|_0 = |a_2 \cdots a_{m-1}|_1 \quad \text{and} \quad |a_{m+1} \cdots a_n|_0 = |a_{m+1} \cdots a_n|_1. \tag{7.21}$$

By the hypothesis of induction it follows that

$$S \overset{*}{\Rightarrow}_G a_2 \cdots a_{m-1} \quad \text{and} \quad S \overset{*}{\Rightarrow}_G a_{m+1} \cdots a_n. \tag{7.22}$$

Therefore the string w satisfies

$$S \Rightarrow_G 0S1S \overset{*}{\Rightarrow}_G 0a_2 \cdots a_{m-1}1a_{m+1} \cdots a_n = w \tag{7.23}$$

(in case $a_1 = 0$ and $a_m = 1$) or

$$S \Rightarrow_G 1S0S \overset{*}{\Rightarrow}_G 1a_2 \cdots a_{m-1}0a_{m+1} \cdots a_n = w \tag{7.24}$$

(in case $a_1 = 1$ and $a_m = 0$). 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 will 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

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

The language BAL is context free; here is a simple CFG that generates it:

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

See if you can convince yourself that this CFG indeed generates BAL!

A more advanced example

Sometimes it is more challenging to come up with a context-free grammar for a given language. The following example concerns one such language.

Example 7.13. Let $\Sigma = \{0,1\}$ and $\Gamma = \{0,1,\#\}$, and define

$$A = \big\{u\#v \,:\, u,v \in \Sigma^* \text{ and } u \neq v\big\}. \tag{7.29}$$

Here is a context-free grammar for A:

$$\begin{aligned} S &\rightarrow W_0 1 Y \,\big|\, W_1 0 Y \,\big|\, Z \\ W_0 &\rightarrow X W_0 X \,\big|\, 0 Y \# \\ W_1 &\rightarrow X W_1 X \,\big|\, 1 Y \# \\ Z &\rightarrow X Z X \,\big|\, X Y \# \,\big|\, \# X Y \\ X &\rightarrow 0 \,\big|\, 1 \\ Y &\rightarrow X Y \,\big|\, \varepsilon. \end{aligned} \tag{7.30}$$

The idea behind this CFG is as follows. First, the variable Z generates strings of the form $u\#v$ where u and v have different lengths. The variable W_0 generates strings that look like this:

$$\underbrace{\square\square\cdots\square}_{n\text{ bits}}0\underbrace{\square\square\cdots\square}_{m\text{ bits}}\#\underbrace{\square\square\cdots\square}_{n\text{ bits}} \tag{7.31}$$

(where $\square$ means either 0 or 1), so that $W_0 1 Y$ generates strings that look like this:

$$\underbrace{\square\square\cdots\square}_{n\text{ bits}}0\underbrace{\square\square\cdots\square}_{m\text{ bits}}\#\underbrace{\square\square\cdots\square}_{n\text{ bits}}1\underbrace{\square\square\cdots\square}_{k\text{ bits}} \tag{7.32}$$

(for any choice of $n, m, k \in \mathbb{N}$). Similarly, $W_1 0 Y$ generates strings that look like this:

$$\underbrace{\square\square\cdots\square}_{n\text{ bits}}1\underbrace{\square\square\cdots\square}_{m\text{ bits}}\#\underbrace{\square\square\cdots\square}_{n\text{ bits}}0\underbrace{\square\square\cdots\square}_{k\text{ bits}} \tag{7.33}$$

Taken together, these two possibilities generate $u\#v$ for all binary strings u and v that differ in at least one position (and that may or may not have the same length). The three options together cover all possible $u\#v$ for which u and v are non-equal binary strings.

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 *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:

$$S \rightarrow 0S1S \;\big|\; 1S0S \;\big|\; \varepsilon. \tag{8.1}$$

Again we will 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:

$$S \Rightarrow 0S1S \Rightarrow 01S0S1S \Rightarrow 010S1S \Rightarrow 0101S \Rightarrow 0101. \tag{8.3}$$

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:

$$0\underline{S}1S \Rightarrow 0\underline{1S0S}1S. \tag{8.4}$$

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 does not 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 is not really anything important about right-most derivations that is not already represented by the notion of a left-most derivation, at least from the viewpoint of this course. For this reason, we will not 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:

1. 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 labeled ε when rules of the form $V \rightarrow \varepsilon$ are applied.
2. 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, variable, or ε 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.

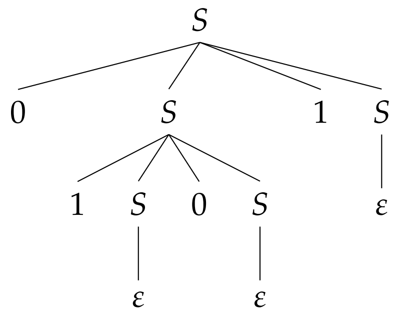


Figure 8.1: The parse tree corresponding to the derivation (8.3) of the string 0101.

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 left-most derivation of the string 0101 by the CFG (8.1) that is different from (8.3) is

$$S \Rightarrow 0S1S \Rightarrow 01S \Rightarrow 010S1S \Rightarrow 0101S \Rightarrow 0101. \tag{8.5}$$

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, 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 *every* 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

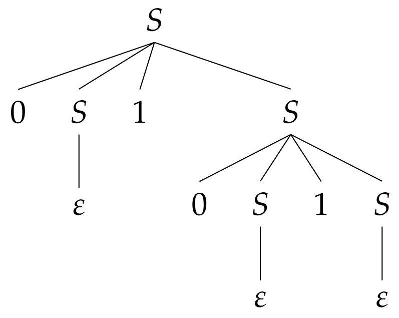


Figure 8.2: The parse tree corresponding to the derivation (8.5) of the string 0101.

L(G) described in (8.2) that, unlike the CFG (8.1), is unambiguous. Here is such a CFG:

$$\begin{aligned} S &\rightarrow 0X1S \mid 1Y0S \mid \varepsilon \\ X &\rightarrow 0X1X \mid \varepsilon \\ Y &\rightarrow 1Y0Y \mid \varepsilon \end{aligned} \tag{8.6}$$

We will not take the time to go through a proof that this CFG is unambiguous, but if you think about it for a few moments you should be able 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 when you read from left to right, and the variable Y is similar except the role of the 0s and 1s is reversed. If you try to generate a particular string by a left-most derivation with this CFG, you will never have more than one option as to which rule to apply.

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

$$\Sigma = \{a, b, +, *, (\,,\,)\} \tag{8.7}$$

along with a CFG

$$S \rightarrow S + S \mid S * S \mid (\,S\,) \mid a \mid b \tag{8.8}$$

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

For example, the string $(a + b) * a + b$ corresponds to such an expression, and one derivation for this string is as follows:

$$\begin{aligned} S &\Rightarrow S * S \Rightarrow (S) * S \Rightarrow (S + S) * S \Rightarrow (a + S) * S \Rightarrow (a + b) * S \\ &\Rightarrow (a + b) * S + S \Rightarrow (a + b) * a + S \Rightarrow (a + b) * a + b. \end{aligned} \tag{8.9}$$

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

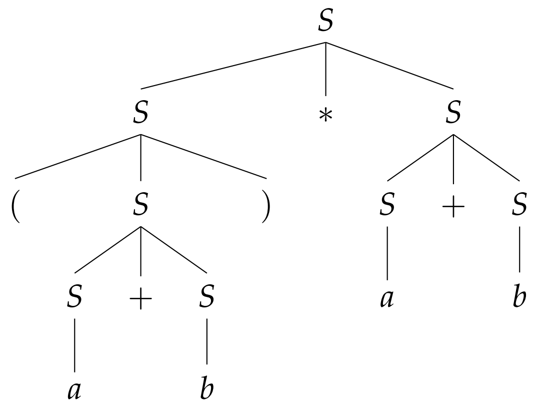


Figure 8.3: Parse tree for $(a+b)*a+b$ corresponding to the derivation (8.9).

Figure 8.3. You can of course 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 (8.8) for the sake of simplicity.

The CFG (8.8) is ambiguous. For instance, a different (left-most) derivation for the same string $(a+b)*a+b$ as before is

$$\begin{aligned} S &\Rightarrow S+S \Rightarrow S*S+S \Rightarrow (S)*S+S \\ &\Rightarrow (S+S)*S+S \Rightarrow (a+S)*S+S \Rightarrow (a+b)*S+S \\ &\qquad\qquad\qquad \Rightarrow (a+b)*a+S \Rightarrow (a+b)*a+b, \end{aligned} \tag{8.10}$$

and the parse tree for this derivation is shown in Figure 8.4.

Notice that the parse tree illustrated in Figure 8.4 is appealing because 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 according to the standard order of precedence for arithmetic operations. In contrast, the parse tree shown in Figure 8.3 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 (8.8), 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 (8.8) would be completely inadequate.

We can, however, come up with a new CFG for the same language that is much better, in the sense that it is unambiguous and properly captures the meaning of

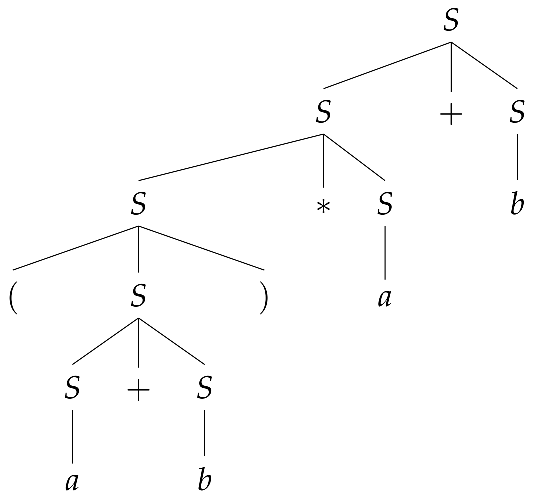


Figure 8.4: Parse tree for $(a+b)*a+b$ corresponding to the derivation (8.10).

arithmetic expressions (given that we give multiplication higher precedence than addition). Here it is:

$$\begin{aligned} S &\rightarrow T \mid S + T \\ T &\rightarrow F \mid T * F \\ F &\rightarrow I \mid (\,S\,) \\ I &\rightarrow a \mid b \end{aligned} \tag{8.11}$$

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

To better understand the CFG (8.11), 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. An example of an inherently ambiguous context-free language is this one:

$$\{0^n 1^m 2^k \,:\, n = m \text{ or } m = k\}. \tag{8.12}$$

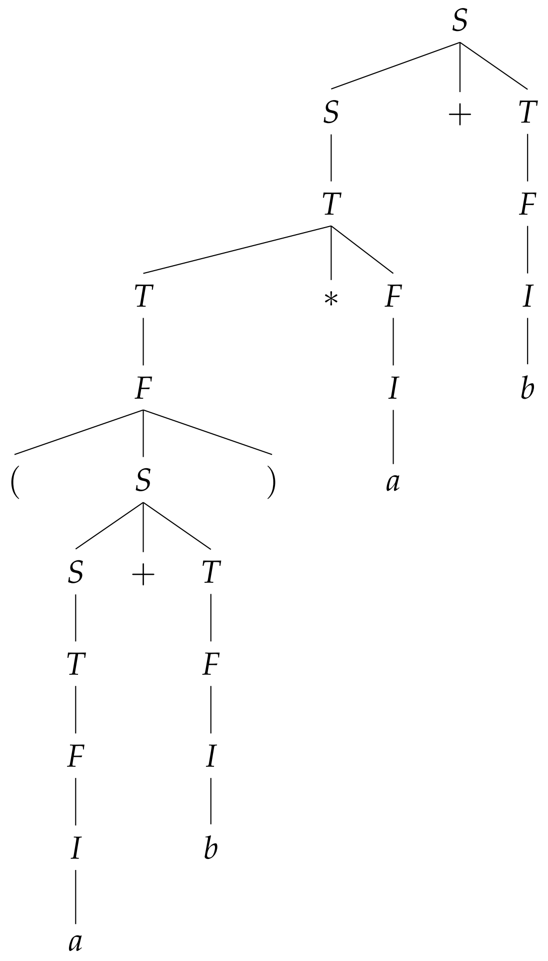


Figure 8.5: Unique parse tree for $(a + b) * a + b$ for the CFG (8.11).

We will not prove that this language is inherently ambiguous, but the intuition is that no matter what CFG you come up with for this language, 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

$$S \to SSSS \mid \varepsilon \tag{8.13}$$

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

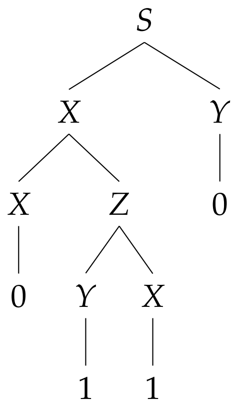


Figure 8.6: A hypothetical example of a parse tree for a CFG in Chomsky normal form.

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 $\mathrm{L}(H) = \mathrm{L}(G)$, 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 are talking about, it is appropriate at this point to define *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. $X \rightarrow a$, for a variable X and a symbol a, 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 does not have two 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.

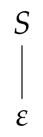


Figure 8.7: The unique parse tree for ε for a CFG in Chomsky normal form, assuming it includes the rule $S \to \varepsilon$.

If the rule $S \to \varepsilon$ is present in a CFG in Chomsky normal form, then we have a special case that does not match 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 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.

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 $\mathrm{L}(H) = \mathrm{L}(G)$. The conversion is, in fact, fairly straightforward—a summary of the steps one may perform to do this conversion for an arbitrary CFG $G = (V, \Sigma, R, S)$ appear below. To illustrate how these steps work, let us start with the following CFG, which generates the balanced parentheses language BAL from the previous lecture:

$$S \to (\,S\,)\,S \mid \varepsilon \tag{8.14}$$

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

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

   Applying this step to the CFG (8.14) yields

$$\begin{aligned} S_0 &\to S \\ S &\to (\,S\,)\,S \mid \varepsilon \end{aligned} \tag{8.15}$$

2. Introduce a new variable X_a for each symbol $a \in \Sigma$.

   First include the new rule $X_a \rightarrow a$. Then, for every other rule in which a appears on the right-hand side, except for the cases when a appears all by itself on the right-hand side, replace each a with X_a.

   Continuing with our example, the CFG (8.15) is transformed into this CFG (where we will use the names L and R rather than the weird-looking variables $X_($ and $X_)$ in the interest of style):

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

3. Split up rules of the form $X \rightarrow Y_1 \cdots Y_m$, whenever $m \geq 3$, using auxiliary variables in a straightforward way.

   In particular, $X \rightarrow Y_1 \cdots Y_m$ can be broken up as

$$\begin{aligned} X &\rightarrow Y_1 Z_2 \\ Z_2 &\rightarrow Y_2 Z_3 \\ &\vdots \\ Z_{m-2} &\rightarrow Y_{m-2} Z_{m-1} \\ Z_{m-1} &\rightarrow Y_{m-1} Y_m \end{aligned} \tag{8.17}$$

   Note that we must use separate auxiliary variables for each rule so that there is no "cross talk" between different rules—so do not reuse the same auxiliary variables to break up multiple rules.

   Transforming the CFG (8.16) in this way results in the following CFG:

$$\begin{aligned} S_0 &\rightarrow S \\ S &\rightarrow LZ_2 \mid \varepsilon \\ Z_2 &\rightarrow SZ_3 \\ Z_3 &\rightarrow RS \\ L &\rightarrow (\\ R &\rightarrow) \end{aligned} \tag{8.18}$$

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

Aside from the special case $S_0 \to \varepsilon$, there is never any need for rules of the form $X \to \varepsilon$; 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 each possible combination. You might introduce new ε-rules in this way, but they can be handled recursively—and any time a new ε-rule is generated that was already eliminated, it is not added back in.

Transforming the CFG (8.18) in this way results in the following CFG:

$$\begin{aligned} S_0 &\to S \mid \varepsilon \\ S &\to LZ_2 \\ Z_2 &\to SZ_3 \mid Z_3 \\ Z_3 &\to RS \mid R \\ L &\to (\\ R &\to) \end{aligned} \tag{8.19}$$

Note that we do end up with the ε-rule $S_0 \to \varepsilon$, but we do not eliminate this one because $S_0 \to \varepsilon$ is the special case that we allow as an ε-rule.

5. Eliminate unit rules, which are rules of the form $X \to Y$.

   Rules like this are never necessary, and they can be eliminated provided that we also include the rule $X \to w$ in the CFG whenever $Y \to w$ appears as a rule. If you obtain a new unit rule that was already eliminated (or is the unit rule currently being eliminated), it is not added back in.

   Transforming the CFG (8.19) in this way results in the following CFG:

$$\begin{aligned} S_0 &\to LZ_2 \mid \varepsilon \\ S &\to LZ_2 \\ Z_2 &\to SZ_3 \mid RS \mid) \\ Z_3 &\to RS \mid) \\ L &\to (\\ R &\to) \end{aligned} \tag{8.20}$$

   At this point we are finished; this context-free grammar is in Chomsky normal form.

The description above is only meant to give you the basic idea of how the construction works and does not constitute a formal 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.

We will make use of the theorem from time to time. In particular, 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.

Lecture 9

Properties of context-free languages

In this lecture we will examine various properties of the class of context-free languages, including the fact that it is closed under the regular operations, that every regular language is context free, and that the intersection of a context-free language and a regular language is always context free.

9.1 Closure under the regular operations

Let us begin by proving that the context-free languages are closed under 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 context-free grammars

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

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 $S \Rightarrow S_A S_B$, and then 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:

$$S \rightarrow S\, S_A \;\big|\; \varepsilon. \tag{9.6}$$

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:

$$S \rightarrow S\, S_A \;\big|\; \varepsilon$$

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

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 is therefore the case that $\mathrm{L}(K) = A^*$, so that A^* is context free. □

9.2 Relationships to regular languages

This section discusses relationships between context-free languages and regular languages. In particular, we will prove that every regular language is context free, and (more generally) that the intersection between a context-free language and a regular language is always context free.

Every regular language is context free

Let us begin with the first fact suggested above, which is that every regular language is also context free. We will discuss two different ways to prove this fact.

Theorem 9.2. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a regular language. The language 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 $R = a$ for $a \in \Sigma$, then G is the CFG
$$S \rightarrow a, \tag{9.11}$$
which generates the language $\{a\}$.
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.

In each case, we observe 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, we find 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 $\mathrm{L}(M) = A$.

We will define a CFG G that effectively simulates M, generating exactly those strings that are accepted by M. In particular, we will define

$$G = \big(V, \Sigma, R, X_{q_0}\big) \tag{9.13}$$

where the variables are $V = \{X_q : q \in Q\}$ (one variable for each state of M) and the following rules are to be included in R:

1. For each choice of $p, q \in Q$ and $a \in \Sigma$ satisfying $\delta(p, a) = q$, the rule
$$X_p \rightarrow a\,X_q \tag{9.14}$$
is included in R.
2. For each state $q \in F$, the rule
$$X_q \rightarrow \varepsilon \tag{9.15}$$
is included in R.

Now, by examining the rules suggested above, we see that every derivation of a string by G begins with the start variable (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_q corresponds to an accept state $q \in F$. By considering the rules of the first type, it is evident that

$$\big[X_{q_0} \overset{*}{\Rightarrow} wX_q\big] \;\Leftrightarrow\; \big[\delta^*(q_0, w) = q\big]. \tag{9.16}$$

We therefore have $X_{q_0} \overset{*}{\Rightarrow} w$ if and only if there exists a choice of $q \in F$ for which $\delta^*(q_0, w) = q$. This is equivalent to the statement that $\mathrm{L}(G) = \mathrm{L}(M)$, which completes the proof. $\square$

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, as we will now prove. The proof is more complicated than most of the other proofs we have seen thus far in the course—if it is not immediately clear, just do your best to try to understand the idea behind it.

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.*

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 is 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 $X \rightarrow a$ in G, include the rule

   $$X_{p,q} \rightarrow a \tag{9.19}$$

   in H for every pair of states $p, q \in Q$ for which $\delta(p, a) = q$.
2. For each rule of the form $X \rightarrow YZ$ in G, include the rule

   $$X_{p,q} \rightarrow Y_{p,r} Z_{r,q} \tag{9.20}$$

   in H for every choice of states $p, q, r \in Q$.

3. If the rule $S \rightarrow \varepsilon$ is included in G and $q_0 \in F$ (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 accept 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 $|w| = 1$ (because we are assuming $w \neq \varepsilon$), and in this case we must have $X_{p,q} \Rightarrow_H a$ for some $a \in \Sigma$. The only rules that allow such a derivation are of the first type above, which require $\delta(p,a) = q$. In the general case in which $|w| \geq 2$, it must be 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 $y, z \in \Sigma^*$ for which $w = yz$. By the hypothesis of induction we conclude that $\delta^*(p,y) = r$ and $\delta^*(r,z) = q$, so that $\delta^*(p,w) = q$.

Finally, we can prove

$$\left[\left(X \overset{*}{\Rightarrow}_G w\right) \wedge \left(\delta^*(p,w) = q\right)\right] \Rightarrow \left[X_{p,q} \overset{*}{\Rightarrow}_H w\right], \tag{9.28}$$

again by induction on the length of w. The base case is $|w| = 1$, which is straightforward: if $X \Rightarrow_G a$ and $\delta(p,a) = q$, then $X_{p,q} \Rightarrow_H a$ 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 is the case 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. □

Remark 9.4. Notice that Theorem 9.3 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 two proofs of Theorem 9.2 that we already discussed are much simpler than the one above, however, it makes sense that we considered them first.

9.3 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

$$\mathrm{Prefix}(A) = \left\{x \in \Sigma^* : \text{there exists } v \in \Sigma^* \text{ such that } xv \in A\right\}, \tag{9.30}$$

$$\mathrm{Suffix}(A) = \left\{x \in \Sigma^* : \text{there exists } u \in \Sigma^* \text{ such that } ux \in A\right\}, \tag{9.31}$$

$$\mathrm{Substring}(A) = \left\{x \in \Sigma^* : \text{there exist } u, v \in \Sigma^* \text{ such that } uxv \in A\right\}. \tag{9.32}$$

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 $\mathrm{L}(G) = A$.

We will need to make one additional assumption on the grammar G, which is that none of the variables in G generates the empty language. A variable that generates the empty language is called a *useless variable*, and it should not be hard to convince yourself that useless variables are indeed useless (with one exception). That is, if you have any CFG G in Chomsky normal form that generates a nonempty language, you can easily come up with a new CFG in Chomsky normal form for the same language that does not contain any useless variables simply by removing the useless variables and every rule in which a useless variable appears.

The one exception is the empty language itself, which by definition requires that the start variable is useless (and you will need at least one additional useless variable to ensure that the grammar has a nonempty set of rules and obeys the conditions of a CFG in Chomsky normal form). However, we do not need to worry about this case because Prefix$(\varnothing)$, Suffix$(\varnothing)$, and Substring$(\varnothing)$ are all equal to the empty language, and are therefore context free.

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 $X \rightarrow YZ$ in G, include these rules in H:

$$\begin{aligned} X &\rightarrow YZ \\ X_0 &\rightarrow YZ_0 \mid Y_0 \end{aligned} \tag{9.33}$$

2. For every rule of the form $X \rightarrow a$ in G, include these rules in H:

$$\begin{aligned} X &\rightarrow a \\ X_0 &\rightarrow a \mid \varepsilon \end{aligned} \tag{9.34}$$

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. 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 $X \rightarrow YZ$ in G, include these rules in K:

$$\begin{aligned} X &\rightarrow YZ \\ X_1 &\rightarrow Y_1 Z \mid Z_1 \end{aligned} \tag{9.35}$$

2. For every rule of the form $X \to a$ in G, include these rules in K:

$$\begin{aligned} X &\to a \\ X_1 &\to a \mid \varepsilon \end{aligned} \tag{9.36}$$

Finally, we take S_1 to be the start variable of K.

To obtain a CFG J for $\text{Substring}(A)$, 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 $X \to Y\,Z$ in G, include these rules in J:

$$\begin{aligned} X &\to Y Z \\ X_0 &\to Y Z_0 \mid Y_0 \\ X_1 &\to Y_1 Z \mid Z_1 \\ X_2 &\to Y_1 Z_0 \mid Y_2 \mid Z_2 \end{aligned} \tag{9.37}$$

2. For every rule of the form $X \to a$ in G, include these rules in J:

$$\begin{aligned} X &\to a \\ X_0 &\to a \mid \varepsilon \\ X_1 &\to a \mid \varepsilon \\ X_2 &\to a \mid \varepsilon. \end{aligned} \tag{9.38}$$

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. The main idea is that 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. *$uv^i xy^i z \in A$ for all $i \in \mathbb{N}$.*

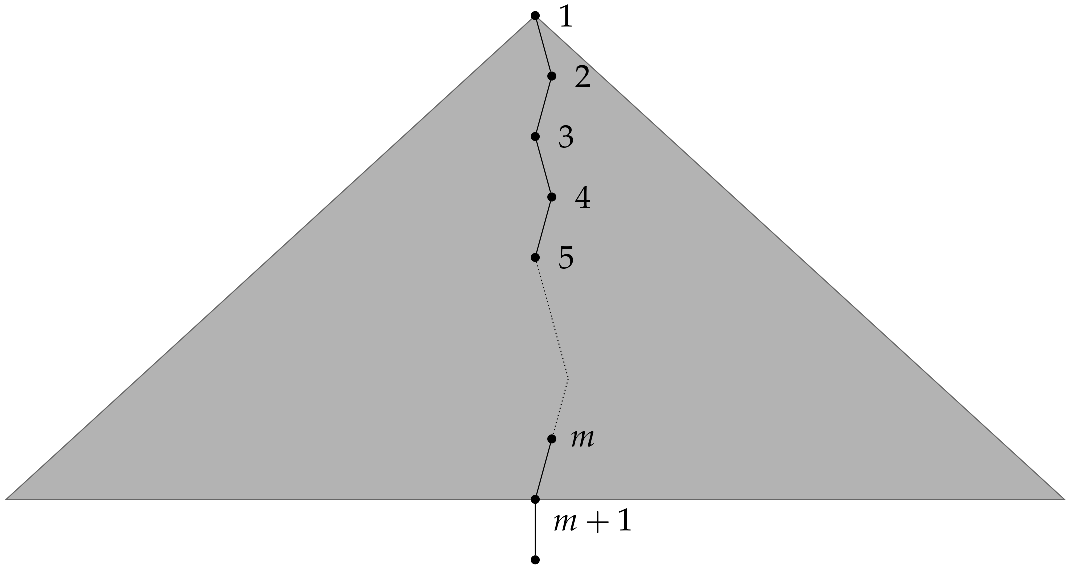


Figure 10.1: At least one path from the root to a leaf in a CNF parse tree for a string of length 2^m or more must have $m+1$ or more variable nodes. If this were not so, the total number of variable nodes (which are collectively represented by the shaded region) would be at most $2^m - 1$, contradicting 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

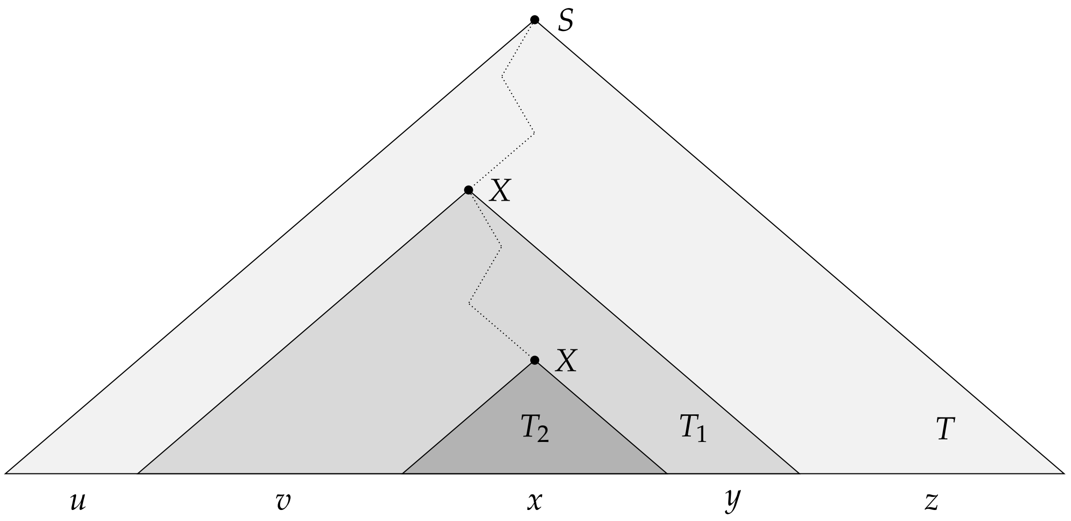


Figure 10.2: An illustration of the subtrees T_1 and T_2 of T.

variable (let us call it X) must appear at least twice within the $m+1$ variable nodes 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, with T_2 being the smaller of these two trees. 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.

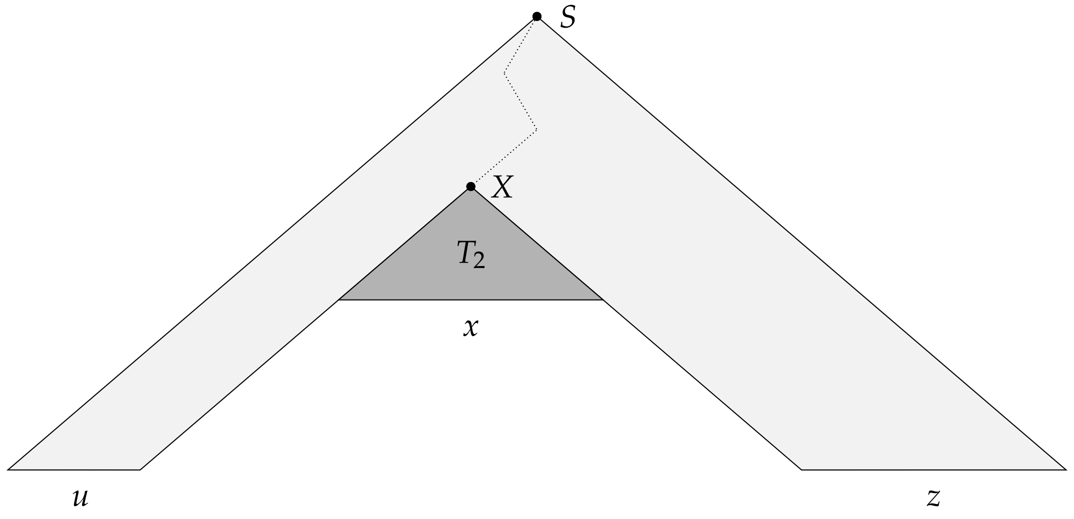


Figure 10.3: By replacing the subtree T_1 by the subtree T_2 in T, a parse tree for the string $uxz = uv^0xy^0z$ is obtained.

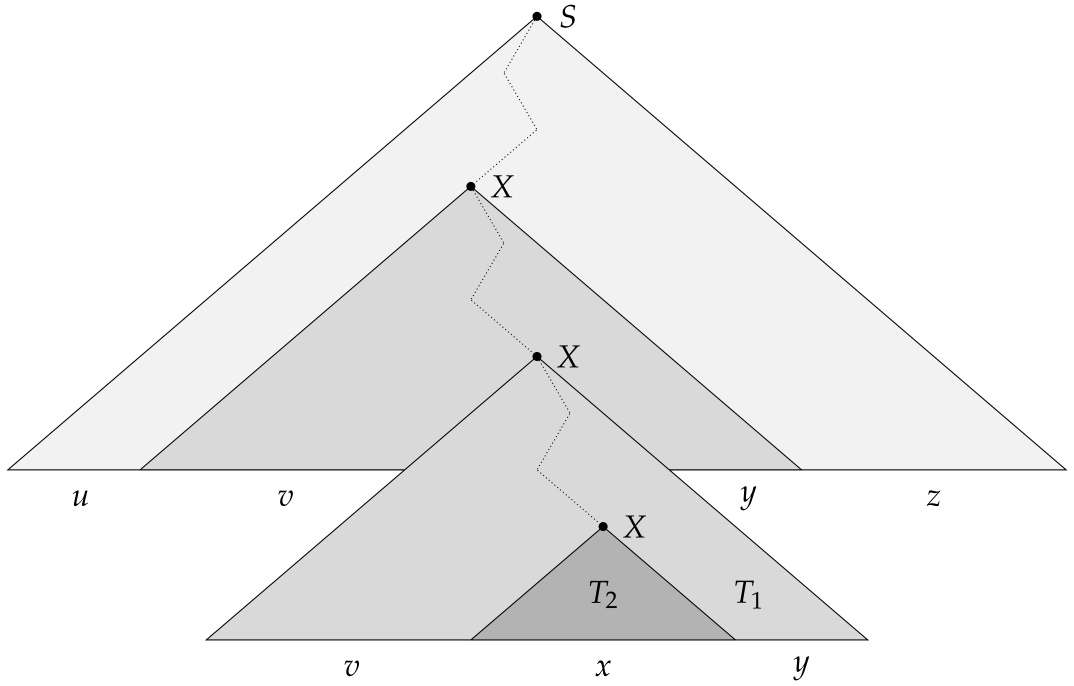


Figure 10.4: By replacing the subtree T_2 by the subtree T_1 in T, a parse tree for the string uv^2xy^2z is obtained. By repeatedly replacing T_2 with T_1 in this way, a parse tree for the string uv^ixy^iz is obtained for any positive integer $i \geq 2$.

Next, the fact that $vy \neq \varepsilon$ follows from the fact that every parse tree for a string 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, we have $|vxy| \leq n$ because the subtree T_1 has at most $2^m = n$ leaf nodes, as was already 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 for A. We will fix such a pumping length n for the remainder of the proof.

Let

$$w = 0^n 1^n 2^n. \tag{10.2}$$

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) $uv^ixy^iz \in A$ 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

$$uv^0xy^0z = uxz \tag{10.3}$$

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

$$|uxz|_1 < |uxz|_0 \quad \text{or} \quad |uxz|_2 < |uxz|_0, \tag{10.4}$$

and if the symbol 2 does not appear in vy, then it must be that either

$$|uxz|_0 < |uxz|_2 \quad \text{or} \quad |uxz|_1 < |uxz|_2. \tag{10.5}$$

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. □

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 recall the language*

$$\text{SQUARE} = \left\{0^{m^2} \,:\, m \in \mathbb{N}\right\} \tag{10.6}$$

defined in Lecture 5. The language SQUARE *is not context free.*

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

Define

$$w = 0^{n^2}. \tag{10.7}$$

We see that $w \in$ SQUARE 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. $uv^ixy^iz \in$ SQUARE 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 $1 \leq k \leq n$. Observe that

$$uv^ixy^iz = 0^{n^2+(i-1)k} \tag{10.8}$$

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

$$uv^2xy^2z = 0^{n^2+k}. \tag{10.9}$$

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

$$(n+1)^2 = n^2 + 2n + 1, \tag{10.10}$$

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

Having obtained a contradiction, we conclude that SQUARE is not context free, as claimed. □

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 SQUARE is nonregular, we obtain a different proof that SQUARE 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 B over Σ as follows:*

$$B = \{r\#s \,:\, r, s \in \{0,1\}^*,\ r \text{ is a substring of } s\}. \tag{10.11}$$

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 exists a pumping length n for B. We will fix such a pumping length n for the remainder of the proof.

Let

$$w = 0^n 1^n \# 0^n 1^n. \tag{10.12}$$

It is the case that $w \in B$ (because 0^n1^n is a substring of itself) and $|w| = 4n + 1 \geq n$. The pumping lemma therefore guarantees that there 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) $uv^ixy^iz \in B$ 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

$$uv^0xy^0z = 0^n1^n\#0^{n-j}1^{n-k} \tag{10.13}$$

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 (10.13) is not contained in B, 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 B. This is in contradiction with the third property, which guarantees that $uv^0xy^0z \in B$, 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

$$uv^2xy^2z = 0^n1^{n+j}\#0^{n+k}1^n, \tag{10.14}$$

which is not in B 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

$$uv^0xy^0z = 0^n1^{n-j}\#0^{n-k}1^n \tag{10.15}$$

in this case, which is not contained in B 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 B 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

$$uv^2xy^2z = r\#0^n1^n \tag{10.16}$$

for some string r that has length strictly larger than $2n$. The string (10.16) is not contained in B, 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 B 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:

$$\begin{aligned} A &= \{0^n 1^n 2^m \,:\, n, m \in \mathbb{N}\}, \\ B &= \{0^n 1^m 2^m \,:\, n, m \in \mathbb{N}\}. \end{aligned} \tag{10.17}$$

These are certainly context-free languages—a CFG generating A is given by

$$\begin{aligned} S &\rightarrow X\,Y \\ X &\rightarrow 0\,X\,1 \mid \varepsilon \\ Y &\rightarrow 2\,Y \mid \varepsilon \end{aligned} \tag{10.18}$$

and a CFG generating B is given by

$$\begin{aligned} S &\rightarrow X\,Y \\ X &\rightarrow 0\,X \mid \varepsilon \\ Y &\rightarrow 1\,Y\,2 \mid \varepsilon \end{aligned} \tag{10.19}$$

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

$$D = \{w \in \{0,1,2\}^* \,:\, |w|_0 = |w|_1 = |w|_2\}. \tag{10.20}$$

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,

$$D \cap \mathrm{L}(0^*1^*2^*) = \big\{0^m 1^m 2^m \,:\, m \in \mathbb{N}\big\}, \tag{10.21}$$

which we already know is not context free. Having obtained a contradiction, we conclude that D is not context free, as required.

Lecture 11

Pushdown automata

This is the last lecture of the course devoted to context-free languages. We will, however, refer to context-free languages from time to time throughout the remainder of the course, just as for regular languages.

The first part of the lecture focuses on the pushdown automata model of computation, which provides an alternative characterization of context-free languages to the definition based on CFGs. The second part of the lecture is devoted to some further properties of context-free languages that we have not discussed thus far, and that happen to be useful for understanding pushdown automata.

11.1 Pushdown automata

The *pushdown automaton* (or PDA) model of computation is essentially what you get if you equip an NFA with a stack. As we shall see, the class of languages recognized by PDAs is precisely the class of context-free languages, which provides a useful tool for reasoning about this class of languages.

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 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

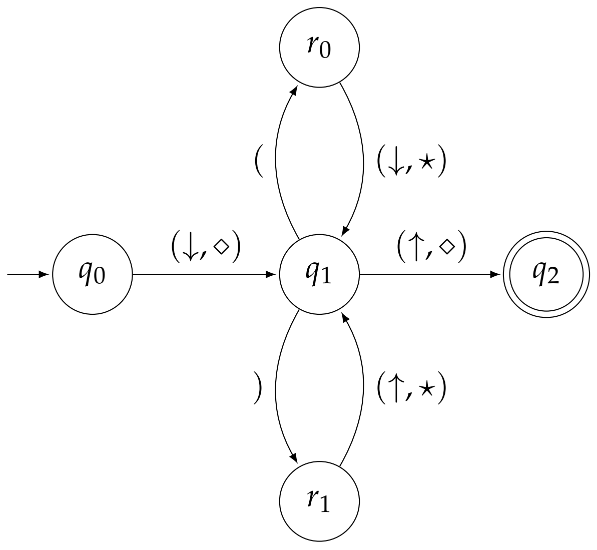


Figure 11.1: The state diagram of a PDA recognizing BAL.

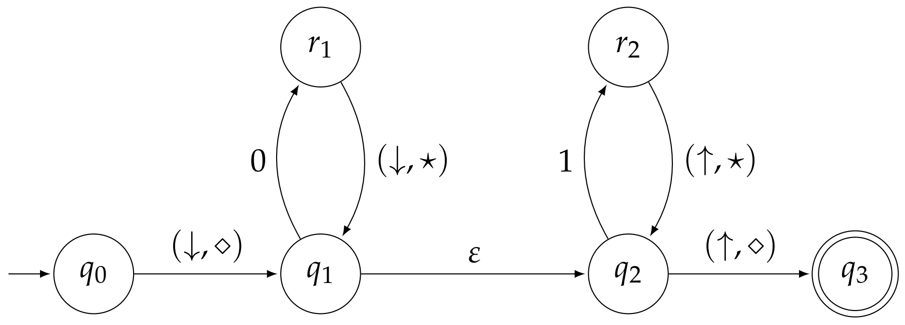


Figure 11.2: The state diagram of a PDA recognizing SAME.

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 $\star$ onto the stack (in the case that we just read a left-parenthesis) or pop the symbol $\star$ 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 on the top of the stack.

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 pushes and 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 recognizes the language

$$\text{SAME} = \{0^n 1^n \,:\, n \in \mathbb{N}\}. \tag{11.1}$$

In this case the stack is essentially used as a counter: we push a star for every 0, 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 alter the form of the transition function so that it specifies how the stack operates.

Before we get to the definition, let us introduce some notation that will be useful for discussing stack operations. For any alphabet Γ, which we will refer to as the *stack alphabet*, the *stack operation alphabet* $\updownarrow\Gamma$ is defined as

$$\updownarrow\Gamma = \{\downarrow, \uparrow\} \times \Gamma. \tag{11.2}$$

The alphabet $\updownarrow\Gamma$ 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.

Definition 11.1. A *pushdown automaton* (or PDA for short) is 6-tuple

$$P = (Q, \Sigma, \Gamma, \delta, q_0, F) \tag{11.3}$$

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*), which must satisfy $\Sigma \cap \updownarrow\Gamma = \varnothing$, δ is a function of the form

$$\delta : Q \times \big(\Sigma \cup \updownarrow\Gamma \cup \{\varepsilon\}\big) \to \mathcal{P}(Q), \tag{11.4}$$

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

The way to interpret a transition function having the above form is that the set of possible labels on transitions is $\Sigma \cup \updownarrow\Gamma \cup \{\varepsilon\}$; we can either read a symbol a, 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 any stack alphabet Γ, and let the stack operation alphabet $\updownarrow\Gamma$ be as defined previously.

We can view a string $v \in (\updownarrow\Gamma)^*$ as either representing or failing to represent a valid sequence of stack operations, assuming we read it from left to right and imagine starting 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:

$$\begin{gathered}(\downarrow,0)(\downarrow,1)(\uparrow,1)(\downarrow,0)(\uparrow,0)(\uparrow,0),\\ (\downarrow,0)(\downarrow,1)(\uparrow,1)(\downarrow,0)(\uparrow,0).\end{gathered} \tag{11.5}$$

In the first case the stack is transformed like this (where the left-most symbol represents the top of the stack):

$$\varepsilon \rightarrow 0 \rightarrow 10 \rightarrow 0 \rightarrow 00 \rightarrow 0 \rightarrow \varepsilon. \tag{11.6}$$

The second case is similar, except that we do not leave the stack empty at the end:

$$\varepsilon \rightarrow 0 \rightarrow 10 \rightarrow 0 \rightarrow 00 \rightarrow 0. \tag{11.7}$$

On the other hand, these strings are invalid stack strings:

$$\begin{gathered}(\downarrow,0)(\downarrow,1)(\uparrow,0)(\downarrow,0)(\uparrow,1)(\uparrow,0),\\ (\downarrow,0)(\downarrow,1)(\uparrow,1)(\downarrow,0)(\uparrow,0)(\uparrow,0)(\uparrow,1).\end{gathered} \tag{11.8}$$

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 1 even through the stack is empty.

It is the case that the language over the alphabet $\updownarrow\Gamma$ 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 (11.5) 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

$$S \to (\downarrow, a)\, S\, (\uparrow, a)\, S \tag{11.9}$$

for every symbol $a \in \Gamma$, as well as the rule $S \to \varepsilon$. 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

$$a_1, \ldots, a_m \in \Sigma \cup \updownarrow\Gamma \cup \{\varepsilon\} \tag{11.10}$$

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 $\updownarrow\Gamma$ from $a_1 \cdots a_m$, the input string w is obtained.
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 how stacks work, and this is represented by the fourth property.

As you would expect, for a given PDA P, we let $\mathrm{L}(P)$ denote the *language recognized* by P, which is the language of all strings accepted by P.

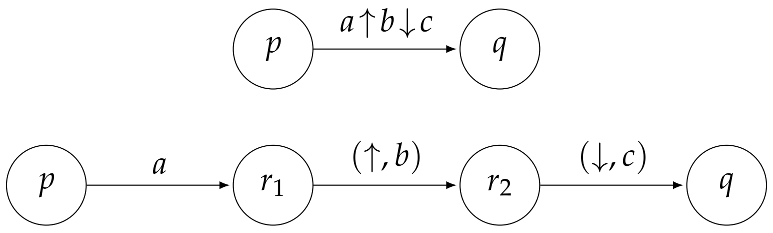


Figure 11.3: The shorthand notation for PDAs appears on the top, and the actual PDA states and transitions represented by this shorthand notation appears on the bottom.

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

$$a{\uparrow}b{\downarrow}c, \tag{11.11}$$

the meaning is that the symbol a is read, b is popped off of the stack, and then c 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

$$a{\uparrow}b_1b_2b_3{\downarrow}c_1c_2c_3c_4 \tag{11.12}$$

means that a is read from the input, $b_1b_2b_3$ is popped off the top of the stack, and then $c_1c_2c_3c_4$ 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 b_1 on the top of the stack, b_2 next on the stack, and b_3 third on the stack—and when the entire operation is done, c_1 is on top of the stack, c_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

$$a{\uparrow}b \tag{11.13}$$

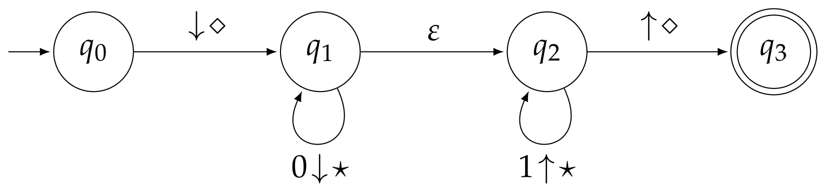


Figure 11.4: The state diagram of a PDA for SAME.

means "read a from the input, pop a off of the stack, and push nothing," the transition label

$$\uparrow b \downarrow c_1 c_2 \tag{11.14}$$

means "read nothing from the input, pop b off of the stack, and push c_1c_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. One 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 fact, 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.

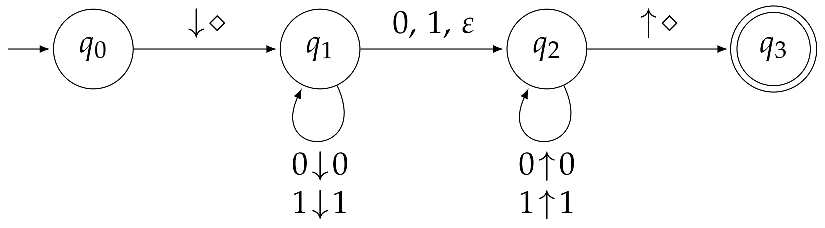


Figure 11.5: A PDA recognizing the language PAL.

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 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, because

$$A \,\triangle\, \Sigma^* = \overline{A}. \tag{11.15}$$

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 these strings. 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 language $A \triangle B$ 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 $\overline{A} \cap B$ is context free.

Finally, given that we have proved that both $A \cap \overline{B}$ and $\overline{A} \cap B$ are context free, it follows that $A \triangle B = (A \cap \overline{B}) \cup (\overline{A} \cap B)$ 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. We call this operation the *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 project all of the strings in A onto the alphabet Σ, you are 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*

$$B = \left\{ w \in \Sigma^* \;:\; \begin{array}{l}\text{there exists a string } x \in A \text{ such that } w \text{ is}\\ \text{obtained from } x \text{ by deleting all symbols in } \Gamma\end{array} \right\}. \tag{11.16}$$

The language 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 $X \to YZ$ appearing in G, include the same rule in H. Also, if the rule $S \to \varepsilon$ appears in G, include this rule in H as well.
2. For every rule of the form $X \to a$ in G, where $a \in \Sigma$, include the same rule $X \to a$ in H.

3. For every rule of the form $X \rightarrow b$ in G, where $b \in \Gamma$, 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 when they are projected onto the alphabet Σ, 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*

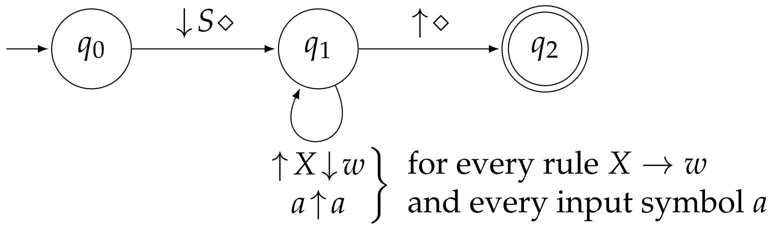
$$B = \left\{ x \in (\Sigma \cup \Gamma)^* \;:\; \begin{array}{l} \textit{the string } w \textit{ obtained from } x \textit{ by deleting} \\ \textit{all symbols in } \Gamma \textit{ satisfies } w \in A \end{array} \right\}. \tag{11.17}$$

The language 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
$$W \rightarrow bW \tag{11.18}$$
in H for each $b \in \Gamma$, as well as the rule $W \rightarrow \varepsilon$, for a new variable W 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 $X \rightarrow YZ$ in G, include the same rule in H without modifying it.
3. For each rule of the form $X \rightarrow a$ in G, include this rule in H:
$$X \rightarrow WaW \tag{11.19}$$
4. If the rule $S \rightarrow \varepsilon$ is contained in G, then include this rule in H:
$$S \rightarrow W \tag{11.20}$$

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. We have that $\mathrm{L}(H) = B$, and therefore B is context free. $\square$



q_0
$\downarrow S\diamond$
q_1
$\uparrow\diamond$
q_2
$\uparrow X \downarrow w$
$a \uparrow a$
for every rule $X \rightarrow w$
and every input symbol a

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. 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

$$G = (V, \Sigma, R, S) \tag{11.21}$$

is a context-free grammar, 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 provides a way to store the symbols and variables needed to carry out a derivation with respect to 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 have 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 would not be all that different from the method just mentioned, but we will 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

$$\delta : Q \times \big(\Sigma \cup \updownarrow\Gamma \cup \{\varepsilon\}\big) \rightarrow \mathcal{P}(Q), \tag{11.22}$$

so if we wanted to, we could think of P as being an NFA for some language over the alphabet $\Sigma \cup \updownarrow\Gamma$. Slightly more formally, let N be the NFA defined as

$$N = \big(Q, \Sigma \cup \updownarrow\Gamma, \delta, q_0, F\big); \tag{11.23}$$

we do not even need to change the transition function because it already has the right form of a transition function for an NFA over the alphabet $\Sigma \cup \updownarrow\Gamma$. Also define $B = \mathrm{L}(N) \subseteq (\Sigma \cup \updownarrow\Gamma)^*$ to be the language recognized by N. In general, the strings in B include symbols in both Σ and $\updownarrow\Gamma$. Even though symbols in $\updownarrow\Gamma$ 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 does not have a stack with which to check this condition.

Now let us consider a second language $C \subseteq (\Sigma \cup \updownarrow\Gamma)^*$. This will be the language consisting of all strings over the alphabet $\Sigma \cup \updownarrow\Gamma$ 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 $\updownarrow\Gamma$ 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 $\updownarrow\Gamma$ 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

Turing machines

In this lecture we will discuss the *Turing machine* model of computation. This model is named after Alan Turing (1912–1954), who proposed it in 1936. It is difficult to overstate Alan Turing's influence on the subject of this course—theoretical computer science effectively started with Turing's work, and for this reason he is sometimes referred to as the father of theoretical computer science.

The Church–Turing thesis

The intention of the Turing machine model is to provide a simple mathematical abstraction of general computations. 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, but understand that 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.

Note that 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 to 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.

There are alternative models of computation that offer abstractions of general computations. One example is λ-calculus, which was proposed by Alonzo Church a short time prior to Turing's introduction of what we now call Turing machines. These two models, Turing machines and λ-calculus, are equivalent in the sense that any computation that can be performed by one of them can be performed by the other. Turing sketched a proof of this fact in his 1936 paper. We will see another example in Lecture 14 when we show that a model called the *stack machine* model is equivalent to the Turing machine model. Based on the stack machine model, or directly on the Turing machine model, it is not conceptually difficult to envision the simulation of a model of computation abstracting the notion of a *random access machine*.

While machines behaving like Turing machines have been built, this is mainly a recreational activity. The Turing machine model was never intended to serve as a practical approach to performing computations, but rather was intended to provide a rigorous mathematical foundation for reasoning about computation—and it has served this purpose very well since its introduction.

12.1 Definition of the Turing machine model

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.
2. 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.
3. The *tape head*. The tape head can move left and right on the tape, and is understood to be scanning exactly one of the tape squares at the start of each computational step. The tape head can read which symbol is stored in the square it scans, and it can write a new symbol into that square.

Figure 12.1 illustrates these three components. It is natural to imagine that the tape head is connected in some way to the finite state control.

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 the single symbol stored in 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 in the tape square being scanned, and 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 variant of the model.

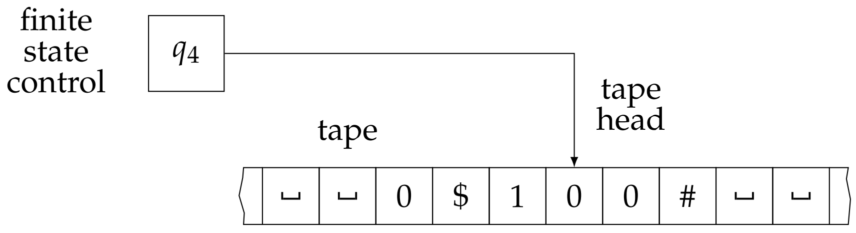


Figure 12.1: An illustration of the three components of a Turing machine: the finite state control, the tape head, and the tape. The tape is infinite in both directions (although it appears that this Turing machine's tape was torn at both ends to fit it into the figure).

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. We need an actual symbol to represent the blank symbol in these notes, and we will use the symbol ␣ 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.

We also give Turing machines an opportunity to stop the computational process and produce an output by requiring them to have two special states: an *accept* state q_{acc} and a *reject* state q_{rej}. These two states are deemed *halting states*, and all other states are *non-halting states*. If the machine enters a halting state, the computation immediately stops and *accepts* or *rejects* accordingly. When we discuss language recognition, our focus is naturally on whether or not a given Turing machine eventually reaches one of the states q_{acc} or q_{rej}, but we can also use the Turing machine model to describe the computation of functions by taking into account the contents of the tape if and when a halting state is reached.

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 ␣, Γ is an alphabet called the

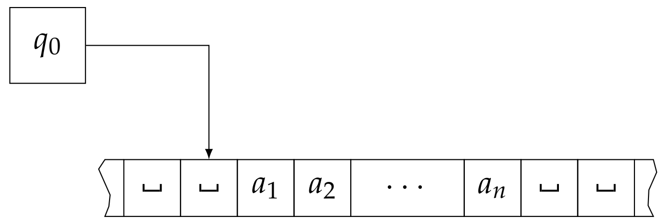


Figure 12.2: The initial configuration of a DTM when run on input $w = a_1 a_2 \cdots a_n$.

tape alphabet, which must satisfy $\Sigma \cup \{␣\} \subseteq \Gamma$, 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 must 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 $p \in Q$, the symbol stored in the tape square being scanned by the tape head is $a \in \Gamma$, and it is the case that

$$\delta(p, a) = (q, b, D) \tag{12.3}$$

for $D \in \{\leftarrow, \rightarrow\}$. The action performed by the DTM is then to

1. change state to q,
2. overwrite the contents of the tape square being scanned by the tape head with b, and
3. 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 halts once it reaches one of these two states.

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 every 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. Thus, 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 it might.

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. This includes

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, like 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

$$u(q, a)v \tag{12.4}$$

for some state $q \in Q$, a tape symbol $a \in \Gamma$, and (possibly empty) strings of tape symbols u and v such that

$$u \in \Gamma^* \backslash \{\sqcup\}\Gamma^* \quad \text{and} \quad v \in \Gamma^* \backslash \Gamma^*\{\sqcup\}. \tag{12.5}$$

In words, u and v are strings of tape symbols, u does not start with a blank, and v does not end with a blank. The interpretation of the expression (12.4) is that it refers to the configuration in which 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

$$0\$1(q_4, 0)0\# \tag{12.6}$$

while the configuration of the DTM in Figure 12.2 is

$$(q_0, \sqcup)w \tag{12.7}$$

for $w = a_1 \cdots a_n$.

When working with descriptions of configurations, it is convenient to define a few functions as follows. We define $\alpha : \Gamma^* \rightarrow \Gamma^* \backslash \{\sqcup\}\Gamma^*$ and $\beta : \Gamma^* \rightarrow \Gamma^* \backslash \Gamma^*\{\sqcup\}$ recursively as

$$\begin{aligned} \alpha(w) &= w && (\text{for } w \in \Gamma^* \backslash \{\sqcup\}\Gamma^*) \\ \alpha(\sqcup w) &= \alpha(w) && (\text{for } w \in \Gamma^*) \end{aligned} \tag{12.8}$$

and

$$\begin{aligned} \beta(w) &= w && (\text{for } w \in \Gamma^* \backslash \Gamma^*\{\sqcup\}) \\ \beta(w\sqcup) &= \beta(w) && (\text{for } w \in \Gamma^*), \end{aligned} \tag{12.9}$$

and we define

$$\gamma : \Gamma^*(Q \times \Gamma)\Gamma^* \rightarrow \big(\Gamma^* \backslash \{\sqcup\}\Gamma^*\big)(Q \times \Gamma)\big(\Gamma^* \backslash \Gamma^*\{\sqcup\}\big) \tag{12.10}$$

as

$$\gamma(u(q, a)v) = \alpha(u)(q, a)\beta(v) \tag{12.11}$$

for all $u, v \in \Gamma^*$, $q \in Q$, and $a \in \Gamma$. This is not as complicated as it might appear: 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

In order to formally define what it means for a DTM to accept, reject, compute a function, and so on, we will define a *yields relation*, similar to what we did for context-free grammars.

Definition 12.2. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ be a DTM. The *yields* relation $\vdash_M$, defined on pairs of configurations of M, includes exactly these pairs:

1. *Moving right.* For every choice of $p \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

$$\delta(p, a) = (q, b, \rightarrow), \tag{12.12}$$

the yields relation includes these pairs for all $u \in \Gamma^*\backslash\{\sqcup\}\Gamma^*$, $v \in \Gamma^*\backslash\Gamma^*\{\sqcup\}$, and $c \in \Gamma$:

$$\begin{aligned} u(p,a)cv &\vdash_M \gamma\big(ub(q,c)v\big) \\ u(p,a) &\vdash_M \gamma\big(ub(q,\sqcup)\big). \end{aligned} \tag{12.13}$$

2. *Moving left.* For every choice of $p \in Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\}$, $q \in Q$, and $a, b \in \Gamma$ satisfying

$$\delta(p,a) = (q, b, \leftarrow), \tag{12.14}$$

the yields relation includes these pairs for all $u \in \Gamma^*\backslash\{\sqcup\}\Gamma^*$, $v \in \Gamma^*\backslash\Gamma^*\{\sqcup\}$, and $c \in \Gamma$:

$$\begin{aligned} uc(p,a)v &\vdash_M \gamma\big(u(q,c)bv\big) \\ (p,a)v &\vdash_M \gamma\big((q,\sqcup)bv\big). \end{aligned} \tag{12.15}$$

We also let $\vdash_M^*$ denote the reflexive, transitive closure of $\vdash_M$. That is, we have

$$u(p,a)v \vdash_M^* y(q,b)z \tag{12.16}$$

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

$$w_k(r_k, c_k)x_k \vdash_M w_{k+1}(r_{k+1}, c_{k+1})x_{k+1} \tag{12.17}$$

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

A more intuitive description of these relations is that the expression

$$u(p,a)v \vdash_M y(q,b)z \tag{12.18}$$

means that by running M for one step we move from the configuration $u(p,a)v$ to the configuration $y(q,b)z$; and

$$u(p,a)v \vdash_M^* y(q,b)z \tag{12.19}$$

means that by running M for some number of steps, possibly zero steps, we will move from the configuration $u(p,a)v$ to the configuration $y(q,b)z$.

12.2 Semidecidable and decidable languages; computable functions

Now we will define the classes of *semidecidable* and *decidable* languages as well as the notion of a *computable function*.

To define the classes of decidable and semidecidable languages, we must first express formally, in terms of the yields relation defined in the previous section, what it means for a DTM to accept or reject.

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

$$(q_0, \sqcup)w \vdash_M^* u(q_{\text{acc}}, a)v, \tag{12.20}$$

then M *accepts* w. If there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_0, \sqcup)w \vdash_M^* u(q_{\text{rej}}, a)v, \tag{12.21}$$

then M *rejects* w. If neither of these conditions hold, then M *runs forever* on input w.

In words, if a DTM is set in its initial configuration, for some input string w, and starts running, it *accepts* w if it eventually enters its accept state, it *rejects* w if it eventually enters its reject state, and it *run forever* if neither of these possibilities holds. It is perhaps obvious, but nevertheless worth noting, that accepting and rejecting are mutually exclusive—because DTMs are deterministic, each configuration has a unique next configuration, and it follows that a DTM M cannot simultaneously accept a string w and reject w.

Similar to what we have done for other computational models, we write $\text{L}(M)$ to denote the language of all strings that are accepted by a DTM M. In the case of DTMs, the language $\text{L}(M)$ does not really tell the whole story—a string $w \notin \text{L}(M)$ might either be rejected or it may cause M to run forever—but the notation is useful nevertheless.

We now define the class of semidecidable languages to be those languages recognized by some DTM.

Definition 12.4. Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language A is *semidecidable* if there exists a DTM M such that $A = \text{L}(M)$.

The name *semidecidable* reflects the fact that if $A = \text{L}(M)$ for some DTM M, and $w \in A$, then running M on w will necessarily lead to acceptance; but if $w \notin A$, then M might either reject or run forever on input w. That is, M does not really decide whether a string w is in A or not, it only "semidecides"; for if $w \notin A$, you might never learn this with certainty as a result of running M on w. There are several alternative names that people often use in place of semidecidable, including *Turing recognizable*, *partially decidable*, and *recursively enumerable* (or *r.e.* for short).

Next, as the following definition makes clear, we define the class of *decidable* languages to be those languages for which there exists a DTM that correctly answers whether or not a given string is in the language, never running forever.

Definition 12.5. Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language A is *decidable* if there exists a DTM M with these two properties:

1. M accepts every string $w \in A$.
2. M rejects every string $w \in \overline{A}$.

The names *recursive* and *computable* are sometimes used in place of *decidable*.

Finally, let us define what it means for a function to be *computable*. We do this for functions mapping strings to strings, but not necessarily having the same input and output alphabets, as this generality will be important in future lectures.

Definition 12.6. Let Σ and Γ be alphabets and let $f : \Sigma^* \to \Gamma^*$ be a function. The function f is *computable* if there exists a DTM $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ such that the relation

$$(q_0, \sqcup)w \vdash_M^* (q_{\text{acc}}, \sqcup)f(w) \tag{12.22}$$

holds for every string $w \in \Sigma^*$. In this case we also say that DTM M *computes* f.

In this definition, Δ can be any tape alphabet; by definition it must include all of the symbols in the input alphabet Σ, and it must also include all symbols from the alphabet Γ that appear in any output string in order for the relation (12.22) to hold.

In words, a function is computed by a DTM if, when run on any choice of an input string to that function, it eventually accepts, leaving the correct output written on the tape surrounded by blanks, with the tape head one square left of this output string.

12.3 A simple example of a Turing machine

Let us now see an example of a DTM, which we will describe using a state diagram. In the DTM case, we can represent the property that the transition function satisfies $\delta(p, a) = (q, b, \rightarrow)$ with a transition of the form

$$\begin{tikzcd} p \arrow[r, "{a, b \rightarrow}"] & q \end{tikzcd}$$

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 recognizes the language

$$\text{SAME} = \{0^n 1^n : n \in \mathbb{N}\}. \tag{12.23}$$

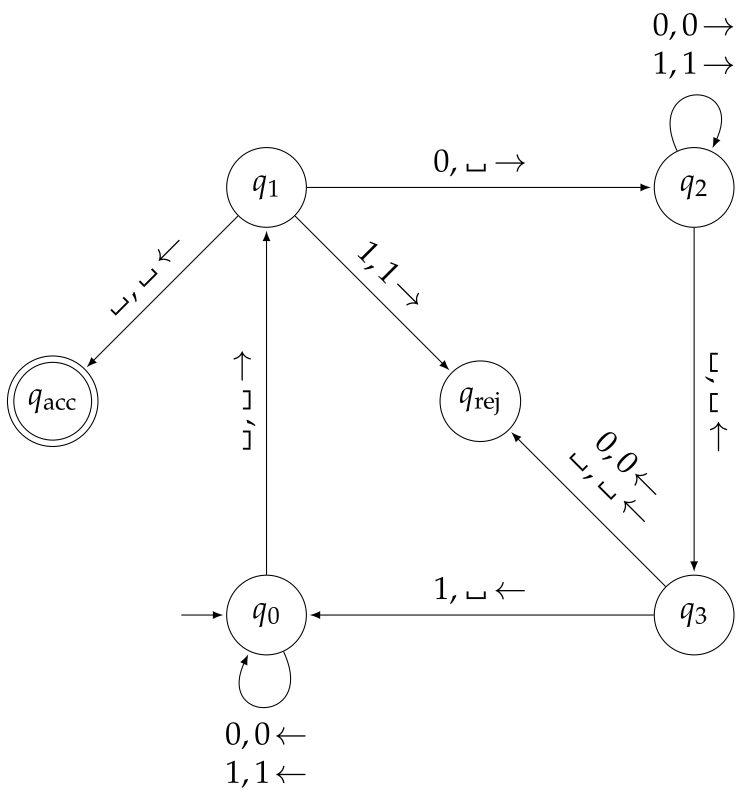


Figure 12.3: A DTM M for the language SAME $= \{0^n 1^n : n \in \mathbb{N}\}$.

To be more precise, M accepts every string in SAME and rejects every string in $\overline{\text{SAME}}$, so never does the DTM run forever. Thus, SAME 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 $0^n 1^n$ 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 SAME. 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 were not 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 does not tell you precisely how the DTM works—but if you did not 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 is enough to convey the basic idea of how this DTM operates. We could, for instance, 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 step 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.

The DTM M being discussed is 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 rarely employ them. Doing so would be analogous to describing a complex program using machine language instructions.

The more usual way to describe DTMs is in terms of *algorithms*, often expressed in the form of pseudo-code or high-level descriptions like the last description of M above. It may not be immediately apparent precisely which high-level algorithm descriptions can be run on Turing machines, but as an intuitive guide one may have confidence that if an algorithm can be implemented using your favorite programming language, then it can also be run on a deterministic Turing machine. The discussions in the two lectures to follow this one are primarily intended to help to build this intuition.

Lecture 13

Variants of Turing machines

In this lecture we will continue to discuss the Turing machine model, focusing on ways in which the model can be changed without affecting its power.

13.1 Simple 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 are 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 simply 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 (perhaps more 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 defined in the previous lecture 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 rules that define the yields relation for ordinary DTMs:

3. *Remaining stationary.* 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 does not 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

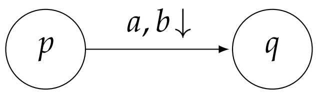
$$\eta(p,a) = \begin{cases} (q, b, \leftarrow) & \text{if } \delta(p,a) = (q, b, \leftarrow) \\ (q, b, \rightarrow) & \text{if } \delta(p,a) = (q, b, \rightarrow) \\ (q', b, \leftarrow) & \text{if } \delta(p,a) = (q, b, \downarrow) \end{cases} \tag{13.8}$$

   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:

$$\begin{tikzcd} p \arrow[r, "a{,}\,b\leftarrow"] & q' \arrow[r, "c{,}\,c\rightarrow"] & q \end{tikzcd}$$

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 decided or semidecided by a stationary-head-DTM is also decided or semidecided 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 for us to be concerned with it—we went through this equivalence in detail only because it is a first example, and 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 k different tracks to have possibly different tape alphabets $\Gamma_1, \ldots, \Gamma_k$. 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.

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 $\Gamma_1 = \{0, 1, \sqcup\}$, the second track stores symbols from the tape alphabet $\Gamma_2 = \{\#, \sqcup\}$, and the third track stores symbols from the tape alphabet $\Gamma_3 = \{\clubsuit, \heartsuit, \diamondsuit, \spadesuit, \sqcup\}$.

It turns out that this is not really even a variant of the DTM definition at all—it is just an ordinary DTM whose tape alphabet Γ is equal to the Cartesian product

$$\Gamma = \Gamma_1 \times \cdots \times \Gamma_k. \tag{13.10}$$

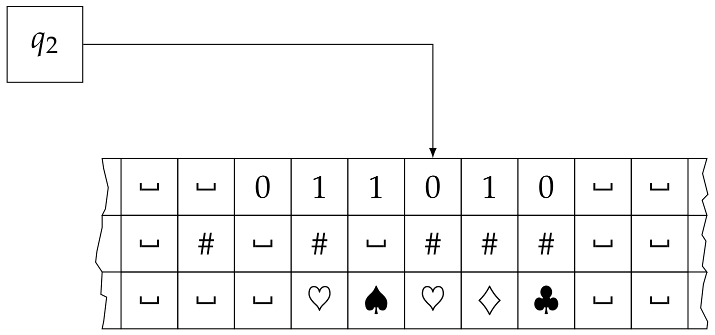


Figure 13.1: A DTM with a three-track tape.

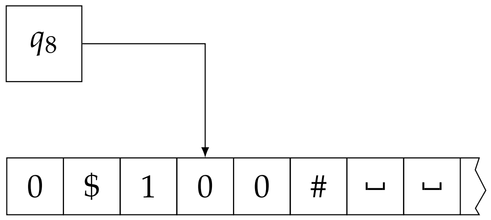


Figure 13.2: A DTM with a one-way infinite tape.

The tape alphabet of any DTM must include the input alphabet and a blank symbol, and so it should be understood that we identify each input 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.

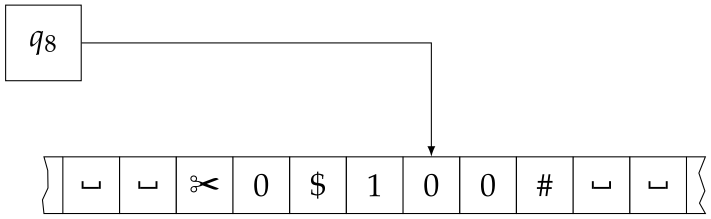


Figure 13.3: A DTM with a two-way infinite tape can easily simulate a DTM with a one-way infinite tape, like the one pictured in Figure 13.2, by writing a special symbol on the tape (in this case the symbol is ✂) that indicates where we should imagine the tape has been cut. When the tape head scans this symbol, the DTM adjusts its behavior accordingly.

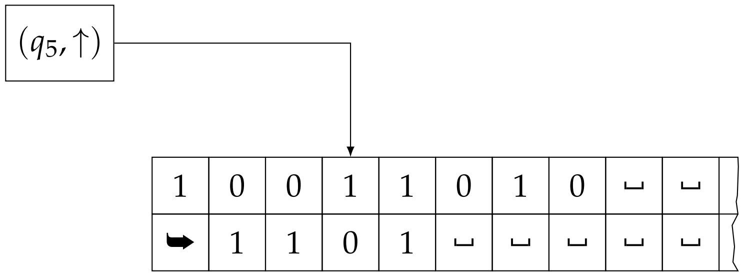


Figure 13.4: A DTM with a one-way infinite tape that simulates an ordinary DTM having a two-way infinite tape. The top track represents the portion of the two-way infinite tape that extends to the right and the bottom track represents the portion extending to the left.

This exactly mimics the behavior of the DTM with a one-way infinite tape that was suggested above.

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 represents the tape of the original DTM being simulated, 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

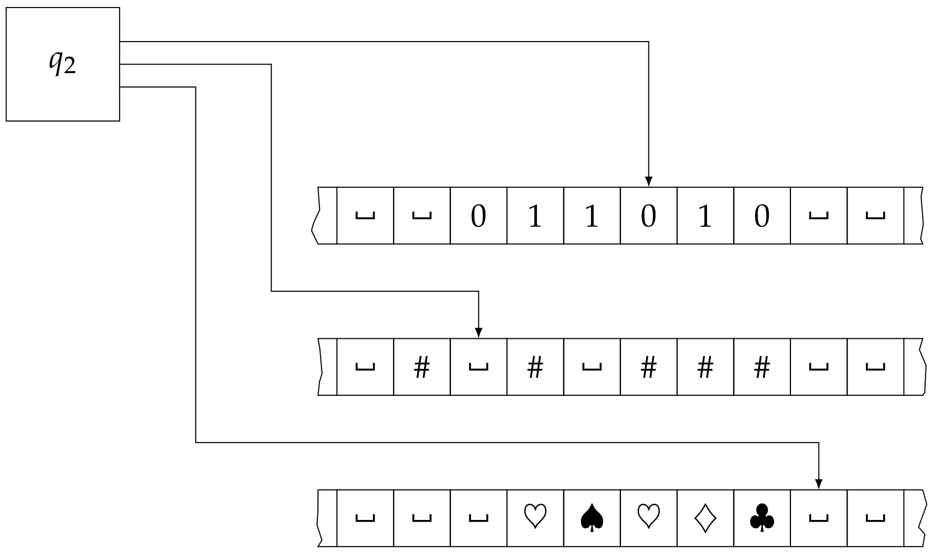


Figure 13.5: A DTM with three tapes.

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 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.

13.2 Multi-tape Turing machines

The last variant of the Turing machine model that we will consider is perhaps the most useful variant. 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-tape 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:

$$\delta : Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma_1 \times \cdots \times \Gamma_k \to Q \times \Gamma_1 \times \cdots \times \Gamma_k \times \{\leftarrow, \downarrow, \rightarrow\}^k. \tag{13.11}$$

If we make the simplifying assumption that the same alphabet is used for each tape (which 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 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

$$\delta(p, a_1, \ldots, a_k) = (q, b_1, \ldots, b_k, D_1, \ldots, D_k), \tag{13.13}$$

then if the DTM is in the state p and is reading the symbols $a_1, \ldots, a_k$ on its k tapes, then

1. the new state becomes q,
2. the symbols $b_1, \ldots, b_k$ are written onto the k tapes (overwriting the symbols $a_1, \ldots, a_k$), and
3. the j-th tape head either moves or remains stationary depending on the value $D_j \in \{\leftarrow, \downarrow, \rightarrow\}$, 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 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.

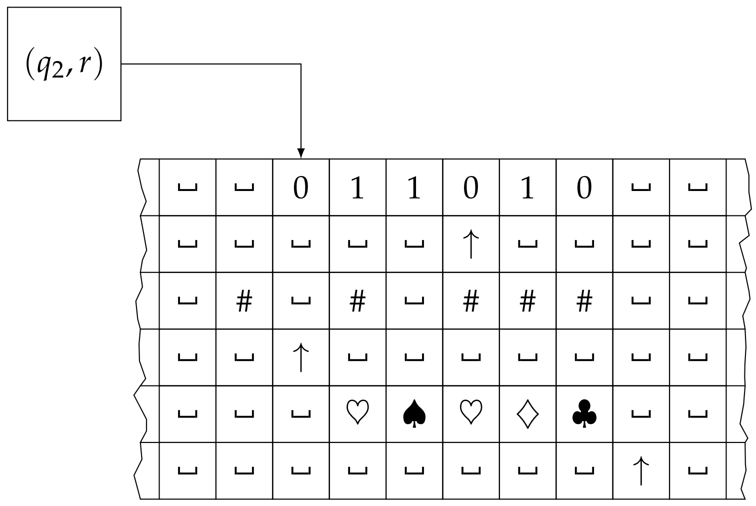


Figure 13.6: A single-tape DTM with a multi-track tape can simulate a multi-tape DTM. Here, the odd numbered tracks represent the contents of the tapes of the DTM illustrated in Figure 13.5, while the even numbered tracks store the locations of the tape heads of the multi-tape DTM. The finite state control 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.

Lecture 14

Stack machines

We will now define a new computational model, the *stack machine model*, and observe that it is equivalent in power to the deterministic Turing machine model. There are two principal motives behind the introduction of the stack machine model at this point in the course:

1. Stack machines are easier to work with than Turing machines, at least at lower levels of detail.

   Whereas formally specifying Turing machines for even the most basic computations is a tedious task prone to errors, the analogous task for stack machines is simpler in comparison. Through the equivalence of the two models, one might therefore be more easily convinced that Turing machines abstract the capabilities of ordinary computers.

2. The equivalence of stack machines and Turing machines provides us with a nice example in support of the Church–Turing thesis.

   The stack machine model is natural and intuitive, particularly within the context of this course and the other models we have studied, and you can certainly imagine building one—or even just emulating one yourself using sheets of paper to implement stacks. That is, they represent mechanical processes. As the Church–Turing thesis predicts, anything you can compute with this model can also be computed with a deterministic Turing machine.

The stack machine model resembles the pushdown automata model, but unlike that model the stack machine model permits the use of *multiple stacks*. This makes all the difference in the world. In fact, as we will see, just two stacks endow a stack machine with universal computational power. Also unlike the pushdown automata model, we will only consider a deterministic variant of the stack machine model (although one can easily define nondeterministic stack machines).

14.1 Definition of stack machines

In this section we will formally define the deterministic stack machine model, but before proceeding to the definition it will be helpful to first mention a few specific points about the model.

1. As was already suggested, this model may be viewed as a deterministic, multi-stack version of a PDA. We may have any number r of stacks, so long as the number is fixed. Formally we do this by defining r-stack deterministic stack machines for every positive integer r.
2. For simplicity we assume that every stack has the same stack alphabet Δ.[1] This alphabet must include a special bottom-of-the-stack symbol $\diamond$, and it must also include the input alphabet Σ (which itself may not include the symbol $\diamond$). We require that the input alphabet is contained in the stack alphabet because the input is assumed to be stored (left-most symbol on top) in the first stack when the computation begins.
3. The state set Q of a stack machine must include a start state q_0 as well as the *halting states* q_{acc} and q_{rej} (which cannot be equal, naturally).
4. Each non-halting state of a stack machine, meaning any element of Q except q_{acc} and q_{rej}, always has exactly one of the r stacks associated with it, and must be either a *push state* or a *pop state*. When the stack machine transitions from a push state to another state, a symbol is always pushed onto the stack associated with that state, and similarly when a stack machine transitions from a pop state to another state, a symbol is popped off of the stack associated with that state (assuming it is non-empty).

Definition 14.1. An r-stack deterministic stack machine (r-DSM, for short) is an 7-tuple

$$M = (Q, \Sigma, \Delta, \delta, q_0, q_{acc}, q_{rej}), \tag{14.1}$$

where

1. Q is a finite and nonempty set of *states*,
2. Σ is an *input alphabet*, which may not include the bottom-of-the-stack symbol $\diamond$,
3. Δ is a *stack alphabet*, which must satisfy $\Sigma \cup \{\diamond\} \subseteq \Delta$,

[1] It would be straightforward to generalize the model to allow for each stack to have a different stack alphabet—we are just opting for a simpler definition.

4. δ is a *transition function* of the form

$$\delta : (Q\backslash\{q_{acc}, q_{rej}\}) \to \big(\{1,\ldots,r\} \times \Delta \times Q\big) \cup \big(\{1,\ldots,r\} \times Q^{\Delta}\big), \tag{14.2}$$

   and

5. $q_0, q_{acc}, q_{rej} \in Q$ are the *initial state*, *accept state*, and *reject state*, respectively, which must satisfy $q_{acc} \neq q_{rej}$.

When we refer to a DSM without specifying the number of stacks r, we simply mean an r-DSM for some choice of a fixed positive integer r. Sometimes we may not care specifically how many stacks a given DSM has, and we leave the number unspecified for simplicity.

In the specification of the transition function δ, the notation Q^{Δ} refers to the set of all functions from Δ to Q. The interpretation of the transition function δ is as follows:

1. If p is a non-halting state and $\delta(p) \in \{1,\ldots,r\} \times \Delta \times Q$, then the state p is a push state; if it is the case that

$$\delta(p) = (k, a, q), \tag{14.3}$$

   then when the machine is in state p, it pushes the symbol a onto stack number k and transitions to state q.

2. If p is a non-halting state and $\delta(p) \in \{1,\ldots,r\} \times Q^{\Delta}$, then the state p is a pop state; if it is the case that

$$\delta(p) = (k, f) \tag{14.4}$$

   for $f : \Delta \to Q$, then when the machine is in state p, it pops whatever symbol $a \in \Delta$ is on the top of stack number k and transitions to the state $f(a)$. Notice specifically that this allows for conditional branching: the state $f(a)$ that the machine transitions to may depend on the symbol a that is popped.

   If it so happens that stack number k is empty in this situation, the machine simply transitions to the state q_{rej}. That is, popping an empty stack immediately causes the machine to reject.

The computation of a stack machine M on an input $x \in \Sigma^*$ begins in the initial state q_0 with the input string $x \in \Sigma^*$ is stored in stack 1. Specifically, the top symbol of stack 1 is the first symbol of x, the second to top symbol of stack 1 contains the second symbol of x, and so on. At the bottom of stack 1, underneath all of the input symbols, is the bottom-of-the-stack symbol $\diamond$. All of the other stacks initially contain just the bottom-of-the-stack symbol $\diamond$.

The computation then continues in the natural way so long as the machine is in a non-halting state. If either of the states q_{acc} or q_{rej} is reached, the computation halts, and the input is accepted or rejected accordingly. Of course, just like a Turing machine, there is also a possibility for computations to carry on indefinitely, and in such a situation we will refer to the machine *running forever*.

State diagrams for DSMs

Deterministic stack machines may be represented by state diagrams in a way that is similar to, but in some ways different from, the other models we have discussed. As usual, states are represented by nodes in a directed graph, directed edges (with labels) represent transitions, and the accept and reject states are labeled as such. You will be able to immediately recognize that a state diagram represents a DSM in these notes from the fact that the nodes are square-shaped (with slightly rounded corners) rather than circle or oval shaped.

In the state diagram of a DSM, the nodes themselves, rather than the transitions, indicate which operation (push or pop) is performed as the machine transitions from a given state, and to which stack the operation refers. Each push state must have a single transition leading from it, with the label indicating which symbol is pushed, with the transition pointing to the next state. Each pop state must have one directed edge leading from it for each possible stack symbol, indicating to which state the computation is to transition (depending on the symbol popped).

We will also commonly assign names like X, Y, and Z to different stacks, rather than calling them *stack 1*, *stack 2*, and so on, as this makes for more natural, algorithmically focused descriptions of stack machines, where we view the stacks as being akin to variables in a computer program.

Figure 14.1 gives an example of a state diagram of a 3-DSM. In this diagram, stack 1 (which stores the input when the computation begins) is named X and the other two stacks are named Y and Z. This DSM accepts every string in the language

$$\{w\#w : w \in \{0,1\}^*\} \tag{14.5}$$

and rejects every string in the complement of this language (over the alphabet $\{0,1,\#\}$).

Two additional comments on state diagrams for DSMs are in order. First, aside from the accept and reject states, we tend not to include the names of individual states in state diagrams. This is because the names we choose for the states are irrelevant to the functioning of a given machine, and omitting them makes for less cluttered diagrams. In rare cases in which it is important to include the name of a state in a state diagram, we will just write the state name above or beside its corresponding node.

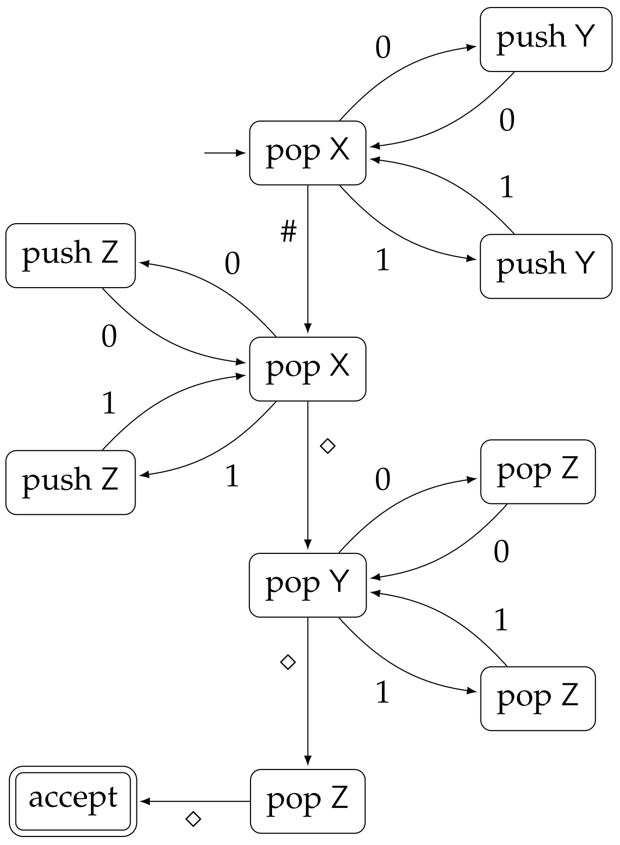


Figure 14.1: A 3-DSM for the language $\{w\#w : w \in \{0,1\}^*\}$.

Second, although every deterministic stack machine is assumed to have a reject state, we often do not bother to include it in state diagrams. Whenever there is a state with which a pop operation is associated, and one or more of the possible stack symbols does not appear on any transition leading out of this state, it is assumed that the "missing" transitions lead to the reject state.

For example, in Figure 14.1, there is no transition labeled $\diamond$ leading out of the initial state, so it is implicit that if $\diamond$ is popped off of X from this state, the machine enters the reject state.

Subroutines

Just like we often do with ordinary programming languages, we can define *subroutines* for stack machines. This can sometimes offer a major simplification to the descriptions of stack machines.

For example, consider the DSM whose state diagram is shown in Figure 14.2.

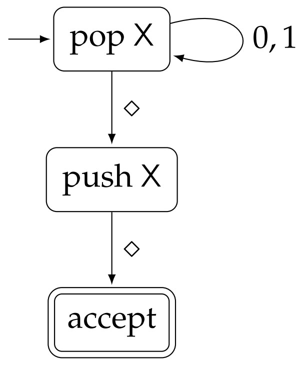


Figure 14.2: An example of a state diagram describing a 1-DSM, whose sole stack is named X. This particular machine is not very interesting from a language-recognition viewpoint—it accepts every string—but it performs the useful task of erasing the contents of a stack. Here the stack alphabet is assumed to be $\Delta = \{0, 1, \diamond\}$, but the idea is easily extended to other stack alphabets.

Before discussing this machine, let us agree that whenever we say that a particular stack *stores* a string x, we mean that the bottom-of-the-stack marker $\diamond$ appears on the bottom of the stack, and the symbols of x appear above this bottom-of-the-stack marker on the stack, with the leftmost symbol of x on the top of the stack. We will never use this terminology in the situation that the symbol $\diamond$ itself appears in x. Using this terminology, the behavior of the DSM illustrated in Figure 14.2 is that if its computation begins with X storing an arbitrary string $x \in \{0, 1\}^*$, then the computation always results in acceptance, with X storing ε. In other words, the DSM erases the string stored by X and halts.

The simple process performed by this DSM might be useful as a subroutine inside of some more complicated DSM, and of course a simple modification allows us to choose any stack in place of X that gets erased. Rather than replicating the description of the DSM from Figure 14.4 inside of this more complicated hypothetical DSM, we can simply use the sort of shorthand suggested by Figure 14.3.

More explicitly, the diagram on the left-hand side of Figure 14.3 suggests a small part of a hypothetical DSM, where the DSM from Figure 14.2 appears inside of the dashed box. Note that we have not included the accept state in the dashed box because, rather than accepting, we wish for control to be passed to the state labeled "another state" as the erasing process completes. We also do not have that the "pop X" state is the initial state any longer, because rather than starting at this state, we have that control passes to this state from the state labeled "some state." There could, in fact, be multiple transitions from multiple states leading to the "pop X" state in the hypothetical DSM being considered.

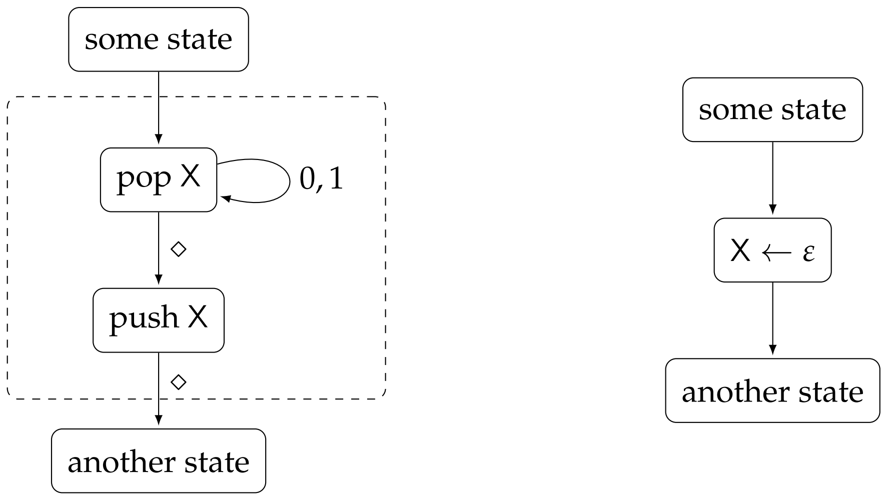


Figure 14.3: The diagrams on the left and right describe equivalent portions of a larger DSM; the contents of the dotted rectangle in the left-hand diagram is viewed as a *subroutine* that is represented by a single rectangle labeled "X ← ε" in the right-hand diagram.

In the diagram on the right-hand side of Figure 14.3 we have replaced the dashed box with a single rectangle labeled "X ← ε." This is just a label that we have chosen, but of course it is a fitting label in this case. The rectangle labeled "X ← ε" looks like a state, and we can think of it as being like a state with which a more complicated operation than push or pop is associated, but the reality is that it is just a short-hand for the contents of the dashed box on the left-hand side diagram.

The same general pattern can be replicated for just about any choice of a DSM. That is, if we have a DSM that we would like to use as a subroutine, we can always do this as follows:

1. Let the original start state of the DSM be the state to which some transition points.
2. Remove the accept state, modifying transitions to this removed accept state so that they point to some other state elsewhere in the larger DSM to which control is to pass once the subroutine is complete.

Naturally, one must be careful when defining and using subroutines like this, as computations could easily become corrupted if subroutines modify stacks that are being used for other purposes elsewhere in a computation. The same thing can, of course, be said concerning subroutines in ordinary computer programs.

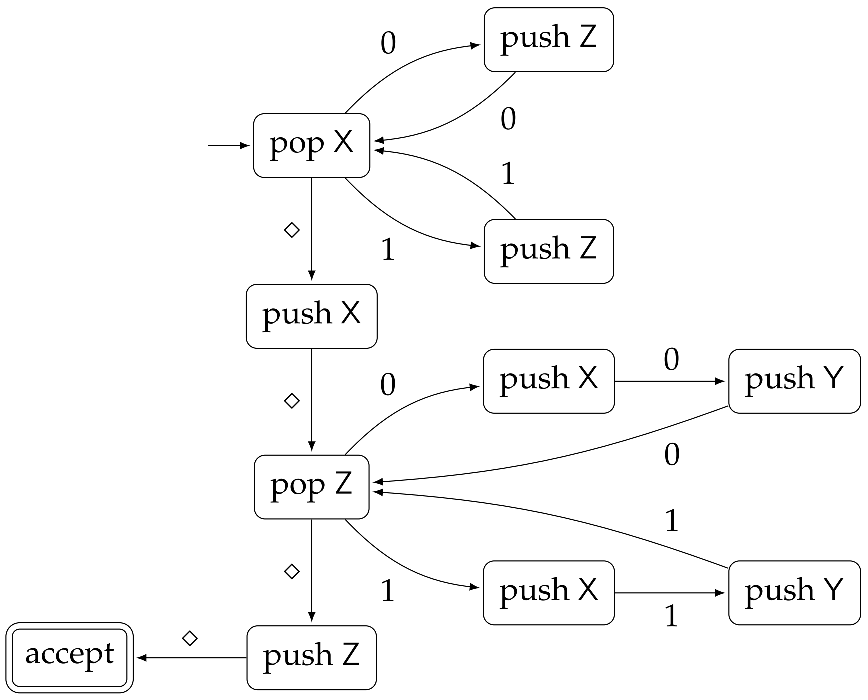


Figure 14.4: An example of a state diagram describing a 3-DSM (with stacks named X, Y, and Z). This machine performs the task of copying the contents of one stack to another: X is copied to Y. The stack Z is used as workspace to perform this operation.

Another example of a subroutine is illustrated in Figure 14.4. This stack machine copies the contents of one stack to another, using a third stack as an auxiliary (or workspace) stack to accomplish this task. Specifically, under the assumption that a stack X stores a string $x \in \{0,1\}^*$ and stacks Y and Z store the empty string, the illustrated 3-DSM will always lead to acceptance—and when it does accept, the stacks X and Y will both store the string x, while Z will revert to its initial configuration in which it stores the empty string. In summary, if initially $(\mathsf{X},\mathsf{Y},\mathsf{Z})$ stores $(x,\varepsilon,\varepsilon)$, then upon certain acceptance $(\mathsf{X},\mathsf{Y},\mathsf{Z})$ will store (x,x,ε).

If we wish to use this DSM as a subroutine in a more complicated DSM, we could again represent the entire DSM (minus the accept state) by a single rectangle, just like we did in Figure 14.3. A fitting label in this case is "Y ← X."

One more example of a DSM that is useful as a subroutine is pictured in Figure 14.5. Notice that in this state diagram we have made use of the two previous subroutines to make the figure simpler. After each new subroutine is defined, we are naturally free to use it to describe new DSMs. The DSM in the figure *reverses* the string stored by X. It uses a workspace stack Y to accomplish this task—but in

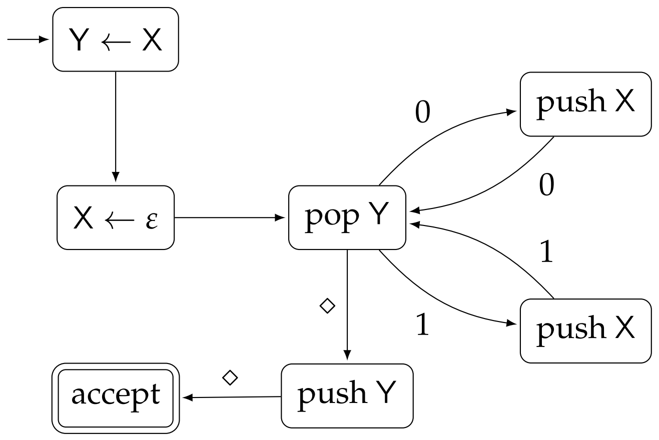


Figure 14.5: A DSM that reverses the string stored by a stack X.

fact it also uses a workspace stack Z, which is hidden inside the subroutine labeled "Y ← X." In summary, it transforms (X, Y, Z) from $(x, \varepsilon, \varepsilon)$ to $(x^R, \varepsilon, \varepsilon)$.

Note, by the way, that we do not in general need to list all of the workspace stacks used by stack machines—we have only done this here to make their use clear. So long as workspace stacks are used correctly, they can safely be ignored.

14.2 Equivalence of DTMs and DSMs

We will now argue that deterministic Turing machines and deterministic stack machines are equivalent computational models. This will require that we establish two separate facts:

1. Given a DTM M, there exists a DSM K that simulates M.
2. Given a DSM M, there exists a DTM K that simulates M.

Just like in the previous lecture, this does not necessarily mean that one step of the original machine corresponds to a single step of the simulator: the simulator might require many steps to simulate one step of the original machine. A consequence of both facts listed above is that, for every input string w, K accepts w whenever M accepts w, K rejects w whenever M rejects w, and K runs forever on w whenever M runs forever on w.

The two simulations are described in the subsections that follow. These descriptions are not intended to be formal proofs, but they should provide enough information to convince you that the two models are indeed equivalent.

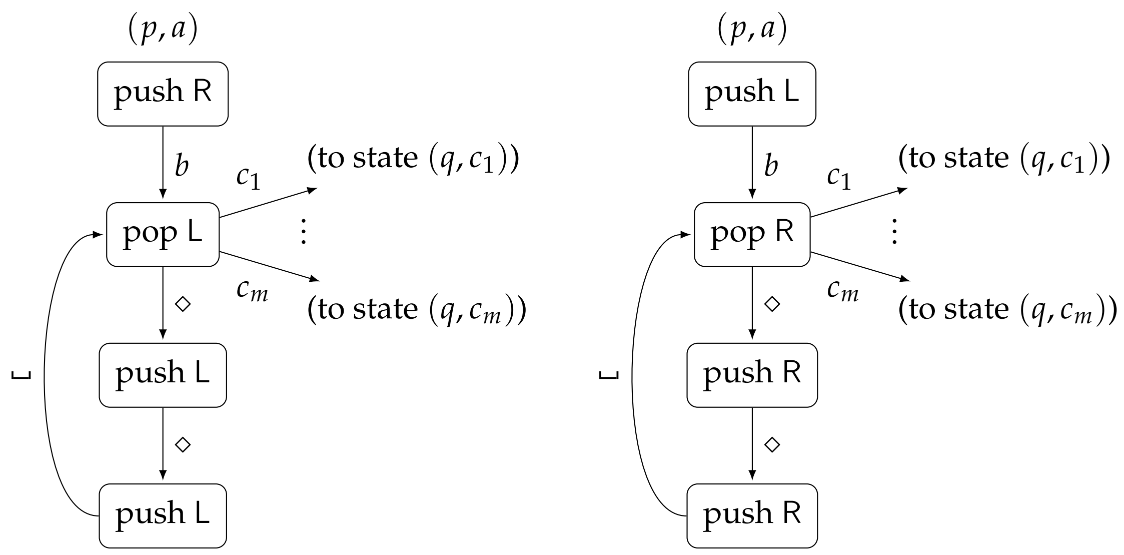


Figure 14.6: For each state/symbol pair $(p,a) \in (Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\}) \times \Gamma$ of M, there are two possibilities: if $\delta(p,a) = (q,b,\leftarrow)$, then K includes the states and transitions in the left-hand diagram, and if $\delta(p,a) = (q,b,\rightarrow)$, then K includes the states and transitions in the right-hand diagram. (The diagrams are symmetric under swapping L and R.)

Simulating a DTM with a DSM

First we will discuss how a DSM can simulate a DTM. To simulate a given DTM M, we will define a DSM K having two stacks, called L and R (for "left" and "right," respectively). The stack L will represent the contents of the tape of M to the left of the tape head (in reverse order, so that the topmost symbol of L is the symbol immediately to the left of the tape head of M) while R will represent the contents of the tape of M to the right of the tape head. The symbol in the tape square of M that is being scanned by its tape head will be stored in the internal state of K, so this symbol does not need to be stored on either stack. Our main task will be to define K so that it pushes and pops symbols to and from L and R in a way that mimics the behavior of M.

To be more precise, suppose that $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ is the DTM to be simulated. The DSM K will require a collection of states for every state/symbol pair $(p,a) \in Q \times \Gamma$. Figure 14.6 illustrates these collections of states in the case that p is a non-halting state. If it is the case that $\delta(p,a) = (q,b,\leftarrow)$, then the states and transitions on the left-hand side of Figure 14.6 mimic the actions of M in this way:

1. The symbol b gets written to the tape of M and the tape head moves left, so K

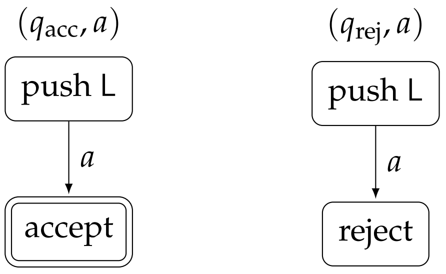


Figure 14.7: For each state/symbol pair $(p, a) \in \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma$ of M, the DSM K simply transitions to its accept or reject state accordingly. The symbol a stored in the finite state memory of K is pushed onto the stack L as this transition is followed. (The choice to push this symbol onto L rather than R is more or less arbitrary, and this operation is not important if one is only interested in whether M accepts, rejects, or runs forever; this operation only has relevance when the contents of the tape of M when it halts are of interest.)

pushes the symbol b onto R to record the fact that the symbol b is now to the right of the tape head of M.

2. The symbol that was one square to the left of the tape head of M becomes the symbol that M scans because the tape head moved left, so K pops a symbol off of L in order to learn what this symbol is and stores it in its finite state memory. In case K pops the bottom-of-the-stack marker, it pushes this symbol back, pushes a blank, and tries again; this has the effect of inserting extra blank symbols as M moves to previously unvisited tape squares.

3. As K pops the top symbol off of L, as described in the previous item, it transitions to the new state (q, c), for whatever symbol c it popped. This sets up K to simulate the next step of M.

The situation is analogous in the case $\delta(p, a) = (q, b, \rightarrow)$, with left and right (and the stacks L and R) swapped.

For each pair (p, a) where $p \in \{q_{\text{acc}}, q_{\text{rej}}\}$, there is no next-step of M to simulate, so K simply transitions to its accept or reject state accordingly, as illustrated in Figure 14.7. Note that if we only care about whether M accepts or rejects, as opposed to what is left on its tape in case it halts, we could alternatively eliminate all states of the form (q_{acc}, a) and (q_{rej}, a), and replace transitions to these eliminated states with transitions to the accept or reject state of K.

The start state of K is the state $(q_0, \sqcup)$ and it is to be understood that stack R is stack number 1 (and therefore contains the input along with the bottom-of-the-stack marker) while L is stack number 2. The initial state of K therefore represents the initial state of M, where the tape head scans a blank symbol and the input is written in the tape squares to the right of this blank tape square.

Simulating a DSM with a DTM

Now we will explain how a DSM can be simulated by a DTM. The idea behind this simulation is fairly straightforward: the DTM will use its tape to store the contents of all of the stacks of the DSM it is simulating, and it will update this information appropriately so as to mimic the DSM. This will require many steps in general, as the DTM will have to scan back and forth on its tape to manipulate the information representing the stacks of the original DSM.

In greater detail, suppose that $M = (Q, \Sigma, \Delta, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ is an r-DSM. The DTM K that we will define to simulate M will have the Cartesian product alphabet:

$$\Gamma = \bigl(\Delta \cup \{\#, \sqcup\}\bigr)^r. \tag{14.6}$$

That is, we are thinking about K as having a tape with multiple tracks, just like in the previous lecture. We assume that # is a special symbol that is not contained in the stack alphabet Δ of M, and that the blank symbol $\sqcup$ is also not contained in Δ. Figure 14.8 provides an illustration of how the tape will simulate k stacks. As before, it is to be understood that the true blank symbol of K is the symbol $(\sqcup, \ldots, \sqcup)$, and that an input string $a_1 \cdots a_n \in \Sigma^*$ of M is to be identified with the string of tape symbols

$$(a_1, \sqcup, \ldots, \sqcup) \cdots (a_n, \sqcup, \ldots, \sqcup) \in \Gamma^*. \tag{14.7}$$

The purpose of the symbol # is to mark a position on the tape of K; the contents of the stacks of M will always be to the left of these # symbols. The first thing that K does, before any steps of M are simulated, is to scan the tape (from left to right) to find the end of the input string. In the first tape square after the input string, it places the bottom-of-the-stack marker $\diamond$ in every track, and in the next square to the right it places the # symbol in every track. Once these # symbols are written to the tape, they will remain there for the duration of the simulation. The DTM then moves its tape head to the left, so that it is positioned over the # symbols, and begins simulating steps of the DSM M.

The DTM K will store the current state of M in its internal memory, and one way to think about this is to imagine that K has a collection of states for every state $q \in Q$ of M (which is similar to the simulation in the previous subsection, except there we had a collection of states for every state/symbol pair rather than just for each state). The DTM K is defined so that this state will initially be set to q_0 (the start state of M) when it begins the simulation.

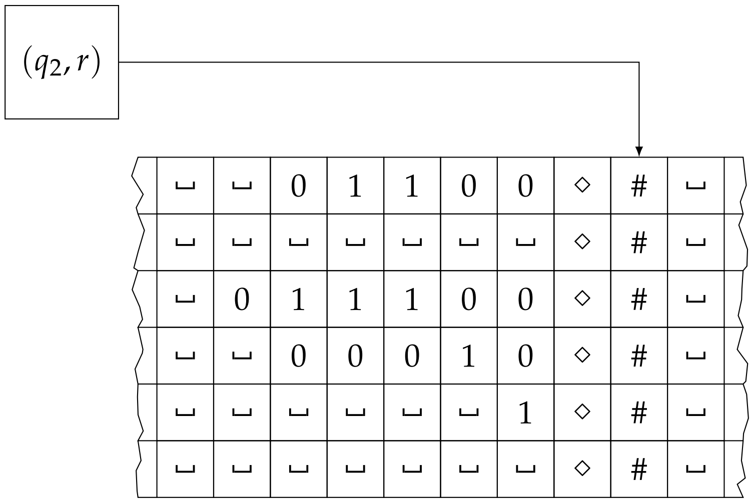


Figure 14.8: An example of a DTM whose tape has 6 tracks, each representing a stack. This figure is consistent with the stack alphabet of the DSM that is being simulated being $\Delta = \{0, 1, \diamond\}$; the situation pictured is that the DSM stacks 0 through 5 store the strings 01100, ε, 011100, 00010, 1, and ε, respectively.

There are two possibilities for each non-halting state $q \in Q$ of M: it is either a push state or a pop state. In either case, there is a stack index k that is associated with this state. The behavior of K is as follows for these two possibilities:

1. If q is a push state, then there must be a symbol $a \in \Gamma$ that is to be pushed onto stack k. The DTM K scans left until it finds a blank symbol on track k, overwrites this blank with the symbol a, and changes the state of M stored in its internal memory exactly as M does.
2. If q is a pop state, then K needs to find out what symbol is on the top of stack k. It scans left to find a blank symbol on track k, moves right to find the symbol on the top of stack k, changes the state of M stored in its internal memory accordingly, and overwrites this symbol with a blank. Naturally, in the situation where M attempts to pop an empty stack, K will detect this (as there will be no non-blank symbols to the left of the # symbols), and it immediately transitions to its reject state.

In both cases, after the push or pop operation was simulated, K scans its tape head back to the right to find the # symbols, so that it can simulate another step of M.

Finally, if K stores a halting state of M when it would otherwise begin simulating a step of M, it accepts or rejects accordingly. In a situation in which the contents of the tape of K after the simulation are important, such as when M computes a function rather than simply accepting or rejecting, one may of course define K so that it first removes the # symbols and $\diamond$ symbols from its tape prior to accepting or rejecting.

Lecture 15

Encodings; examples of decidable languages

Now that we have studied basic aspects of the Turing machine model, including variants of Turing machines and the computationally equivalent stack machine model, it is time to discuss some examples of decidable languages. In this lecture we will focus on examples based on finite automata and context-free grammars. These languages will have a somewhat different character from most of the languages we discussed previously in the course; their definitions are centered on fundamental mathematical concepts, as opposed to simple syntactic patters.

Before doing this, however, we will discuss *encodings*, which allow us to represent complicated mathematical objects using strings over a given alphabet. 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 objects of multiple types. We will make use of the notion of an encoding for the remainder of the course.

15.1 Encodings of interesting mathematical objects

The idea that we can encode different sorts of objects as strings will be familiar to students of computer science, and for this reason we will not belabor this issue—but it will nevertheless be helpful to establish a few conventions and introduce useful ideas concerning the encoding of different objects of interest.

Encoding multiple strings into one

Let us begin by considering the following task that concerns two hypothetical individuals: Alice and Bob. Alice has two binary strings $x \in \{0,1\}^*$ and $y \in \{0,1\}^*$, and she would like to communicate these two strings to Bob. However, for some hypothetical reason, Alice is only allowed to transmit a single binary string to Bob, so somehow x and y must be packed into a single string $z \in \{0,1\}^*$ from which Bob can recover both x and y. The two of them may agree ahead of time on a method through which this will be done, but naturally the method must be agreed upon prior to Alice knowing which strings x and y are to be communicated. That is, the method must work for an arbitrary choice of binary strings x and y. There are different methods through which this task may be accomplished, but let us describe just one method.

The first step is to introduce a new symbol, which we will call #. We then choose to encode the pair (x, y) into a single string $x\#y \in \{0,1,\#\}^*$. Obviously, if Alice were to send this string to Bob, he could recover x and y without difficulty, so it is a good method in that sense—but unfortunately it does not solve the original problem because it makes use of the alphabet $\{0,1,\#\}$ rather than $\{0,1\}$.

The second step of the method will take us back to the binary alphabet: we can encode the string $x\#y$ as a binary string by substituting the individual symbols according to this pattern:[1]

$$\begin{aligned} 0 &\rightarrow 00 \\ 1 &\rightarrow 01 \\ \# &\rightarrow 1. \end{aligned} \tag{15.1}$$

The resulting binary string will be the *encoding* of the two strings x and y that Alice sends to Bob.

For example, if the two strings are $x = 0110$ and $y = 01111$, we first consider the string 0110#01111, and then perform the substitution suggested above to obtain

$$\langle 0110, 01111 \rangle = 0001010010001010101. \tag{15.2}$$

Here we have used a notation that we will use frequently throughout much of the remainder of the course: whenever we have some object X, along with an encoding scheme that encodes a class of objects that includes X as strings, we write $\langle X \rangle$ to denote the string that encodes X. In the equation above, the notation $\langle 0110, 01111 \rangle$ therefore refers to the encoding of the two strings 0110 and 01111, viewed as an ordered pair.

Let us make a few observations about the encoding scheme just described:

1. It is easy to recover the strings x and y from the encoding $\langle x, y \rangle$. Specifically, so long as we find the symbol 0 in each odd-numbered position, the symbols

[1] There are other patterns that would work equally well. The one we have selected is an example of a *prefix-free code*; because none of the strings appearing on the right-hand side of (15.1) is a prefix of any of the other strings, we are guaranteed that by concatenating together a sequence of these strings we can recover the original string without ambiguity.

in the even-numbered positions that follow belong to x; and once we find a 1 in an odd-numbered position, we know that x is determined and it is time to recover y through a similar process.

2. The scheme works not only for two strings x and y, but for any finite number of binary strings $x_1, \ldots, x_n$; such a list of strings may be encoded by first forming the string $x_1 \# x_2 \# \cdots \# x_n \in \{0, 1, \#\}^*$, and then performing the substitutions described above to obtain $\langle x_1, \ldots, x_n \rangle \in \{0, 1\}^*$.

3. Every n-tuple $(x_1, \ldots, x_n)$ of binary strings has a unique encoding $\langle x_1, \ldots, x_n \rangle$, but it is not the case that every binary string encodes an n-tuple of binary strings. In other words, the encoding is one-to-one but not onto. For instance, the string 10 does not decode to any string over the alphabet $\{0, 1, \#\}$, and therefore does not encode an n-tuple of binary strings. This is not a problem; most of the encoding schemes we will consider in this course have the same property that not every string is a valid encoding of some object of interest.

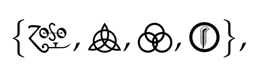
4. You could easily generalize this scheme to larger alphabets by adding a new symbol to mark the division between strings over the original alphabet, and then choosing a suitable encoding in place of (15.1).

This one method of encoding multiple strings into one turns out to be incredibly useful, and by using it repeatedly we can devise encoding schemes for highly complex mathematical objects.

Encoding strings over arbitrary alphabets using a fixed alphabet

Next, let us consider the task of encoding a string over an alphabet Γ whose size we do not know ahead of time by a string over a fixed alphabet Σ. In the interest of simplicity, let us take $\Sigma = \{0, 1\}$ to be the binary alphabet.

Before we discuss a particular scheme through which this task can be performed, let us take a moment to clarify the task at hand. In particular, it should be made clear that we are not looking for a way to encode strings over any possible alphabet Γ that you could ever imagine. For instance, consider the alphabet

$\Gamma =$  (15.3)

from the very first lecture of the course. Some might consider this to be an interesting alphabet, but in some sense there is nothing special about it—all that is really relevant from the viewpoint of the theory of computation is that it has four symbols, so there is little point in differentiating it from the alphabet $\Gamma = \{0, 1, 2, 3\}$.[2] That is, when we think about models of computation, all that really matters is the number of symbols in our alphabet, and sometimes the order we choose to put them in, but not the size, shape, or color of the symbols.

With this understanding in place, we will make the assumption that our encoding task is to be performed for an alphabet of the form

$$\Gamma = \{0, 1, \ldots, n-1\} \tag{15.4}$$

for some positive integer n, where we are imagining that each integer between 0 and $n-1$ is a single symbol.

The method from the previous subsection provides a simple means through which the task at hand can be accomplished. First, for every nonnegative integer $k \in \mathbb{N}$, let us decide that the encoding $\langle k \rangle$ of this number is given by its representation using binary notation:

$$\langle 0 \rangle = 0, \quad \langle 1 \rangle = 1, \quad \langle 2 \rangle = 10, \quad \langle 3 \rangle = 11, \quad \langle 4 \rangle = 100, \quad \text{etc.} \tag{15.5}$$

Then, to encode a given string $k_1 k_2 \cdots k_m$, we simply encode the m binary strings $\langle k_1 \rangle, \langle k_2 \rangle, \ldots, \langle k_m \rangle$ into a single binary string

$$\big\langle \langle k_1 \rangle, \langle k_2 \rangle, \ldots, \langle k_m \rangle \big\rangle \tag{15.6}$$

using the method from the previous subsection.

For example, let us consider the string 001217429, which we might assume is over the alphabet $\{0, \ldots, 9\}$ (although this assumption will not influence the encoding that is obtained). The method from the previous subsection suggests that we first form the string

$$0\#0\#1\#10\#1\#111\#100\#10\#1001 \tag{15.7}$$

and then encode this string using the substitutions (15.1). The binary string we obtain is

$$\langle 001217429 \rangle = 00100101101001011010101101000010100101000001. \tag{15.8}$$

Finally, let us briefly discuss the possibility that the alphabet Σ is the unary alphabet $\Sigma = \{0\}$ rather than the binary alphabet. You can still encode strings over any alphabet $\Gamma = \{0, \ldots, n-1\}$ using this alphabet, although (not surprisingly) it

[2] You could of course consider encoding schemes that represent the shapes and sizes of different alphabet symbols—the symbols appearing in (15.3), for example, are in fact the result of a binary string encoding obtained from an image compression algorithm—but this is not what we are talking about.

will be extremely inefficient. One way to do this is to first encode strings over Γ as strings over the binary alphabet, exactly as discussed above, and then to encode binary strings as unary strings with respect to the lexicographic ordering:

$$\begin{aligned} \varepsilon &\rightarrow \varepsilon \\ 0 &\rightarrow 0, \\ 1 &\rightarrow 00, \\ 10 &\rightarrow 000, \\ 11 &\rightarrow 0000, \\ 100 &\rightarrow 00000, \end{aligned} \tag{15.9}$$

and so on.

This means that you could, in principle, encode an entire book in unary. Think of an ordinary book as a string over the alphabet that includes upper- and lower-case letters, spaces, and punctuation marks, and imagine encoding this string over the unary alphabet as just described. You open the unary-encoded book and see that every page is filled with 0s, and as you are reading the book you have absolutely no idea what it is about. All you can do is to eliminate the possibility that the book corresponds to a shorter string of 0s than the number you have seen so far, just like when you rule out the possibility that it is 3 o'clock when the bells at City Hall have (thus far) rung four times. Finally you finish the book and in an instant it all becomes clear, and you say "Wow, what a great book!"

Numbers, vectors, and matrices

We already used the elementary fact that nonnegative integers can be encoded as binary strings using binary notation in the previous subsection. One can also encode arbitrary integers using binary notation by interpreting 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 using the method from earlier in the lecture. One could also consider floating point representations, which are of course very common in practice, but also have the disadvantage that they only represent rational numbers for which the denominator is a power of two.

With a method for encoding numbers as binary strings in mind, one can represent vectors by simply encoding the entries as strings, and then encoding these multiple strings into a single string using the method described at the start of the lecture. Matrices can be represented as lists of vectors. Indeed, once you know how to encode lists of strings as strings, you can very easily devise encoding schemes for highly complex mathematical objects.

An encoding scheme for DFAs and NFAs

Now let us devise an encoding scheme for DFAs and NFAs. We will start with DFAs, and once we are finished we will observe how the scheme can be easily modified to obtain an encoding scheme for NFAs.

What we are aiming for is a way to encode every possible DFA

$$M = (Q, \Gamma, \delta, q_0, F) \tag{15.10}$$

as a binary string $\langle M \rangle$.[3] Intuitively speaking, given the binary string $\langle M \rangle \in \Sigma^*$, it should be possible to recover a description of exactly how M operates without difficulty. There are, of course, many possible encoding schemes that one could devise—we are just choosing one that works but otherwise is not particularly special.

Along similar lines to the discussion above concerning the encoding of strings over arbitrary alphabets, we will make the assumption that the alphabet Γ of M takes the form

$$\Gamma = \{0, \ldots, n-1\} \tag{15.11}$$

for some positive integer n. For the same reasons, we will assume that the state set of M takes the form

$$Q = \{q_0, \ldots, q_{m-1}\} \tag{15.12}$$

for some positive integer m.

There will be three parts of the encoding:

1. A positive integer n representing $|\Gamma|$. This number will be represented using binary notation.
2. A specification of the set F together with the number of states m. These two things together can be described by a binary string s of length m. Specifically, the string

   $$s = b_0 b_1 \cdots b_{m-1} \tag{15.13}$$

   specifies that

   $$F = \big\{q_k : k \in \{0, \ldots, m-1\},\ b_k = 1\big\}, \tag{15.14}$$

   and of course the number of states m is given by the length of the string s. Hereafter we will write $\langle F \rangle$ to refer to the encoding of the subset F that is obtained in this way.

[3] Notice that we are taking the alphabet of M to be Γ rather than Σ to be consistent with the conventions used in the previous subsections: Γ is an alphabet having an arbitrary size, and we cannot assume it is fixed as we devise our encoding scheme, while $\Sigma = \{0, 1\}$ is the alphabet we are using for the encoding.

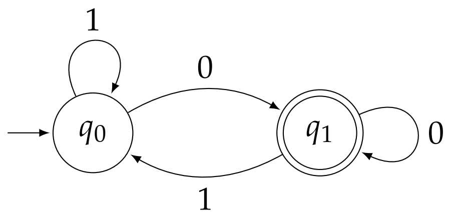


Figure 15.1: A simple example of a DFA.

3. The transition function δ will be described by listing all of the inputs and outputs of this function, in the following way. First, for $j, k \in \{0, \ldots, m-1\}$ and $a \in \Gamma = \{0, \ldots, n-1\}$, the string
$$\langle \langle j \rangle, \langle a \rangle, \langle k \rangle \rangle \tag{15.15}$$
specifies that
$$\delta(q_j, a) = q_k. \tag{15.16}$$
Here, $\langle j \rangle$, $\langle k \rangle$, and $\langle a \rangle$ refer to the strings obtained from binary notation, which makes sense because j, k, and a are all nonnegative integers. We then encode the list of all of these strings, in the natural ordering that comes from iterating over all pairs (j, a), into a single string $\langle \delta \rangle$.

   For example, the DFA depicted in Figure 15.1 has a transition function δ whose encoding is
$$\langle \delta \rangle = \langle \langle 0, 0, 1 \rangle, \langle 0, 1, 0 \rangle, \langle 1, 0, 1 \rangle, \langle 1, 1, 0 \rangle \rangle. \tag{15.17}$$
(We will leave it in this form rather than expanding it out as a binary string in the interest of clarity.)

Finally, the encoding of a given DFA M is just a list of the three parts just described:
$$\langle M \rangle = \langle \langle n \rangle, \langle F \rangle, \langle \delta \rangle \rangle. \tag{15.18}$$

This encoding scheme can easily be modified to obtain an encoding scheme for NFAs. This time, the values the transition function takes are subsets of Q rather than elements of Q, and we must also account for the possibility of ε-transitions. Fortunately, we already know how to encode subsets of Q; we did this for the set F, and exactly the same method can be used to encode any one of the subsets $\delta(q_j, a)$ as a binary string $\langle \delta(q_j, a) \rangle$ having length equal to the total number of states in Q. That is, the string
$$\langle \langle j \rangle, \langle a \rangle, \langle \delta(q_j, a) \rangle \rangle \tag{15.19}$$

describes the value of the transition function for the pair (q_j, a). To specify the ε-transitions of M, we may use the string

$$\langle \langle j \rangle, \varepsilon, \langle \delta(q_j, \varepsilon) \rangle \rangle, \tag{15.20}$$

which takes advantage of the fact that we never have $\langle a \rangle = \varepsilon$ for any symbol $a \in \Gamma$. As before, we simply list all of the strings corresponding to the different inputs of δ in order to encode δ.

Encoding schemes for regular expressions, CFGs, PDAs, etc.

We could continue on and devise encoding schemes through which regular expressions, CFGs, PDAs, DTMs, and DSMs can be specified. Because of its importance, we will in fact return to the case of DTMs in the next lecture, but for the others I will leave it to you to think about how you might design encoding schemes. There are countless specific ways to do this, but it turns out that the specifics are not really all that important—the reason why we did this carefully DFAs and NFAs is to illustrate how it can be done for those models, with the principal aim being to clarify the concept rather than to create an encoding scheme whose specific aspects are conceptually relevant.

15.2 Decidability of formal language problems

Now let us turn our attention toward languages that concern the models of computation we have studied previously in the course.

Languages based on DFAs, NFAs, and regular expressions

The first language we will consider is this one:

$$\mathrm{A}_{\mathrm{DFA}} = \big\{ \langle \langle D \rangle, \langle w \rangle \rangle \,:\, D \text{ is a DFA and } w \in \mathrm{L}(D) \big\}. \tag{15.21}$$

Here we assume that $\langle D \rangle$ is the encoding of a given DFA D, $\langle w \rangle$ is the encoding of a given string w, and $\langle \langle D \rangle, \langle w \rangle \rangle$ is the encoding of the two strings $\langle D \rangle$ and $\langle w \rangle$, all as described earlier in the lecture. Thus, $\langle D \rangle$, $\langle w \rangle$, and $\langle \langle D \rangle, \langle w \rangle \rangle$ are all binary strings. It could be the case, however, that the alphabet of D is any alphabet of the form $\Gamma = \{0, \ldots, n-1\}$, and likewise for the string w.

It is a natural question to ask whether or not the language $\mathrm{A}_{\mathrm{DFA}}$ is decidable. It certainly is. For a given input string $x \in \{0,1\}^*$, one can easily check that it takes the form $x = \langle \langle D \rangle, \langle w \rangle \rangle$ for a DFA D and a string w, and then check whether or not D accepts w by simply *simulating*, just like you would do with a piece of paper

The DTM M operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle\langle D\rangle, \langle w\rangle\rangle$ for D being a DFA and w being a string over the alphabet of D, then reject.
2. Simulate D on input w; accept if D accepts w and reject if D rejects w.

Figure 15.2: A high-level description of a DTM M that decides the language A_{DFA}.

and a pencil if you were asked to make this determination for yourself. Figure 15.2 gives a high-level description of a DTM M that decides the language A_{DFA} along these lines.

Now, you might object to the claim that Figure 15.2 describes a DTM that decides A_{DFA}. It does describe the main idea of how M operates, which is that it simulates D on input w, but it offers hardly any detail at all. It seems more like a suggestion for how to design a DTM than an actual description of a DTM.

This is a fair criticism, but as we move forward with the course, we will need to make a transition along these lines. The computations we will consider will become more and more complicated, and in the interest of both time and clarity we must abandon the practice of describing the DTMs that perform these computations explicitly. Hopefully our discussions and development of the DTM model have convinced you that the process of taking a high-level description of a DTM, such as the one in Figure 15.2, and producing an actual DTM that performs the computation described would be a more or less routine task.

Because the description of the DTM M suggested by Figure 15.2 is our first example of such a high-level DTM description, let us take a moment to consider in greater detail how it could be turned into a formal specification of a DTM. Because we know that any deterministic stack machine can be simulated by a DTM, it suffices to describe a DSM M that operates as described in Figure 15.2.

1. The input x to the DSM M is initially stored in stack number 1, which we might instead choose to name X for clarity. The first step in Figure 15.2 is to check that the input takes the form $x = \langle\langle D\rangle, \langle w\rangle\rangle$. Assuming that the input does take this form, it is convenient for the sake of the second step of M (meaning the simulation of D on input w) that the input is split into two parts, with the string $\langle D\rangle$ being stored in a stack called D and $\langle w\rangle$ being stored in a stack called W. This splitting could easily be done as a part of the check that the input does take the form $x = \langle\langle D\rangle, \langle w\rangle\rangle$.

2. To simulate D on input w, the DSM M will need to keep track of the current state of D, so it is natural to introduce a new stack Q for this purpose. At the start of the simulation, Q is initialized it so that it stores 0 (the encoding of the state q_0).
3. The actual simulation proceeds in the natural way, which is to examine the encodings of the symbols of w stored in W, one at a time, updating the state contained in Q accordingly. While an explicit description of the DSM states and transitions needed to do this would probably look rather complex, it could be done in a conceptually simple manner. In particular, each step of the simulation would presumably involve M searching through the transitions of D stored in D to find a match with the current state encoding stored in Q and the next input symbol encoding stored in W, after which Q is updated. Naturally, M can make use of additional stacks and make copies of strings as needed so that the encoding $\langle D \rangle$ is always available at the start of each simulation step.
4. Once the simulation is complete, an examination of the state stored in Q and the encoding $\langle F \rangle$ of the accepting states of D leads to acceptance or rejection appropriately.

All in all, it would be a tedious task to write down the description of a DSM M that behaves in the manner just described—but I hope you will agree that with a bit of time, patience, and planning, it would be feasible to do this. An explicit description of such a DSM M would surely be made more clear if a thoughtful use of subroutines was devised (not unlike the analogous task of writing a computer program to perform such a simulation). Once the specification of this DSM is complete, it can then be simulated by a DTM as described in the previous lecture.

Next let us consider a variant of the language A_{DFA} for NFAs in place of DFAs:

$$A_{\text{NFA}} = \big\{ \langle \langle N \rangle, \langle w \rangle \rangle \,:\, N \text{ is an NFA and } w \in L(N) \big\}. \tag{15.22}$$

Again, it is our assumption that the encodings with respect to which this language is defined are as discussed earlier in the lecture. The language A_{NFA} is also decidable. This time, however, it would not be reasonable to simply describe a DSM that "simulates N on input w," because it is not at all clear how a deterministic computation can simulate a nondeterministic finite automaton computation.

What we can do instead is to make use of the process through which NFAs are converted to DFAs that we discussed in Lecture 3; this is a well-defined deterministic procedure, and it can certainly be performed by a DTM. Figure 15.3 gives a high-level description of a DTM M that decides A_{NFA}. Once again, although it would be a time-consuming task to explicitly describe a DTM that performs this

The DTM M operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle \langle N \rangle, \langle w \rangle \rangle$ for N being an NFA and w being a string over the alphabet of N, then reject.
2. Convert N into an equivalent DFA D using the subset construction described in Lecture 3.
3. Simulate D on input w; accept if D accepts w and reject if D rejects w.

Figure 15.3: A high-level description of a DTM M that decides the language A_{NFA}.

computation, it is reasonable to view this as a straightforward task in a conceptual sense.

One can also define a language similar to A_{DFA} and A_{NFA}, but for regular expressions in place of DFAs and NFAs:

$$\text{A}_{\text{REX}} = \big\{ \langle \langle R \rangle, \langle w \rangle \rangle \,:\, R \text{ is a regular expression and } w \in \text{L}(R) \big\}. \tag{15.23}$$

We did not actually discuss an encoding scheme for regular expressions, so it will be left to you to devise or imagine your own encoding scheme—but as long as you picked a reasonable one, the language A_{REX} would be decidable. In particular, given a reasonable encoding scheme for regular expressions, a DTM could first convert this regular expression into an equivalent DFA, and then simulate this DFA on the string w.

Here is a different example of a language, which we will argue is also decidable:

$$\text{E}_{\text{DFA}} = \big\{ \langle D \rangle \,:\, D \text{ is a DFA and } \text{L}(D) = \varnothing \big\}. \tag{15.24}$$

In this case, one cannot decide this language simply by "simulating the DFA D," because a priori there is no particular string on which to simulate it; we care about every possible string that could be given as input to D and whether or not D accepts any of them. Deciding the language E_{DFA} is therefore not necessarily a straightforward simulation task.

What we can do instead is to treat the decision problem associated with this language as a *graph reachability* problem. The DTM M suggested by Figure 15.4 takes this approach and decides E_{DFA}. By combining this DTM with ideas from the previous examples, one can prove that analogously defined languages E_{NFA} and E_{REX} are also decidable:

$$\begin{aligned} \text{E}_{\text{NFA}} &= \big\{ \langle N \rangle \,:\, N \text{ is an NFA and } \text{L}(N) = \varnothing \big\}, \\ \text{E}_{\text{REX}} &= \big\{ \langle R \rangle \,:\, R \text{ is a regular expression and } \text{L}(R) = \varnothing \big\}. \end{aligned} \tag{15.25}$$

The DTM M operates as follows on input $x \in \{0,1\}^*$:

```
1. If it is not the case that $x = \langle D \rangle$ for $D$ being a DFA, then reject.
2. Set $S \leftarrow \{0\}$.
3. Set $a \leftarrow 1$.
4. For every pair of integers $j, k \in \{0, \ldots, m-1\}$, where $m$ is the number
   of states of $D$, do the following:
   4.1 If $j \in S$ and $k \notin S$, and $D$ includes a transition from $q_j$ to $q_k$, then
       set $S \leftarrow S \cup \{k\}$ and $a \leftarrow 0$.
5. If $a = 0$ then goto step 3.
6. Reject if there exists $k \in S$ such that $q_k \in F$ (i.e., $q_k$ is an accept state
   of $D$), otherwise accept.
```

Figure 15.4: A high-level description of a DTM M that decides the language E_{DFA}.

One more example of a decidable language concerning DFAs is this language:

$$\text{EQ}_{\text{DFA}} = \big\{ \langle \langle A \rangle, \langle B \rangle \rangle \,:\, A \text{ and } B \text{ are DFAs and } \text{L}(A) = \text{L}(B) \big\}. \tag{15.26}$$

Figure 15.5 gives a high-level description of a DTM that decides this language. One natural way to perform the construction in step 2 is to use the Cartesian product construction described in Lecture 4.

Languages based on CFGs

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:

$$\begin{aligned} \text{A}_{\text{CFG}} &= \big\{ \langle \langle G \rangle, \langle w \rangle \rangle \,:\, G \text{ is a CFG and } w \in \text{L}(G) \big\}, \\ \text{E}_{\text{CFG}} &= \big\{ \langle G \rangle \,:\, G \text{ is a CFG and } \text{L}(G) = \varnothing \big\}. \end{aligned} \tag{15.27}$$

Once again, although we have not explicitly described an encoding scheme for context-free grammars, it is not difficult to come up with such a scheme (or to just imagine that such a scheme has been defined). A DTM that decides the first language is described in Figure 15.6. It is worth noting that this is a ridiculously inefficient way to decide the language A_{CFG}, but right now we do not care! We are

The DTM M operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle \langle A \rangle, \langle B \rangle \rangle$ for A and B being DFAs, then reject.
2. Construct a DFA C for which $\mathrm{L}(C) = \mathrm{L}(A) \triangle \mathrm{L}(B)$.
3. Accept if $\langle C \rangle \in \mathrm{E}_{\mathrm{DFA}}$ and otherwise reject.

Figure 15.5: A high-level description of a DTM M that decides the language $\mathrm{EQ}_{\mathrm{DFA}}$.

The DTM M operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle \langle G \rangle, \langle w \rangle \rangle$ for G a CFG and w a string, then reject.
2. Convert G into an equivalent CFG H in Chomsky normal form.
3. If $w = \varepsilon$ then *accept* if $S \rightarrow \varepsilon$ is a rule in H and *reject* otherwise.
4. 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.

Figure 15.6: A high-level description of a DTM M that decides the language $\mathrm{A}_{\mathrm{CFG}}$. This DTM is ridiculously inefficient, but there are more efficient ways to decide this language.

just trying to prove that this language 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. A DTM that decides this language is described in Figure 15.7.

Now, you may be wondering about this next language, as it is analogous to one concerning DFAs from above:

$$\mathrm{EQ}_{\mathrm{CFG}} = \bigl\{ \langle \langle G \rangle, \langle H \rangle \rangle \;:\; G \text{ and } H \text{ are CFGs and } \mathrm{L}(G) = \mathrm{L}(H) \bigr\}. \tag{15.28}$$

As it turns out, this language is *undecidable*. We will not go through the proof because it would take us a bit too far off the path of the rest of the course—but some

```
The DTM $M$ operates as follows on input $x \in \{0,1\}^*$:
  1. If it is not the case that $x = \langle G \rangle$ for $G$ a CFG, then reject.
  2. Set $T \leftarrow \Sigma$ (for $\Sigma$ being the alphabet of $G$).
  3. Set $a \leftarrow 1$.
  4. For each rule $X \rightarrow w$ of $G$ do the following:
      4.1 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 $a \leftarrow 0$.
  5. If $a = 0$ then goto step 3.
  6. Reject if the start variable of $G$ is contained in $T$, otherwise accept.
```

Figure 15.7: A high-level description of a DTM M that decides the language E_{CFG}.

of the facts we will prove in the next lecture may shed some light on why this language is undecidable. Some other examples of undecidable languages concerning context-free grammars are these:

$$\begin{gathered}\big\{\langle G\rangle \;:\; G \text{ is a CFG that generates all strings over its alphabet}\big\},\\ \big\{\langle G\rangle \;:\; G \text{ is an ambiguous CFG}\big\},\\ \big\{\langle G\rangle \;:\; G \text{ is a CFG and } \mathrm{L}(G) \text{ is inherently ambiguous}\big\}.\end{gathered} \tag{15.29}$$

Lecture 16

Universal Turing machines and undecidable languages

In this lecture we will describe a *universal Turing machine*. This is a deterministic Turing machine that, when given the encoding of an arbitrary DTM, can *simulate* that machine on a given input.

To describe such a universal machine, we must naturally consider *encodings* of DTMs, and this will be the first order of business for the lecture. The very notion of an encoding scheme for DTMs allows us to obtain our first example of a language that is not semidecidable (and is therefore not decidable). Through the non-semidecidability of this language, many other languages can be shown to be either undecidable or non-semidecidable. We will see two simple examples in this lecture and more in the lecture following this one.

16.1 An encoding scheme for DTMs

In the previous lecture we discussed in detail an encoding scheme for DFAs, and we observed that this scheme is easily adapted to obtain an encoding scheme for NFAs. While we did not discuss specific encoding schemes for regular expressions and context-free grammars, we made use of the fact that one can devise encoding schemes for these models without difficulty.

We could follow a similar route for DTMs, as there are no new conceptual difficulties that arise for this model in comparison to the other models just mentioned. However, given the high degree of importance that languages involving encodings of DTMs will have in the remainder of the course, it is fitting to take a few moments to be careful and precise about this encoding. As is the case for just about every encoding scheme we consider, there are many alternatives to the encoding of DTMs we will define—our focus on the specifics of this encoding scheme is done in the interest of clarity and precision, and not because the specifics themselves are essential to the study of computability.

Throughout the discussion that follows, we will assume that

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \tag{16.1}$$

is a given DTM whose encoding is to be described. We make the assumption that the state set Q of M takes the form

$$Q = \{q_0, \ldots, q_{m-1}\} \tag{16.2}$$

for some positive integer m, and that the input and tape alphabets of M take the form

$$\Sigma = \{0, \ldots, k-1\} \quad \text{and} \quad \Gamma = \{0, \ldots, n-1\} \tag{16.3}$$

for positive integers k and n. Given that Σ is properly contained in Γ, it follows that $k < n$; it is to be assume that the blank symbol $\sqcup \in \Gamma$ corresponds to the last symbol $n-1$ of Γ.

First, the encoding of a state $q \in Q$ should be understood as referring to the string $\langle q \rangle \in \{0,1\}^*$ obtained by expressing the index of that state written in binary notation, so that

$$\langle q_0 \rangle = 0, \quad \langle q_1 \rangle = 1, \quad \langle q_2 \rangle = 10, \quad \text{etc.} \tag{16.4}$$

Tape symbolds are encoded in a similar way, so that

$$\langle 0 \rangle = 0, \quad \langle 1 \rangle = 1, \quad \langle 2 \rangle = 10, \quad \text{etc.} \tag{16.5}$$

We shall encode the directions left and right as

$$\langle \leftarrow \rangle = 0 \quad \text{and} \quad \langle \rightarrow \rangle = 1. \tag{16.6}$$

Next we need a way to encode the transition function δ, and this will be done in a similar way to the encoding of DFAs from the previous lecture. For each choice of states $q \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\}$ and $r \in Q$, tape symbols $a, b \in \Gamma$, and a direction $D \in \{\leftarrow, \rightarrow\}$, the encoding

$$\big\langle \langle q \rangle, \langle a \rangle, \langle r \rangle, \langle b \rangle, \langle D \rangle \big\rangle \in \{0,1\}^* \tag{16.7}$$

indicates that

$$\delta(q, a) = (r, b, D). \tag{16.8}$$

The encoding of the transition function δ is then obtained by encoding the list of binary strings of the form above, for each pair $(q, a) \in Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma$ in lexicographic order.

Finally, by means of these encodings, we obtain from every DTM M the binary string encoding $\langle M \rangle \in \{0,1\}^*$ as follows:

$$\langle M \rangle = \big\langle \langle m \rangle, \langle k \rangle, \langle n \rangle, \langle \delta \rangle, \langle q_{\text{acc}} \rangle, \langle q_{\text{rej}} \rangle \big\rangle. \tag{16.9}$$

Example 16.1. Consider the DTM M for the language SAME whose state diagram is shown in Figure 12.3. We have

$$\begin{aligned}\delta(q_0,0) &= (q_0,0,\leftarrow),\\ \delta(q_0,1) &= (q_0,1,\leftarrow),\\ \delta(q_0,\sqcup) &= (q_1,\sqcup,\rightarrow),\\ &\vdots\\ \delta(q_3,0) &= (q_{\text{rej}},0,\leftarrow),\\ \delta(q_3,1) &= (q_0,\sqcup,\leftarrow),\\ \delta(q_3,\sqcup) &= (q_{\text{rej}},\sqcup,\leftarrow).\end{aligned} \tag{16.10}$$

We will make the identification $q_{\text{acc}} = q_4$ and $q_{\text{rej}} = q_5$, and we note explicitly that there are $m = 6$ states, $k = 2$ input symbols, and $n = 3$ tape symbols, with the blank symbol $\sqcup$ being identified with the last tape symbol 2.

The transition $\delta(q_0,0) = (q_0,0,\leftarrow)$ is encoded as

$$\big\langle \langle q_0\rangle, \langle 0\rangle, \langle q_0\rangle, \langle 0\rangle, \langle\leftarrow\rangle \big\rangle = \langle 0,0,0,0,0\rangle, \tag{16.11}$$

the transition $\delta(q_0,1) = (q_0,1,\leftarrow)$ is encoded as

$$\big\langle \langle q_0\rangle, \langle 1\rangle, \langle q_0\rangle, \langle 1\rangle, \langle\leftarrow\rangle \big\rangle = \langle 0,1,0,1,0\rangle, \tag{16.12}$$

and, skipping to the last one, the transition $\delta(q_3,2) = (q_5,2,\leftarrow)$ is encoded as

$$\begin{aligned}&\big\langle \langle q_3\rangle, \langle \sqcup\rangle, \langle q_{\text{rej}}\rangle, \langle \sqcup\rangle, \langle\leftarrow\rangle \big\rangle\\ &\quad = \big\langle \langle q_3\rangle, \langle 2\rangle, \langle q_5\rangle, \langle 2\rangle, \langle\leftarrow\rangle \big\rangle = \langle 11,10,101,10,0\rangle.\end{aligned} \tag{16.13}$$

The entire transition function δ is encoded as

$$\begin{aligned}\langle\delta\rangle &= \big\langle \big\langle \langle q_0\rangle, \langle 0\rangle, \langle q_0\rangle, \langle 0\rangle, \langle\leftarrow\rangle \big\rangle, \big\langle \langle q_0\rangle, \langle 1\rangle, \langle q_0\rangle, \langle 1\rangle, \langle\leftarrow\rangle \big\rangle,\\ &\qquad \ldots, \big\langle \langle q_3\rangle, \langle \sqcup\rangle, \langle q_{\text{rej}}\rangle, \langle \sqcup\rangle, \langle\leftarrow\rangle \big\rangle \big\rangle\\ &= \big\langle \langle 0,0,0,0,0\rangle, \langle 0,1,0,1,0\rangle, \ldots, \langle 11,10,101,10,0\rangle \big\rangle.\end{aligned} \tag{16.14}$$

And, finally, and M is encoded as

$$\langle M\rangle = \big\langle \langle 6\rangle, \langle 2\rangle, \langle 3\rangle, \langle \delta\rangle, \langle 0\rangle, \langle 4\rangle, \langle 5\rangle \big\rangle. \tag{16.15}$$

16.2 A universal Turing machine

Now that we have defined an encoding scheme for DTMs, we can consider the computational task of simulating a given DTM on a given input. A *universal Turing machine* is a DTM that can perform such a simulation—it is *universal* in the sense that it is one single DTM that is capable of simulating all other DTMs.

Recall from Lecture 12 that a *configuration* of a DTM

$$M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}) \tag{16.16}$$

may be expressed in the form

$$u(q, a)v \tag{16.17}$$

for a state $q \in Q$, a tape symbol $a \in \Gamma$, and strings of tape symbols $u, v \in \Gamma^*$, where u does not start with a blank and v does not end with a blank. Such a configuration may be encoded as

$$\langle u(q, a)v \rangle = \langle \langle u \rangle, \langle \langle q \rangle, \langle a \rangle \rangle, \langle v \rangle \rangle \tag{16.18}$$

where $\langle u \rangle$, $\langle q \rangle$, $\langle a \rangle$, and $\langle v \rangle$ refer to fitting encoding schemes we have already discussed in this lecture.

Now, if we wish to simulate the computation of a given DTM M on a given input string w, a natural approach is to keep track of the configurations of M and update them appropriately, as the computations themselves dictate. Specifically, we will begin with the initial configuration of M on input w, which is

$$(q_0, \sqcup)w. \tag{16.19}$$

We then repeatedly compute the *next* configuration of M, over and over, until perhaps we eventually reach a configuration whose state is q_{acc} or q_{rej}, at which point we can stop. We might never reach such a configuration; if M runs forever on input w, our simulation will also run forever. There is no way to avoid this, as we shall see.

With this approach in mind, let us focus on the task of simply determining the *next configuration*, meaning the one that results from one computational step, for a given DTM M and a given configuration of M. That is, we will consider the function

$$\text{next} : \{0,1\}^* \to \{0,1\}^* \tag{16.20}$$

defined as follows. For every DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})$ and every *non-halting* configuration $u(q, a)v$ of M, we define

$$\text{next}(\langle \langle M \rangle, \langle u(q, a)v \rangle \rangle) = \langle x(p, b)y \rangle, \tag{16.21}$$

for $x(p,b)y$ being the configuration obtained by running M for one step starting from the configuration $u(q,a)v$, i.e., the unique configuration for which

$$u(q,a)v \vdash_M x(p,b)y. \tag{16.22}$$

We recall that Definition 12.2 describes this relation in precise terms. For every *halting* configuration $u(q,a)v$ of M, let us define (as a matter of convenience)

$$\text{next}(\langle\langle M\rangle, \langle u(q,a)v\rangle\rangle) = \langle u(q,a)v\rangle. \tag{16.23}$$

On inputs $x \in \{0,1\}^*$ *not* having the form $\langle\langle M\rangle, \langle u(q,a)v\rangle\rangle$, for M a DTM and $u(q,a)v$ a configuration of M, we shall take $\text{next}(x) = \varepsilon$.

Let us consider how this function might be computed using a stack machine. A natural first step is to process the input so that, upon conclusion of this processing, we have a stack M that stores $\langle M\rangle$, as well as four stacks, S, T, L, and R, that initially store the encodings $\langle q\rangle$, $\langle a\rangle$, $\langle u\rangle$, and $\langle v\rangle$, respectively. During this processing, we naturally stop and output ε if the input is not an encoding as we expect.

Next, the encoding $\langle M\rangle$, which includes within it the encoding $\langle\delta\rangle$ of the transition function δ, must be examined to determine $\delta(q,a) = (p,b,D)$. This requires a scan through $\langle\delta\rangle$ to obtain, after finding a match involving the strings $\langle q\rangle$ and $\langle a\rangle$ stored by S and T, the encodings $\langle p\rangle$, $\langle b\rangle$, and $\langle D\rangle$. These encodings may each be stored in their own stacks, which we need not name. A processing of S, T, L, and R that updates their contents to $\langle p\rangle$, $\langle b\rangle$, $\langle x\rangle$, and $\langle y\rangle$, for p, b, x, and y satisfying (16.22), is then performed.

Finally, these strings are recombined to form the output string

$$\langle\langle x\rangle, \langle\langle p\rangle, \langle b\rangle\rangle, \langle y\rangle\rangle. \tag{16.24}$$

It would be a time-consuming process to explicitly describe a DSM that operates in this way, but at a conceptual level this computation could reasonably be described as fairly straightforward. String matching subroutines would surely be helpful.

With the function next in hand, one can simulate the computation of a given DTM M on a given input w in the manner suggested above, by starting with the initial configuration $(q_0, \sqcup)w$ of M on w and repeatedly applying the function next.

Now consider the following language, which is the natural DTM analogue of the languages A_{DFA}, A_{NFA}, A_{REG}, and A_{CFG} discussed in the previous lecture:

$$\text{A}_{\text{DTM}} = \bigl\{\langle\langle M\rangle, \langle w\rangle\rangle \;:\; M \text{ is a DTM and } w \in \text{L}(M)\bigr\}. \tag{16.25}$$

We conclude that A_{DTM} is semidecidable: the DSM U described in Figure 16.1 is such that $\text{L}(U) = \text{A}_{\text{DTM}}$. This DSM has been named U to reflect the fact that it is a *universal DSM*. As described in Lecture 14, the DSM U can be simulated by a DTM.

The DSM U operates as follows on input $x \in \{0,1\}^*$:

1. If x takes the form $x = \langle \langle M \rangle, \langle w \rangle \rangle$, for M being a DTM and w being a string over the alphabet of M, then initialize the stack M so that it stores $\langle M \rangle$ and initialize C so that it stores $\langle (q_0, \sqcup) w \rangle$. Reject if not.
2. Repeat the following steps:

   2.1 If C stores a halting configuration of M, then halt and *accept* or *reject* accordingly. If C stores a non-halting configuration of M, the computation continues.

   2.2 Compute the configuration $\text{next}(\langle M \rangle, \langle u(q,a)v \rangle)$, for the encodings $\langle M \rangle$ and $\langle u(q,a)v \rangle$ stored in M and C, and update C so that it stores this new configuration.

Figure 16.1: A high-level description of a DSM U that recognizes the language $\mathrm{A_{DTM}}$.

Proposition 16.2. *The language* $\mathrm{A_{DTM}}$ *is semidecidable.*

Before moving on to the next section, let us note that the following language is decidable:

$$\mathrm{S_{DTM}} = \left\{ \langle \langle M \rangle, \langle w \rangle, \langle t \rangle \rangle \,:\, \begin{array}{l} M \text{ is a DTM, } w \text{ is an input string to } M, \\ t \in \mathbb{N}, \text{ and } M \text{ accepts } w \text{ within } t \text{ steps} \end{array} \right\}. \tag{16.26}$$

This language could be decided by a DTM following essentially the same simulation described previously, but where the simulation cuts off and rejects after t steps if M has not yet halted on input w.

16.3 A few undecidable languages

It is natural at this point to ask whether or not $\mathrm{A_{DTM}}$ is decidable, given that it is semidecidable. It is undecidable, as we will soon prove. Before doing this, however, we will consider a different language and prove that this language is not even semidecidable. Here is the language:

$$\mathrm{DIAG} = \{ \langle M \rangle \,:\, M \text{ is a DTM and } \langle M \rangle \notin \mathrm{L}(M) \}. \tag{16.27}$$

That is, the language DIAG contains all binary strings $\langle M \rangle$ that, with respect to the encoding scheme we discussed at the start of the lecture, encode a DTM M that

The DTM K operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle M \rangle$ for M being a DTM, then reject.
2. Run T on input $\langle \langle M \rangle, \langle M \rangle \rangle$. If T accepts, then *reject*, otherwise *accept*.

Figure 16.2: A DTM that decides DIAG, assuming that there exists a DTM T for A_{DTM}.

does not accept this encoding of itself. Note that if it so happens that the string $\langle M \rangle$ encodes a DTM whose input alphabet has just one symbol, so that it does not include 0 and 1, then it will indeed be the case that $\langle M \rangle \notin \text{L}(M)$.

Theorem 16.3. *The language* DIAG *is not semidecidable.*

Proof. Assume toward contradiction that the language DIAG is semidecidable. There must therefore exist a DTM M such that $\text{L}(M) = \text{DIAG}$.

Now, consider the encoding $\langle M \rangle$ of M. By the definition of the language DIAG one has

$$\langle M \rangle \in \text{DIAG} \;\Leftrightarrow\; \langle M \rangle \notin \text{L}(M). \tag{16.28}$$

On the other hand, because M recognizes DIAG, it is the case that

$$\langle M \rangle \in \text{DIAG} \;\Leftrightarrow\; \langle M \rangle \in \text{L}(M). \tag{16.29}$$

Consequently,

$$\langle M \rangle \notin \text{L}(M) \;\Leftrightarrow\; \langle M \rangle \in \text{L}(M), \tag{16.30}$$

which is a contradiction. We conclude that DIAG is not semidecidable. □

Remark 16.4. 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. It is remarkable how simple this proof of the non-semidecidability of DIAG is; it has used essentially none of the specifics of the DTM model or the encoding scheme we defined.

Now that we know DIAG is not semidecidable, we may prove that A_{DTM} is not decidable.

Theorem 16.5. *The language* A_{DTM} *is undecidable.*

Proof. Assume toward contradiction that A_{DTM} is decidable. There must therefore exist a DTM T that decides A_{DTM}. Define a new DTM K as described in Figure 16.2.

The DTM K operates as follows on input $x \in \{0,1\}^*$:

1. If it is not the case that $x = \langle\langle M\rangle, \langle w\rangle\rangle$ for M being a DTM and w an input string to M, then reject.
2. Run T on input $\langle\langle M\rangle, \langle w\rangle\rangle$ and *reject* if T rejects. Otherwise, continue to the next step.
3. Simulate M on input w; *accept* if M accepts and *reject* if M rejects.

Figure 16.3: A DTM that decides A_{DTM}, assuming that there exists a DTM T that decides HALT.

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 $\langle M\rangle \in \text{DIAG}$, then by the definition of DIAG it is the case that $\langle M\rangle \notin \text{L}(M)$, and therefore $\langle\langle M\rangle, \langle M\rangle\rangle \notin \text{A}_{\text{DTM}}$ (because M does not accept $\langle M\rangle$). This implies that T rejects the input $\langle\langle M\rangle, \langle M\rangle\rangle$, and so K must accept the input $\langle M\rangle$. If, on the other hand, it is the case that $\langle M\rangle \notin \text{DIAG}$, then $\langle M\rangle \in \text{L}(M)$, and therefore $\langle\langle M\rangle, \langle M\rangle\rangle \in \text{A}_{\text{DTM}}$. This implies that T accepts the input $\langle\langle M\rangle, \langle M\rangle\rangle$, and so K must reject the input $\langle M\rangle$. One final possibility is 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 non-semidecidable (and is therefore undecidable). Having obtained a contradiction, we conclude that A_{DTM} is undecidable, as required. $\square$

Here is another example, which is a famous relative of A_{DTM}.

$$\text{HALT} = \big\{\langle\langle M\rangle, \langle w\rangle\rangle \,:\, M \text{ is a DTM that halts on input } w\big\}. \tag{16.31}$$

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 semidecidable, we just run a modified version of our universal Turing machine U on input $\langle\langle M\rangle, \langle w\rangle\rangle$, 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 $\langle\langle M\rangle, \langle w\rangle\rangle$.

Theorem 16.6. *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 in Figure 16.3. The DTM K decides A_{DTM}, as a case analysis reveals:

1. If it is the case that M accepts w, then T will accept $\langle\langle M\rangle,\langle w\rangle\rangle$ (because M halts on w), and the simulation of M on input w will result in acceptance.
2. If it is the case that M rejects w, then T will accept $\langle\langle M\rangle,\langle w\rangle\rangle$ (because M halts on w), and the simulation of M on input w will result in rejection.
3. If it is the case that M runs forever on w, then T will reject $\langle\langle M\rangle,\langle w\rangle\rangle$, 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. □

Lecture 17

More undecidable languages; reductions

In the previous lecture we saw a few examples of undecidable languages: DIAG is not semidecidable, and therefore not decidable, while A_{DTM} and HALT are semidecidable but not decidable. In this lecture we will see a few more examples. We will also introduce the notion of a *reduction* from one language to another, which can be utilized when proving languages are undecidable.

Before doing this, however, we will take a few moments to observe a couple of basic but useful tricks involving Turing machines.

17.1 A couple of basic Turing machine tricks

This section describes two ideas that can be helpful when proving certain languages are undecidable (or non-semidecidable), and in other situations as well. The first involves a simple way to manage a search over an infinite domain and the second is concerned with the surprisingly powerful technique of hard coding inputs of Turing machines.

Limiting infinite search spaces

Sometimes we would like a Turing machine to effectively search over an infinitely large search space, but it may not always be immediately clear how this can be done. One way to address this issue is to set up a loop in which a single positive integer serves simultaneously as a bound on multiple parameters in the search space.

The proof of the following theorem, which is very important in its own right, illustrates this idea.

```
The DTM $M$ operates as follows on input $x \in \{0,1\}^*$:
  1. Set $t \leftarrow 1$.
  2. Run $M_0$ on input $w$ for $t$ steps. If $M_0$ has accepted, then *accept*.
  3. Run $M_1$ on input $w$ for $t$ steps. If $M_1$ has accepted, then *reject*.
  4. Set $t \leftarrow t + 1$ and goto step 2.
```

Figure 17.1: A high-level description of a DTM M that decides A, assuming that M_0 and M_1 are DTMs satisfying $\mathrm{L}(M_0) = A$ and $\mathrm{L}(M_1) = \overline{A}$.

Theorem 17.1. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language such that both A and $\overline{A}$ are semidecidable. The language A is decidable.*

Proof. Because A and $\overline{A}$ are semidecidable 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 described in Figure 17.1.

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. □

To be clear, the technique being suggested involves the use of the variable t in the description of the DTM M in Figure 17.1; it is a single variable, but it is used to limit the simulations of both M_0 and M_1 (in steps 2 and 3) so that neither runs forever. We will see further examples of this technique later in this lecture and in the next lecture.

Hard-coding input strings

Suppose that we have a DTM M along with a string x over the input alphabet of M. Consider the new DTM M_x described in Figure 17.2.

This may seem like a rather useless Turing machine: M_x always leads to the same outcome regardless of what input string it is given. In essence, the input string x has been "hard-coded" directly into the description of M_x. We will see, however, that it is sometimes useful to consider a DTM defined in this way, particularly when proving that certain languages are undecidable or non-semidecidable.

The DTM M operates as follows on input $w \in \{0,1\}^*$:

1. Discard w and run M on input x.

Figure 17.2: The DTM M_x erases its input and runs M on the string x.

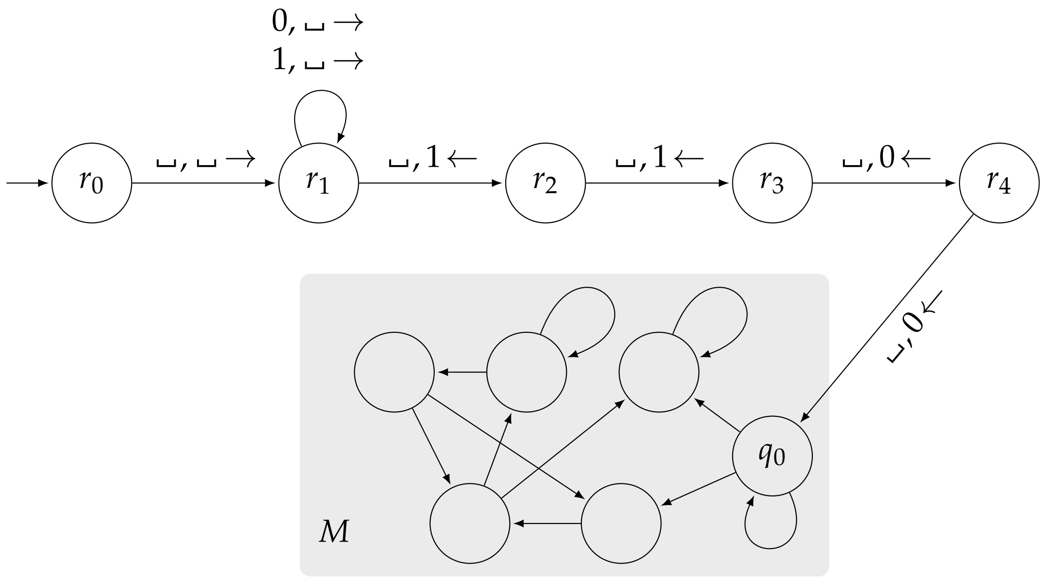


Figure 17.3: A state diagram for the DTM M_{0011}. The contents of the gray box is intended to represent the state diagram of an arbitrary DTM M having start state q_0.

Let us also note that if we have an encoding $\langle\langle M\rangle, \langle x\rangle\rangle$ of both a DTM M and a string x over the input alphabet of M, it is not difficult to compute an encoding $\langle M_x\rangle$ of the DTM M_x. The DTM M_x can be described as having three phases:

1. Move the tape head to the right across whatever input string w has been given, replacing the symbols of w with blanks, until all of w has been erased.
2. Starting at the end of the string x, write each symbol of x on the tape and move the tape head left.
3. Once the string x has been written to the tape, pass control to M.

The first phase can easily be done with a couple of states, and the second phase can be done using one state of M_x for each symbol of x. The third phase operates exactly like M. Figure 17.1 illustrates what the state diagram of the DTM M_{0011} looks like.

The DTM K operates as follows on input $w \in \{0,1\}^*$:

1. Reject unless $w = \langle\langle M\rangle, \langle x\rangle\rangle$ for M being a DTM and x being an input string to M.
2. Compute an encoding $\langle M_x\rangle$ of the DTM M_x defined from M as described in Figure 17.1.
3. Run T on input $\langle M_x\rangle$: if T accepts $\langle M_x\rangle$, then *reject*, otherwise *accept*.

Figure 17.4: The DTM K used in the proof of Proposition 17.2.

Here is an example that illustrates the usefulness of this construction. Define a language

$$\mathrm{E_{DTM}} = \big\{\langle M\rangle \,:\, M \text{ is a DTM with } \mathrm{L}(M) = \varnothing\big\}. \tag{17.1}$$

Proposition 17.2. *The language* $\mathrm{E_{DTM}}$ *is undecidable.*

Proof. Assume toward contradiction that $\mathrm{E_{DTM}}$ is decidable, so that there exists a DTM T that decides this language. Define a new DTM K as in Figure 17.4.

Now, suppose that M is a DTM and $x \in \mathrm{L}(M)$, and consider the behavior of K on input $\langle\langle M\rangle, \langle x\rangle\rangle$. Because M accepts x, it is the case that M_x accepts *every* string over its alphabet—because whatever string you give it as input, it erases this string and runs M on x, leading to acceptance. It is therefore certainly not the case that $\mathrm{L}(M_x) = \varnothing$, so T must reject $\langle M_x\rangle$, and therefore K accepts $\langle\langle M\rangle, \langle w\rangle\rangle$.

On the other hand, if M is a DTM and $x \notin \mathrm{L}(M)$ then K will reject the input $\langle\langle M\rangle, \langle w\rangle\rangle$. Either x is not a string over the alphabet of M, which immediately leads to rejection, or M either rejects or runs forever on input x. In this second case, M_x either rejects or runs forever on every string, and therefore $\mathrm{L}(M_w) = \varnothing$. The DTM T therefore accepts $\langle M_x\rangle$, causing K to reject the input $\langle\langle M\rangle, \langle w\rangle\rangle$.

Thus, K decides $\mathrm{A_{DTM}}$, which contradicts the fact that this language is undecidable. We conclude that $\mathrm{E_{DTM}}$ is undecidable, as required. □

17.2 Proving undecidability through reductions

The proofs that we have seen so far that establish certain languages to be undecidable or non-semidecidable have followed a general pattern that can often be used to prove that a chosen language A is undecidable:

1. Assume toward contradiction that A is decidable.

2. Use that assumption to construct a DTM that decides a language B that we already know to be undecidable.
3. Having obtained a contradiction from the assumption that A is decidable, we conclude that A is undecidable.

A similar approach can sometimes be used to prove that a language A is non-semidecidable, and in both cases we might potentially obtain a contradiction by using our assumption toward contradiction about A to semidecide a language B that we already know to be non-semidecidable.

A different method through which languages may be proved to be undecidable or non-semidecidable makes use of the notion of a *reduction*.

Reductions

The notion of a reduction is, in fact, very general, and many different types of reductions are considered in theoretical computer science—but for now we will consider just one type of reduction (sometimes called a *mapping reduction* or *many-to-one reduction*), which is defined as follows.

Definition 17.3. Let Σ and Γ be alphabets and let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be languages. It is said that A *reduces* to B if there exists a computable function $f : \Sigma^* \rightarrow \Gamma^*$ such that

$$w \in A \Leftrightarrow f(w) \in B \tag{17.2}$$

for all $w \in \Sigma^*$. One writes $A \leq_m B$ 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 17.5 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 $y \in \Gamma^*$ 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 semidecidability.

Theorem 17.4. *Let Σ and Γ be alphabets, let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be languages, and assume $A \leq_m B$. The following two implications hold:*

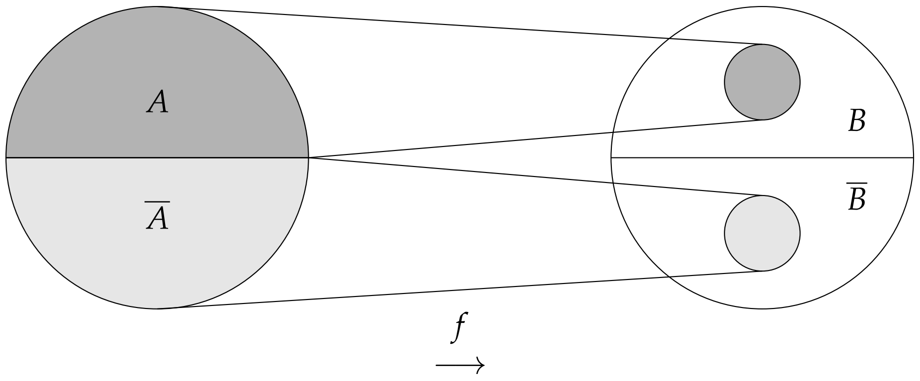


Figure 17.5: An illustration of a reduction f from A to B.

The DTM M_A operates as follows on input $w \in \Sigma^*$:

1. Compute $y = f(w)$.
2. Run M_B on input y.

Figure 17.6: Given a reduction f from A to B, and assuming the existence of a DTM M_B that either decides or semidecides B, the DTM M_A described either decides or semidecides A.

1. *If B is decidable, then A is decidable.*
2. *If B is semidecidable, then A is semidecidable.*

Proof. Let $f : \Sigma^* \rightarrow \Gamma^*$ 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 semidecidable, there must exist a DTM M_B such that $B = \mathrm{L}(M_B)$. Define a new DTM M_A as described in Figure 17.6. 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 accepts y. Along similar lines, if it is the case that $w \notin A$, 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 semidecidable.

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 in Figure 17.6 then decides A, and therefore A is decidable. □

We will soon use this theorem to prove that certain languages are undecidable (or non-semidecidable), but let us first take a moment to observe two useful facts about reductions.

Proposition 17.5. *Let Σ, Γ, and Δ be alphabets and let $A \subseteq \Sigma^*$, $B \subseteq \Gamma^*$, and $C \subseteq \Delta^*$ be languages. If $A \leq_m B$ and $B \leq_m C$, then $A \leq_m C$. (In 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 $f : \Sigma^* \rightarrow \Gamma^*$ and $g : \Gamma^* \rightarrow \Delta^*$ such that

$$w \in A \Leftrightarrow f(w) \in B \quad \text{and} \quad y \in B \Leftrightarrow g(y) \in C \tag{17.3}$$

for all $w \in \Sigma^*$ and $y \in \Gamma^*$.

Define a function $h : \Sigma^* \rightarrow \Delta^*$ 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 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 17.6. *Let Σ and Γ be alphabets and let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be languages. It is the case that $A \leq_m B$ if and only if $\overline{A} \leq_m \overline{B}$.*

Proof. For a given function $f : \Sigma^* \rightarrow \Gamma^*$ and a string $w \in \Sigma^*$, the statements $w \in A \Leftrightarrow f(w) \in B$ and $w \in \overline{A} \Leftrightarrow f(w) \in \overline{B}$ 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. □

Undecidability through reductions

It is possible to use Theorem 17.4 to prove that certain languages are either decidable or semidecidable, but we will focus mainly on using it to prove that languages are either undecidable or non-semidecidable. When using the theorem in this way, we consider the two implications in the contrapositive form. That is, if two languages $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ satisfy $A \leq_m B$, then the following two implications hold:

```
The DTM $K_M$ operates as follows on input $w \in \Sigma^*$:
  1.  Run $M$ on input $w$.
      1.1  If $M$ accepts $w$ then *accept*.
      1.2  If $M$ rejects $w$, then *run forever*.
```

Figure 17.7: Given a DTM M, we can easily obtain a DTM K_M that behaves as described by replacing any transitions to the accept state of M with transitions to a state that intentionally causes an infinite loop.

1. If A is undecidable, then B is undecidable.
2. If A is non-semidecidable, then B is non-semidecidable.

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-semidecidable. The examples that follow illustrate how this may be done.

Example 17.7. For our first example of a reduction, we shall prove

$$\text{A}_{\text{DTM}} \leq_m \text{HALT}. \tag{17.4}$$

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 described in Figure 17.7. The idea behind the DTM K_M is very simple: if M accepts a string w, then so does K_M, if M rejects w then K_M runs forever on w, and of course if M runs forever on input w then so does K_M. Thus, K_M halts on input w if and only if M accepts w. Note that 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: just replace the reject state of M with a new state that purposely causes an infinite loop (by repeatedly moving the tape head right, say).

Now let us define a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ as follows:

$$f(x) = \begin{cases} \langle\langle K_M\rangle, \langle w\rangle\rangle & \text{if } x = \langle\langle M\rangle, \langle w\rangle\rangle \text{ for a DTM } M \text{ and a string } w \\ & \text{over the alphabet of } M \\ x_0 & \text{otherwise,} \end{cases} \tag{17.5}$$

where $x_0 \in \{0,1\}^*$ is any fixed string that is not contained in HALT. (For example, we could take $x_0 = \varepsilon$, because ε does not encode a DTM together with an input string—but it is not important which string we choose as x_0, so long as it is not in 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 $\langle\langle M\rangle, \langle w\rangle\rangle$ of a DTM M and a string w over the alphabet of M, and if so it replaces the encoding of M with the encoding of the DTM K_M suggested above.

Now let us check to see that f is a reduction from A_{DTM} to HALT. Suppose first that we have an input $\langle\langle M\rangle, \langle w\rangle\rangle \in \text{A}_{\text{DTM}}$. These implications hold:

$$\begin{aligned}\langle\langle M\rangle, \langle w\rangle\rangle \in \text{A}_{\text{DTM}} &\Rightarrow M \text{ accepts } w \Rightarrow K_M \text{ halts on } w\\ &\Rightarrow \langle\langle K_M\rangle, \langle w\rangle\rangle \in \text{HALT} \Rightarrow f(\langle\langle M\rangle, \langle w\rangle\rangle) \in \text{HALT}.\end{aligned} \tag{17.6}$$

We therefore have

$$\langle\langle M\rangle, \langle w\rangle\rangle \in \text{A}_{\text{DTM}} \Rightarrow f(\langle\langle M\rangle, \langle w\rangle\rangle) \in \text{HALT}, \tag{17.7}$$

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 $\langle\langle M\rangle, \langle w\rangle\rangle$ for a DTM M and a string w over the alphabet of M, and the other is that it does not. For the first case, we have these implications:

$$\begin{aligned}\langle\langle M\rangle, \langle w\rangle\rangle \notin \text{A}_{\text{DTM}} &\Rightarrow M \text{ does not accept } w\\ &\Rightarrow K_M \text{ runs forever on } w \Rightarrow \langle\langle K_M\rangle, \langle w\rangle\rangle \notin \text{HALT}\\ &\Rightarrow f(\langle\langle M\rangle, \langle w\rangle\rangle) \notin \text{HALT}.\end{aligned} \tag{17.8}$$

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 $\langle\langle M\rangle, \langle w\rangle\rangle$ for a DTM M and a string w over the alphabet of M, and in this case it trivially holds that $f(x) = x_0 \notin \text{HALT}$. (This is why we defined f as we did in this case, and we will generally do something similar for other examples).

We have therefore proved that

$$x \in \text{A}_{\text{DTM}} \Leftrightarrow f(x) \in \text{HALT}, \tag{17.9}$$

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 was not particularly difficult). We have, however, established a closer relationship between $\mathrm{A}_{\mathrm{DTM}}$ 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-semidecidable).

Example 17.8. For our next example of a reduction, we will prove

$$\mathrm{DIAG} \leq_m \mathrm{E}_{\mathrm{DTM}}, \tag{17.10}$$

where we recall that $\mathrm{E}_{\mathrm{DTM}}$ is defined as follows:

$$\mathrm{E}_{\mathrm{DTM}} = \big\{ \langle M \rangle \,:\, M \text{ is a DTM and } \mathrm{L}(M) = \varnothing \big\}. \tag{17.11}$$

We will now prove that $\mathrm{DIAG} \leq_m \mathrm{E}_{\mathrm{DTM}}$. Because we already know that DIAG is non-semidecidable, we conclude from this reduction that $\mathrm{E}_{\mathrm{DTM}}$ is not just undecidable, but in fact it is also non-semidecidable.

For this one we will again use the hardcoding trick from the beginning of the lecture: for a given DTM M, let us define a new DTM $M_{\langle M \rangle}$ just like in Figure 17.2, for the specific choice of the hardcoded string $x = \langle M \rangle$. This actually only makes sense if the input alphabet of M includes the symbols $\{0,1\}$ used in the encoding $\langle M \rangle$, so let us agree that $M_{\langle M \rangle}$ immediately rejects if this is not the case.

Now let us define a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ as follows:

$$f(x) = \begin{cases} \langle M_{\langle M \rangle} \rangle & \text{if } x = \langle M \rangle \text{ for a DTM } M \\ x_0 & \text{otherwise,} \end{cases} \tag{17.12}$$

for any fixed binary string x_0 not contained in $\mathrm{E}_{\mathrm{DTM}}$. If you think about it for a moment, it should not be hard to convince yourself that f is computable. It remains to verify that f is a reduction from DIAG to $\mathrm{E}_{\mathrm{DTM}}$.

For any string $x \in \mathrm{DIAG}$ we have that $x = \langle M \rangle$ for some DTM M that satisfies $\langle M \rangle \notin \mathrm{L}(M)$. In this case we have that $f(x) = \big\langle M_{\langle M \rangle} \big\rangle$, and because $\langle M \rangle \notin \mathrm{L}(M)$ it must therefore be that $M_{\langle M \rangle}$ never accepts, and so $f(x) = \big\langle M_{\langle M \rangle} \big\rangle \in \mathrm{E}_{\mathrm{DTM}}$.

Now suppose that $x \notin \mathrm{DIAG}$. There are two cases: either $x = \langle M \rangle$ for a DTM M such that $\langle M \rangle \in \mathrm{L}(M)$, 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 $\langle M \rangle \in \mathrm{L}(M)$, we have that $M_{\langle M \rangle}$ accepts *every* string over its alphabet, and therefore $f(x) = \big\langle M_{\langle M \rangle} \big\rangle \notin \mathrm{E}_{\mathrm{DTM}}$. If it is the case that x does not encode a DTM, then it trivially holds that $f(x) = x_0 \notin \mathrm{E}_{\mathrm{DTM}}$.

We have proved that

$$x \in \mathrm{DIAG} \Leftrightarrow f(x) \in \mathrm{E}_{\mathrm{DTM}}, \tag{17.13}$$

so the proof that $\mathrm{DIAG} \leq_m \mathrm{E}_{\mathrm{DTM}}$ is complete.

Example 17.9. Our third example of a reduction is

$$\text{A}_{\text{DTM}} \leq_m \text{AE}, \tag{17.14}$$

where the language AE is defined like this:

$$\text{AE} = \big\{\langle M\rangle : M \text{ is a DTM that accepts } \varepsilon\big\}. \tag{17.15}$$

The name AE stands for "accepts the empty string."

To prove this reduction, we can use the same hardcoding trick that we have now used twice already. For every DTM M and every string x over the alphabet of M, define a new DTM M_x as in Figure 17.2, and define a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ as follows:

$$f(w) = \begin{cases} \langle M_x\rangle & \text{if } w = \langle\langle M\rangle, \langle x\rangle\rangle \text{ for a DTM } M \text{ and a string } x \\ & \text{over the alphabet of } M \\ w_0 & \text{otherwise,} \end{cases} \tag{17.16}$$

where, as you likely now expect, w_0 is any fixed binary string not contained in AE. Now let us check that f is a valid reduction from A_{DTM} to AE.

First, for any string $w \in \text{A}_{\text{DTM}}$ we have $w = \langle\langle M\rangle, \langle x\rangle\rangle$ for a DTM M that accepts the string x. In this case, $f(w) = \langle M_x\rangle$. We have that M_x accepts every string, including the empty string, because M accepts x. Therefore $f(w) = \langle M_x\rangle \in$ AE.

Now consider any string $w \notin \text{A}_{\text{DTM}}$, for which there are two cases. If it is the case that $w = \langle\langle M\rangle, \langle x\rangle\rangle$ for a DTM M and x a string over the alphabet of M, then $w \notin \text{A}_{\text{DTM}}$ implies that M does not accept x. In this case we have $f(w) = \langle M_x\rangle \notin$ AE, because M_x does not accept any strings at all (including the empty string). If $w \neq \langle\langle M\rangle, \langle x\rangle\rangle$ for a DTM M and string x over the alphabet of M, then $f(w) = w_0 \notin$ AE.

We have shown that $w \in \text{A}_{\text{DTM}} \Leftrightarrow f(w) \in \text{AE}$ for every string $w \in \{0,1\}^*$, and therefore $\text{A}_{\text{DTM}} \leq_m \text{AE}$, as required.

Example 17.10. The last example of a reduction for the lecture will be a bit more difficult than the others. We will prove that

$$\text{E}_{\text{DTM}} \leq_m \text{DEC} \tag{17.17}$$

where

$$\text{DEC} = \big\{\langle M\rangle : M \text{ is a DTM such that L}(M) \text{ is decidable}\big\}. \tag{17.18}$$

```
The DTM $K_M$ operates as follows on input $x \in \{0,1\}^*$:

1.  Set $t \leftarrow 1$.
2.  For every string $w$ over the input alphabet of $M$ satisfying $|w| \leq t$:
    2.1  Run $M$ for $t$ steps on input $w$, and if $M$ accepts then goto step 4.
3.  Set $t \leftarrow t + 1$ and goto step 2.
4.  Run $H$ on $x$.
```

Figure 17.8: The DTM K_M in Example 17.10. We assume that H is any fixed DTM with $\mathrm{L}(H) = \mathrm{HALT}$.

Notice that the inclusion $\langle M \rangle \in \mathrm{DEC}$ does not imply that M always halts—but rather that there exists some DTM K, not necessarily M, that decides the language recognized by M.

Given an arbitrary DTM M, let us define a new DTM K_M as in Figure 17.8. In this description, assume H is any fixed DTM satisfying $\mathrm{L}(H) = \mathrm{HALT}$. We know there is such an H; we can easily adapt a universal DTM U so that it semidecides HALT. If one asks why we define K_M in this way, the answer is nothing more than that it makes the reduction work—but notice that within the definition of K_M we are making use of the infinite search technique from the start of the lecture.

Now let us define a function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ as

$$f(x) = \begin{cases} \langle K_M \rangle & \text{if } x = \langle M \rangle \text{ for a DTM } M \\ x_0 & \text{otherwise,} \end{cases} \tag{17.19}$$

where x_0 is any fixed binary string not contained in DEC. This is a computable function, and it remains to verify that it is a reduction from $\mathrm{E}_{\mathrm{DTM}}$ to DEC.

Suppose $\langle M \rangle \in \mathrm{E}_{\mathrm{DTM}}$. We therefore have that $\mathrm{L}(M) = \varnothing$; and by considering the way that K_M behaves we see that $\mathrm{L}(K_M) = \varnothing$ as well; the computation alternates between steps 2 and 3 forever if M never accepts. The empty language is decidable, and therefore $f(\langle M \rangle) = \langle K_M \rangle \in \mathrm{DEC}$.

On the other hand, if M is a DTM and $\langle M \rangle \notin \mathrm{E}_{\mathrm{DTM}}$, then M must accept at least one string. This means that $\mathrm{L}(K_M) = \mathrm{HALT}$, because K_M will eventually find a string accepted by M, reach step 4, and then accept if and only if $x \in \mathrm{HALT}$. Therefore $f(\langle M \rangle) = \langle K_M \rangle \notin \mathrm{DEC}$. The remaining case in which x does not encode a DTM is, as always, straightforward: in this case we have $f(x) = x_0 \notin \mathrm{DEC}$.

We have shown that $x \in \mathrm{E}_{\mathrm{DTM}} \Leftrightarrow f(x) \in \mathrm{DEC}$ for every string $x \in \{0,1\}^*$, and therefore $\mathrm{E}_{\mathrm{DTM}} \leq_m \mathrm{DEC}$, as required.

We conclude that the language DEC is non-semidecidable, as we already know that $\mathrm{E}_{\mathrm{DTM}}$ is non-semidecidable.

Lecture 18

Further discussion of computability

In this lecture we will discuss a few points relating to Turing machines and computability that were not covered in previous lectures. We will begin with some basic closure properties of decidable and semidecidable languages. We will then briefly discuss nondeterministic Turing machines, and relate them to the semidecidable and decidable languages. Finally we will prove an interesting characterization of semidecidability, which is that a nonempty language is semidecidable if and only if it is equal to the *range* of a computable function.

18.1 Closure properties of decidable and semidecidable languages

The decidable and semidecidable languages are closed under many (but not all) of the operations on languages that we have considered thus far in the course. Some examples are described in this section.

Closure properties of decidable languages

First let us observe that the decidable languages are closed under the regular operations. The proof is quite straightforward.

Proposition 18.1. *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 DTMs described in Figures 18.1, 18.2, and 18.3 decide the languages $A \cup B$, AB, and A^*, respectively. It follows that these languages are all decidable. □

```
The DTM $M$ operates as follows on input $w \in \Sigma^*$:

1. Run $M_A$ on input $w$.
2. Run $M_B$ on input $w$.
3. If either $M_A$ or $M_B$ has accepted, then accept, otherwise reject.
```

Figure 18.1: A DTM M that decides $A \cup B$, given DTMs M_A and M_B that decide A and B, respectively.

```
The DTM $M$ operates as follows on input $w \in \Sigma^*$:

1. For every choice of strings $u, v \in \Sigma^*$ satisfying $w = uv$:
    1.1 Run $M_A$ on input $u$.
    1.2 Run $M_B$ on input $v$.
    1.3 If both $M_A$ and $M_B$ have accepted, then accept.
2. Reject.
```

Figure 18.2: A DTM M that decides AB, given DTMs M_A and M_B that decide A and B, respectively.

```
The DTM $M$ operates as follows on input $w \in \Sigma^*$:

1. If $w = \varepsilon$, then accept.
2. For every way of writing $w = u_1 \cdots u_m$ for nonempty strings
   $u_1, \ldots, u_m$:
    2.1 Run $M_A$ on each of the strings $u_1, \ldots, u_m$.
    2.2 If $M_A$ has accepted on all of these runs, then accept.
3. Reject.
```

Figure 18.3: A DTM M that decides A^*, given a DTM M_A that decides A.

The decidable languages are also closed under complementation, as the next proposition states. Perhaps this one is simple enough that we can safely skip the proof; it is clear that if a DTM M decides A, and we simply swap the accept and reject states of M, we obtain a DTM deciding $\overline{A}$.

Proposition 18.2. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a decidable language. The language $\overline{A}$ is decidable.*

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 its input string and then deciding whether or not that string 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 18.3. The language $\text{Prefix}(A)$ might not be decidable, even if A is decidable. To construct an example that illustrates that this is so, let us first take H to be a DTM satisfying $\mathrm{L}(H) = \text{HALT}$. We may then define a language $A \subseteq \{0, 1, \#\}^*$ as follows:

$$A = \{w\#0^t \;:\; H \text{ accepts } w \text{ within } t \text{ steps}\}. \tag{18.1}$$

This is a decidable language, but $\text{Prefix}(A)$ is not—for if $\text{Prefix}(A)$ were decidable, then one could easily decide HALT by using the fact that a string $w \in \{0, 1\}^*$ is contained in HALT if and only if $w\# \in \text{Prefix}(A)$; both inclusions are equivalent to the existence of a positive integer t such that $w\#0^t \in A$.

Closure properties of semidecidable languages

The semidecidable 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 semidecidable languages are indeed closed. In this case, one needs to be a bit more careful than in the proof of the analogous proposition for decidable languages; the DTMs semideciding the languages in question might run forever on inputs not in those languages. However, it is nothing that the method for searching infinite spaces from the previous lecture cannot handle.

```
The DTM $M$ operates as follows on input $w \in \Sigma^*$:
 1. Set $t \leftarrow 1$.
 2. Run $M_A$ on input $w$ for $t$ steps.
 3. Run $M_B$ on input $w$ for $t$ steps.
 4. If either $M_A$ or $M_B$ has accepted, then accept.
 5. Set $t \leftarrow t + 1$ and goto step 2.
```

Figure 18.4: A DTM M that semidecides $A \cup B$, given DTMs M_A and M_B that semidecide A and B, respectively.

```
The DTM $M$ operates as follows on input $w \in \Sigma^*$:
 1. Set $t \leftarrow 1$.
 2. For every choice of strings $u, v$ satisfying $w = uv$:
     1.1 Run $M_A$ on input $u$ for $t$ steps.
     1.2 Run $M_B$ on input $v$ for $t$ steps.
     1.3 If both $M_A$ and $M_B$ have accepted, then accept.
 3. Set $t \leftarrow t + 1$ and goto step 2.
```

Figure 18.5: A DTM M that semidecides AB, given DTMs M_A and M_B that semidecide A and B, respectively.

Proposition 18.4. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be semidecidable languages. The languages $A \cup B$, AB, and A^* are semidecidable.*

Proof. Because the languages A and B are semidecidable, there must exist DTMs M_A and M_B such that $\mathrm{L}(M_A) = A$ and $\mathrm{L}(M_B) = B$. The DTMs described in Figures 18.4, 18.5, and 18.6 semidecide the languages $A \cup B$, AB, and A^*, respectively. It follows that these languages are all semidecidable. □

The semidecidable 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 do not actually need to be as careful about running forever.

```
The DTM $M$ operates as follows on input $w \in \Sigma^*$:
  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
     $u_1, \ldots, u_m$:
       3.1 Run $M_A$ on each of the strings $u_1, \ldots, u_m$ for $t$ steps.
       3.2 If $M_A$ has accepted in all $m$ of these runs, then accept.
  4. Set $t \leftarrow t + 1$ and goto step 3.
```

Figure 18.6: A DTM M that semidecides A^*, given a DTM M_A that semidecides A.

```
The DTM $M$ operates as follows on input $w \in \Sigma^*$:
  1. Run $M_A$ on input $w$. If $M_A$ rejects $w$, then reject.
  2. Run $M_B$ on input $w$. If $M_B$ rejects $w$, then reject.
  3. Accept.
```

Figure 18.7: A DTM M that semidecides $A \cap B$, given DTMs M_A and M_B that semidecide A and B, respectively. Note that if either M_A or M_B runs forever on input w, then so does M, but this does not change the fact that M semidecides $A \cap B$.

Proposition 18.5. *Let Σ be an alphabet and let $A, B \subseteq \Sigma^*$ be semidecidable languages. The language $A \cap B$ is semidecidable.*

Proof. Because the languages A and B are semidecidable, there must exist DTMs M_A and M_B such that $\mathrm{L}(M_A) = A$ and $\mathrm{L}(M_B) = B$. The DTM M described in Figure 18.7 semidecides $A \cap B$, which implies that $A \cap B$ is semidecidable. □

The semidecidable languages are not closed under complementation. This follows from Theorem 17.1 from the previous lecture. In particular, if $\overline{A}$ were semidecidable for every semidecidable language A, then we would conclude from that theorem that every semidecidable language is decidable, which is not the case. For example, HALT is semidecidable but not decidable.

Finally, there are some operations under which the semidecidable languages are closed, but under which the decidable languages are not. For example, if A is semidecidable, then so are the languages $\text{Prefix}(A)$, $\text{Suffix}(A)$, and $\text{Substring}(A)$.

18.2 Nondeterministic Turing machines

We have focused on a deterministic variant of the Turing machine model, but it is possible to define a nondeterministic variant of it. This is done in a way that is completely analogous to the definition of nondeterministic finite automata, as compared with deterministic finite automata.

Definition 18.6. A *nondeterministic Turing machine* (or NTM, for short) is a 7-tuple

$$N = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}}), \tag{18.2}$$

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

$$\delta : Q \backslash \{q_{\text{acc}}, q_{\text{rej}}\} \times \Gamma \rightarrow \mathcal{P}\big(Q \times \Gamma \times \{\leftarrow, \rightarrow\}\big), \tag{18.3}$$

$q_0 \in Q$ is the initial state, and $q_{\text{acc}}, q_{\text{rej}} \in Q$ are the accept and reject states, which must satisfy $q_{\text{acc}} \neq q_{\text{rej}}$.

As one would naturally guess, nondeterministic Turing machines may have multiple choices for which actions can be performed on a given step. Specifically, for a given nonhalting state $q \in Q\backslash\{q_{\text{acc}}, q_{\text{rej}}\}$ and tape symbol a, the value $\delta(q, a)$ taken by the transition function is a subset of $Q \times \Gamma \times \{\leftarrow, \rightarrow\}$, rather than a single element of this set, indicating a set of choices for the machine's actions when it is in state q and reading the tape symbol a.

For example, if we have an NTM N whose transition function satisfies

$$\delta(p, a) = \{(q, b, \leftarrow), (r, c, \rightarrow)\}, \tag{18.4}$$

then when N is in state p and scanning a tape square containing the symbol a, then it may either change state to q, overwrite a with b on the tape, and move the tape head left; or it may change state to r, overwrite a with c, and move the tape head right.

Yields relation and acceptance for an NTM

The yields relation of an NTM is defined in a similar manner to DTMs, allowing for multiple choices of configurations that are reachable from a given one. Specifically, the definition is precisely the same as Definition 12.2, except that the conditions

$$\delta(p,a) = (q,b,\rightarrow) \quad \text{and} \quad \delta(p,a) = (q,b,\leftarrow) \tag{18.5}$$

are replaced by

$$(q,b,\rightarrow) \in \delta(p,a) \quad \text{and} \quad (q,b,\leftarrow) \in \delta(p,a), \tag{18.6}$$

respectively.

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. Formally speaking, an NTM N accepts an input string w if there exist strings $u, v \in \Gamma^*$ and a symbol $a \in \Gamma$ such that

$$(q_0, \sqcup)w \vdash_N^* u(q_{\text{acc}}, a)v. \tag{18.7}$$

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.

Computation trees

When we think about the computation of an NTM N on an input string w, it is natural to envision a *computation tree*. This is a (possibly infinite) tree in which each node is labeled by a configuration: the root node of the tree is labeled by the initial configuration of N on input w, and in general each node in the tree has one child labeled by each configuration that can be reached in one step from the one labeling the original node. As a point of clarification, note that distinct nodes in the computation tree may be labeled by the same configuration. That is to say, there could be more than one *computational path* that leads to a given configuration. Finally, note that the leaves of the computation tree are the ones labeled by halting configurations.

It should perhaps be noted that the computation tree of a *deterministic* Turing machine on a given input can be defined in exactly the same way, but it just does not happen to be very interesting. It is a stick, finite or infinite, without branching.

In terms of the computation tree of N on input w, saying that N accepts w is equivalent to saying that *somewhere* in the computation tree there exists an accepting configuration. There might or might not exist rejecting configurations in the computation tree, and there might or might not exist infinitely long branches in the tree, but it does not matter: 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.

Semidecidability and decidability through NTMs

As the following theorem states, the class of languages recognized by NTMs is exactly the same as for DTMs.

Theorem 18.7. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language A is semidecidable if and only if there exists an NTM N such that $\mathrm{L}(N) = A$.*

It is not difficult to prove this theorem, but these notes will only briefly discuss the main idea behind the proof.

It is clear that every semidecidable language is recognized by some NTM, as we may view a DTM as being an NTM just like we can view a DFA as being an NFA. The other implication required to prove the theorem is that every language recognized by an NTM is semidecidable. That is, one must prove that if $A = \mathrm{L}(N)$ for an NTM N, then A is semidecidable. The key idea through which this may be proved is to perform a *breadth-first search* of the computation tree of N on a given input, using a DTM. 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.

It should perhaps be noted 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.

One may also characterize decidable languages through nondeterministic Turing machines, although the required conditions on nondeterministic computations for decidable languages is not quite as simple to state. The following theorem provides such a characterization.

Theorem 18.8. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language A is decidable if and only if there exists an NTM N for which the following two conditions are met:*

1. $A = \mathrm{L}(N)$.
2. *For every input string $w \in \Sigma^*$, the computation tree of N on input w is finite.*

This theorem can be proved using exactly the same technique, using breadth-first search, as the previous theorem. In this case, the DTM performing the search

The DTM M operates as follows on input $w \in \Sigma^*$:

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 Γ^* and goto step 2.

Figure 18.8: A DTM M that semidecides $A = \operatorname{range}(f)$ for a computable function $f : \Gamma^* \rightarrow \Sigma^*$.

rejects whenever the entire computation tree has been searched and an accepting configuration has not been found.

18.3 The range of a computable function

Finally, we will observe an interesting alternative characterization of semidecidable languages (excepting 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 $f : \Gamma^* \rightarrow \Sigma^*$ is defined as follows:

$$\operatorname{range}(f) = \{ f(w) : w \in \Gamma^* \}. \tag{18.8}$$

Theorem 18.9. *Let Σ and Γ be alphabets and let $A \subseteq \Sigma^*$ be a nonempty language. The following two statements are equivalent:*

1. *A is semidecidable.*
2. *There exists a computable function $f : \Gamma^* \rightarrow \Sigma^*$ such that $A = \operatorname{range}(f)$.*

Proof. Let us first prove that the second statement implies the first. That is, we will prove that if there exists a computable function $f : \Gamma^* \rightarrow \Sigma^*$ such that $A = \operatorname{range}(f)$, then A is semidecidable. Consider the DTM M described in Figure 18.8. In essence, this DTM searches over Γ^* to find a string that f maps to a given input string w. If it is the case that $w \in \operatorname{range}(f)$, then M will eventually find $x \in \Gamma^*$ such that $f(x) = w$ and accept, while M will run forever if $w \notin \operatorname{range}(f)$. Thus, we have $\mathrm{L}(M) = \operatorname{range}(f) = A$, which implies that A is semidecidable.

For the other implication, which is slightly more difficult, suppose that A is semidecidable. Thus, there exists a DTM M such that $\mathrm{L}(M) = A$. 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.

Now define a function $f : \Gamma^* \to \Sigma^*$ as follows:

$$f(x) = \begin{cases} w & \text{if } x = \langle \langle w \rangle, \langle t \rangle \rangle, \text{ for } w \in \Sigma^* \text{ and } t \in \mathbb{N}, \\ & \text{and } M \text{ accepts } w \text{ within } t \text{ steps} \\ w_0 & \text{otherwise.} \end{cases} \tag{18.9}$$

Here we assume that $\langle \langle w \rangle, \langle t \rangle \rangle$ refers to any encoding scheme through which the string $w \in \Sigma^*$ and the natural number $t \in \mathbb{N}$ are encoded into a single string over the alphabet Γ. As we discussed earlier in the course, this is possible for any alphabet Γ, even if it contains only a single symbol.

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 $\langle \langle w \rangle, \langle t \rangle \rangle$, 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 be that $w = f(\langle \langle w \rangle, \langle t \rangle \rangle)$ for some sufficiently large natural number t, so $A \subseteq \text{range}(f)$. 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 is therefore the case that $A = \text{range}(f)$, which completes the proof. □

Remark 18.10. 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.

Theorem 18.9 provides a very useful characterization of semidecidable languages. For instance, one can use this theorem to come up with alternative proofs for all of the closure properties of the semidecidable languages stated in the first section of this lecture.

For example, suppose that $A, B \subseteq \Sigma^*$ are nonempty semidecidable languages. By Theorem 18.9, for whatever alphabet Γ we choose, there must exist computable functions $f : \Gamma^* \to \Sigma^*$ and $g : \Gamma^* \to \Sigma^*$ such that $\text{range}(f) = A$ and $\text{range}(g) = B$. Define a new function

$$h : (\Gamma \cup \{\#\})^* \to \Sigma^* \tag{18.10}$$

as follows:

$$g(x) = \begin{cases} f(y)g(z) & \text{if } x = y\#z \text{ for } y \in \Gamma^* \text{ and } z \in \Gamma^* \\ f(\varepsilon)g(\varepsilon) & \text{otherwise.} \end{cases} \tag{18.11}$$

Here, we are naturally assuming that $\# \notin \Gamma$. One sees that h is computable, and $\text{range}(h) = AB$, which implies that AB is semidecidable.

The DTM M operates as follows on input $w \in \Sigma^*$:

1. Set $x \leftarrow \varepsilon$.
2. Compute $y \leftarrow f(x)$.
3. If $y = w$ then *accept*.
4. If $y > w$ (with respect to the lexicographic ordering of Σ^*) then *reject*.
5. Increment x with respect to the lexicographic ordering of Σ^* and goto step 2.

Figure 18.9: A DTM M for Corollary 18.11.

Here is another example of an application of Theorem 18.9, establishing that every infinite semidecidable language must have an *infinite* decidable language as a subset.

Corollary 18.11. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be any infinite semidecidable language. There exists an infinite decidable language $B \subseteq A$.*

Proof. Because A is infinite (and therefore nonempty), there exists a computable function $f : \Sigma^* \rightarrow \Sigma^*$ such that $A = \text{range}(f)$.

We will now define a language B by first defining a DTM M and then taking $B = \text{L}(M)$. In order for us to be sure that B satisfies the requirements of the corollary, it will need to be proved that M never runs forever (so that B is decidable), that M only accepts strings that are contained in A (so that $B \subseteq A$), and that M accepts infinitely many different strings (so that B is infinite). The DTM M is described in Figure 18.9.

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 $f(x) > w$, causing 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:

$$\left\{ w \in \Sigma^* \; : \; \begin{array}{l} \text{there exists } x \in \Sigma^* \text{ such that } w = f(x) \\ \text{and } w > f(z) \text{ for all } z < x \end{array} \right\}. \tag{18.12}$$

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 $B = \mathrm{L}(M)$ therefore satisfies the requirements of the corollary, which completes the proof. $\square$

Lecture 19

Time-bounded computations

In the final couple of lectures of the course, we will discuss the topic of *computational complexity theory*, which is concerned with the inherent difficulty (or *hardness*) of computational problems and the effect of resource constraints on models of computation. We will only have time to scratch the surface; complexity theory is a rich subject, and many researchers 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.

In this lecture we will focus on the most important resource (from the view of computational complexity theory), which is *time*. The motivation is, in some sense, obvious: in order to be useful, computations generally need to be performed within a reasonable amount of time. In an extreme situation, if we have some computational task that we would like to perform, and someone gives us a computational device that will perform this computational task, but only after running for one million years or more, it is practically useless. One can also consider other resources besides time, such as space (or memory usage), communication in a distributed scenario, or a variety of more abstract notions concerning resource usage.

We will start with a definition of the running time of a DTM.

Definition 19.1. Let M be a DTM with input alphabet Σ. For each string $w \in \Sigma^*$, let $T(w)$ denote the number of steps (possibly infinite) for which M runs on input w. The *running time* of M is the function $t : \mathbb{N} \rightarrow \mathbb{N} \cup \{\infty\}$ defined as

$$t(n) = \max\{T(w) \,:\, w \in \Sigma^*,\ |w| = n\} \tag{19.1}$$

for every $n \in \mathbb{N}$. In words, $t(n)$ is the maximum number of steps required for M to halt, over all input strings of length n.

We will restrict our attention to DTMs whose running time is finite for all input lengths.

19.1 DTIME and time-constructible functions

Deterministic time complexity classes

For every function $f : \mathbb{N} \rightarrow \mathbb{N}$, we define a class of languages called DTIME(f), which represents those languages decidable in time $O(f(n))$.

Definition 19.2. Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a function. A language A is contained in the class DTIME(f) if there exists a DTM M that decides A and whose running time t satisfies $t(n) = O(f(n))$.

We define 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 generally reserve 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 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

$$\text{DTIME}\big(n\sqrt{n}\big) \quad \text{or} \quad \text{DTIME}\big(n^2\log(n)\big), \tag{19.2}$$

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 nonnegative integer. For instance, DTIME$(n^2\log(n))$ means DTIME(f) for

$$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \lceil n^2\log(n) \rceil & \text{otherwise.} \end{cases} \tag{19.3}$$

Example 19.3. The DTM for the language SAME $= \{0^m1^m : m \in \mathbb{N}\}$ from Lecture 12 runs in time $O(n^2)$ on inputs of length n, and therefore SAME is contained in the class DTIME(n^2).

It is, in fact, possible to do better: it is the case that SAME $\in$ DTIME$(n\log(n))$. To do this, one may define 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, SAME 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 SAME. The answer is that this is not possible. This is a consequence of the following theorem (which we will not prove).

Theorem 19.4. *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*

$$\lim_{n\to\infty} \frac{t(n)}{n\log(n)} = 0, \tag{19.4}$$

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 SAME, 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 from the viewpoint of computational complexity—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

$$\text{DTIME}(f) = \text{DTIME}(g), \qquad \text{for } g(n) = 2^{f(n)}; \tag{19.5}$$

even though g is exponentially larger than f, they both result in exactly the same deterministic time complexity class. This does not necessarily imply something important about time complexity, it is 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 19.5. Let $f : \mathbb{N} \rightarrow \mathbb{N}$ 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 outputs $f(n)$ (written in binary notation), 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 does not take more than $O(f(n))$ steps—and then it has the number $f(n)$ stored in binary notation so that it can then 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 function $f(n) = n^k$ is time constructible.
2. For any choice of an integer $k \geq 2$, the function $f(n) = k^n$ is time constructible.
3. For any choice of an integer $k \geq 1$, the functions

   $$f(n) = \begin{cases} 0 & \text{if } n = 0 \\ \lceil n^k \log(n) \rceil & \text{otherwise} \end{cases} \tag{19.6}$$

   and

   $$f(n) = \left\lceil n^k \sqrt{n} \right\rceil \tag{19.7}$$

   are time constructible.
4. If f and g are time-constructible functions, then the functions

   $$h_1(n) = f(n) + g(n), \quad h_2(n) = f(n)g(n), \quad \text{and} \quad h_3(n) = f(g(n)) \tag{19.8}$$

   are also time constructible.

19.2 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

The DTM K operates as follows on input $w \in \{0,1\}^*$:

1. If the input w does not take the form $w = \langle M \rangle 01^k$ for a DTM M with input alphabet $\{0,1\}$ and $k \in \mathbb{N}$, then reject.
2. Compute $t = f(|w|)$.
3. Simulate M on input w for t steps. If M has rejected w within t steps, then accept, otherwise reject.

Figure 19.1: This DTM decides a language that cannot be decided in time $o(f(n))$.

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).

Suppose that a time-constructible function f has been selected, and define a DTM K as described in Figure 19.1. It is not immediately apparent what the running time is for K, because this depends on precisely how the simulation of M in step 3 is done; different ways of performing the simulation could of course lead to different running times. For the time being, let us take g to be the running time of K, and we will worry later about how specifically g relates to f.

Next, let us think about the language $\mathrm{L}(K)$ decided by K. This is a language over the binary alphabet, and 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 $o(f(n))$.

To this end, 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

$$w \in \mathrm{L}(K) \;\Leftrightarrow\; w \notin \mathrm{L}(M). \tag{19.9}$$

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 is 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 are 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 $f : \mathbb{N} \rightarrow \mathbb{N}$, the proper subset relation

$$\mathrm{DTIME}(h) \subsetneq \mathrm{DTIME}(g) \tag{19.10}$$

holds whenever $h(n) = o(f(n))$, where g is the running time of K (which depends somehow on f).

Remark 19.6. There is an aspect of the argument just presented that is worth noting. We obtained the language $\mathrm{L}(K)$, which is contained in $\mathrm{DTIME}(g)$ but not $\mathrm{DTIME}(h)$ assuming $h(n) = o(f(n))$, not by actually describing the language explicitly, but by simply describing the DTM K that decides it. Indeed, in this case it is hard to imagine a description of the language $\mathrm{L}(K)$ that would be significantly more concise than the description of K itself. This technique can be useful in other situations. Sometimes, when you wish to prove the existence of a language having a certain property, rather than explicitly defining the language, it is possible to define a DTM M that operates in a particular way, and then take the language you are looking for to be $\mathrm{L}(M)$.

If you work very hard to make K run as efficiently as possible, the following theorem can be obtained.

Theorem 19.7 (Time-hierarchy theorem). *If $f, g : \mathbb{N} \rightarrow \mathbb{N}$ are time-constructible functions for which $f(n) = o(g(n)/\log(g(n)))$, then*

$$\mathrm{DTIME}(f) \subsetneq \mathrm{DTIME}(g). \tag{19.11}$$

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 the proof. In particular, notice that it is another example of a proof that uses the diagonalization technique; while it is a bit more technical, it has a very similar flavor to the proof that DIAG is not semidecidable, 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 19.8. *For all $k \geq 1$, it is the case that* $\text{DTIME}(n^k) \subsetneq \text{DTIME}(n^{k+1})$.

19.3 Polynomial and exponential time

We will 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:

$$\text{P} = \bigcup_{k\geq 1} \text{DTIME}(n^k) \quad \text{and} \quad \text{EXP} = \bigcup_{k\geq 1} \text{DTIME}\big(2^{n^k}\big). \tag{19.12}$$

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 fixed 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 fixed 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.

By the time-hierarchy theorem, we can conclude 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:

$$\text{P} \subseteq \text{DTIME}(2^n) \subsetneq \text{DTIME}(2^{2n}) \subseteq \text{EXP}. \tag{19.13}$$

There are many examples of languages contained in the class P. If we restrict our attention to languages we have discussed thus far in the course, we may say the following.

- The languages A_{DFA}, E_{DFA}, EQ_{DFA}, and E_{CFG} from Lecture 15 are all certainly contained in P; if you analyzed the running times of the DTMs we described for those languages, you would find that they run in polynomial time.

- The languages A_{NFA}, A_{REX}, E_{NFA}, and E_{REX} are also in P, but the DTMs we described for these languages in Lecture 15 do not actually show this. Those DTMs have exponential running time, because the conversion of an NFA or regular expression to a DFA could result in an exponentially large DFA. It is, however, not too hard to decide these languages in polynomial time through different methods.

  In particular, we can decide A_{NFA} in polynomial time through a more direct simulation in which we keep track of the set of all states that a given NFA could be in when reading a given input string, and we can decide A_{REX} by performing a polynomial-time conversion of a given regular expression into an equivalent NFA, effectively reducing the problem in polynomial time to A_{NFA}. The language E_{NFA} can be decided in polynomial time by treating it as a graph reachability problem, and E_{REX} can be reduced to E_{NFA} in polynomial time.

- The language A_{CFG} is also contained in P, but once again, the DTM for this language that we discussed in Lecture 15 does not establish this. A more sophisticated approach based on the algorithmic technique of *dynamic programming* does, however, allow one to decide A_{CFG} in polynomial time. This fact allows one to conclude that every context-free language is contained in P.

There does not currently exist a proof that the languages EQ_{NFA} and EQ_{REX} fall outside of the class P, but this is conjectured to be the case. This is because these languages are complete for the class PSPACE of languages that are decidable within a polynomial amount of space. If EQ_{NFA} and EQ_{REX} are in P, then it would then follow that $\text{P} = \text{PSPACE}$, which seems highly unlikely. It is the case, however, that EQ_{NFA} and EQ_{REX} are contained in EXP, for in exponential time one can afford to perform a conversion of NFAs or regular expressions to DFAs and then test the equivalence of the two (possibly exponential size) DFAs in the same way that we considered earlier.

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. For example, 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, are critically important in theoretical computer science, and algorithms courses typically discuss many practically important examples of such functions.[1]

[1] Algorithms courses usually consider computational models that represent machines having random access memory, as opposed to Turing machines. However, because a Turing machine can simulate such a model with no more than a polynomial slowdown, the class of polynomial-time computable functions is the same for the two types of models.

Lecture 20

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 notion of completeness for the class NP.

Remark 20.1. The concept of NP-completeness is certainly among the most important contributions theoretical computer science has made to science in general; NP-complete problems, so recognized, are ubiquitous throughout the mathematical sciences. It is therefore fitting that this concept should be mentioned in a course titled *Introduction to the Theory of Computing*.

At the University of Waterloo, however, the complexity class NP and the notion of NP-completeness are covered in a different course, *CS 341 Algorithms*. For this reason, in the present course we do not place the focus on these notions that they deserve—here we are essentially just treating the class NP as an important example of a complexity class. In particular, we will not cover techniques for proving specific languages are NP-complete, but those new to this topic may rest assured that thousands of interesting examples, including ones of great practical importance, are known.

20.1 The complexity class NP

There are two equivalent ways to view the complexity class NP. The first way is where NP gets its name, as the class of languages decidable in *nondeterministic polynomial time*. The second way, which is more intuitive and more easily applied, is as the class of languages that are *verifiable in polynomial time*, given a polynomial-length *certificate* for the membership of any string in the language in question.

NP as polynomial-time nondeterministic computations

As suggested above, we may define the class NP as its name suggests, as *nondeterministic polynomial time*.

To begin, let us first be precise about what is meant by the running time of a nondeterministic Turing machine.

Definition 20.2. Let N be an NTM having input alphabet Σ. For each input string $w \in \Sigma^*$, let $T(w)$ be the *maximum* number of steps (possibly infinite) for which N runs on input w, over all possible nondeterministic computation paths. The *running time* of N is the function $t : \mathbb{N} \rightarrow \mathbb{N} \cup \{\infty\}$ defined as

$$t(n) = \max\{T(w) \,:\, w \in \Sigma^*,\ |w| = n\} \tag{20.1}$$

for every $n \in \mathbb{N}$. In words, $t(n)$ is the maximum number of steps required for N to halt, over all input strings of length n and over all nondeterministic choices of N.

As in the previous lecture, we shall restrict our attention to NTMs having finite running times for all input lengths.

Let us recall explicitly from Lecture 18 how decidability is is characterized by nondeterministic Turing machines: an NTM N decides a language A if $A = \mathrm{L}(N)$ and N has a finite computation tree for every input string. Thus, if N is an NTM having running time t and w is an input string to N, the computation tree of N on input w always has depth bounded by $t(|w|)$. It is an *accepting* computation if there exists at least one accepting configuration in the computation tree, and otherwise, if there are no accepting configurations at all in the tree, it is a *rejecting* computation.

We may now define NTIME(f), for every function $f : \mathbb{N} \rightarrow \mathbb{N}$, to be the complexity class of all languages that are decided by a nondeterministic Turing machine running in time $O(f(n))$. Having defined NTIME(f) in this way, we define NP as we did for P:

$$\mathrm{NP} = \bigcup_{k \geq 1} \mathrm{NTIME}(n^k). \tag{20.2}$$

NP as certificate verification

The second way to view NP is as the class of languages that are verifiable in polynomial time, given polynomial-length certificates for membership. We shall state a theorem that equates this notion to membership in the complexity class NP, as defined by (20.2).

Before we state the theorem, however, let us introduce some simplifying terminology. Hereafter, a function of the form $p : \mathbb{N} \to \mathbb{N}$ is a *polynomially bounded function* if it is time constructible and satisfies $p(n) = O(n^k)$ for some positive integer k.

Theorem 20.3. *Let Σ be an alphabet and let $A \subseteq \Sigma^*$ be a language. The language A is contained in* NP *if and only if there exists a polynomially bounded function p and a language $B \in$* P *such that*

$$A = \Big\{ x \in \Sigma^* : \exists y \in \{0,1\}^* \text{ such that } |y| \leq p(|x|) \text{ and } \langle x, y \rangle \in B \Big\}. \tag{20.3}$$

An interpretation of this theorem is that the string y plays the role of a *certificate* or *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 *witness* is another alternative name to certificate.

The basic idea behind the proof of two implications needed to imply the theorem, which we shall not cover in detail, are as follows.

1. Given a language A characterized by the equation (20.3) in the theorem statement, one may conclude that $A \in$ NP by defining a polynomial-time NTM N so that it first nondeterministically guesses a binary string y having length at most $p(|x|)$ and then decides membership of the string $\langle x, y \rangle$ in B.
2. If $A \in$ NP, so that we have a polynomial-time NTM N that decides A, we may encode each nondeterministic computation path of N on any input x by some binary string y having length at most $p(|x|)$, for some polynomially bounded function p. We then define B to be the language of strings $\langle x, y \rangle$ for which y encodes an accepting computation path of N on input x, which can be decided in polynomial time.

Relationships among P, NP, and EXP

Let us now observe the following inclusions:

$$\text{P} \subseteq \text{NP} \subseteq \text{EXP}. \tag{20.4}$$

The first of these inclusions, P $\subseteq$ NP, is straightforward. Every DTM is equivalent to an NTM that never has multiple nondeterministic options, so

$$\text{DTIME}(f) \subseteq \text{NTIME}(f) \tag{20.5}$$

```
The DTM $M$ operates as follows on input $x \in \Sigma^*$:
  1. Set $y \leftarrow \varepsilon$.
  2. If $\langle x, y \rangle \in B$, then accept.
  3. Increment $y$ with respect to the lexicographic ordering of the binary
     alphabet.
  4. If $|y| > p(|x|)$ then reject, else goto step 2.
```

Figure 20.1: A DTM M that decides a given NP-language in exponential time.

for all functions $f : \mathbb{N} \rightarrow \mathbb{N}$. This implies that

$$\text{DTIME}(n^k) \subseteq \text{NTIME}(n^k) \tag{20.6}$$

for all $k \geq 1$, and therefore $\text{P} \subseteq \text{NP}$.

Alternatively, with respect to the characterization of NP given by Theorem 20.3, suppose $A \subseteq \Sigma^*$ is a language over an alphabet Σ, and assume $A \in \text{P}$. We may then define a language B as follows:

$$B = \big\{ \langle x, \varepsilon \rangle \,:\, x \in A \big\}. \tag{20.7}$$

It is evident that $B \in \text{P}$; as A in polynomial time, we can easily decide B in polynomial time as well. For any choice whatsoever of a polynomially bounded function p it is the case that (20.3) holds, and therefore $A \in \text{NP}$.

Now let us observe that $\text{NP} \subseteq \text{EXP}$. Suppose $A \subseteq \Sigma^*$ is language over an alphabet Σ, and assume $A \in \text{NP}$. By Theorem 20.3, there exists a polynomially bounded function p and a language $B \in \text{P}$ such that (20.3) holds. Define a DTM M as described in Figure 20.1. It is evident that M decides A, as it simply searches over the set of all binary strings y with $|y| \leq p(|x|)$ 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 $x \in \Sigma^*$ and a binary string y satisfying $|y| \leq p(|x|)$. This test takes a number of steps that is polynomial in $|x|$, and the reason why is as follows. First, we have $|y| \leq p(|x|)$, and therefore the length of the string $\langle x, y \rangle$ is polynomially bounded (in the length of x). Now, because $B \in \text{P}$, 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 polynomially bounded functions is another polynomially bounded function, we find that the time required to test whether $\langle x, y\rangle \in B$ is polynomial in the length of x. Step 3 can also be performed in time polynomial in the length of x, as can the test $|y| > p(|x|)$ in step 4.

Finally, again using the assumption that f is polynomially bounded, so that $f(n) = O(n^k)$ for some positive integer k, we find that the total number of times the steps just considered are executed is at most

$$2^{p(n)+1} - 1 = O\Big(2^{n^{k+1}}\Big). \tag{20.8}$$

Using the rather coarse upper-bound that every polynomially bounded function g satisfies $g(n) = O(2^n)$, we find that the entire computation of M runs in time

$$O\Big(2^{n^{k+2}}\Big). \tag{20.9}$$

We have established that M runs in exponential time, so $A \in \text{EXP}$.

Now we know that

$$\text{P} \subseteq \text{NP} \subseteq \text{EXP}, \tag{20.10}$$

and we also know that

$$\text{P} \subsetneq \text{EXP} \tag{20.11}$$

by the time-hierarchy theorem. Of course this means that one (or both) of the following proper containments must hold: (i) $\text{P} \subsetneq \text{NP}$, or (ii) $\text{NP} \subsetneq \text{EXP}$. Neither one has yet been proved, and a correct proof of either one would be a major breakthrough in complexity theory. Indeed, determining whether or not $\text{P} = \text{NP}$ is viewed by many as being among the greatest unsolved mathematical challenges of our time.

20.2 Polynomial-time reductions and NP-completeness

We discussed reductions in Lecture 17 and used them to prove that certain languages are undecidable or non-semidecidable. *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 20.4. Let Σ and Γ be alphabets and let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be languages. It is said that A *polynomial-time reduces* to B if there exists a polynomial-time computable function $f : \Sigma^* \to \Gamma^*$ such that

$$w \in A \Leftrightarrow f(w) \in B \tag{20.12}$$

for all $w \in \Gamma^*$. One writes

$$A \leq_m^p B \tag{20.13}$$

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 20.5. Let Σ be an alphabet and let $B \subseteq \Sigma^*$ be a language.

1. It is said that B is NP-hard if $A \leq_m^p B$ for every language $A \in \text{NP}$.
2. It is said that B is NP-complete if B is NP-hard and $B \in \text{NP}$.

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 us a key to unlocking the computational difficulty of the class NP in its entirety. Figure 20.2 illustrates the relationship among the classes P and NP, and the NP-hard and NP-complete languages, under the assumption that $\text{P} \neq \text{NP}$.

Here are some basic facts concerning polynomial-time reductions and NP. For all of these facts, it is to be assumed that A, B, and C are languages over arbitrary alphabets.

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 $\text{P} = \text{NP}$.

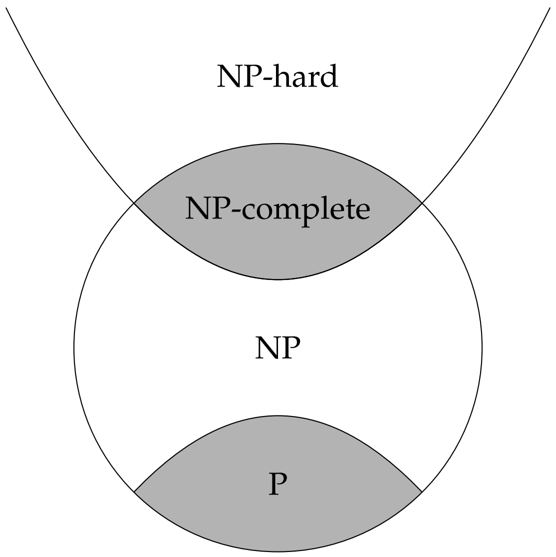


Figure 20.2: The relationship among the classes P and NP, and the NP-hard and NP-complete languages. The figure assumes P $\neq$ NP.

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 $A \leq_m^p B$.

The proofs of the statements listed above are all fairly straightforward, and you might try proving them for yourself if you are interested. Let us pick just one of the statements and prove it.

Proposition 20.6. *Let $A \subseteq \Sigma^*$ and $B \subseteq \Gamma^*$ be languages, for alphabets Σ and Γ, and assume $A \leq_m^p B$ and $B \in \text{NP}$. It is the case that $A \in \text{NP}$.*

Proof. Let us begin by gathering some details concerning the assumptions of the proposition.

First, because $A \leq_m^p B$, we know that there exists a polynomial-time computable function $f : \Sigma^* \rightarrow \Gamma^*$ such that

$$x \in A \;\Leftrightarrow\; f(x) \in B \tag{20.14}$$

for all $x \in \Sigma^*$. Because f is polynomial-time computable, there must exist a polynomially bounded function p such that $|f(x)| \leq p(|x|)$ for all $x \in \Sigma^*$.

Second, by the assumption that $B \in \text{NP}$, there exists a polynomially bounded function q and a language $C \in \text{P}$ for which

$$B = \Big\{ x \in \Gamma^* \;:\; \exists y \in \{0,1\}^* \text{ such that } |y| \leq q(|x|) \text{ and } \langle x, y \rangle \in C \Big\}. \tag{20.15}$$

The DTM M operates as follows on input $w \in \Sigma^*$:

1. If w does not take the form $w = xx$ for some string $x \in \Sigma^*$, then reject.
2. Accept if $x \in A$, otherwise reject.

Figure 20.3: The DTM M from the proof of Corollary 20.7.

Now, define a new language

$$D = \{\langle x, y\rangle : \langle f(x), y\rangle \in C\}. \tag{20.16}$$

It is evident that $D \in \mathrm{P}$ because one may simply compute $\langle f(x), y\rangle$ from $\langle x, y\rangle$ in polynomial time (given that f is polynomial-time computable), and then test if $\langle f(x), y\rangle \in C$, which requires polynomial time because $C \in \mathrm{P}$.

Finally, observe that

$$A = \Big\{x \in \Sigma^* : \exists y \in \{0,1\}^* \text{ such that } |y| \leq q(p(|x|)) \text{ and } \langle x, y\rangle \in D\Big\}. \tag{20.17}$$

As the composition $q \circ p$ is a polynomially bounded function and $D \in \mathrm{P}$, it follows that $A \in \mathrm{NP}$. □

Let us conclude the lecture with the following corollary, which is meant to be a fun application of the previous proposition along with the time-hierarchy theorem.

Corollary 20.7. $\mathrm{NP} \neq \mathrm{DTIME}(2^n)$.

Proof. Assume toward contradiction that $\mathrm{NP} = \mathrm{DTIME}(2^n)$. Let Σ be any alphabet, let $A \subseteq \Sigma^*$ be an arbitrarily chosen language in $\mathrm{DTIME}(4^n)$, and define

$$B = \{xx : x \in A\}.$$

First we observe that $B \in \mathrm{DTIME}(2^n)$. In particular, the DTM M described in Figure 20.3 decides B in time $O(2^n)$. The reason why M runs in time $O(2^n)$ is as follows: the first step can easily be performed in polynomial time, and the second step requires $O(4^{n/2}) = O(2^n)$ steps, as A can be decided in time $O(4^m)$ on inputs of length m, and here we are deciding membership in A on a string of length $m = n/2$. The running time of M is therefore $O(2^n)$. As we have assumed that $\mathrm{NP} = \mathrm{DTIME}(2^n)$, it follows that $B \in \mathrm{NP}$.

Now define a function $f : \Sigma^* \to \Sigma^*$ as

$$f(x) = xx$$

for all $x \in \Sigma^*$. The function f can easily be computed in polynomial time, and it is immediate from the definition of B that

$$x \in A \Leftrightarrow f(x) \in B.$$

We therefore have that $A \leq_m^p B$. By Proposition 20.6, it follows that $A \in \text{NP}$, and given the assumption $\text{NP} = \text{DTIME}(2^n)$, it follows that $A \in \text{DTIME}(2^n)$.

However, as A was an arbitrarily chosen language in $\text{DTIME}(4^n)$, we conclude that $\text{DTIME}(4^n) \subseteq \text{DTIME}(2^n)$. This contradicts the time hierarchy theorem, so our assumption $\text{NP} = \text{DTIME}(2^n)$ was incorrect. □