CSCE 355

Foundations of Computation

Stephen Fenner
Daniel Padé
Duncan A. Buell

University of South Carolina

These notes are based on two lectures per week. Sections beginning with a star (*) are optional.

Last modified January 20, 2021

Contents

Lecture 1

This lecture will outline the topics and requirements of the course. We will also jump into some review of discrete math.

Example of a two-state automaton modeling a light switch.

Some basic definitions so that we're all on the same page.

Definition 1.0.1. A *natural number* is any whole number that is at least zero. We can list the natural numbers as $0, 1, 2, 3, \ldots$. We let $\mathbb{N}$ denote the set of all natural numbers.

Some mathematicians, especially those working in algebra or number theory, define the natural numbers to start at 1 and exclude 0. Logicians and computer scientists usually define them as we did above, and we'll stick to that.

A more formal way of defining the natural numbers is as the least collection of numbers satisfying

- 0 is a natural number, and
- if x is any natural number, then $x + 1$ is a natural number.

This definition is the basis of a method of proof called *mathematical induction*, which we'll describe later.

Definition 1.0.2. A number n is an *integer* if either x or $-x$ is a natural number. The integers form a doubly infinite list: $\ldots, -2, -1, 0, 1, 2, \ldots$. We let $\mathbb{Z}$ denote the set of all integers.

So the integers are all the whole numbers—positive, negative, or zero. Speaking of which,

Definition 1.0.3. Let x be any real number. We say that x is *positive* just in the case that $x > 0$. We say that x is *negative* just in the case that $x < 0$ (equivalently, $-x$ is positive). Additionally, we say that x is *nonnegative* to mean that $x \geq 0$, i.e., that x is either zero or positive.

So that means that for any real number x, exactly *one* of the following three statements is true:

- x is positive
- $x = 0$
- x is negative

Definition 1.0.4. A real number x is *rational* just in case that $x = a/b$ for some integers a and b with $b \neq 0$. By negating both the numerator and denominator if necessary, we can always assume that $b > 0$. If x is not rational, then we say that x is *irrational*. We let $\mathbb{Q}$ denote the set of all rational numbers.

1.1 Logical Connectives

Conditionals

Many theorems are of the form, "If H then C," where H and C are statements. This is called a *conditional statement*: H is the *hypothesis* and C is the *conclusion*. This conditional statement can be written symbolically as $H \to C$. H and C may have variables, in which case the statement must be proven true for all appropriate values of the variables. If there is any doubt, we may quantify exactly what values those are.

Other equivalent ways of saying "if H then C" are:

- "H implies C"
- "C follows from H"
- "C if H"
- "H only if C"
- "H is a sufficient condition for C"
- "C is a necessary condition for H"
- "it cannot be the case that H is true and C is false"

Example:

For all integers x, if x^2 is even, then x is even.

Here, the hypothesis is "x^2 is even" and the conclusion is "x is even." We quantified x over the integers, that is, we said that the statement holds for all integers x. So the statement says nothing about x if x is not an integer (π, say). (By the way, this statement is true, and we'll prove it later.)

The hypothesis or the conclusion may be more complicated. Here is a statement where the hypothesis is two simple statements joined by "and":

> For all integers x, if $x > 2$ and x is prime, then x is odd.

This statement is also true.

Biconditionals

A statement of the form "H if and only if C" is called a *biconditional*. It asserts that both H implies C and that C implies H, i.e., C and H both follow from each other. In other words, C and H are equivalent (have the same truth value). The phrase "if and only if" is often abbreviated by "iff." A proof of a biconditional usually requires two subproofs: one that H implies C (the *forward* direction, or "only if" part), and one that C implies H (the *reverse* direction, or "if" part).

The *converse* of a conditional statement "if H then C" is the conditional statement "if C then H." Thus a biconditional asserts both the conditional (forward direction) and its converse (reverse direction).

Here are some other ways of saying "H if and only if C":

- "H iff C"
- "C iff H"
- "H implies C and conversely"
- "H and C are equivalent"
- "if H then C and if C then H"
- "H is a necessary and sufficient condition for C"
- "C is a necessary and sufficient condition for H"
- "H and C are either both true or both false"

Symbolically, we write "$H \leftrightarrow C$," and this asserts that $H \to C$ and $C \to H$.

1.2 Methods of proof

We look at several techniques to prove statements:

- direct proof
- proof by cases
- proof by contradiction
- proof by induction (and variants)

Many complex proofs combine some or all of these ingredients together.

Direct proofs

Theorem 1.2.1. *For any integer x, if $x \geq 4$, then $2^x \geq x^2$.*

Proof. Notice that $2^4 = 16 = 4^2$, so the statement is true for $x = 4$.[1] Now consider the sequence

$$2^4, 2^5, 2^6, \ldots$$

of values on the left-hand side and the sequence

$$4^2, 5^2, 6^2, \ldots$$

of values on the right-hand side. Taking the ratio of adjacent terms in each sequence, we see that

$$\frac{2^{x+1}}{2^x} = \frac{2^1}{2^0} = 2,$$

and

$$\frac{(x+1)^2}{x^2} = \left(\frac{x+1}{x}\right)^2.$$

If $x \geq 4$, then $(x+1)/x \leq 5/4 = 1.25$, and so

$$\left(\frac{x+1}{x}\right)^2 \leq \left(\frac{5}{4}\right)^2 = \frac{25}{16} < 2.$$

So the left-hand sequence values increase by a factor of 2 each time, but the right-hand values increase by a factor of less than 2 each time. This will make all the left-hand values at least as big as the corresponding right-hand values. □

[1] We could check that the statement is also true for $x = 5, 6, 7, \ldots 19$, but this is not sufficient to prove the statement, because we are only proving it true for some finite sample of values whereas the theorem asserts the result for *all* values at least 4. It still may be useful to check a few cases, however, to give a hint about the general argument.

This is a direct proof. We start by assuming the hypothesis, infer some new statements based on the hypothesis and using easy and familiar facts about numbers (what I'll call "high school math"), and eventually reach the conclusion. The proof above is not completely formal, because we don't bother proving these facts from high school math (e.g., the fact that $(a/b)^2 = a^2/b^2$ for all real a and b), but that's fine; these facts are so easy and intuitively obvious that proving them would be a tedious waste of time and obscure the key points of the whole proof itself.

1.3 Notation

Now is good a time as any to mention the notation that is common in mathematics and theoretical computer science. We define symbols as follows:

- $\mathbb{N}$: the natural numbers, as mentioned above
- $\mathbb{Z}$: the integers (in German, the *Zahlennummer*, or "counting numbers"), being all numbers
$$\ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots$$
- $\mathbb{Q}$: the rational numbers, which are the quotients a/b of two integers a and b
- $\mathbb{R}$: the real numbers
- $\mathbb{C}$: the complex numbers

In this course we will use $\mathbb{N}$ and $\mathbb{Z}$ a lot, $\mathbb{Q}$ a little, and $\mathbb{R}$ and $\mathbb{C}$ almost not at all.

These symbols are in what is called "blackboard bold" font. Decades ago, in books and journals, the letters would be printed in boldface. But it's not really possible on the blackboard to write a different symbol for a boldface letter to distinuish it from an ordinary letter, so the extra stroke would be used on the board to indicate that the symbol was different.

Mathematics is what used to be referred to as "penalty copy" due to the large number of special symbols, superscripts, subscripts, and the like. When Knuth developed the TeX typesetting package, he deliberately included a font for blackboard bold. With the advent of TeX and its derivatives, mathematics stopped being penalty copy for publishers (in part because the typesetting task was transferred to the mathematicians doing the writing).

Nonetheless, in the 14th edition (1993) of the University of Chicago's *Manual of Style*, the manual specifically stated "Blackboard bold should be confined to the classroom" and that authors should indicate whether that which would be written on the board in blackboard bold should be typeset as boldface, sans serif, or some other font.

It was the third author of these notes who later received a promise from the editors of the *Manual of Style* to delete that outdated point of view from subsequent

editions. This came after submitting to those editors a journal paper, published by the American Mathematical Society (which should know these things), that used in the same paper, for three different things, the symbols Q, $\boldsymbol{Q}$, and $\mathbb{Q}$. The third author argued that if the world's primary math society would publish papers simultaneously using all three symbols, surely the curmudgeons at the U of Chicago were just not up with the times.

Lecture 2

2.1 Methods of Proof (cont.)

Proof by cases

Theorem 2.1.1. *There exist irrational numbers $a, b > 0$ such that a^b is rational.*

Proof. Consider $\sqrt{2}$, which is known to be irrational (we'll actually prove this later).

Case 1: $\sqrt{2}^{\sqrt{2}}$ is rational. Then we set $a = b = \sqrt{2}$ and we are done.

Case 2: $\sqrt{2}^{\sqrt{2}}$ is irrational. Set $a = \sqrt{2}^{\sqrt{2}}$ and $b = \sqrt{2}$. Then

$$a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = (\sqrt{2})^{\sqrt{2}\sqrt{2}} = \sqrt{2}^2 = 2,$$

which is rational. So we are done.

In either case we have found irrational numbers a, b such that a^b is rational. Since one of the two cases must hold, the Theorem must be true. □

Notice that the proof does not depend on which case holds, because we can prove the theorem in either case. (It is actually known that Case 2 holds.) This is how proof by cases works. You can split the hypothesis into two (or more cases) and prove the conclusion in each case. This particular proof is *nonconstructive* in that it doesn't actually give us two numbers a and b, but merely shows us that such numbers exist. It gives us two possibilities for the pair of values and asserts that at least one of them is correct, but does not tell us which one. Constructive proofs are usually preferable, but there are some theorems in math that have no known constructive proof.

In any proof by cases, the cases must be *exhaustive*, that is, it must always be that at least one of the cases holds. We will see more proofs by cases below.

2.2 Summations

Sigma notation is shorthand for the sum over a sequence indexed by integers:

$$\sum_{i=m}^{n} = a_m + a_{m+1} + \cdots + a_{n-1} + a_n$$

The index i starts at i, and is incremented by 1 each term until it reaches n (inclusive). Generalizations of the notation exist where the indices are taken from some finite set:

- $\sum_{m \leq i \leq n} f(i)$
- $\sum_{i \in S} f(i)$
- $\sum_{i,j=0}^{n}$

Formally, we define summation (inductively!) as follows:

Definition 2.2.1.

$$\sum_{i=m}^{n} f(i) = \begin{cases} 0 & (n < m) \\ f(n) + \sum_{i=m}^{n-1} f(i) & (n \geq m) \end{cases}$$

by definition 2.2.1, we can prove the following are valid

- $\sum_{i=m}^{n} C \cdot f(i) = C \cdot \sum_{i=m}^{n} f(i)$ for C a constant
- $\sum_{i=m}^{n} f(i) + \sum_{i=m}^{n} g(i) = \sum_{i=m}^{n} [f(i) + g(i)]$
- $\sum_{i=m}^{n} f(i) = \sum_{i=m+j}^{n+j} f(i-j)$
- $\sum_{i=m}^{j} f(i) + \sum_{i=j+1}^{n} f(i) = \sum_{i=m}^{n} f(i)$

2.3 Proof by induction

This method of proof is extremely useful and has many variants. It is used to prove statements about the natural numbers. In its basic form, induction is used to prove that some statement $P(n)$ is true for every natural number n. The argument is in two stages:

Base case: Prove that $P(0)$ is true. (This is often trivial to do.)

Inductive step: Prove that for any natural number $n \geq 0$, if $P(n)$ is true then $P(n+1)$ is true.

The base case provides the starting point for the induction, and the inductive step provides a template for getting S to hold for the next natural number given that you've established it for the current one. So if we unwind the argument, we establish that

- $P(0)$ is true (this is the base case)
- $P(1)$ is true (this applies the inductive step with $n = 0$ and the fact that we've already established $P(0)$)
- $P(2)$ is true (by the inductive step again, this time with $n = 1$, as well as the previous proof of $P(1)$)
- etc.

The point is that once we've established $P(n)$ for some value of n, then we can conclude $P(n + 1)$ by the inductive step. So if we prove both the base case and the inductive step for general n, we must conclude that $P(n)$ holds for all natural numbers n.

A common variant is to start the induction with some natural number other than 0 for the base case, for example, 1. So here the base case is to prove $P(1)$ and the induction step is to prove $P(n) \to P(n + 1)$ for any $n \geq 1$. From this we conclude that S holds for all *positive* integers (not necessarily for 0). Similarly, you can use any other integer as the base case—for an arbitary example, you can prove $P(17)$ as the base case then prove $P(k) \to P(k + 1)$ for all integers $k \geq 17$. Conclude that $P(n)$ holds for all integers $n \geq 17$. You could also start the induction with a negative integer if you want.

For our first example of induction, we reprove Theorem 1.2.1. Proofs by induction tend to be more formally correct and less "hand-wavy" than alternatives.

Proof of Theorem 1.2.1 by induction. We let $P(n)$ be the statement that $2^n \geq n^2$, then we wish to prove $P(n)$ for all integers $n \geq 4$. Thus we start the induction at 4 as our base case.

Base case: Clearly, $2^4 = 16 = 4^2$, so $P(4)$ is true.

Inductive case: Here we must show for all integers $n \geq 4$ that $P(n)$ implies $P(n+1)$. Fix an arbitrary integer $n \geq 4$, and assume that $P(n)$ holds, i.e., that $2^n \geq n^2$. (This assumption is called the *inductive hypothesis*.) We want to infer that

$P(n+1)$ holds, i.e., that $2^{n+1} \geq (n+1)^2$. We can do this by a direct chain of inequalities:

$$\begin{aligned}
2^{n+1} &= 2(2^n) && \text{(sum of exponents rule)} \\
&\geq 2n^2 && \text{(inductive hypothesis)} \\
&= n^2 + n^2 \\
&\geq n^2 + 4n && \text{(since } n \geq 4\text{, we have } n^2 \geq 4n \text{ by multiplying both sides by } n\text{)} \\
&= n^2 + 2n + 2n \\
&\geq n^2 + 2n + 1 && \text{(because } 2n \geq 2(4) = 8 \geq 1\text{)} \\
&= (n+1)^2.
\end{aligned}$$

□

In the proof above we set things up to make use of the inductive hypothesis. If an inductive proof does not make use of the inductive hypothesis somewhere, it is surely suspect.

Here is a more useful example. First, a familiar definition.

Definition 2.3.1. Let x be any integer. We say that x is *even* iff $x = 2k$ for some integer k. We say that x is *odd* to mean that x is not even.

Is 0 even? Yes, because $0 = 2 \cdot 0$ and 0 is an integer. Is 18 even? Yes, because $18 = 2 \cdot 9$ and 9 is an integer. Is -4 even? Yes, because $-4 = 2(-2)$ and -2 is an integer. Is 3 even? No, 3 is odd.

Now for the theorem we prove by induction. The proof will also use cases.

Theorem 2.3.2. *For every integer $n \geq 1$, either n is even or $n-1$ is even.*

Proof. Let $P(n)$ be the statement, "either n is even or $n-1$ is even." We prove by induction that $P(n)$ holds for all integers $n \geq 1$ (so we'll start the induction at 1 instead of 0.

Base case: To see that $P(1)$ holds, we just note that $0 = 1 - 1$ and 0 is even.

Inductive step: Fix any integer $n \geq 1$. We prove directly that if $P(n)$ holds then $P(n+1)$ holds. Assume that $P(n)$ holds, i.e., that either n is even or $n-1$ is even (this is the inductive hypothesis), and consider the statement $P(n+1)$: "either $n+1$ is even or $(n+1)-1$ is even."

Case 1: n is even. Then since $(n+1)-1 = n$, we have that $(n+1)-1$ is even in this case, which implies that $P(n+1)$ holds, and so we are done.

Case 2: n is odd, i.e., n is not even. Since the inductive hypothesis $P(n)$ (which we assume is true) says that either n is even or $n-1$ is even, we must have then that $n-1$ is even. By the definition of evenness, this means that $n-1 = 2k$ for some integer k. But then, by "high school math",

$$n+1 = (n-1)+2 = 2k+2 = 2(k+1).$$

Since $k+1$ is an integer, this shows that $n+1$ is even. Thus $P(n+1)$ holds in this case as well.

We've established $P(n+1)$ assuming $P(n)$ in either case. Since the cases are exhaustive, we have $P(n) \to P(n+1)$ for all $n \geq 1$.

We can now conclude by induction that $P(n)$ holds for all integers $n \geq 1$. □

A *corollary* of a theorem is a new theorem that follows easily from the old one. The theorem we just proved has a corollary that strengthens it:

Corollary 2.3.3. *If n is any integer, then either n is even or $n-1$ is even.*

Note that in the corollary, we've dropped the restriction that $n \geq 1$.

Proof. Let n be any integer. We know that either $n > 0$ or $n \leq 0$, and we prove the statement in each case.

Case 1 If $n > 0$, then $n \geq 1$ (because n is an integer), so Theorem 2.3.2 applies directly to this case.

Case 2 If $n \leq 0$, then negating both sides gives $-n \geq 0$, and adding 1 to both sides gives $1-n \geq 1$. Since $1-n$ is an integer at least 1 we can apply Theorem 2.3.2 to $1-n$ to get that either $1-n$ is even or $(1-n)-1 = -n$ is even.

We then look at *these* two cases separately: If $1-n$ is even, then $1-n = 2k$ for some integer k. Then negating both sides gives $n-1 = -(1-n) = -2k = 2(-k)$, and so $n-1$ is even because $-k$ is an integer. Likewise, if $-n$ is even, then we can write $-n = 2\ell$ for some integer ℓ. Negating both sides, we get $n = -2\ell = 2(-\ell)$. So since $-\ell$ is an integer, n is even.

So in both cases, either n is even or $n-1$ is even. □

Induction on the length of a string

We will do many proofs in this course by induction on the length of a string of symbols. The following is an example of how this might work.

Theorem 2.3.4. *Let $s = 1\ldots1$ be a string of n 1-bits. Then as a binary number the string s represents $2^n - 1$.*

Proof. **Base case:** Let $s = 1$, a string of length $n = 1$. As a binary number, this is the number $1 = 2^1 - 1$.

Inductive case: Let $s = 1...1$ be a string of n 1-bits. By the inductive hypothesis, s represents the integer $2^n - 1$. Our goal is to prove that the string $s1$ of $n + 1$ 1-bits represents the integer $2^{n+1} - 1$. Consider first the string $s0$, of n 1-bits followed by a single 0. Appending the 0 to the right end of the string is equivalent to multiplying the represented integer by 2, so $s0$ represents the integer

$$2 \cdot (2^n - 1) = 2^{n+1} - 2.$$

Changing the 0 to a 1 is now equivalent to adding 1 to the represented integer, so the string $s1$ represents

$$2 \cdot (2^n - 1) + 1 = 2^{n+1} - 2 + 1 = 2^{n+1} - 1,$$

which is exactly what we were trying to prove.

□

2.4 Proof by contradiction

The next theorem's proof uses a new proof techique: *proof by contradiction*. To prove a statement S by contradiction, you start out assuming the negation of S (i.e., that S is false) then from that assumption you prove a known falsehood (a "contradiction"), such as $0 = 1$ or some such. You can then conclude that S must be true, because its being false implies something absurd and impossible.

To prove a conditional statement "if H then C" by contradiction, you start by assuming that the conditional is not true, i.e., that H is true but C is false, then from that you prove a contradiction, perhaps that H is false (and so H is both true and false, which is a contradiction). Proof by contradiction may be useful if you don't see any direct way of proving a statement.

Theorem 2.4.1. *An integer n is odd iff $n = 2k + 1$ for some integer k.*

Proof. The statement is a biconditional, and we prove each direction separately.

Forward direction: (For this direction, we assume that n is odd and prove that $n = 2k + 1$ for some integer k.) Assume n is odd. Then n is not even, and so by Corollary 2.3.3, we must have that $n - 1$ is even. So $n - 1 = 2k$ for some integer k (definition of being even). So we have $n = (n - 1) + 1 = 2k + 1$.

Reverse direction: (For this direction, we assume that $n = 2k + 1$ for some integer k and prove that n is odd.) Assume that $n = 2k + 1$ for some integer k. Now here is where we use proof by contradiction: We want to show that n is odd,

but we have no direct way of proving this. So we will assume (for the sake of contradiction) that n is *not* odd, i.e., that n is even. (From this we will derive something that is obviously not true.) Assuming n is even, we must have $n = 2\ell$ for some integer ℓ (definition of evenness). Then we have $2k + 1 = n = 2\ell$. Subtracting $2k$ from both sides, we get $1 = 2\ell - 2k = 2(\ell - k)$. Dividing by 2 then gives

$$\ell - k = \frac{1}{2}.$$

But ℓ and k are both integers, and so $\ell - k$ is an integer, but $1/2$ is not an integer, and so they cannot be equal. This is a contradiction,[1] which means that our assumption that n is even must be wrong. Thus n is odd.

□

The next corollary says that odd times odd is odd.

Corollary 2.4.2. *Let a and b be integers. If a and b are both odd, then their product ab is odd.*

Proof. Assuming a and b are both odd, by Theorem 2.4.1 (forward direction) we can write $a = 2k + 1$ and $b = 2\ell + 1$ for some integers k and ℓ. Then

$$\begin{aligned} ab &= (2k + 1)(2\ell + 1) \\ &= 4k\ell + 2k + 2\ell + 1 \\ &= 2(2k\ell + k + \ell) + 1 \\ &= 2m + 1 \end{aligned}$$

where $m = 2k\ell + k + \ell$. Since m is clearly an integer, we use Theorem 2.4.1 again (reverse direction this time) to conclude that ab is odd. □

2.5 And Finally ...

We close this with one of the most famous proofs in mathematics. This comes from the first problem set in Royden's *Real Analysis*, a textbook for the graduate level analysis course that virtually all mathematics doctoral students take.

The purpose of this example is simply to show that sometimes one must do so thinking in orthogonal ways to "the usual way".

Theorem 2.5.1. *If x is a member of the empty set, then x is a green-eyed lion.*

[1] A contradiction is often indicated symbolically by $\Rightarrow\Leftarrow$.

Proof. What we have in more notation is

$$x \in \emptyset \Rightarrow x \text{ is a green-eyed lion}$$

□

Now, how would we prove or disprove this?

Well, obviously if it were possible to find an element x in the empty that is not in fact a green-eyed lion, then this theorem would be false.

And if it is not possible to find an element x in the empty set that isn't a green-eyed lion, clearly the theorem is true.

But ... there is by definition no element x that is in the empty set.

So there can be no such element x that is in the empty set and for which the property "is a green-eyed lion" fails to be true.

Lecture 3

3.1 Strong induction and the well-ordering principle

Strong induction is a kind of mathematical induction. Fix an integer c to start the induction. To prove a that a statement $P(n)$ holds for all integers $n \geq c$, it suffices to prove that $P(n)$ follows from $P(c), P(c+1), P(c+2), \ldots, P(n-1)$. In other words, it is enough to prove $P(n)$ assuming that $P(k)$ holds for all integers k with $c \leq k \leq n-1$. This still requires proving $P(c)$ outright with no assumptions, but then you can establish $P(c+1)$ given $P(c)$, because you've just proved $P(c)$. Then you can establish $P(c+2)$ assuming both $P(c)$ and $P(c+1)$ since you've proved both of the latter, and so on. So strong induction gives a template to iterate the proof to all $n \geq c$.

In regular induction, you can only assume $P(n)$ to prove $P(n+1)$, so it appears that regular induction is more restrictive than strong induction. It turns out that regular induction and strong induction actually have the same proving power, that is, any proof using strong induction can be converted into one using regular induction, and *vice versa*. Sometimes, however, just assuming $P(n)$ is not enough to directly prove $P(n+1)$, so strong induction may work easily in some cases where it is difficult or clunky to apply regular induction. That said, why would you ever use regular induction when you can always use strong induction? Why, indeed; I don't have a good answer. Perhaps regular induction is conceptually simpler when it can be applied.

The *well-ordering principle* of the natural numbers states

> If X is any nonempty set of natural numbers, then X has a least element. That is, there is some z in X such that $z \leq w$ for all w in X.

This should be pretty intuitive, and we can use it freely.

* Equivalence of strong induction and the well-ordering principle

Strong induction (starting at 0) and the well-ordering principle are actually equivalent: it is easy to prove one from the other and vice versa.

Proof of the well-ordering principle using strong induction. Let X be any nonempty set of natural numbers. We use strong induction and proof-by-contradiction to show that X has a least element. For every natural number n, let $P(n)$ be the property that $n \notin X$, i.e., that n is not an element of X. We now prove by strong induction that $P(n)$ is true for every natural number n, hence X must be empty, which contradicts the fact that X is nonempty.

Base case: If $P(0)$ were false, then that would mean that $0 \in X$, and since 0 is the least integer, 0 must be the least element of X, contradicting our assumption that X has no least element. So $P(0)$ must be true. (See how this is a mini-proof by contradiction.)

Inductive step: Fix any natural number n and assume (inductive hypothesis) that $P(m)$ is true for all natural numbers $m \leq n$. This means that $m \notin X$ for all natural numbers $m \leq n$. Then $P(n+1)$ must also be true, for if $P(n+1)$ were false, then $n+1$ would be the least element of X. Again, a contradiction. Thus $P(n+1)$ is true.

To reiterate: by strong induction, we have that $P(n)$ (equivalently, $n \notin X$) is true for all natural numbers n, and hence X must be empty. This finishes the proof of the well-ordering principle. □

Proof of strong induction using the well-ordering principle. Let S be any property of numbers. Suppose that $P(0)$ is true, and for any natural number n, we know that if $P(0), \ldots, P(n)$ are all true then $P(n+1)$ must also be true. Then we use the well-ordering principle and proof-by-contradition to show that $P(n)$ is true for all natural numbers n. Let X be the set of all natural numbers n such that $P(n)$ is false, i.e.,

$$X = \{n \in \mathbb{N} \mid P(n) \text{ is false}\}.$$

It suffices to show that X is empty. Suppose, for the sake of contradiction, that X is nonempty. Then by the well-ordering principle, X must have a least element, say n_0. Since $n_0 \in X$ we have that $P(n_0)$ is false, so in particular, $n_0 \neq 0$. Let $n = n_0 - 1$. Then n is a natural number, and since n_0 is the *least* element of X, we have that $0, \ldots, n \notin X$. Thus $P(0), \ldots, P(n)$ are all *true*, but $P(n+1) \iff P(n_0)$, which is false, violating our assumptions about the property S. Thus X must be empty. □

3.2 Proof that $\sqrt{2}$ is irrational

We'll now use the well-ordering principle together with contradiction to prove that $\sqrt{2}$ is irrational — a fact that has been known since ancient times.

Theorem 3.2.1. *There is no rational number q such that $q^2 = 2$.*

Proof. For the sake of contradiction, let's assume that there does exist $q \in \mathbb{Q}$ such that $q^2 = 2$. We can set $q = a/b$ for integers a, b with $b > 0$, and so b is a natural number, and $(a/b)^2 = 2$. Now let X be the set of all natural numbers $b > 0$ such that $(a/b)^2 = 2$ for some integer a, i.e.,

$$X = \{b \in \mathbb{N} \mid \ b > 0 \text{ and there exists } a \in \mathbb{Z} \text{ such that } (a/b)^2 = 2 \ \}.$$

By our assumption, X is nonempty, and so by the well-ordering principle, X must have some least element $n > 0$ where there exists some integer m such that $(m/n)^2 = 2$. We then have

$$2 = \left(\frac{m}{n}\right)^2 = \frac{m^2}{n^2}.$$

Multiplying both sides by n^2, we get

$$m^2 = 2n^2$$

And thus m^2 is even. This means that m itself must be even (if m were odd, then $m^2 = mm$ would also be odd, by Corollary 2.4.2—that's a mini-proof by contradiction). So we can write $m = 2k$ for some integer k. Then we have

$$2n^2 = m^2 = (2k)^2 = 4k^2.$$

Dividing by 2 gives

$$n^2 = 2k^2$$

whence n^2 is even. Thus n is even by an argument similar to the one for m. So we can write $n = 2\ell$ for some integer $\ell > 0$. Now we have

$$\left(\frac{k}{\ell}\right)^2 = \left(\frac{2k}{2\ell}\right)^2 = \left(\frac{m}{n}\right)^2 = 2$$

This means that ℓ is in the set X, because there is an integer k such that $(k/\ell)^2 = 2$. But $\ell = n/2$, which is less than n, and this contradicts the fact that n is the *least* element of X. Thus our original assumption about the existence of q must be false. □

Lecture 4

4.1 "Proofs" that fail

Theorem 4.1.1. *All horses are the same color*

Proof. The proof proceeds by induction:

Base case: All horses are the same color in a set with one horse.

Inductive step: We assume all horses have the same color in a set with n horses. Take the set $\{h_1, h_2, \ldots, h_{n+1}\}$ and split it into two sets:

$$S_1 = \{h_1, \ldots, h_n\}$$
$$S_2 = \{h_2, \ldots, h_{n+1}\}$$

By the inductive hypothesis, all horses in S_1 have the same color and all horses in S_2 have the same color. Since these two sets overlap, then all horses have the same color.

□

Theorem 4.1.2. *1 = 2*

Proof. Let $a = b$

$$\begin{aligned} a^2 &= ab \\ a^2 + a^2 &= a^2 + ab \\ 2a^2 &= a^2 + ab \\ 2a^2 - 2ab &= a^2 - ab \\ 2(a^2 - ab) &= a^2 - ab \\ 2 &= 1 \end{aligned}$$

□

Lecture 5

Let's review some basic facts about sets. A set is a collection of things (its *members* or *elements*). For any object x and set S, we write $x \in S$ to mean that x is a member of set S (equivalently, x is in S). We write $x \notin S$ to mean that x is not a member of S (x is not in S).

A set can be an essentially arbitrary collection of things, and it is completely determined by its members. No other information is carried by the set. That is, if A and B are sets, then $A = B$ if all members of A are also members of B and vice versa (i.e., they have the same members). This is worth stating formally:

Fact 5.0.1 (Axiom of Extensionality)**.** *If two sets have the same members, then they are equal. That is, for any sets A and B, if $z \in A \iff z \in B$ for all z, then $A = B$.*

Given any object x and set S, there are only two possibilities: either $x \in S$ or $x \notin S$. There is no sense in which "x appears in S some number of times" or "x appears in one place in S and not another", etc.; these notions are not relevant to sets.

5.1 Describing sets

Listing the elements of a set

If the members of a set are easily listable, then we can denote the set by listing its members, separated by commas and enclosed in braces (curly brackets). For example,

$$\{1, 4, 9, 16, 25\} \tag{5.1}$$

denotes the set whose members are the five smallest squares of positive integers. In keeping with the notion of set above, the members can appear in any order, and duplicate occurrences of a member don't matter. In particular, all the following expressions represent the same set (5.1), above:

- $\{1, 4, 9, 16, 25\}$

- $\{4, 25, 16, 1, 9\}$
- $\{9, 1, 9, 9, 16, 1, 4, 25\}$
- etc.

In some cases — only when it is intuitively clear—the listing can omit some elements and use an ellipsis (...) instead. For example, if n is a natural number, then the set of all natural numbers between 0 and n inclusive can be written as

$$\{0, 1, 2, \ldots, n\}$$

or even just

$$\{0, \ldots, n\}$$

if the context is clear enough. Here, we are omitting some number of elements in the listing (although they are in the set), using an ellipsis instead. A good reason for doing this is that we may not have a specific value of n in mind (we may be arguing something for all n), so we can't give a completely explicit listing that works in all cases. The ellipsis can also be used to denote infinite sets, e.g.,

$$\mathbb{N} = \{0, 1, 2, \ldots\},$$
$$\mathbb{Z} = \{\ldots, -2, -1, 0, 1, 2, \ldots\}.$$

Definition 5.1.1. For any finite set A (i.e., A has a finite number of elements), we let $\|A\|$ denote the number of elements of A. This number is always a natural number (for finite sets) and is called the *cardinality* of A.

So for example, $\|\{1, 4, 9, 16, 25\}\| = 5$.

Set formers

If the members of a set are not so easily listable, even using ellipses (e.g., the set has many members that don't form a regular pattern, or the set is infinite, or there is no easy way to express some of the set's members), then a *set former* may be used to describe the set. In general, a set former is an expression of the form

$$\{\langle \text{expression} \rangle \mid \langle \text{property} \rangle\}.$$

Here, ⟨expression⟩ is some arbitrary expression, usually involving one or more variable names, e.g., $x, y, z, \ldots$, and ⟨property⟩ is some statement about the variables used in the ⟨expression⟩. The set former above denotes the set whose members are all possible values of the expression as the variables range over all possible

values satisfying the property. The divider (|) can be read as "such that", and the set former itself can be read as, "the set of all ⟨expression⟩ such that ⟨property⟩."

For example, the set (5.1) above can be denoted by the set former

$$\{x^2 \mid x \in \mathbb{Z} \wedge 1 \leq x \wedge x \leq 5\}$$

Informally, this is the set of all squares of integers in the range 1 to 5, inclusive.[1] The two inequalities involving x can be contracted to the shorthand, "$1 \leq x \leq 5$", so the set former can be written,

$$\{x^2 \mid x \in \mathbb{Z} \wedge 1 \leq x \leq 5\}.$$

Generally, a set may have more than one set former denoting it. The set former

$$\{x^2 \mid x \in \mathbb{N} \wedge 0 < x < 6\}$$

denote the same set.

Any variable name introduced in the expression part of a set former is local to the set former itself. Such a variable is called a *dummy variable*. The actual name chosen for this variable does not affect the set, provided the name is used consistently throughout the set former. For example, we can change the name x to y in the set former above to get a new set former for the same set:

$$\{y^2 \mid y \in \mathbb{Z} \wedge 1 \leq y \leq 5\}.$$

Here is another example using two dummy variables to denote the set of rational numbers:

$$\mathbb{Q} = \left\{\frac{a}{b} \mid a, b \in \mathbb{Z} \wedge b \neq 0\right\}.$$

We can rename each dummy variable consistently throughout to obtain another set former for the same set:

$$\mathbb{Q} = \left\{\frac{x}{y} \mid x, y \in \mathbb{Z} \wedge y \neq 0\right\}.$$

The dummy variables used in a set former have no meaning outside of the set former. They are "local" to the set former. This is similar to variables local to a function in a programming language; they cannot be accessed outside the body of the function.

[1] We will use the wedge symbol ($\wedge$) to mean "and" (conjunction), the vee symbol ($\vee$) to mean "or" (disjunction), and the prefix $\neg$ to mean "not" (negation). Following standard logical convention, we will always use "or" inclusively. That is, for statements P and Q, the statement $P \vee Q$ is true just when P is true or Q is true *or both*, i.e., when at least one of P, Q is true. If we ever mean the exclusive version, we will say so explicitly.

Don't confuse a set with its members!

A set is a single mathematical object that is intended to group together some number of mathematical objects into a single whole. A set should never by confused with its elements, even if the set has only one element. {17} is the set consisting of the number 17 as its only member, but {17} itself is not a number.

5.2 Subsets and the empty set

Definition 5.2.1. For any sets A and B, we say *A is a subset of B*, and write $A \subseteq B$, to mean that every element of A is also an element of B. More formally, $A \subseteq B$ iff for all z, $z \in A \implies z \in B$.

We write $A \not\subseteq B$ to mean that A is not a subset of B, in other words, there is at least one element of A that is not an element of B.

Be careful not to confuse the two relations $A \subseteq B$ and $A \in B$. The former says that everything in A is also in B, whereas the latter says that the set A itself is an element of B. Remember that the set A is a single object distinct from its members.

The *empty set* (sometimes called the *null set*) is the set with no members. (By the Axiom of Extensionality, there can be only one such set, hence we are justified in calling it *the* empty set.) It is usually denoted by the symbol $\emptyset$. Here are some other ways to denote it:

$$\emptyset = \{\} = \{x \mid x \in \mathbb{Z} \wedge x \notin \mathbb{Z}\} = \{x \mid 0 = 1\}$$

For each of the set formers, the point is that the property is not satisfied by any x, so the denoted set has no elements. Notice that $\|\emptyset\| = 0$, and $\emptyset$ is the only set whose cardinality is 0.

Here are some easy properties of the subset relation:

Fact 5.2.2. *For any sets A, B, and C,*

1. *$\emptyset \subseteq A$ ($\emptyset$ is a subset of every set),*
2. *$A \subseteq A$ (every set is a subset of itself, i.e., the subset relation is* reflexive*),*
3. *if $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$ (the subset relation is* transitive*),*
4. *if $A \subseteq B$ and $B \subseteq A$, then $A = B$ (the subset relation is* antisymmetric*).*

Proving two sets equal

The last item in Fact 5.2.2 (antisymmetry of $\subseteq$) deserves some comment. It is true because if everything in A is in B and vice versa, then A and B have the same elements, and so must be equal by Extensionality. We will often need to prove that

two sets are equal, and we can use antisymmetry to do this. Suppose we have sets A and B that we want to prove equal. Antisymmetry says that our proof can consist of two subproofs: one that $A \subseteq B$, and the other that $B \subseteq A$. To prove "subsethood," e.g., that $A \subseteq B$, we show that any element of A must also lie in B. Thus we can follow this template:

> Let z be any element of A. Then blah blah blah ... and therefore, $z \in B$.

We will see some examples of this type of proof shortly.

5.3 Boolean set operations

Definition 5.3.1. Let A and B be any sets. We define

$$A \cup B := \{z \mid z \in A \vee z \in B\},$$
$$A \cap B := \{z \mid z \in A \wedge z \in B\},$$
$$A - B := \{z \mid z \in A \wedge z \notin B\}.$$

$A \cup B$ is called the *union* of A and B; $A \cap B$ is the *intersection* of A and B; $A - B$ is the *complement* of B in A (also called the complement of B *relative* to A).

These three operations are called Boolean because they correspond to the Boolean connectives OR, AND, and NOT, respectively. Informally, $A \cup B$ is the set of all things that are either in A or in B (or both). $A \cap B$ is the set of all things common to (in both) A and B. $A - B$ is the set of all things in A which are not in B. (It could be read, "A except B.")

For example, let $A = \{1, 3, 4, 6\}$ and let $B = \{0, 2, 4, 6, 7\}$. Then $A \cup B = \{0, 1, 2, 3, 4, 6, 7\}$, $A \cap B = \{4, 6\}$ and $A - B = \{1, 3\}$.

It turns out that the intersection operation can be defined in terms of the other two. This will give us our first example of a proof of set equality.

Proposition 5.3.2. *For any sets A and B,*

$$A \cap B = A - (A - B).$$

Proof. To show equality, it suffices to show (1) that $A \cap B \subseteq A - (A - B)$ and (2) that $A - (A - B) \subseteq A \cap B$.

1. Let z be any element of $A \cap B$. We show that $z \in A - (A - B)$. Since $z \in A \cap B$, we have by definition that $z \in A$ and $z \in B$. Since $A - B = \{x \mid x \in A \wedge x \notin B\}$, the element z (being in B) fails this criterion, and thus $z \notin A - B$. But since $z \in A$, we then have $z \in A - (A - B)$, again by definition. Since z was chosen arbitrarily from $A \cap B$, it follows that $A \cap B \subseteq A - (A - B)$.

2. Now let z be any element of $A - (A - B)$. We show that $z \in A \cap B$. From $z \in A - (A - B)$ it follows by definition that $z \in A$ and $z \notin A - B$. Recalling that $A - B = \{x \mid x \in A \wedge x \notin B\}$, if $z \notin A - B$, then z must violate this condition, i.e., it is not the case that both $z \in A$ and $z \notin B$. That is, either $z \notin A$ (violating the first statement) or $z \in B$ (violating the second). We know by assumption that $z \in A$, so it must be the second: $z \in B$. Thus $z \in A$ and $z \in B$, so by definition $z \in A \cap B$. Since z is an arbitrary element of $A - (A - B)$, it follows that $A - (A - B) \subseteq A \cap B$.

□

The preceding proof can be "condensed" to a string of equivalences involving an arbitrary object z (using 0 to mean FALSE):

$$\begin{aligned}
z \in A - (A - B) &\iff z \in A \wedge z \notin (A - B) \\
&\iff z \in A \wedge \neg(z \in (A - B)) \\
&\iff z \in A \wedge \neg(z \in A \wedge z \notin B) \\
&\iff z \in A \wedge (z \notin A \vee z \in B) \\
&\iff (z \in A \wedge z \notin A) \vee (z \in A \wedge z \in B) \\
&\iff 0 \vee (z \in A \wedge z \in B) \\
&\iff z \in A \wedge z \in B \\
&\iff z \in A \cap B.
\end{aligned}$$

This derivation shows the parallels between the Boolean set operations and their logical counterparts (AND, OR, NOT). Although it may look more formal, such a derivation is not necessarily preferable: the Boolean transformations are hard to pick through, and justifying the steps requires some Boolean identities (De Morgan's Law and a distributive law, for example) that you may or may not know. A more prosaic proof like the first one above is perfectly fine, and it works in cases where no formal chain of equalities/equivalences is possible.

The next fact, given without proof, gives several basic identities satisfied by the Boolean set operators.

Fact 5.3.3. *For any sets A, B, and C,*

- $A \cup B = B \cup A$ *and* $A \cap B = B \cap A$. *(Union and intersection are both commutative.)*
- $(A \cup B) \cup C = A \cup (B \cup C)$ *and* $(A \cap B) \cap C = A \cap (B \cap C)$. *(Union and intersection are both associative. This justifies dropping parentheses for repeated applications of the same operation, e.g.,* $A \cup B \cup C$ *and* $A \cap B \cap C$.*)*
- $A \cup A = A \cap A = A$.

- $A \cap B \subseteq A \subseteq A \cup B$.
- $A - B \subseteq A$.
- $A \cup \emptyset = A$ *and* $A \cap \emptyset = \emptyset$.
- $A \subseteq B$ *iff* $A \cup B = B$ *iff* $A \cap B = A$ *iff* $A - B = \emptyset$.

Here is another example of a proof that two sets are equal. It is one of the distributive laws for $\cup$ and $\cap$.

Theorem 5.3.4 (Intersection distributes over union)**.** *For any sets A, B, and C,*

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C).$$

Proof. First, we show that $A \cap (B \cup C) \subseteq (A \cap B) \cup (A \cap C)$. Let z be any element of $A \cap (B \cup C)$. Then $z \in A$, and $z \in B \cup C$, which means that either $z \in B$ or $z \in C$.

Case 1: $z \in B$. Then since $z \in A$, we have $z \in A \cap B$. Thus $z \in (A \cap B) \cup (A \cap C)$ (because $z \in (A \cap B) \cup$ (anything)).

Case 2: $z \in C$. Similarly, since $z \in A$, we have $z \in A \cap C$ and so $z \in (A \cap B) \cup (A \cap C)$.

In any case, we have $z \in (A \cap B) \cup (A \cap C)$.

Second, we show that $(A \cap B) \cup (A \cap C) \subseteq A \cap (B \cup C)$. Let z be any element of $(A \cap B) \cup (A \cap C)$. Then either $z \in A \cap B$ or $z \in A \cap C$.

Case 1: $z \in A \cap B$. Then $z \in A$ and $z \in B$. Since $z \in B$, it surely follows that $z \in B \cup C$ as well. Thus $z \in A \cap (B \cup C)$.

Case 2: $z \in A \cap C$. Similarly, we get $z \in A$ and $z \in C$, whence it follows that $z \in B \cup C$, and so $z \in A \cap (B \cup C)$ as before.

In either case, $z \in A \cap (B \cup C)$. □

5.4 Sets of sets, ordered pairs, Cartesian product

Sets are objects themselves, so we can form sets of sets. For example, the set

$$\{\emptyset, \{3, 4\}, \{3\}, \{4\}\}$$

is a set containing four elements, each a set of integers drawn from the set $\{3, 4\}$. In fact, this is the set of all subsets of $\{3, 4\}$. We can forms sets whose elements are sets whose elements are also sets of ....

The empty set is an actual object, despite having no elements. And so, $\emptyset \neq \{\emptyset\}$, because the second set is not empty (it has one member, namely $\emptyset$).

Given any mathematical objects a and b, we can form the *ordered pair* of a and b as a single object, denoted (a, b). Don't confuse this with $\{a, b\}$; the latter is sometimes called the *unordered pair* of a and b. In (a, b), the order matters, and so $(a, b) \neq (b, a)$ unless $a = b$. Duplicates also matter, so $(a, a) \neq a$. Given the ordered pair (a, b), a is called the *first coordinate* of the pair, and b is the *second coordinate*. The key fact about ordered pairs is that they just completely identify their coordinates and nothing else:

Fact 5.4.1. *For any ordered pairs (a, b) and (c, d),*

$$(a, b) = (c, d) \iff (a = c \wedge b = d).$$

That is, two ordered pairs are equal iff their corresponding coordinates are both equal. This is the *only* relevant fact about ordered pairs. Any correct "implementation" of ordered pairs only needs to satisfy this one fact.

Definition 5.4.2. Let A and B be any sets. We define the *Cartesian product of A and B as follows:*

$$A \times B := \{(a, b) \mid a \in A \wedge b \in B\}.$$

For example,

$$\{1, 2, 3\} \times \{3, 4\} = \{(1, 3), (1, 4), (2, 3), (2, 4), (3, 3), (3, 4)\}.$$

We take all combinations of an element from A with an element from B. A has three elements, B has two elements, and their Cartesian product has $3 \cdot 2 = 6$ elements. This should suggest to you the following fact:

Fact 5.4.3. *If A and B are finite sets, then so is $A \times B$, and*

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

Notice that

$$\{3, 4\} \times \{1, 2, 3\} = \{(3, 1), (3, 2), (3, 3), (4, 1), (4, 2), (4, 3)\} \neq \{1, 2, 3\} \times \{3, 4\}$$

so Cartesian product is not commutative in general.

Proving the following distributive laws will be a homework exercise.

Fact 5.4.4 (Cartesian product distributes over union and intersection)**.** *For any sets A, B, and C,*

$$A \times (B \cup C) = (A \times B) \cup (A \times C),$$
$$(A \cup B) \times C = (A \times C) \cup (B \times C),$$
$$A \times (B \cap C) = (A \times B) \cap (A \times C),$$
$$(A \cap B) \times C = (A \times C) \cap (B \times C).$$

We must state both types of distributive law for each operation (union and intersection), because Cartesian product is not commutative.

* Ordered pairs as sets

One standard, traditional way to define an ordered pair as a set is as follows:

Definition 5.4.5. Let a and b be any mathematical objects. Then the ordered pair of a and b is defined as

$$(a, b) := \{\{a\}, \{a, b\}\}.$$

It can be shown that this definition of ordered pairs satisfies Fact 5.4.1, and so it is a legitimate way to implement ordered pairs as sets. There are other ways, but all correct implementations must satisfy Fact 5.4.1.

Exercise: With this definition, what are $(3, 4)$, $(3, 3)$, and $((3, 4), 5)$ as sets? Write them as compactly as possible in standard set notation (i.e., comma separated list between braces).

5.5 Relations and functions

I will just give the basic notions here. I hope that this is mostly review from MATH 374 at least.

Given two sets A and B, a *(binary) relation from A to B* is any subset R of $A \times B$. That is, R consists entirely of ordered pairs of the form (a, b) for some $a \in A$ and $b \in B$. We sometimes write aRb to mean $(a, b) \in R$. If $B = A$, then we say that R is a binary relation *on* A. For example, $\leq$ is a binary relation on $\mathbb{R}$, consisting of all ordered pairs (x, y) of real numbers such that $x \leq y$. (Notice that we usually write "$x \leq y$" instead of "$(x, y) \in \leq$", which looks silly even though it is more formally correct.) For another example, the equality relation "$=$" is the binary relation (on any set A) consisting of the ordered pairs (x, x) for all $x \in A$. There are lots of interesting possible types of binary relations on a set: equivalence relation, pre-order, partial order, total order, tournament, etc. We will not need these concepts.

A relation f from set A to set B is called a *function mapping A into B* iff for every $a \in A$ there exists a unique (that is, exactly one) $b \in B$ such that $(a, b) \in f$. If this is the case, we may write $f : A \to B$, and we say that A is the *domain* of f and that B is a *codomain* of f. Also, for every $a \in A$, we let $f(a)$ denote the unique $b \in B$ such that $(a, b) \in f$ (read this as "f of a" or "f applied to a"), and we say that f *maps a to b*. If $f(a) = f(b)$ implies $a = b$ (for all $a, b \in A$), then we say that f is *one-to-one*. If for all $b \in B$ there exists $a \in A$ such that $b = f(a)$, then we can say that f maps *A onto B* (rather than simply *into*).

5.6 The pigeonhole principle

The *pigeonhole principle* is a useful tool in mathematical proofs. Here it is, stated formally using functions. It is a reasonably obvious fact about mappings between finite sets, and we will not prove it (although there is a fairly straightforward proof by induction).

Theorem 5.6.1 (Pigeonhole Principle)**.** *Let A and B be finite sets, and suppose $f : A \to B$ is any function mapping A into B. If $\|B\| < \|A\|$, then f cannot be one-to-one, that is, there must exist distinct $a, b \in A$ such that $f(a) = f(b)$.*

Less formally, however you associate to each element of a finite set A some element of a smaller set B, you must wind up associating the same element of B to (at least) two different elements of A. The principle gets its name from homing pigeons: if you have m pigeons and each must fly through one of n holes, where $n < m$, then two pigeons must fly through the same hole.

Here is an example adapted from Wikipedia: There must be at least two residents of Los Angeles with the same number of hairs on their heads. The average number of hairs on a human head is about $150,000$, and it is reasonable to assume that nobody has more than $1,000,000$ hairs on their head. Since there are more than $1,000,000$ people living in Los Angeles, at least two have the same number of hairs on their heads. That is, the function mapping each Angelino to the number of hairs on his or her head cannot be one-to-one.

Here is another, classic example that combines the pigeonhole principle with proof-by-cases:

Proposition 5.6.2. *In any graph with at least two vertices, there exist two vertices with the same degree.*

Stated another way, at a party with $n \geq 2$ people, there are always two different people who shake hands with the same number of people at the party.

Proof. Let G be a graph with n vertices, where $n \geq 2$. Then the degree of any vertex is in the set $\{0, 1, \ldots, n-1\}$. Let V be the set of vertices of G, and let $d : V \to \{0, 1, \ldots, n-1\}$ be the function mapping each vertex to its degree. We have $\|V\| = n$.

Case 1: G has an isolated vertex (that is, there exists a $v \in V$ such that $d(v) = 0$). Then no vertex has degree $n-1$, and so in fact, $d(V) \subset \{0, 1, \ldots, n-2\}$. Since the set on the right has $n-1$ elements, by the pigeonhole principle, there exist vertices $u \neq v$ such that $d(u) = d(v)$.

Case 2: G has no isolated vertices. Then $d(V) \subseteq \{1, 2, \ldots, n-1\}$ and the set on the right has $n-1$ elements. Thus as in Case 1, there exist $u \neq v$ such that $d(u) = d(v)$.

□

There is a stronger version of the pigeonhole principle:

Theorem 5.6.3 (Strong Pigeonhole Principle)**.** *Let A and B be finite sets with $\|A\| = m$ and $\|B\| = n > 0$, and suppose $f : A \to B$ is any function mapping A into B. Then there exists an element $b \in B$ such that $b = f(a)$ for at least m/n many $a \in A$.*

This version can be proved by contradiction: If each of the n points $b \in B$ had fewer than m/n many pre-images (i.e., $a \in A$ such that $f(a) = b$), then there would be fewer than $n(m/n) = m$ pre-images in all. But then this would not account for all the m elements of A, each of which is a pre-image of some $b \in B$.

The strong pigeonhole principle implies the (standard) pigeonhole principle: if $m > n$, then $m/n > 1$, and so there must be some $b \in B$ with at least two pre-images (since the number of pre-images must be a natural number).

There are versions of the pigeonhole principle involving infinite sets. Here is one:

Theorem 5.6.4. *Let A and B be sets such that A is infinite and B is finite. For any function $f : A \to B$ there must exist $b \in B$ such that $b = f(a)$ for infinitely many $a \in A$.*

*Application: Hall's theorem

Your organization has a number of members. Among its members there are some committees made up of groups of members, which may overlap (the same person could belong to more than one committee). Each committee must have a *chair*, who must be a member of the committee. To ease workloads, you don't want the same person to chair more than one committee (although a chair may still be a *member* of multiple committees). Hall's theorem addresses when this is possible.

Definition 5.6.5. A (finite) *set system* is a pair (D, C), where D is any finite set (the "domain") and C is any collection of subsets of D.

Think of D as the organization (i.e., the set of its members), the elements of C being the committees.

Definition 5.6.6. Let (D, C) be a set system. A *system of distinct representatives (sdr)* for (D, C) is a one-to-one mapping $r : C \to D$ such that $r(S) \in S$ for all $S \in C$.

Think of $r(S)$ as being the chair of committee S. An sdr is then a choice of chair for each committee with no person chairing more than one committee (r is one-to-one).

Example. Let $D := \{1, 2, 3, 4\}$ and $C := \{\{1, 2\}, \{1, 3\}, \{2, 3\}, \{2, 3, 4\}\}$. One possible sdr maps

$$\begin{aligned} \{1, 2\} &\mapsto 1 \\ \{1, 3\} &\mapsto 3 \\ \{2, 3\} &\mapsto 2 \\ \{2, 3, 4\} &\mapsto 4 \end{aligned}$$

There is one other sdr for this example. Find it.

Here is a set system with no sdr: $D := \{1, 2, 3\}$ and $C := \{\{1\}, \{1, 2\}, \{1, 3\}, \{2, 3\}\}$.

There is one obvious case where an sdr does *not* exist. If some set of j many committees comprises a total of fewer than j many members, then there are not enough different committee members to be chairs of all these j committees. This is the pigeonhole principle in action. In the second example above, there are four sets containing a total of three elements, so no sdr.

Hall's theorem[2] says that this is the *only* case where there is no sdr.

Definition 5.6.7. A set system (D, C) has the $(\star)$-*property* if, for every $j > 0$ and j many (pairwise distinct) sets $S_1, S_2, \ldots, S_j \in C$, we have

$$\left|S_1 \cup S_2 \cup \cdots \cup S_j\right| \geq j \ .$$

If (D, C) has an sdr, then by the pigeonhole principle it must have the $(\star)$-property. Hall's theorem is the converse.

Theorem 5.6.8 (Hall's theorem)**.** *If set system (D, C) has the $(\star)$-property, then (D, C) has an sdr.*

Proof. We use induction[3] on $k := |C|$. For the base case, if $k = 0$, then $C = \emptyset$, whence (D, C) has an sdr, namely, the empty mapping.

Now assume $k > 0$ and that the theorem holds for all $j < k$, that is, for all $j < k$ and for any set system (D', C') such that $|C'| = j < k$, if (D', C') has the $(\star)$-property, then (D', C') has an sdr. (This is the inductive hypothesis.) We need to show that (D, C) has an sdr. We have two cases:

Case 1: For every positive $j < k$ and for every collection of j many sets $S_1, \ldots, S_j \in C$, we have

$$|S_1 \cup \cdots \cup S_j| > j$$

[2] Quite amazing, IMHO.

[3] strong induction, actually

(note the strict inequality). Then choose an arbitrary $S \in C$ and some arbitrary $x \in S$. (S is nonempty because (D, C) has the $(\star)$-property.) Define

$$\begin{aligned} D' &:= D - \{x\} \ , \\ C' &:= \{T \cap D' \mid T \in C \wedge T \neq S\} \\ &= \{T - \{x\} \mid T \in C \wedge T \neq S\} \ . \end{aligned}$$

(D', C') is a set system. Let $\ell := |C'|$. We have $\ell < k$, because in the set former, T ranges over all sets in C except for S. We next show that (D', C') has the $(\star)$-property: Let $T_1, \ldots, T_j$ be j many arbitrary (pairwise distinct) sets in C', for some $0 < j \leq \ell$, and let $T'_1, \ldots, T'_j$ be elements of $C - \{S\}$ such that $T_i = T'_i - \{x\}$ for all $1 \leq i \leq j$. We have

$$T_1 \cup \cdots \cup T_j = (T'_1 - \{x\}) \cup \cdots \cup (T'_j - \{x\}) = (T'_1 \cup \cdots \cup T'_j) - \{x\} \ ,$$

and thus

$$|T_1 \cup \cdots \cup T_j| \geq |T'_1 \cup \cdots \cup T'_j| - 1 \ . \tag{5.2}$$

By the Case 1 assumption (and the fact that $0 < j < k$), we have

$$|T'_1 \cup \cdots \cup T'_j| > j \ .$$

Then combining this with (5.2), we have

$$|T_1 \cup \cdots \cup T_j| \geq j \ .$$

Since $T_1, \ldots, T_j$ were chosen arbitrarily from C', this establishes that (D', C') has the $(\star)$-property. Now since $|C'| = \ell < k$, we can apply the inductive hypothesis to (D', C') to get that (D', C') has an sdr $r' : C' \to D'$. Now extend r' to a function $r : C \to D$ by defining

$$\begin{aligned} r(S) &:= x \ , \\ r(T) &:= r'(T - \{x\}) \qquad\qquad \text{(for all } T \in C - \{S\}). \end{aligned}$$

The function r' is one-to-one because it is an sdr, and since $x \notin D'$, the function r is also one-to-one. Since $r(S) = x \in S$ and $r(T) \in T$ for all $T \in C$, we have that r is an sdr for (D, C).

Case 2: Not Case 1, that is, there exists a positive $j < k$ and j many sets $S_1, \ldots, S_j \in C$ such that

$$|S_1 \cup \cdots \cup S_j| \leq j \ .$$

By the $(\star)$-property for (D, C), we actually have equality:

$$|S_1 \cup \cdots \cup S_j| = j \ .$$

Now let $C' := \{S_1, \ldots, S_j\}$. Then $|C'| = j < k$, and the set system (D, C') has the $(\star)$-property, because all the sets in C' are also in C. Then by the inductive hypothesis (D, C') has an sdr $r' : C' \to D$. Let R be the range of r', i.e.,

$$R := \mathrm{rng}(r') = \{r(S_i) \mid 1 \leq i \leq j\} .$$

Since r' is one-to-one, $|R| = j$. Since $r'(S_i) \in S_i$ for all $1 \leq i \leq j$, we have $R \subseteq S_1 \cup \cdots \cup S_j$. But then, $R = S_1 \cup \cdots \cup S_j$, because both sets have size j. Define

$$\begin{aligned} D' &:= D - R , \\ C'' &:= \{T - R \mid T \in C \wedge T \notin C'\} = \{T \cap D' \mid T \in C - C'\} . \end{aligned}$$

(D', C'') is a set system, and $|C''| \leq |C - C'| = k - j < k$, so we can apply the inductive hypothesis to (D', C'') provided we can show that it has the $(\star)$-property.

Claim 5.6.9. (D', C'') has the $(\star)$-property.

We prove Claim 5.6.9 afterward.

By Claim 5.6.9 and the inductive hypothesis, (D', C'') has an sdr $r'' : C'' \to D'$. Now we combine r' and r'' into a single sdr r for (D, C) by defining, for all $T \in C$,

$$r(T) := \begin{cases} r'(T) & \text{if } T \in C', \\ r''(T - R) & \text{if } T \in C - C'. \end{cases}$$

Observe that r is well-defined. Why? Because the domains of r' and r'' are disjoint: for every $T' \in C'$ and for every $T'' \in C''$, we have $T' \subseteq R$ and $T'' \cap R = \emptyset$, and since both T' and T'' are nonempty, we must have $T' \neq T''$.

The maps r' and r'' are both one-to-one, and their ranges are disjoint (empty intersection): $\mathrm{rng}(r') = R$ and $\mathrm{rng}(r'') \subseteq D' = D - R$. Thus r is one-to-one. We also have $r(T) \in T$ for all $T \in C$, and so r is an sdr for (D, C).

In both cases then, (D, C) has an sdr. □

Proof of Claim 5.6.9. Let $T_1, \ldots, T_q \in C''$ be any q many pairwise distinct sets in C', where $q > 0$. We want to show that

$$|T_1 \cup \cdots \cup T_q| \geq q .$$

By the definition of C'', for each i with $1 \leq i \leq q$ there exists a set $T_i' \in C - C'$ such that $T_i' - R = T_i$. We observe that the sets $T_1', \ldots, T_q', S_1, \ldots, S_j$ are all pairwise distinct; in particular, no T_i' can equal any $S_{i'}$ because the latter is in C' and the former is in $C - C'$. So by the $(\star)$-property of (D, C) and recalling that $R = S_1 \cup \cdots \cup S_j$, we have

$$|T_1' \cup \cdots \cup T_q' \cup R| = |T_1' \cup \cdots \cup T_q' \cup S_1 \cup \cdots \cup S_j| \geq q + j . \tag{5.3}$$

Also, $T_i \subseteq T_i' \subseteq T_i \cup R$ for all i, and so

$$T_1 \cup \cdots \cup T_q \cup R = T_1' \cup \cdots \cup T_q' \cup R \,,$$

which combines with (5.3) to give

$$|T_1 \cup \cdots \cup T_q \cup R| \geq q + j \,.$$

Therefore, we finally have

$$\begin{aligned} |T_1 \cup \cdots \cup T_q| &= |(T_1 \cup \cdots \cup T_q \cup R) - R| \qquad \text{(because } (T_1 \cup \cdots \cup T_q) \cap R = \emptyset\text{)} \\ &= |T_1 \cup \cdots \cup T_q \cup R| - |R| \\ &\geq q + j - j \\ &= q \,, \end{aligned}$$

and we are done. □

Discussion. Here are some things to ponder:

1. Which case does the first example given earlier fall into?
2. Use the proof to recursively build an sdr for that example step by step.
3. In the $k = 1$ case, you have a set system of the form $(D, \{S\})$ for some $S \subseteq D$. How does the size of S determine which case to apply?
4. Although Hall's theorem is conceptually elegant, it does not suggest any efficient algorithm to find an sdr (or even just determine if one exists), since one presumably would need to exhaustively check all subsets of C to establish whether the $(\star)$-property holds. There *is*, however, an efficient algorithm for this problem: an sdr corresponds to a sufficiently large matching in a certain bipartite graph, which can be found by an efficient max-flow algorithm due to Edmonds and Karp.

5.7 Countable sets

In this lecture we will talk about how to "count" the number of elements of an infinite set and present a counting technique that is absolutely fundamental to thinking about some later things in this course.

Definition 5.7.1. We say that a set has *infinite cardinality* (shortened to "is infinite" or "is an infinite set") if the number of elements in the set is larger than any given integer n. We say that a set has *finite cardinality* if there exists an integer n that is larger than the number of elements in the set.

Note that any set, under these definitions, is either of finite cardinality or of infinite cardinality, but not both.

Fact 5.7.2. *Let $S = \{1, 2, 3, ...\}$. then S has infinite cardinality.*

Proof. Assume not, that is, assume that there exists an integer m that is larger than the number of elements in the set. However, we can enumerate S as

$$S = \{1, ..., m, m + 1, ...\}$$

But now each of the elements j for integers $1 \leq j \leq m$ is in S, which is a total of m elements. But we're not done. There is at least $m + 1$, which is one more. So S has *more* than m elements. So S cannot be of finite cardinality and must be of infinite cardinality. □

Definition 5.7.3. We say that sets S and T *have the same cardinality* if there is a 1-1 and onto function $f : S \to T$.

Definition 5.7.4. We say that a set S is *countably infinite* (sometimes shortened to *countable*) if there is a 1-1 and onto function $f : \mathbb{N} \to S$.

Theorem 5.7.5. *The set $\mathbb{N}$ of natural numbers is countably infinite.*

Proof. We let f be the identity function $f(n) = n$. □

Theorem 5.7.6. *The set $\mathbb{N}^+$ of natural numbers is countably infinite.*

Proof. We let f be the function $f : \mathbb{N} \to \mathbb{N}^+$ such that $f(n) = n + 1$. □

Note that this last implies that we can "count" using either the nonnegative integers

$$\{0, 1, 2, 3, ...\}$$

or the positive integers

$$\{1, 2, 3, ...\}$$

depending on which is more convenient.

Note also that this can be extended so that shifting up (or down) by any finite number of places doesn't change the countable-ness. We can show that

$$\{3, 4, 5, ...\}$$

is countable by shifting by 3 instead of just by 1. We shift down a finite number of places on the "left" end of the set, and this doesn't affect the counting because we can always adjust with the infinite number of things on the right end of the set.

Theorem 5.7.7. *The set of nonnegative even integers is a countable set.*

Proof. We map $\mathbb{N}$ to $\{0, 2, 4, 6, ...\}$ in the obvious way:

$$f(n) = 2n$$

□

so $0 \to 0$, $1 \to 2$, $2 \to 4$, $3 \to 6$, $4 \to 8$, $5 \to 10$, ...

Theorem 5.7.8. *The set of positive odd integers is a countable set.*

Proof.

$$f(n) = 2n + 1$$

□

So among other things, we have two sets, all nonnegative integers and all even nonnegative integers, each of which is infinite, one of which is properly contained within the other, but which have the same cardinality, a concept we usually think of as "having the same number of elements."

This shows that a set, the set of all nonnegative integers, "has the same number of elements" (according to the notion of "the same number of elements" that is Definition 5.7.3) as a set, the set of even nonnegative integers, that we would normally think of as having only half as many elements. We can work the other way as well.

Theorem 5.7.9. *The set $\mathbb{Z}$ of all integers is a countable set.*

Proof. We map from $\mathbb{N}$ to $\mathbb{Z}$ as follows.

$$\begin{aligned} 0 &\to 0 \\ 1 &\to 1 \\ 2 &\to -1 \\ 3 &\to 2 \\ 4 &\to -2 \\ 5 &\to 3 \\ &... \end{aligned}$$

□

Now, let's pause a moment. We start with a countably infinite set, the nonnegative integers.

If we add or delete one element, or five elements, or any finite number of elements, we do not change the cardinality; the set is still countably infinite.

If get rid of half the elements (e.g., all the odd numbers), the set is still countably infinite.

If we double the number of elements (e.g., to get the integers), the set is still countably infinite.

What about an infinite number of copies of a countably infinite set?

Theorem 5.7.10. *Let $R = \{(a, b) : a, b \in \mathbb{N}\}$ be the set of all pairs of nonnegative integers (think of these as the points with integer coordinates in the upper right quadrant of the $x - y$ plane). Then R is a countable set.*

Proof. Note that this is a set that extends infinitely in two dimensions:

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

We enumerate these using a dovetailing approach that is a cornerstone of the theory of computing (and other things as well). We enumerate upwards and to the right along the off-diagonals.

$$\begin{array}{r} 1 \to (0,0) \\ \\ 2 \to (0,1) \\ 3 \to (1,0) \\ \\ 4 \to (0,2) \\ 5 \to (1,1) \\ 6 \to (2,0) \\ \\ 7 \to (0,3) \\ 8 \to (2,1) \\ 9 \to (1,2) \\ 10 \to (3,0) \\ \ldots \end{array}$$

□

Note that we could not count just by starting with the top row, because that's of infinite length, and we would never finish that row and be able to move on the next row. Similarly, we could not count down the left column, because we would never finish that column and be able to move on to the next column. By counting on the diagonals, we never go down an infinite rabbit hole; although each diagonal

is one longer than the previous diagonal, each is of finite length and we know we will finish the diagonal.

We are going to pair the previous argument with the following theorem that will allow us to be a little sloppy in listing things.

This also shows something interesting: the cardinality of a countable set is the smallest cardinality of any set that is not finite. That is, there isn't anything that is infinite but "smaller than" countable.

Theorem 5.7.11. *[The "We can be a little sloppy theorem."] Let T be a countable set. Let S be a subset of T that is not finite. Then S is a countable set.*

Proof. If T is countable, then we can enumerate the elements of T: t_0, t_1, t_2, t_3, t_4, ...

We will build the function $g : \mathbb{N} \to S$ perhaps as follows:

$$\begin{array}{ll} t_0 \in S & 0 \to t_0 \\ t_1 \in S & 1 \to t_1 \\ t_2 \notin S & skip \\ t_3 \notin S & skip \\ t_4 \in S & 2 \to t_4 \\ \ldots & \end{array}$$

Since we assumed S to be infinite, we wind up in the rightmost column with an assignment of every nonnegative integer to an element of S. □

We can combine the dovetailing argument of Theorem 5.7.10 with the ability to be a little sloppy in the following. It might be a little tedious to try to define a tableau that consisted only of rational numbers in lowest terms. The ability to be a little sloppy helps because it doesn't require us to solve all those tedious details.

Theorem 5.7.12. *The set of rational numbers $\mathbb{Q}$ is a countable set.*

Proof. Since the rational numbers consist of all quotients a/b where $a \in \mathbb{Z}$ and $b \in \mathbb{Z}, b \neq 0$, we can write the rationals in a tableau similar to the tableau for pairs of nonnegative integers.

$$\begin{array}{cccccc} (1,1) & (1,2) & (1,3) & (1,4) & (1,5) & \ldots \\ (-1,1) & (-1,2) & (-1,3) & (-1,4) & (-1,5) & \ldots \\ (2,1) & (2,2) & (2,3) & (2,4) & (2,5) & \ldots \\ (-2,1) & (-2,2) & (-2,3) & (-2,4) & (-2,5) & \ldots \\ \ldots & & & & & \end{array}$$

where the element (a, b) represents the rational number a/b, which may or may not be in lowest terms. The rational numbers that are the elements of this tableau clearly form a countably infinite set, enumerated by the dovetailing argument used

for pairs. This set contains all the rationals, but the tableau has repeats since not all these represent a rational number in lowest terms.

However, we can apply the "We can be a little sloppy" Theorem 5.7.11. The set of all rational numbers in lowest terms is certainly an infinite subset of the elements in this tableau. The set of elements in this tableau is countable. So the set of all rational numbers is countable. □

Metaphysics

We have that the set of pairs of elements, where the elements come from a countable set, is a countable set. We can easily extend this to the set of triples of elements, each of which comes from a countable set. The set of triples (a, b, c) is clearly in 1-1 correspondence with the set of pairs $((a, b), c)$. Since the set of pairs (a, b) is countable, showing that the set of pairs $((a, b), c)$ is countable is just a slight variation of the proof of Thereom 5.7.10.

Indeed, for any fixed and finite n, the set of n-tuples, each of whose coordinates comes from a countable set, is countable just by a repeated application of Thereom 5.7.10.

5.8 Uncountable sets

We have shown that several kinds of "infinite" sets that would seem to be "of different sizes" are in fact "the same size" under our Definition 5.7.3 of "size". At this point it is then perfectly legitimate to inquire if there is anything that is not countable.

The answer is yes. (Of course the answer is yes. If the answer were no, then it's unlikely that there would be enough "there" there in this discussion to warrant even having the discussion.)

Theorem 5.8.1. *The set of real numbers in the open interval* $(0, 1)$ *is a set that is infinite but not countably infinite.*

Proof. We will prove this by contradiction. Every number r in the interval from 0.0 to 1.0 inclusive can be written as a decimal expansion

$$0.d_0d_1d_2d_3...$$

Now, let's assume that this set is in fact countable, which means that we can enumerate its elements as $r_0, r_1, r_2, ...$, Since each of these has a decimal expansion, we can write these as

$$\begin{aligned} r_0 &= 0.d_{00}d_{01}d_{02}d_{03}... \\ r_1 &= 0.d_{10}d_{11}d_{12}d_{13}... \\ r_2 &= 0.d_{20}d_{21}d_{22}d_{23}... \\ &... \end{aligned}$$

Let us then construct a real number s by means of its decimal expansion, and we construct s as

$$s = 0.s_0s_1s_2s_3\ldots$$

where

$$\begin{aligned} s_0 &= d_{00} + 1 \pmod{10} \\ s_1 &= d_{11} + 1 \pmod{10} \\ &\ldots \end{aligned}$$

. That is, $s_i = d_{ii} + 1$ if d_{ii} is a digit 0 through 8, and is 0 if d_{ii} is 9. (We don't want to add 1 and go from 9 to 10, because that's two digits and not one digit.)

This is clearly a real number, defined by an infinite decimal expansion, and it's in the range from 0.0 to 1.0 inclusive. We claim that this real number cannot be any of the r_i that we enumerated.

And it's not, because s differs, along the diagonal, from any of the r_i that we enumerated, in exactly the i-th digit d_{ii}.

This is a significant proof by contradiction. We assumed the negation of the conclusion, that the interval was in fact a countable set. We enumerated the elements of the set, which we could do if it were countable.

And then we showed that there was a real number in the set of the hypothesis that was not in our purported enumeration of elements. □

Commentary

This is the second of the three famous "diagonalization" arguments that will appear in this lecture. Like the first one, this is absolutely classic. It is probably fair to say that no graduate student in mathematics anywhere in the world in (at least) the last 75 years has not seen this argument. This is part of the canon.

Theorem 5.8.2. *The cardinality of the interval* $(-1, 1)$ *on the real line is the same as the cardinality of the interval* $(0, 1)$*.*

Proof. The 1-1 function needed for the definition of equal cardinalities is

$$f(x) = 2x - 1.$$

What this function does is map $1/2$ to 0, stretch the interval $(1/2, 1)$ into the interval $(0, 1)$, and stretch the interval $(0, 1/2)$ into the interval $(-1, 0)$. □

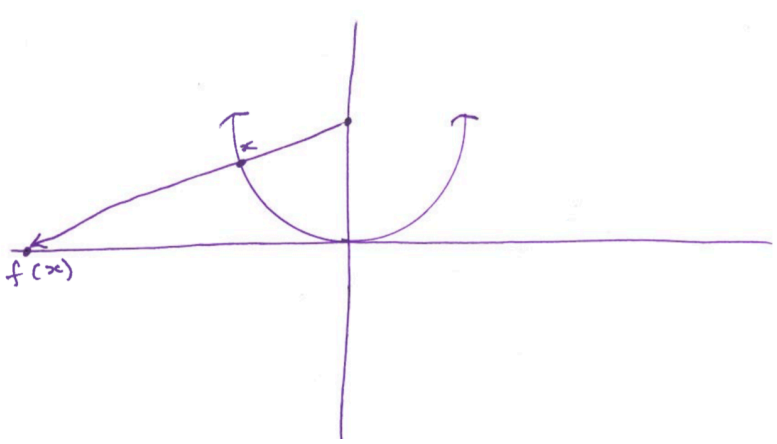
Now we go for another of the classic basic statements of cardinality.

Theorem 5.8.3. *The cardinality of the interval* $(-\pi/2, \pi/2)$ *on the real line is the same as the cardinality of the interval* $(0, 1)$*.*

Proof. This proof is just a variation of the "stretching" proof just completed. □

Theorem 5.8.4. *The cardinality of the (finite) interval* $(-\pi/2, \pi/2)$ *is the same as the cardinality of the entire real line* $\mathbb{R}$*.*

Proof. There is a classic proof by projection. Imagine we take the finite interval and wrap it into a half circle as in the following diagram. We then project as if with a beam of light. Any point x on the half-circle maps to exactly one point $f(x)$ on the real line. And since we carefully chose not to include the endpoints of the interval, the project always hits the real line, however far out on the line that might happen to be. (Projecting the two endpoints would of course be parallel to the real line and thus not intersect it.) □



5.9 Why should we care?

Theorem 5.9.1. *The total number of legal computer programs (in any programming language) is countably infinite.*

Proof. There are clearly infinitely many legal programs, since we can always imagine just being redundant:

```
void main() {
```

```
  int n;
  n = 1;
}
```

and

```
void main() {
  int n;
  n = 1;
  n = 1;
}
```

and

```
void main() {
  int n;
  n = 1;
  n = 1;
  n = 1;
}
```

and so forth. Stupid, but legal, and infinitely many.

So let's think of how we can enumerate the legal programs. Let's restrict ourselves for the moment to Python programs. Since there are only finitely many programming languages (even if we consider all the languages in non-English script), if we show how to demonstrate that each language has only countably many programs, then demonstrating that all together we have only countably many is just using the argument about pairs, triples, etc.

My current keyboard has 81 keys, and I don't think I need anything for a Python program that doesn't appear on my keyboard. So there are 81 possible Python programs that consist of one character. There are 81^2 possible Python programs that consist of two characters, and are 81^3 possible Python programs that consist of three characters, and so forth. Of course, most of these are not legal, but we are not going to care because we have the "we can be a little sloppy" theorem.

Enumerating the possible programs by length, clearly we see that there are only countably many such programs. Indeed, if we assigned a keyboard character to each of the values from 0 through 80, then we would know exactly how to enumerate the programs. Each possible program of k characters would be a k-digit number base 81 (read left to right or right to left, depending on what convention for reading we chose). □

Now we get to the third of the "diagonalization" methods.

Theorem 5.9.2. *There is a computer program $\mathcal{P}$ that will enumerate (i.e., list) all the Python programs that are legal, that require no input, and that terminate (i.e., do not have infinite loops). This program will not, however, terminate.*

Proof. Clearly if we just run Python on an invalid program, it will terminate and say the program is invalid.

And clearly if we run Python on a program that crashes, it will terminate by crashing. That's ok.

If we run Python on a legal program that requires no input and that runs to completion, that's ok.

We have to fudge details a little to talk about a program that might require input. If a program asks for console input, it might (if the system is naively written) keep polling for input. That is, the Python interpreter could keep asking the console, "is there input?", "is there input?", "is there input?", "is there input?", "is there input?", "is there input?", until input appeared. It is more likely on a modern system that the interpreter would wait for an interrupt that said that input was available (that interrupt would probably be triggered by hitting the "enter/return" key). But let's imagine polling.

Programs requiring input, and programs that have infinite loops, will thus run forever.

If we are going to list all the programs that don't run forever, then we have to be careful not to turn Python loose in an uncontrolled way on one of those programs.

This is entirely analogous to not just enumerating all the way across the top row of the tableau in Theorem 5.7.10.

First, we enumerate all the possible Python programs: $P_1, P_2, P_3, P_4, P_5, P_6, \ldots$ Now we do the following:

1. a) Run $\mathcal{P}$ for one step on P_1.
2. a) Run $\mathcal{P}$ for two steps on P_1.
   b) Run $\mathcal{P}$ for one step on P_2.
3. a) Run $\mathcal{P}$ for three steps on P_1.
   b) Run $\mathcal{P}$ for two steps on P_2.
   c) Run $\mathcal{P}$ for one step on P_3.
4. a) Run $\mathcal{P}$ for four steps on P_1.
   b) Run $\mathcal{P}$ for three steps on P_2.
   c) Run $\mathcal{P}$ for two steps on P_3.
   d) Run $\mathcal{P}$ for one step on P_4.
5. and so forth

If the program has terminated after running the program in one of these steps, we print its sequence number.

It doesn't really matter what we use for "step" in this. That could be one second of CPU time, one Python instruction, one machine instruction, etc. What matters is that we never turn total control over to a program that could be infinite in its execution time.

This "program" as described could readily be turned into a linux script as a double loop. If a program is legal, requires no input, and terminates, then it will eventually be executed for more time steps than is needed to run to completion, and the sequence number will be printed.

□

Why is this important?

This last section has had a moderate level of fudging and imprecision about legal versus illegal programs, input, which programming language, and so forth. But the argument about running programs for limited time periods "along the diagonal" is independent of this, and this is a fundamental method used in foundations of computing.

Much of the rest of the course will be an attempt to do this section of this lecture in a clear, crisp way. The arguments will be more abstract, but that will be to eliminate the fudging about keys on keyboards and such.

5.10 Constructing the nonnegative integers

In this lecture we will construct the nonnegative integers in a different way and then prove that what we have constructed is what we would intuitively have thought we were constructing.

Definition 5.10.1. Let $f(x) : \mathbb{R} \to \mathbb{R}$ be defined as follows.

$$f(x) = x + 1$$

Definition 5.10.2. We define the *nonnegative integers* $\mathbb{N}$ as follows.

$$\mathbb{N} = \{0\} \cup \{f(n) | n \in \mathbb{N}\}$$

Note that this is a recursive definition.

Now, clearly we have $0 \in \mathbb{N}$ and $f(0) = 1 \in \mathbb{N}$ and $f(1) = 2 \in \mathbb{N}$ and $f(2) = 3 \in \mathbb{N}$ and so forth.

Now let's define another function.

Definition 5.10.3. Let $g(x) : \mathbb{R} \to \mathbb{R}$ be defined as follows.

$$g(x) = f(f(x)) = f(x + 1) = x + 2$$

and then we will use a recursive definition to define

Definition 5.10.4.

$$\mathbb{N}^E = \{0\} \cup \{g(n) | n \in \mathbb{N}\}$$
$$\mathbb{N}^O = \{1\} \cup \{g(n) | n \in \mathbb{N}\}$$

Theorem 5.10.5.

$$n \in \mathbb{N}^E \iff n = 2m$$

for some $m \in \mathbb{N}$.

$$n \in \mathbb{N}^O \iff n = 2m + 1$$

for some $m \in \mathbb{N}$.

Proof. First off we're going to unwind the recursive definition. We claim that

$$\mathbb{N}^E = \{0\} \cup \{g(0)\} \cup \{g(g(0))\} \cup \{g(g(g(0)))\} \cup \ldots$$

This is straightforward. If $n \in \mathbb{N}$, then $n = n' + 2$ for some $n' \in \mathbb{N}$. And If then $n = n' + 2 = n'' + 4$ for some $n'' \in \mathbb{N}$. And so forth. Since we are making things smaller with each step, this descent cannot continue infinitely, and eventually we get to a nonnegative value less than 2. That value has to be 0 and not 1, because applying g increases the size of the integer, and the initial application of g to 0 skips 1.. The number of times we have to un-apply g to get back to 0 is the value m desired for the theorem.

The proof for $\mathbb{N}^O$ is entirely similar.

□

Now, the whole reason for doing this argument is that the previous argument defined the even numbers to be those that were $2m$ and the odd numbers to be "everything else", without describing what an odd number looked like. Our construction has described two kinds of numbers, which we could call even and odd, but has not yet shown that these are all the numbers we can have. That's the point of the next theorem.

Theorem 5.10.6. *If* $n \in \mathbb{N}$ *then either* $n \in \mathbb{N}^E$ *or* $n \in \mathbb{N}^O$ *but not both.*

Proof. We can unwind the recursive definition of $\mathbb{N}$ to be

$$\mathbb{N} = \{0\} \cup \{f(0)\} \cup \{f(f(0))\} \cup \{f(f(f(0)))\} \cup \ldots$$

and then we just rearrange this list.

$$\begin{aligned}
\mathbb{N} =& \{0\} \cup \{f(f(0))\} \cup \{f(f(f(f(0))))\} \cup \ldots \\
& \cup \{1 = f(0)\} \cup \{f(f(1))\} \cup \{f(f(f(f(1))))\} \cup \ldots \\
=& \{0\} \cup \{g(0)\} \cup \{g(g(0))\} \cup \ldots \\
& \cup \{1 = f(0)\} \cup \{g(1)\} \cup \{g(g(1))\} \cup \ldots
\end{aligned}$$

□

Lecture 6

6.1 Alphabets and Strings

Definition 6.1.1. Let Σ be any nonempty, finite set. A *string* w over Σ is any finite sequence $w_1 w_2 \cdots w_n$, where $w_i \in \Sigma$ for all $1 \leq i \leq n$. Here, n is the *length* of w (denoted $|w|$) and can be any natural number (including zero). For each $i \in \{1, \ldots, n\}$, w_i is the *i'th symbol* of w.

The set Σ we sometimes call the *alphabet*, and the elements of Σ *symbols* or *characters*. We depict a string by juxtaposing the symbols of the string in order from left to right. The same symbol may appear more than once in a string. Unlike with sets, duplicates and order does matter with strings: two strings w and x are equal iff (1) they have the same length (say $n \geq 0$), and (2) for all $i \in \{1, \ldots, n\}$, the i'th symbol of w is equal to the i'th symbol of x. That is, $w = x$ iff w and x look identical when written out. We will consider the symbols of Σ themselves to be strings of length 1.

The concatenation operator

Given any two strings x and y, we can form the *concatenation* of x followed by y, denoted xy. It is the result of appending y onto the end of x. Thus concatenation is a binary operator defined on strings and returning strings. Clearly, the length of the concatenation is the sum of the lengths of the strings:

$$|xy| = |x| + |y|.$$

Concatenation is not generally commutative, that is, it is usually the case that $xy \neq yx$ (give examples where equality holds and where equality does not hold). Concatenation is always *associative*, howerver. That is, if you first concatenate strings x and y, then concatenate the result with a string z, you get the same string as you would by first concatenating y with z then concatenating x with the result. In other words,

$$(xy)z = x(yz)$$

for all strings x, y, and z. Note that the parentheses above are only used to show how the concatenation operator is applied; they are not part of the strings themselves.

Associativity allows us to remove parentheses in multiple concatenations. For example the above string can simply be written xyz. The same hold for concatenations of more than three strings.

The empty string

There is exactly one string of length zero (regardless of the alphabet). This string is called the *empty string* and is usually denoted by ε (the Greek letter epsilon).[1] The symbol ε is special, and it should not be considered part of any alphabet. Therefore it never appears as a literal component of any string (contributing to the length of the string). To be technically correct to a ridiculous extent, the empty string should be denoted as

(it is empty after all!), but this just looks like we forgot to write something, so we use ε as a placeholder instead.

The empty string acts as the *identity* under concatenation. That is, for any string w,

$$w\varepsilon = \varepsilon w = w.$$

ε is the only string with this property; when part of a concatenation, it simply disappears.

Formalities

We will often prove facts involving strings. For example, we may wish to prove that all strings (over a fixed alphabet Σ) have a certain property. The most useful technique for such proofs (in our case, at least) is by induction on the length of a string. We also use induction (or if you like, recursion) to *define* things involving strings. Such definitions and proofs have the advantage of being precise and not relying on our intuitions (which are often wrong). In this subsection, we will do both—we will define the concatenation operator formally by induction, then use this formal definition to (formally) prove that string concatenation is associative. In this subsection, it would be good to ignore the last two subsections entirely and pretend that we are starting from scratch with these concepts. For example, we will pretend that we don't yet know that string concatenation is associative. For

[1] Some books and papers use λ (lambda) to denote the empty string.

all this, we fix an arbitrary alphabet Σ and assume that all strings we mention are over Σ.

Inductions on strings all use the following basic idea, which describes how a string can be built out of single symbols. Definition 6.1.2 specifies what it means to append a symbol to a string to form a longer string, or alternatively, to break up a nonempty string by separating out its last symbol. Since every string can be built up from the empty string ε by repeatedly appending single symbols, Definition 6.1.2 shows us how induction on strings can be done. As a bonus, it also inductively defines the length of a string.

Definition 6.1.2. For any string x, exactly one of the following cases holds:

1. $x = \varepsilon$ (the *empty string*).
2. There exist a unique string y and a unique symbol $a \in \Sigma$ such that $x = ya$.

In the first case, $|x| = 0$. In the second case, $|x| = |y| + 1$, and we say that a is the *last symbol* of x and y is the *principal prefix* of x. We also say that x results from *appending* the symbol a to y.

Case 1 above describes the base case of a string induction, and Case 2 describes the inductive case. In the rest of this subsection, we will use Definition 6.1.2 to do three things—first formally define string concatenation (inductively on the length of the second string), then prove that $\varepsilon x = x$ for any string x, then finally prove that string concatenation is associative. The goal here is not to teach you anything new about strings, but rather to illustrate how induction on strings works.

Definition 6.1.3 defines the concatenation of two strings x and y inductively (recursively) on the length of y. We denote concatenation by juxtaposing the strings being concatenated, i.e., xy is the concatenation of x and y. (We also used juxtaposition in Definition 6.1.2 to denote appending a symbol to a string. These two uses agree if we identify a single symbol with a string of lenth 1.)

Definition 6.1.3. Given any two strings x and y, we define the *concatenation* xy of x and y as follows, inductively on $|y|$:

Base Case: If $y = \varepsilon$, then $xy = x\varepsilon := x$ (that is, we *define* the concatenation of x with ε to be x).

Inductive Case: For $y \neq \varepsilon$, let z be the unique string and a be the unique symbol of Σ such that $y = za$ (that is, a is the last symbol of y and z is the principal prefix of y, referring to Definition 6.1.2 above). Then we define

$$xy = x(za) := (xz)a\,,$$

that is, we define xy to be the result of appending a to the concatenation xz. (So xy is the unique string whose last symbol is a and whose principal prefix is the concatenation xz, assumed to be already defined inductively).

This inductive definition is well-founded (i.e., no infinite recursion) because in the inductive case, $|z| < |y|$, so we can assume (inductive hypothesis) that xz has already been defined. You can think of xz as the result of a recursive call to the concatenation operator with arguments x and z.

The next proposition has a straightforward proof and is used as a warm-up. It says that ε is a left identity under concatenation. (Definition 6.1.3 already defines ε to be a right identity.)

Proposition 6.1.4. *$\varepsilon x = x$ for any string x.*

Proof. We proceed by induction on $|x|$.

Base Case: If $x = \varepsilon$, then $\varepsilon x = \varepsilon\varepsilon = \varepsilon = x$, the second equality by the base case of Definition 6.1.3.

Inductive Case: If $x \neq \varepsilon$, then let a be the last symbol of x and let y be the principal prefix of x (thus $x = ya$). Since $|y| < |x|$, we can assume for the inductive hypothesis that $\varepsilon y = y$. Then

$$\varepsilon x = \varepsilon(ya) = (\varepsilon y)a = ya = x \ .$$

The second equality is by the inductive case of Definition 6.1.3, and the third equality is by the inductive hypothesis.

The proposition is thus proved. □

It is not necessary to regidly structure a proof like we did above by separating the two cases into a list. Proofs can be more prosaic and still be precise. We prove the next proposition in a more relaxed, conversational style. Comments in brackets are optional and can be removed.

Proposition 6.1.5. *String concatenation is associative, that is, for any strings x, y, and z,*

$$(xy)z = x(yz) \ .$$

Proof. We proceed by induction on $|z|$. If $z = \varepsilon$ [base case], then we have $(xy)z = (xy)\varepsilon = xy$, and likewise, $x(yz) = x(y\varepsilon) = xy$. Thus the proposition holds in this case. [Note that both equalities use the base case of Definition 6.1.3.] Otherwise, let string w and symbol a be such that $z = wa$ [noting that $|w| = |z| - 1 < |z|$], and assume (inductive hypothesis) that $(xy)w = x(yw)$. Then

$$\begin{aligned}
(xy)z &= (xy)(wa) \\
&= ((xy)w)a && \text{[by the inductive case of Definition 6.1.3]} \\
&= (x(yw))a && \text{[by the inductive hypothesis]} \\
&= x((yw)a) && \text{[by the inductive case of Definition 6.1.3]} \\
&= x(y(wa)) && \text{[ditto]} \\
&= x(yz) \ .
\end{aligned}$$

[Note that we can only use the inductive case of Definition 6.1.3 when we are appending a single symbol to a string, not for arbitrary concatenations.] This concludes the proof. □

Exercise 6.1.6. Give inductive proofs of the following for all strings x, y, and z. You may assume without proof standard facts about natural numbers.

1. $|x| \geq 0$.
2. $|xy| = |x| + |y|$.
3. (Right Cancellation.) If $xz = yz$, then $x = y$.
4. (Left Cancellation.) If $xy = xz$, then $y = z$.

Exercise 6.1.7. The *reversal* of a string x (denoted x^R) is the string formed by putting the symbols of x in reverse order. (For example, $(abcb)^R = bcba$.)

1. Give a precise, inductive definition of the reversal x^R of a string x.
2. Using your definition, give proofs by induction that $|x^R| = |x|$ and that $(x^R)^R = x$ for any string x.

6.2 Languages

Given an alphabet Σ, we let Σ^* denote the set of all strings over Σ. For our purposes, a *language* over Σ is any set of strings over Σ, i.e., any subset of Σ^*.

Languages as decision problems

The simplest type of computational problem is a *decision problem*. A decision problem has the form, "Given an input object w, does w have some property?" For example, these are all decision problems:

1. Given a graph G, is G connected?
2. Given a natural number n, is n prime?
3. Given an $n \times n$ matrix A with rational entries, is A invertible?
4. Given a binary string x, does x contain 001 as a substring?
5. Given integers a, b, and c, is there a real solution to the equation $ax^2 + bx + c = 0$?
6. Given an ASCII string y, is y a well-formed expression in the C++ programming language?

7. Given a collection of positive integers $\{a_1, \ldots, a_n\}$ and a positive integer t (all numbers given in binary), is there some subset of $\{a_1, \ldots, a_n\}$ whose sum is t?

A decision problem asks a yes/no question about some input object. The given objects are *instances* of the problem. Those for which the answer is "yes" are called yes-instances, and the rest are called no-instances. An algorithmic solution (or *decision procedure*) to a decision problem is some algorithm or computational device which takes an instance of the problem as input and outputs (in some way) the correct answer (yes or no) to the question for that instance. All the examples given above, except for the last one, are known to have efficient algorithmic solutions. (Computational problems that are not decision problems are ones that ask for more than just a yes/no answer. For example, "Given a natural number n, what is the smallest prime number larger than n?"; "Given a graph G and vertices s, t of G, find a path from s to t in G." We won't consider these here, at least for a while.)

All input objects are finite, and so can be ultimately encoded as strings. For example, natural numbers can be given by their binary representation, graphs can be given by their adjacency matrices, texts by their ASCII strings, etc. Any object that could conceivably be the input to an algorithm can be placed in a file of finite length, and in the end, that file is just a finite sequence of bits, i.e., one long binary string. For this reason, we will assume that all inputs in a decision problem are strings over some convenient alphabet Σ.

A decision problem, then, just asks a yes/no question about every string in Σ^*. Given any decision problem, the yes-instances of the problem form subset of Σ^*, i.e., a language over Σ. Conversely given any language L over Σ^*, we can form the decision problem, "Given a string $w \in \Sigma^*$, is w a member of L?" In this way, languages and decision problems are interchangeable; they encode the same information: the answer to a yes/no question for every string in Σ^*.

Put in very general, somewhat vague terms, a computational device A *recognizes* a language L over Σ iff the possible behaviors of A when fed strings $w \in L$ as input are distinguishable from those possible behaviors of A when fed strings $w \notin L$ as input. That is, one can tell whether a string w is in L or not by looking at the behavior of A on input w.

6.3 Finite automata

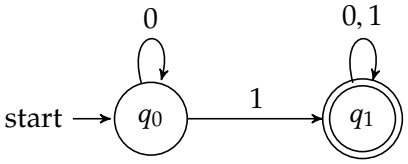
The first computational device we consider is a very simple (and very weak) one: the *deterministic finite automaton*[2], or DFA for short. A DFA has a finite number of states, with a preset collection of allowed transitions between the states labeled

[2] "Automaton" is the singular form of the noun. The plural is "automata."

with symbols from the alphabet Σ. Starting in some designated *start state*, the automaton reads the input string $w \in \Sigma^*$ from left to right, making the designated transition from state to state for each symbol read, until the entire string w is read. The DFA then either *accepts* or *rejects* the input w, depending only on which state the DFA was in at the end.

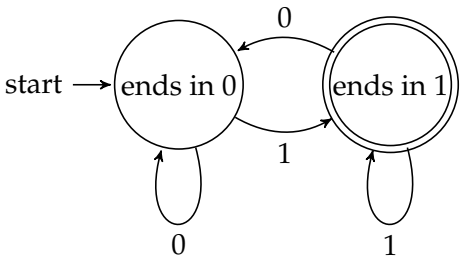
That's it. The DFA has no auxillary memory, and it can't do calculations on the side. We'll define a DFA more formally later, but in the mean time, here is a simple example of a DFA:

A DFA recognizing binary strings that contain at least one 1

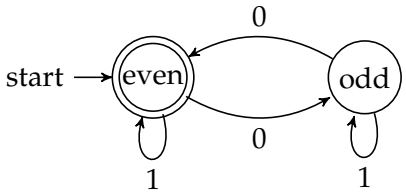


Several examples of automata today:

Example 6.3.1. checking that the last symbol of a binary string is 1

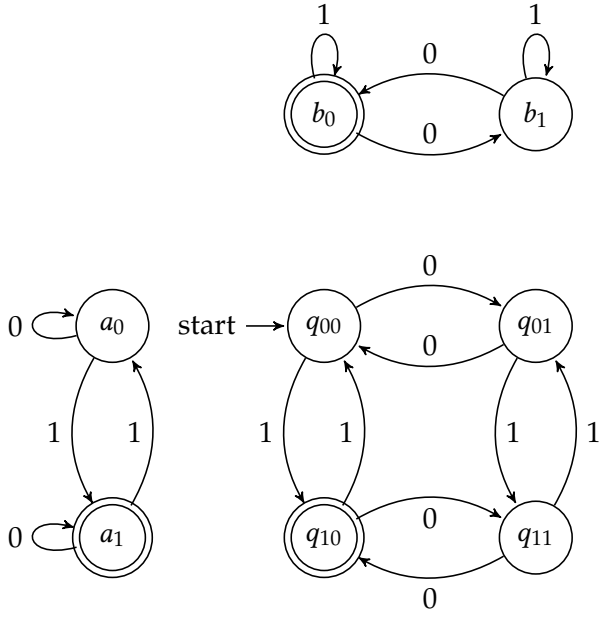


Example 6.3.2. checking for an even number of 0's in a binary string



Example 6.3.3. product construction for even 0's and odd 1's

The product construction.



Instead of a diagram, the transitions can also be described in a more declarative fashion:

	0	1
q_{00}	q_{01}	q_{10}
q_{01}	q_{00}	q_{11}
q_{10}	q_{11}	q_{00}
q_{11}	q_{10}	q_{01}

Assuming that the table allows us to derive Σ and the full set of states, notice that we still need to define the start and accepting states for a complete description.

Example 6.3.4. complementary automata — obtained by swapping accepting and non-accepting states

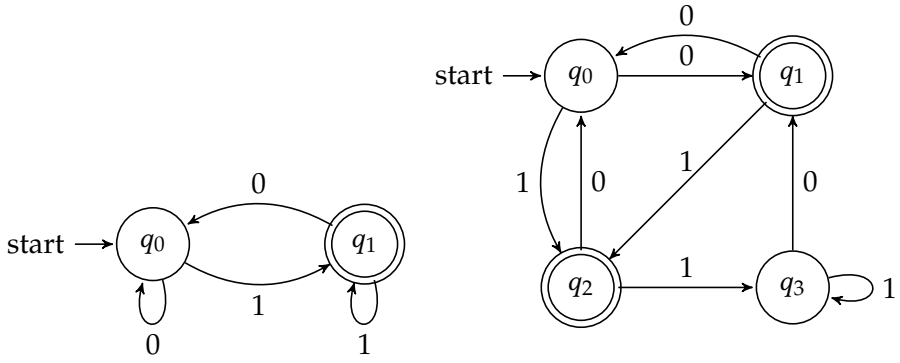
Lecture 7

7.1 String Search

Example 7.1.1 (KMP String Search Algorithm)**.**

7.2 DFAs, formally

Working with transition diagrams is fine, but we have to be careful to be precise. In order to arrive at a precise specification for an automaton, let's take a look at a few past examples:



What do these figures have in common?

- States. Call the set of states Q
- An alphabet to label the transitions. Call it Σ
- One start state, call it $q_0 \in Q$
- Accepting states. Call the set of accepting states $F \subseteq Q$

- Transitions. We can think of a transition as a function taking a state and a symbol and producing a new state. In other words, for $q \in Q, s \in \Sigma$, the transition function can be defined as $\delta : Q \times \Sigma \to Q$

An *automaton* is therefore the ordered collection (or a *tuple*) of those 5 elements

Definition 7.2.1. An *automaton* is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F \rangle$

We want to be able to talk about automata with respect to languages, but the transition function currently only describes what to do with the input one symbol at a time. In order to talk about strings, we need to extend the definition of the transition function to all strings in Σ^*.

Definition 7.2.2. Let $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ be a DFA. We define the function $\hat{\delta} : Q \times \Sigma^* \to Q$ inductively as follows: for any state $q \in Q$,

Base case: We define $\hat{\delta}(q, \varepsilon) := q$;

Inductive case: For any $x \in \Sigma^*$ and $a \in \Sigma$, we define $\hat{\delta}(q, xa) := \delta(\hat{\delta}(q, x), a)$.

$\hat{\delta}(q, w)$ is the state you wind up in when starting in state q and reading w.

Exercise 7.2.3. Check that $\hat{\delta}$ agrees with δ on individual symbols, i.e., strings of length 1.

Defining computation, acceptance, language recognition.

Definition 7.2.4. Given a string $w = (w_1 \dots w_n)$ and a DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$, a *computation* is the sequence of states $(q_0, \delta(q_0, w_1), \dots, \hat{\delta}(q_0, w))$

Definition 7.2.5. For a given string w, we say an automaton *accepts* w iff $\hat{\delta}(q_0, w) \in F$. Any given computation may either *accept* or *reject*

Now, for a given automaton, we can precisely define the language it recognizes.

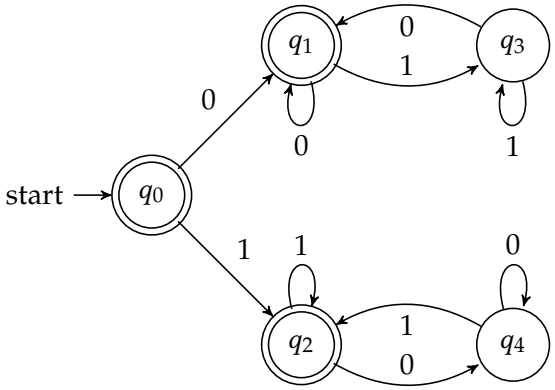
Definition 7.2.6. For a given DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$, the *language* of A is defined as

$$L(A) \coloneqq \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

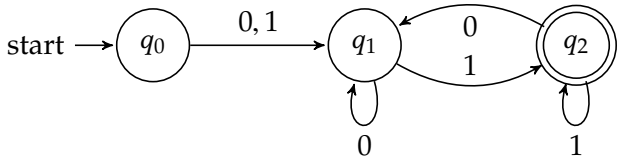
Definition 7.2.7. We say a language L is *regular* if L can be recognized by some DFA

More examples:

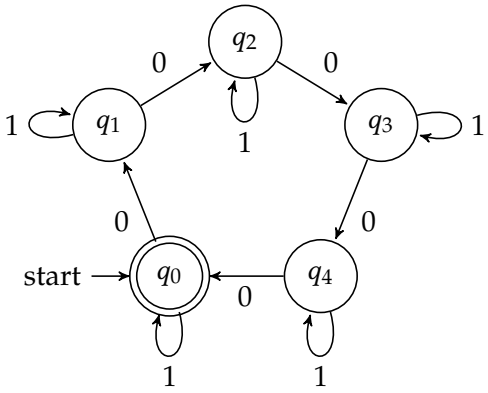
Example 7.2.8. nonempty binary strings that start and end with the same symbol



Example 7.2.9. binary strings of length ≥ 2 whose penultimate symbol is 1



Example 7.2.10. binary strings with a multiple of 5 many 0s



Exercise 7.2.11. binary representations of multiples of 3

DFAs given in tabular form.

7.3 *An Alternate Characterization of DFA Acceptance

Instead of using the extended transition function to define when a DFA accepts a string, we can use the notion of an accepting path. The next lemma makes this connection.

Lemma 7.3.1. *Let $D := \langle Q, \Sigma, \delta, q_0, F\rangle$ be a DFA, let $q, r \in Q$ be states of D, and let $w \in \Sigma^*$ be a string. Then $\hat{\delta}(q, w) = r$ if and only if there exist $k \geq 0$, symbols $w_1, \ldots, w_k \in \Sigma$, and states $s_0, s_1, \ldots, s_k \in Q$ such that*

- $w = w_1 \cdots w_k$,
- $s_0 = q$,
- $s_k = r$, *and*
- *for every* $1 \leq i \leq k$, $s_i = \delta(s_{i-1}, w_i)$.

We call the sequence of states $\langle s_0, \ldots, s_k\rangle$ the *computation path* in D from state q to state r reading w, and k is the *length* of this path. The path and the symbols $w_1, \ldots, w_k$ are uniquely determined, given D, q, and w. Thus we are justified in saying, "*the* computation path ...."

Proof of Lemma 7.3.1. By induction on $k = |w|$:

Base Case: $k = 0$**.** Here, $w = \varepsilon$, so $\hat{\delta}(q, w) = q$. Letting $r := q$ in the lemma, we see that $s_0 = q = r$, and so $\langle q\rangle$ satisfies all the criteria of a computation path from q to r reading w. Conversely, if $\langle s_0\rangle$ is a length-0 computation path from q to some state r reading w, then we must have $\hat{\delta}(q, w) = q = s_0 = r$.

Inductive Case: $k > 0$ **and the lemma holds for strings of length** $k - 1$**.** Write $w = xa$ for unique $x \in \Sigma^*$ and $a \in \Sigma$. First the forward implication: Let $r := \hat{\delta}(q, w)$ and let $s := \hat{\delta}(q, x)$. Since $|x| = k - 1$, by the inductive hypothesis, there exists a computation path $p := \langle s_0, \ldots, s_{k-1}\rangle$ in D from q to s reading x. We have $r = \delta(s, a)$ by definition of $\hat{\delta}$, so letting $s_k := r$ we can extend p to $\langle s_0, \ldots, s_{k-1}, s_k\rangle$, which (it is readily checked) is a computation path from q to r reading w. Conversely, suppose $\langle s_0, \ldots, s_{k-1}, s_k\rangle$ is a computation path from q to some state r reading w. We want to show that $r = \hat{\delta}(q, w)$. By the definition of computation path, we have $r = s_k = \delta(s_{k-1}, a)$. It follows that $\langle s_0, \ldots, s_{k-1}\rangle$ is a computation path from q to s_{k-1} reading x. By the inductive hypothesis, we then have $s_{k-1} = \hat{\delta}(q, x)$, whence

$$r = s_k = \delta(s_{k-1}, a) = \delta(\hat{\delta}(q, x), a) = \hat{\delta}(q, xa) = \hat{\delta}(q, w)$$

as required.

Thus the lemma is proved. □

A computation path from the start state q_0 to an accepting state is called an *accepting path*. A computation path from q_0 to a rejecting state is called a *rejecting path*. The next theorem is an immediate corollary of Lemma 7.3.1.

Theorem 7.3.2. *Let D be a DFA with alphabet Σ, and let $w \in \Sigma^*$ be a string. D accepts w if and only if D has an accepting path reading w.*

7.4 Product and Complement Constructions

Now that we have formal definitions of automata, we can create precise statements (proofs) about certain automata and the languages they recognize.

Here are formal definitions of the complementation and product construction we have used to recognize the intersection of the languages of two DFAs. This is described formally (using slightly different notation) on page 137, if you want to read ahead.

Definition 7.4.1. Let $A = (Q, \Sigma, \delta, q_0, F)$ and $B = (R, \Sigma, \zeta, r_0, G)$ be DFAs with common alphabet Σ.

1. We define the *product* of A and B as the following DFA:

$$A \wedge B := (Q \times R, \Sigma, \eta, (q_0, r_0), F \times G) ,$$

   where

$$\eta((q, r), a) := (\delta(q, a), \zeta(r, a))$$

   for all $q \in Q$, $r \in R$, and $a \in \Sigma$.

2. We define the *complement* of A as the following DFA:

$$\neg A := (Q, \Sigma, \delta, q_0, Q - F) .$$

We'll now prove formally the two fundamental facts about these two constructions. In both, we let Σ denote the common alphabet of the automata.

Theorem 7.4.2. *For any DFA A, $L(\neg A) = \overline{L(A)}$, where $\overline{L(A)} = \Sigma^* - L(A)$.*

Proof. Noticing that A and $\neg A$ share the same state set, transition function, and start state, we have, for every string $w \in \Sigma^*$,

$$\begin{aligned} w \in L(\neg A) &\iff \hat{\delta}(q_0, w) \in Q - F \\ &\iff \hat{\delta}(q_0, w) \notin F \\ &\iff w \notin L(A) \\ &\iff w \in \overline{L(A)} . \end{aligned}$$

Thus $L(\neg A) = \overline{L(A)}$ as required. □

Theorem 7.4.3. *For any DFAs A and B, $L(A \wedge B) = L(A) \cap L(B)$.*

Proof. Let A, B, and $A \wedge B$ be as in the definition above. First we show by induction on the length of a string w that the extended function $\hat{\eta}$ behaves as one would expect given $\hat{\delta}$ and $\hat{\zeta}$. That is, we prove that $\hat{\eta}((q_0, r_0), w) = (\hat{\delta}(q_0, w), \hat{\zeta}(r_0, w))$.

Base case: $\hat{\eta}((q_0, r_0), \varepsilon) = (q_0, r_0) = (\hat{\delta}(q_0, \varepsilon), \hat{\zeta}(r_0, \varepsilon))$.

Inductive case: Let x be a string over Σ and let a be a symbol in Σ. Assume (inductive hypothesis) that the equation holds for x, i.e., that $\hat{\eta}((q_0, r_0), x) = (\hat{\delta}(q_0, x), \hat{\zeta}(r_0, x))$. We show the same equation for the string xa:

$$\begin{aligned}
\hat{\eta}((q_0, r_0), xa) &= \eta(\hat{\eta}((q_0, r_0), x), a) && \text{(definition of } \hat{\eta}) \\
&= \eta((\hat{\delta}(q_0, x), \hat{\zeta}(r_0, x)), a) && \text{(inductive hypothesis)} \\
&= (\delta(\hat{\delta}(q_0, x), a), \zeta(\hat{\zeta}(r_0, x), a)) && \text{(definition of } \eta) \\
&= (\hat{\delta}(q_0, xa), \hat{\zeta}(r_0, xa)) && \text{(definitions of } \hat{\delta} \text{ and } \hat{\zeta})
\end{aligned}$$

So the same equation holds for xa. By induction, the equation holds for all strings w.

Now to prove the theorem, let $w \in \Sigma^*$ be any string. We have

$$\begin{aligned}
w \in L(A \wedge B) &\iff \hat{\eta}((q_0, r_0), w) \in F \times G && \text{(definition of acceptance for } A \wedge B) \\
&\iff (\hat{\delta}(q_0, w), \hat{\zeta}(r_0, w)) \in F \times G && \text{(the equation we just proved inductively)} \\
&\iff \hat{\delta}(q_0, w) \in F \text{ and } \hat{\zeta}(r_0, w) \in G && \text{(definition of Cartesian product)} \\
&\iff w \in L(A) \text{ and } w \in L(B) && \text{(definitions of acceptance for } A \text{ and } B) \\
&\iff w \in L(A) \cap L(B) && \text{(definition of set intersection)}
\end{aligned}$$

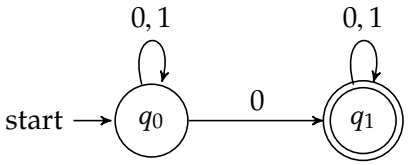
Thus $L(A \wedge B) = L(A) \cap L(B)$, because they have the same elements. □

Lecture 8

8.1 Nondeterministic finite automata (NFAs)

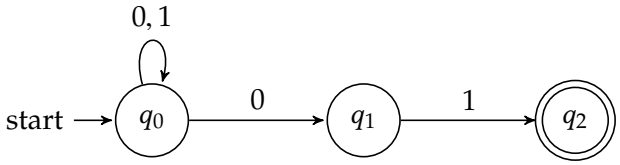
NFAs are like DFAs, where each state can have multiple (or no!) transitions for each symbol in the alphabet. Here are a couple of examples:

- An NFA accepting all strings that contain at least one 0 anywhere



	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
$*q_1$	$\{q_1\}$	$\{q_1\}$

- An NFA accepting all strings that end in 01



	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Notice that every DFA is essentially an NFA where each state has only one transition per symbol.

Definition 8.1.1. An *NFA* is a 5-tuple $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ where:

- Q is a finite set of states,
- Σ is an alphabet of input symbols,
- $q_0 \in Q$ is the start state,
- $F \subseteq Q$ is the set of *final* or *accepting* states, and
- $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ is the transition function taking pairs of a state and an input symbol to a subset of Q. $\mathcal{P}(Q)$ is called the *powerset* of Q (and is sometimes denoted 2^Q).

Extended Transition Function

Again, to define acceptance and computation we extend the transition function from $\delta : Q \times \Sigma \to \mathcal{P}(Q)$ to $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$.

Definition 8.1.2. We define $\hat{\delta}$ inductively:

Basis: $\hat{\delta}(q, \varepsilon) := \{q\}$. In other words, if we don't read in anything we stay in the same state.

Induction: Suppose w is of the form $w = xa$ for some string x and some symbol a. Also suppose $\hat{\delta}(q, x) = \{p_1, p_2, \ldots, p_k\}$. Let

$$\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, r_2, \ldots, r_m\}$$

Then $\hat{\delta}(q, w) := \{r_1, r_2, \ldots, r_m\}$.

Definition 8.1.3. Given an NFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ and a string w, we say *A accepts w* iff

$$\hat{\delta}(q_0, w) \cap F \neq \emptyset$$

Definition 8.1.4. The language of an NFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$ is written $L(A)$ and is defined

$$L(A) \coloneqq \{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

This doesn't really look like computation, does it? On the face of it, an NFA doesn't look like an actual computing device, since it "doesn't know" which transition to make on a symbol. So what's the point of an NFA? Best answer now: NFAs (like a DFAs) can be used to *specify* languages. If you want to communicate to someone a particular language in a precise way with a finite amount of information, you may be able just to provide an NFA recognizing the language. This completely specifies the language, because it pins down exactly which strings are in the language and which are out. Often, an NFA can specify a language much more compactly than the smallest possible DFA.

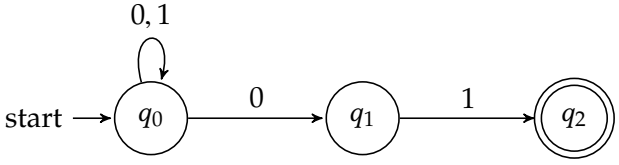
Definition 8.1.5. Two automata A and B are *equivalent* if $L(A) = L(B)$

This suggests the question: are there languages that are recognized by NFAs but not DFAs? Surprisingly, no. We'll prove that for any NFA N, there is a DFA D that recognizes the same language. D may need to have many more states than N, though. The conversion from an arbitrary NFA to an equivalent DFA is known as the *subset construction*, because the states of the DFA will be sets of states of the NFA.

8.2 Subset construction

Here we give a construction to convert some NFAs to DFAs using the example given before

An NFA accepting all strings that end in $0, 1$



	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

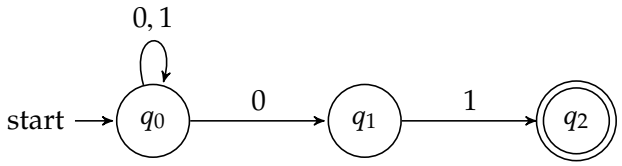
Start with the table of all possible subsets. If any subset contains an accepting state then that whole subset is accepting. Only the subset containing the original start state is the new initial state.

	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$*\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$*\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$*\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Lecture 9

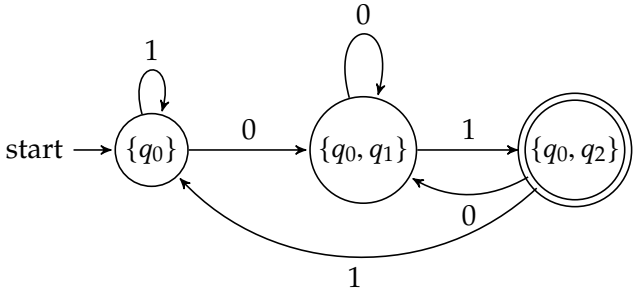
9.1 Optimized/Lazy subset construction

An NFA accepting all strings that end in 0, 1



	0	1
$\rightarrow \{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$*\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

A DFA accepting all strings that end in 0, 1



9.2 Proof of Correctness

The subset construction works in this case, but does it work in general? First, we need to define formally what it is.

We start with an NFA $N = \langle Q, \Sigma, \delta_N, q_0, F_N \rangle$, and the goal is to create a DFA $D = \langle \mathcal{P}(Q), \Sigma, \delta_D, \{q_0\}, F_D \rangle$ such that $L(N) = L(D)$.

Notice that in D, the set of states, the alphabet, and the start state are all "trivially constructable" from the description of N (e.g. the alphabet is the same). All thats left is to construct F_D and δ_D:

- F_D is the subsets of $\mathcal{P}(Q)$ such that for each $S \in \mathcal{P}(Q)$, $S \cap F_N \neq \emptyset$
- For each set $S \in \mathcal{P}(Q)$ and for each symbol $a \in \Sigma$, we define the transition

$$\delta_D(S, a) \coloneqq \bigcup_{p \in S} \delta_N(p, a)$$

Theorem 9.2.1. *If from an NFA $N = \langle Q, \Sigma, \delta_N, q_0, F_N \rangle$ we construct the DFA $D = \langle \mathcal{P}(Q), \Sigma, \delta_D, \{q_0\}, F_D \rangle$ by the subset construction, then $L(D) = L(N)$.*

Proof. First, we prove that the extended transition functions agree on any string w:

$$\hat{\delta}_D(\{q_0\}, w) = \hat{\delta}_N(q_0, w)$$

First note that $\hat{\delta}_D : \mathcal{P}(Q) \times \Sigma \to \mathcal{P}(Q)$ and $\hat{\delta}_N : Q \times \Sigma \to \mathcal{P}(Q)$, so the equivalence above is well-formed (i.e. the return types agree). We prove this by induction on the length of w.

Basis: Let $w = \varepsilon$. From the definitions of the transition functions,

$$\hat{\delta}_D(\{q_0\}, \varepsilon) = \{q_0\} = \hat{\delta}_N(q_0, \varepsilon)$$

Induction: Assume the statement is true for $|w| = n$. Let $|w| = n + 1$. Then $w = xa$ where $x \in \Sigma^*$, $a \in \Sigma$. Then since $|x| = n$, by the inductive hypothesis

$$\hat{\delta}_D(\{q_0\}, x) = \hat{\delta}_N(q_0, x)$$

Call both of these sets $\{p_1, \ldots, p_k\}$.

By the inductive part of the definition of $\hat{\delta}_N$ we have

$$\hat{\delta}_N(q_0, w) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$$

By the subset construction, we also have

$$\delta_D(\{p_1, \ldots, p_k\}, a) = \bigcup_{i=1}^{k} \delta_N(p_i, a)$$

Since $\hat{\delta}_D(\{q_0\}, x) = \{p_1, \ldots, p_k\}$, we can write the following chain of equalities:

$$\begin{aligned}
\hat{\delta}_N(q_0, w) &= \bigcup_{i=1}^{k} \delta_N(p_i, a) \\
&= \delta_D(\{p_1, \ldots, p_k\}, a) \\
&= \delta_D(\hat{\delta}_D(\{q_0\}, x), a) \\
&= \hat{\delta}_D(\{q_0\}, xa) \\
&= \hat{\delta}_D(\{q_0\}, w)
\end{aligned}$$

□

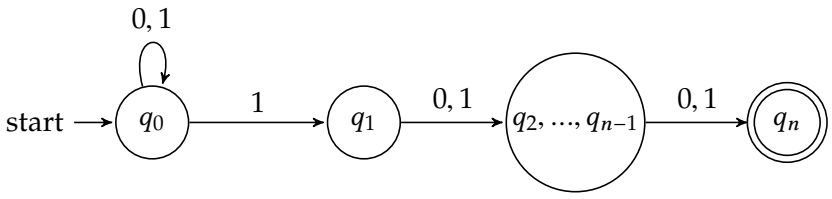
Now observe that D accepts iff $\hat{\delta}_D(\{q_0\}, w) \cap F_N \neq \emptyset$, and N accepts iff $\hat{\delta}_N(q_0, w) \cap F_N \neq \emptyset$. By the above theorem, D accepts iff N accepts.

9.3 An Example of the Worst Case

The reduction from nondeterministic to deterministic could conceivably go from an NFA of $|Q| = n$ states to a DFA with 2^n states, since in fact all the subsets of Q might be needed. Are there examples in which this worst case is "achieved"? Yes, almost.

Consider the NFA below, with $n + 1$ states, that accepts all strings for which the n-th from the last symbol is a 1.

A worst case example of NFA → DFA



The DFA version D of this automaton must be able to remember the last n symbols it has read. If D has read $w = a_1a_2...a_n$, then w will be accepted if $a_1 = 1$. But if $a_1 \neq 1$, and D reads a_{n+1}, then D will accept if $a_2 = 1$, so it needs to be able to remember that symbol.

There are 2^n possible n-bit strings, so if D can be implemented with fewer that 2^n states, then by the pigeonhole principle there would be two different strings $w_1 = a_1a_2...a_n$ and $w_2 = b_1b_2...b_n$ of n bits that took D from the start state to the same state q.

If these are two different strings, then they must differ in some bit, WLOG bit i, so that $a_i \neq b_i$.

We may assume by symmetry that $a_i = 1$ and $b_i = 0$. Now if $i = 1$, then q is a final state because D accepts w_1, but also q is not a final state because D rejects w_2, and that's a contradiction.

If $i \neq 1$, then we can add $i - 1$ zeros to the end of both w_1 and w_2. Now since D reads w_1 and w_2 and moves from the start state to some state q, D must continue on from q by reading those zeros as part of either string to arrive at some state p. That is: D reads either w_1 or w_2 and moves to state q, and reads either $w_1 0 \ldots 0$ or $w_2 0 \ldots 0$ and moves to q and then on to p. But now we have $a_i = 1$ as the n-th symbol from the end, and p must be a final state, but we have $b_i = 0$ as the n-th symbol from the end, and p must be a non-final state.

Again, we have a contradiction.

We must conclude that at least 2^n states are needed in order to remember n symbols.

Lecture 10

10.1 ε-transitions

An ε-NFA (or an NFA with ε-transitions, or ε-moves), is an NFA with an additional type of allowed transition: an edge labeled with ε. When this edge is followed, no symbol from the input is read, i.e., the input pointer is not advanced. These ε-transitions allow more flexibility in designing an automaton for a language.

Good example (from a book exercise): The language of all binary strings that are either one or more repetitions of 01 or one or more repetitions of 010.

Every NFA is essentially an ε-NFA, but even ε-NFAs are no more powerful at recognizing languages than DFAs.

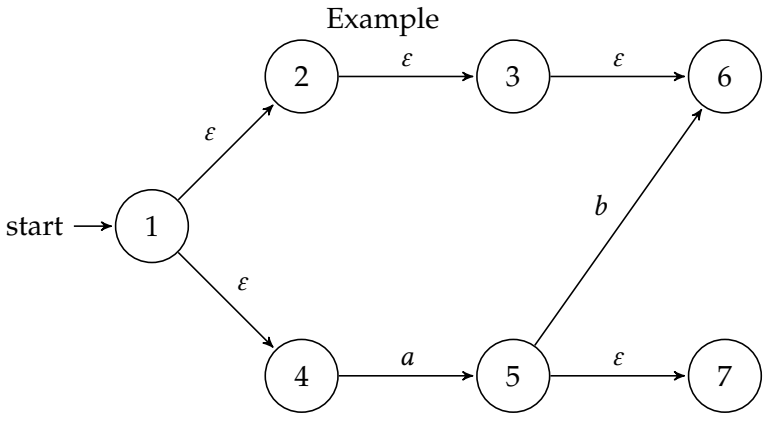
Definition 10.1.1. The *ε-closure* of a state is defined recursively as

Basis State q is in $\text{ECLOSE}(q)$

Induction If state p is in $\text{ECLOSE}(q)$, then $\text{ECLOSE}(q)$ also contains all of $\delta(p, \varepsilon)$

Definition 10.1.2. The *ε-closure* of a set of states S is defined as

$$\text{ECLOSE}(S) = \bigcup_{s \in S} \text{ECLOSE}(s)$$



10.2 ε-NFAs

Definition 10.2.1. An *ε-NFA* is a 5-tuple $\langle Q, \Sigma, \delta, q_0, F\rangle$, where $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \to \mathcal{P}(Q)$

Again, extend the transition function:

Definition 10.2.2. The *extended transition function* $\hat{\delta} : Q \times \Sigma^* \to \mathcal{P}(Q)$

Basis $\hat{\delta}(q, \varepsilon) = \text{ECLOSE}(q)$

Induction Suppose $w = xa$ for $x \in \Sigma^*$ and $a \in \Sigma$. Compute $\hat{\delta}(q, w)$ as follows:

1. Let $\hat{\delta}(q, x) = \{p_1, \ldots, p_k\}$
2. Let $\bigcup_{i=1}^{k} \delta(p_i, a) = \{r_1, \ldots, r_m\}$. Notice this is only *some* of the states reachable from all the p_i's
3. Then $\hat{\delta}(q, w) = \text{ECLOSE}(\{r_1, \ldots, r_m\})$

Definition 10.2.3. Given an ε-NFA $N = \langle Q, \Sigma, \delta, q_0, F\rangle$, the *language* of N is

$$L(N) \coloneqq \left\{w \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\right\}$$

Example: search for "`colo[u]r`"

10.3 *Alternate Characterizations of NFA and ε-NFA Acceptance

In Section 7.3, we gave an alternate characterization of DFA acceptance using accepting paths rather than the extended transition function. Here we present (briefly and without proofs) similar ways of characterizing NFA and ε-NFA acceptance. You should look back at that section to compare it with what follows.

Theorem 10.3.1. *Let $N \coloneqq \langle Q, \Sigma, \delta, q_0, F\rangle$ be an NFA, and let $w \in \Sigma^*$ be a string. Then N accepts w if and only if there exist $k \geq 0$, symbols $w_1, \ldots, w_k \in \Sigma$, and states $s_0, s_1, \ldots, s_k \in Q$ such that*

- $w = w_1 \cdots w_k$,
- $s_0 = q$,
- $s_k \in F$, *and*
- *for every* $1 \leq i \leq k$, $s_i \in \delta(s_{i-1}, w_i)$.

We call $\langle s_0, s_1, \ldots, s_k \rangle$, if one exists, an *accepting path* of N on w. Note the similarity with Lemma 7.3.1 and Theorem 7.3.2. Unlike with DFAs, however, the path may not be unique (although $w_1, \ldots, w_k$ are still uniquely determined from w, and $k = |w|$).

Theorem 10.3.2. *Let $N := \langle Q, \Sigma, \delta, q_0, F \rangle$ be an ε-NFA, and let $w \in \Sigma^*$ be a string. Then N accepts w if and only if there exist $k \geq 0$, strings $w_1, \ldots, w_k \in \Sigma \cup \{\varepsilon\}$, and states $s_0, s_1, \ldots, s_k \in Q$ such that*

- $w = w_1 \cdots w_k$,
- $s_0 = q$,
- $s_k \in F$, *and*
- *for every* $1 \leq i \leq k$, $s_i \in \delta(s_{i-1}, w_i)$.

The only difference between Theorems 10.3.1 and 10.3.2 is that in the latter, some of the w_i may be ε. This means that now the w_i are not even uniquely determined by w, and in fact, we could have $k > |w|$ (we always have $k \geq |w|$).

10.4 Eliminating ε transitions

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an ε-NFA. We define an equivalent NFA N' (without ε-transitions). There are two similar but not identical ways of doing this (this is not in the book):

Method 1

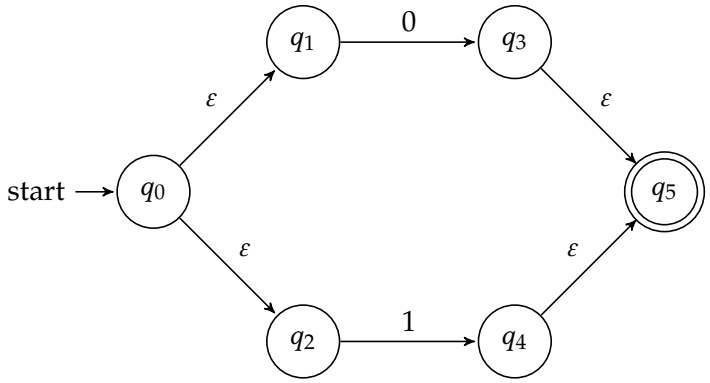
We let $N' = (Q, \Sigma, \delta', q_0, F')$, where

1. For all $q \in Q - \{q_0\}$ and $a \in \Sigma$, define
$$\delta'(q, a) := \text{ECLOSE}(\delta(q, a)) = \bigcup_{r \in \delta(q,a)} \text{ECLOSE}(r) .$$
2. Define
$$F' := \{q \in Q \mid \text{ECLOSE}(q) \cap F \neq \emptyset\} .$$
3. For all $a \in \Sigma$, define
$$\delta'(q_0, a) := \bigcup_{q \in \text{ECLOSE}(q_0)} \text{ECLOSE}(\delta(q, a)) .$$

One can prove that $L(N') = L(N)$.

Example 1

Consider the example to be found two chapters from now. This accepts either a single 0 or a single 1, and nothing else. But it's a simple example that allows the process of removing the ε transitions to be observed.



Let's do the table for δ.

δ	ε	0	1
q_0	$\{q_1, q_2\}$	$\emptyset$	$\emptyset$
q_1	$\emptyset$	$\{q_3\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_4\}$
q_3	$\{q_5\}$	$\emptyset$	$\emptyset$
q_4	$\{q_5\}$	$\emptyset$	$\emptyset$
q_5	$\emptyset$	$\emptyset$	$\emptyset$

Now let's do the ε-closures.

1. $q_0 \in ECLOSE(q_0)$
   $q_1 \in ECLOSE(q_0)$ since $q_1 \in \delta(q_0, \varepsilon)$
   $q_2 \in ECLOSE(q_0)$ since $q_2 \in \delta(q_0, \varepsilon)$
   So $ECLOSE(q_0) = \{q_0, q_1, q_2\}$
   (This shouldn't be surprising–*ECLOSE* of a state is the state together with all the states that can be reached by only ε-moves.)

2. $q_1 \in ECLOSE(q_1)$
   Nothing more, since there are no ε-moves from q_1.
   So $ECLOSE(q_1) = \{q_1\}$

3. $q_2 \in ECLOSE(q_2)$
   Nothing more, since there are no ε-moves from q_2.
   So $ECLOSE(q_2) = \{q_2\}$

4. $q_3 \in ECLOSE(q_3)$
   $q_5 \in ECLOSE(q_3)$ since $q_5 \in \delta(q_3, \varepsilon)$
   So $ECLOSE(q_3) = \{q_3, q_5\}$

5. $q_4 \in ECLOSE(q_4)$
   $q_5 \in ECLOSE(q_4)$ since $q_5 \in \delta(q_4, \varepsilon)$
   So $ECLOSE(q_4) = \{q_4, q_5\}$

6. $q_5 \in ECLOSE(q_5)$
   Nothing more, since there are no ε-moves from q_5.
   So $ECLOSE(q_5) = \{q_5\}$

Now we build δ'.

$$\begin{array}{rcccccc}
\delta'(q_0,0) & = & ECLOSE(\delta(q_0,0)) & \cup & ECLOSE(\delta(q_1,0)) & \cup & ECLOSE(\delta(q_2,0)) \\
& = & \emptyset & \cup & \{q_3,q_5\} & \cup & \emptyset \\
& = & \{q_3,q_5\} & & & & \\
\\
\delta'(q_0,1) & = & ECLOSE(\delta(q_0,1)) & \cup & ECLOSE(\delta(q_1,1)) & \cup & ECLOSE(\delta(q_2,1)) \\
& = & \emptyset & \cup & \emptyset & \cup & \{q_4,q_5\} \\
& = & \{q_4,q_5\} & & & & \\
\\
\delta'(q_1,0) & = & ECLOSE(\delta(q_1,0)) & = & ECLOSE(q_3) & = & \{q_3,q_5\} \\
\delta'(q_1,1) & = & ECLOSE(\delta(q_1,1)) & = & ECLOSE(\emptyset) & = & \emptyset \\
\\
\delta'(q_2,0) & = & ECLOSE(\delta(q_2,0)) & = & ECLOSE(\emptyset) & = & \emptyset \\
\delta'(q_2,1) & = & ECLOSE(\delta(q_2,1)) & = & ECLOSE(q_4) & = & \{q_4,q_5\} \\
\\
\delta'(q_3,0) & = & ECLOSE(\delta(q_3,0)) & = & ECLOSE(\emptyset) & = & \emptyset \\
\delta'(q_3,1) & = & ECLOSE(\delta(q_3,1)) & = & ECLOSE(\emptyset) & = & \emptyset \\
\\
\delta'(q_4,0) & = & ECLOSE(\delta(q_4,0)) & = & ECLOSE(\emptyset) & = & \emptyset \\
\delta'(q_4,1) & = & ECLOSE(\delta(q_4,1)) & = & ECLOSE(\emptyset) & = & \emptyset \\
\\
\delta'(q_5,0) & = & ECLOSE(\delta(q_5,0)) & = & ECLOSE(\emptyset) & = & \emptyset
\end{array}$$

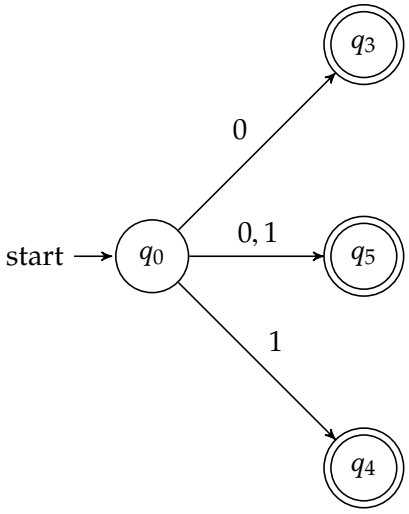
Thus,

δ'	0	1
q_0	$\{q_3, q_5\}$	$\{q_4, q_5\}$
q_1	$\{q_3, q_5\}$	$\emptyset$
q_2	$\emptyset$	$\{q_4, q_5\}$
q_3	$\emptyset$	$\emptyset$
q_4	$\emptyset$	$\emptyset$
q_5	$\emptyset$	$\emptyset$

We have

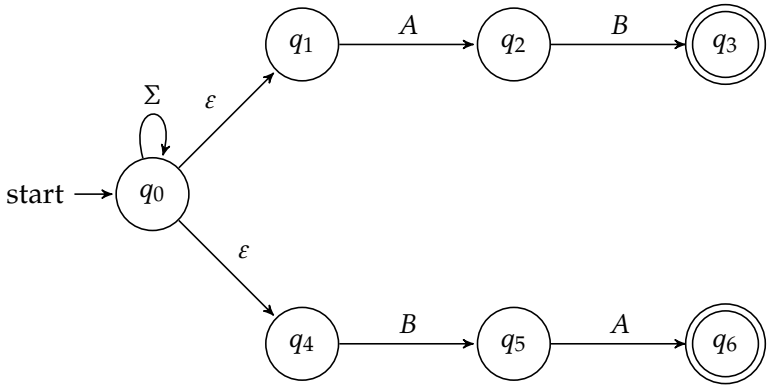
$$\begin{aligned} F' &= \{q \mid ECLOSE(q) \cap \{q_5\} \neq \emptyset\} \\ &= \{q \mid q_5 \in ECLOSE(q)\} \\ &= \{q_3, q_4, q_5\} \end{aligned}$$

The diagram could look like this.



Example 2

Let's look at a simplified automaton similar to that of Figure 2.19. This will accept strings over an alphabet Σ (provided Σ contains at least the symbols A and B) that end either in AB or BA.



Let's do the table for δ.

δ	ε	A	B
q_0	$\{q_1, q_4\}$	$\{q_0\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$	$\emptyset$
q_2	$\emptyset$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\emptyset$	$\emptyset$
q_4	$\emptyset$	$\emptyset$	$\{q_5\}$
q_5	$\emptyset$	$\{q_6\}$	$\emptyset$
q_6	$\emptyset$	$\emptyset$	$\emptyset$

Now let's do the ε-closures.

$$\begin{aligned}
ECLOSE(q_0) &= \{q_0, q_1, q_4\} \\
ECLOSE(q_1) &= \{q_1\} \\
ECLOSE(q_2) &= \{q_2\} \\
ECLOSE(q_3) &= \{q_3\} \\
ECLOSE(q_4) &= \{q_4\} \\
ECLOSE(q_5) &= \{q_5\} \\
ECLOSE(q_6) &= \{q_6\}
\end{aligned}$$

And we have

$$F' = \{q_3, q_6\}$$

Now we build out δ'. We have

$$\begin{array}{rcccccc}
\delta'(q_0, A) & = & ECLOSE(\delta(q_0, A)) & \cup & ECLOSE(\delta(q_1, A)) & \cup & ECLOSE(\delta(q_4, A)) \\
& = & ECLOSE(\{q_0\}) & \cup & ECLOSE(\{q_2\}) & \cup & \emptyset \\
& = & \{q_0, q_1, q_4\} & \cup & \{q_2\} & \cup & \emptyset
\end{array}$$

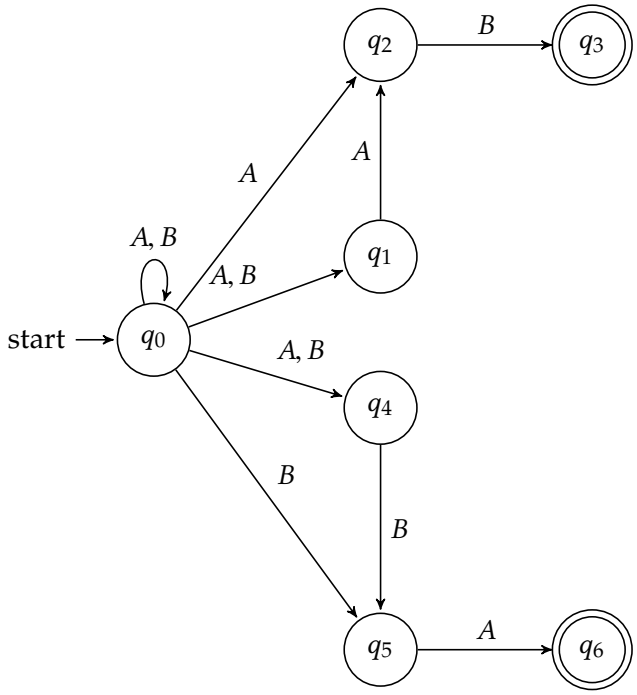
and

$$\begin{array}{rcccccc}
\delta'(q_0, B) & = & ECLOSE(\delta(q_0, B)) & \cup & ECLOSE(\delta(q_1, B)) & \cup & ECLOSE(\delta(q_4, B)) \\
& = & ECLOSE(\{q_0\}) & \cup & ECLOSE(\{q_2\}) & \cup & ECLOSE(\{q_5\}) \\
& = & \{q_0, q_1, q_4\} & \cup & \emptyset & \cup & \{q_5\}
\end{array}$$

The other transitions to nonempty sets are:

$$\begin{aligned}
\delta'(q_1, A) &= ECLOSE(\{q_2\} = \{q_2\} \\
\delta'(q_2, B) &= ECLOSE(\{q_3\} = \{q_3\} \\
\delta'(q_4, B) &= ECLOSE(\{q_5\} = \{q_5\} \\
\delta'(q_5, A) &= ECLOSE(\{q_6\} = \{q_6\}
\end{aligned}$$

From this one can produce a diagram:



Method 2

We construct N' via the algorithm below. In the algorithm, the ε-NFA N' is initially N and is then modified in stages. Each modification leaves the language recognized by N' the same, and hence the output N' at the end is equivalent to N.

1. Set $N' := N$ (that is, all components of N' are equal to those of N).
2. WHILE there exist states $q \in Q - F$ and $r \in F$ such that $r \in \delta(q, \varepsilon)$ DO
   a) $F := F \cup \{q\}$ (that is, add q to F)
3. WHILE there exist $q, r, s \in Q$ and $a \in \Sigma$ such that $r \in \delta(q, \varepsilon)$ and $s \in \delta(r, a)$ but $s \notin \delta(q, a)$ DO
   a) $\delta(q, a) := \delta(q, a) \cup \{s\}$ (that is, add s to $\delta(q, a)$)
4. For all $q \in Q$, set $\delta(q, \varepsilon) := \emptyset$ (that is, remove all ε-transitions from N')
5. Return N' (N' is essentially an NFA)

***Notes on correctness**

- Step 2 can iterate at most $|Q|$ many times, as each iteration increases the size of F by 1.

- No iteration in Step 2 causes N' to reject a string that it previously accepted.
- No iteration in Step 2 adding a state q to F causes N' to accept a string that it previously rejected, because any accepting path ending at q can be extended by a single ε-transition to a state already previously in F.
- Step 3 can iterate at most $|Q|^2|\Sigma|$ times, as it adds a new transition without taking any away.
- No iteration in Step 3 causes N' to reject a string that it previously accepted.
- No iteration in Step 3 adding a state s to $\delta(q, a)$ causes N' to accept a string that it previously rejected, because any accepting path using the new transition $q \overset{a}{\mapsto} s$ can be rerouted to use the previously existing transitions $q \overset{\varepsilon}{\mapsto} r \overset{a}{\mapsto} s$ instead, for some state r.
- Step 4 does not cause N' to accept any string that it previously rejected.
- Step 4 does not cause N' to reject any string that it previously accepted: Let w be any string previously accepted by N' (after Step 3 but before Step 4), and let $w = w_1 \cdots w_n$, where each w_i is in Σ_ε and $p := \langle s_0, \ldots, s_n \rangle$ is a corresponding accepting path, as in Theorem 10.3.2. We can remove all the ε-transitions from p to obtain an accepting path of w with no ε-transitions as follows:
  - If the last symbol $w_n = \varepsilon$, then we can just remove that transition: since $s_n \in F$ and $s_n \in \delta(s_{n-1}, \varepsilon)$, we must also have $s_{n-1} \in F$ by Step 2. Thus $\langle s_0, \ldots, s_{n-1} \rangle$ is an accepting path of $w = w_1 \cdots w_{n-1}$. Repeat this until there are no more ε-transitions at the end of p.
  - After the above, if there are still any ε-transitions in p, then there must be one that is immediately followed by a non-ε-transition. That is, $w_i = \varepsilon$ and $w_{i+1} = a$ for some $i \geq 1$ and $a \in \Sigma$. Then $\langle s_0, \ldots, s_{i-1}, s_{i+1}, \cdots, s_n \rangle$ is an accepting path for $w = w_1 \cdots w_{i-1} w_{i+1} \cdots w_n$, because $s_i \in \delta(s_{i-1}, \varepsilon)$ and $s_{i+1} \in \delta(s_i, a)$ and so by Step 3 we have $s_{i+1} \in \delta(s_{i-1}, a)$. Repeat this until there are no more ε-transitions in p.

Comparing the Two Methods

Methods 1 uses the ε-closure operation to "push forward" non-ε-transitions through ε-transitions. Method 2 "pulls back" accepting states and non-ε-transitions through ε-transitions. Method 1 results in an NFA with the same set of accepting states as the original ε-NFA, but Method 2 may expand the size of the accepting state set considerably.

10.5 Regular expressions

Used to denote (specify) languages. Syntax. Example: Same as the ε-NFA example above.

Regexp for short.
Metasyntax
Uses in Unix/Linux, Perl, text processing, search engines, compilers, etc.
Regular expression syntax and semantics are defined recursively.

Regular expression syntax

Fixing an alphabet Σ, we define a *regular expression (regexp) over* Σ as either

- $\emptyset$
- a (for any symbol $a \in \Sigma$),
- $R + S$ (for any regexps R and S over Σ),
- RS (for any regexps R and S over Σ), or
- R^* (for any regexp R over Σ).

The first two types of regexps are called the *atomic* expressions. (The other types are called nonatomic.) The + operator is called *union*, and the $\cdot$ (juxtaposition) operator is called *concatenation*. These are both binary infix operators and are associative. The unary postfix $*$ operater is called *Kleene closure* or *Kleene star* (named after the mathematician Stephen Kleene, one of the founders of theoretical computer science). We can use parentheses freely to group expressions, and may sometimes drop them assuming the following precedence rules: Kleene star is highest precedence, followed by concatenation, followed by union (lowest precedence).

Regular expression semantics

A regexp R over some alphabet Σ may or may not *match* (or equivalently, be *matched* by) a string $w \in \Sigma^*$ according to the following recursive rules, which mirror the recursive syntax rules for building up regexps given before:

- The regexp $\emptyset$ does not match any string.
- Any regexp a (where $a \in \Sigma$) matches the string a (of length one) and nothing else.
- If R and S are regexps, then $R + S$ matches exactly those strings that either match R or match S (or both).

- If R and S are regexps, then RS matches exactly those strings of the form xy for some string x matching R and some string y matching S.
- If R is a regexp, then R^* matches exactly those strings w of the form $w_1 \cdots w_n$, where n is a natural number and each w_i matches R (that is, w is the concatenation of zero or more strings, each one matching R).

Note that in the last bullet, n could be 0, in which case $w = \varepsilon$. This means that R^* *always* matches ε, regardless of R. In particular, the regexp $\emptyset^*$ matches the empty string ε and nothing else. It is thus natural to use ε as shorthand for the regexp $\emptyset^*$, and pretend that this is another atomic regexp.

Definition 10.5.1. For every regular expression R over Σ, the *language of* R, denoted $L(R)$, is the set of all strings over Σ that are matched by R.

10.6 Buell's additional notes

Example

Let

$$L = \{00, 011\}$$

and

$$M = \{001, 111, 1011\}.$$

Then

$$L + M = \{00, 011, 001, 111, 1011\}$$

and

$$LM = \{00|001, 00|111, 00|1011, 011|001, 011|111, 011|1011\},$$

where the vertical stroke shouldn't be there in fact but is there as a guide to what the concatenations are.

The Kleene closure is probably best looked at as the union of each of the concatenations of lengths 0, 1, 2, ...

$$N = \{0, 1, 01\}$$

$$N^0 = \{\varepsilon\}$$

$$N^1 = N = \{0, 1, 01\}$$

$$N^2 = \{00, 01, 001, 10, 11, 101, 010, 011, 0101\}$$

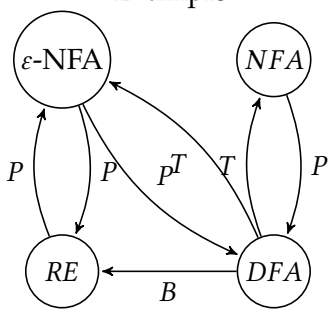
$$N^* = N^0 \cup N^1 \cup N^2 \cup \ldots$$

Note that in doing the N^i we can get repetitions that get discarded because we are taking a set union. The 01 string, for example, is already there in N but is then created again by concatenation in N^2.

Proving all things equivalent

We have so far introduced DFAs, NFAs, ε-NFAs, and now regular expressions (REs). Our goal is to show that all of these are the same thing in that each can be converted into the other and the language accepted/generated by one is accepted/generated by an instance of any of the others.

Example



This represents what we know or can prove, with T meaning "trivial" and P meaning that we have or will prove the connection.

Any DFA is trivially an NFA and thus trivially an ε-NFA.

We have proved that an NFA can be reduced to an equivalent DFA.

The book proves that a DFA can be reduced to an RE.

We will prove that an ε-NFA can be reduced to a RE.

Lecture 11

11.1 Regex Examples

Precedence rules for regular expressions

- Parentheses supersede all precedence
- Kleene star is of the highest precedence
- Concatenation is lower
- union (or +) is lowest

Example: $01^* + 0^*1 = (0(1^*)) + ((0^*)1)$

More examples of regular expressions: more metasyntax. Floating point constants, identifiers, HTML tags, etc.

Lecture 12

12.1 Transforming regular expressions into ε-NFAs

Definition 12.1.1. We will say that an ε-NFA $N = (Q, \Sigma, \delta, q_0, F)$ is *clean* iff

1. it has exactly one final state, and this state is not the start state (that is, $F = \{r\}$ for some state $r \neq q_0$),
2. there are no transitions entering the start state (that is, $q_0 \notin \delta(q, a)$ for any $q \in Q$ and $a \in \Sigma \cup \{\varepsilon\}$), and
3. there are no transitions out of the final state (that is, for $r \in F$ as above, we have $\delta(r, a) = \emptyset$ for all $a \in \Sigma \cup \{\varepsilon\}$).

For every ε-NFA $N = (Q, \Sigma, \delta, q_0, F)$, we can construct an equivalent clean ε-NFA N' as follows:

1. Add a new start state $q_0' \notin Q$ with a single ε-transition from q_0' to q_0 (making q_0 a non-start state of N').
2. Add a new final state $r \notin Q \cup \{q_0'\}$ with ε-transitions from each final state of N to r.
3. Make all the final states of N non-final states of N'.

Every regexp has an equivalent ε-NFA.

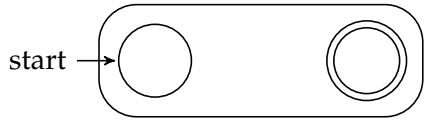
Theorem 12.1.2. *For every regular expression R there exists an ε-NFA N such that $L(N) = L(R)$.*

This theorem is proved by explicit construction, following the recursive definition of regexp syntax, above.

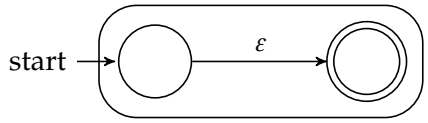
Proof.

Basis: Three base cases, one for each atomic regular expression.

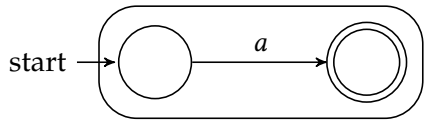
1. Nothing ($\emptyset$)



2. The empty string

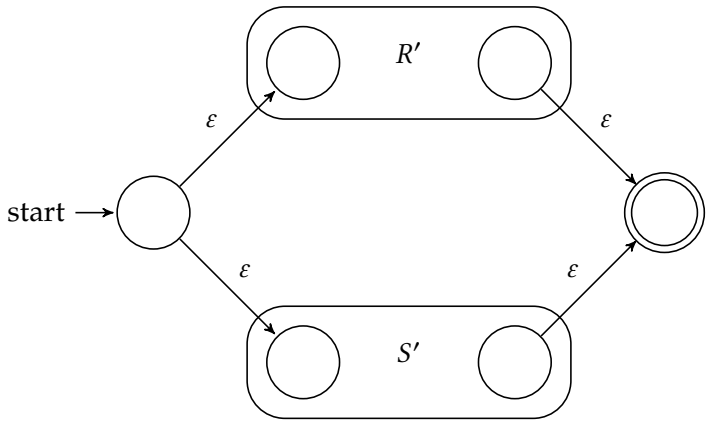


3. A symbol

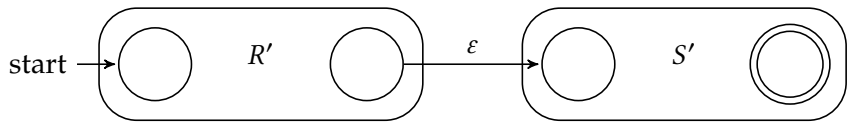


Induction: The inductive hypothesis assumes we have (clean) ε-NFAs for the subexpressions R and S.

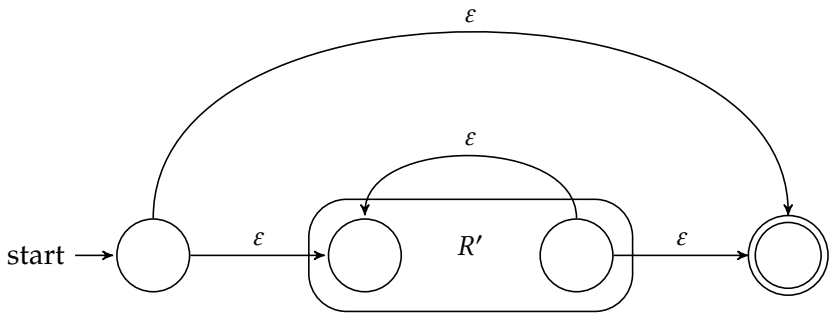
1. $R + S$. Assume we have automata for R and S. Convert these into clean automata R' and S' respectively. Connect the final states in R' and S' to a single final state, and change the final states in R' and S' to non-final states. Finally, connect them with a single start state as shown:



2. RS



3. R^*

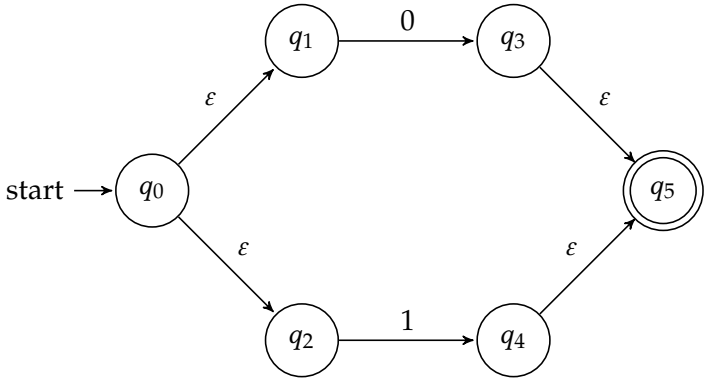


Possible example from the book: $(0 + 1)^*1(0 + 1)$

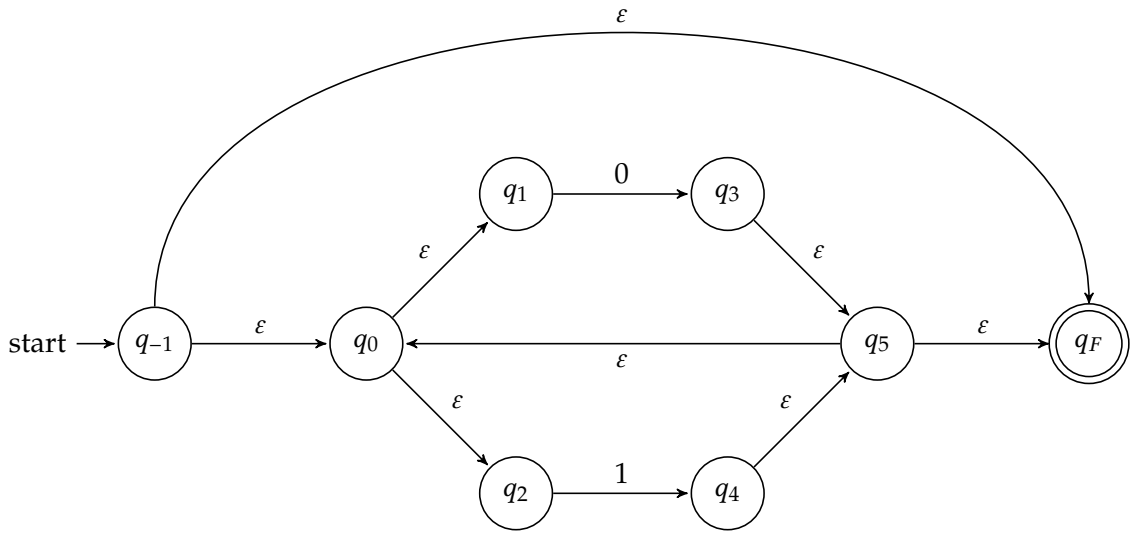
□

Example

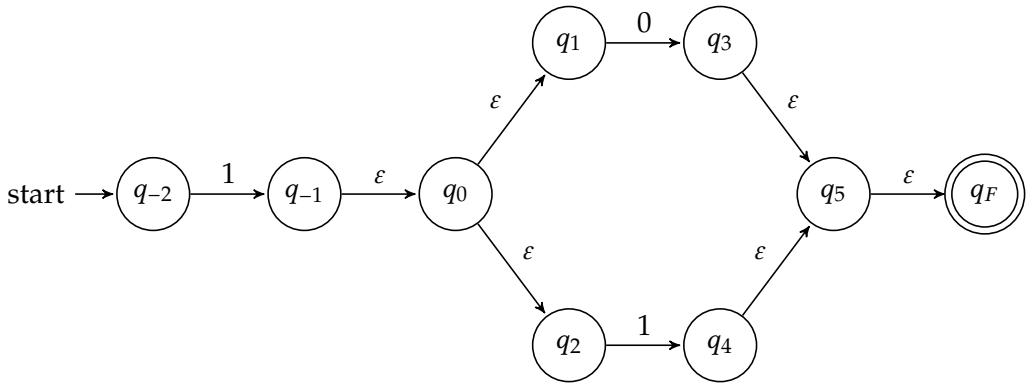
Step 1: $(0 + 1)$



Step 2: $(0 + 1)^*$


ε
start
q_{-1}
ε
q_0
ε
q_1
0
q_3
ε
ε
q_5
ε
q_F
ε
ε
q_2
1
q_4
ε

Step 3: 1(0 + 1)


start
q_{-2}
1
q_{-1}
ε
q_0
ε
q_1
0
q_3
ε
ε
q_5
ε
q_F
ε
ε
q_2
1
q_4

Lecture 13

13.1 Transforming ε-NFAs into regular expressions

Note that the book goes from DFAs to regexps. Starting with ε-NFAs is no harder, so we'll do that.

We will essentially do the state elimination method. We first define an NFA/regexp hybrid:

Definition 13.1.1. Given an alphabet Σ, let REG_Σ be the set of all regular expressions over Σ. A *generalized finite automaton (GFA)* with alphabet Σ is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a nonempty, finite set (the *state set*),
- δ is a function mapping ordered pairs of states to regular expressions over Σ, that is, $\delta : Q \times Q \to \mathrm{REG}_\Sigma$,
- q_0 is an element of Q (the *start state*), and
- F is a subset of Q (the set of *final* or *accepting* states).

Example from the quiz. Give transition diagram and tabular form. Other possible examples: multiples of 3 in binary, binary strings that *don't* contain 010 as a substring (start with a DFA to find 010, complement it, then convert to regular expression).

Define reachability of r from q on w. Define acceptance.

Definition 13.1.2. Let $G = (Q, \Sigma, \delta, q_0, F)$ be a GFA and let $w \in \Sigma^*$ be a string. For any states $q, r \in Q$, we say that r *is reachable from* q *reading* w iff there exist $n \in \mathbb{N}$, states $s_0, s_1, \ldots, s_n \in Q$ and strings $w_1, \ldots, w_n \in \Sigma^*$ such that

1. $w = w_1 \cdots w_n$,
2. $s_0 = q$ and $s_n = r$, and

3. for all $1 \leq i \leq n$, the string w_i matches the regexp $\delta(s_{i-1}, s_i)$ (that is, $w_i \in L(\delta(s_{i-1}, s_i))$).

We say that G *accepts* w iff there exists a final state $f \in F$ that is reachable from the start state q_0 reading w. We let $L(G)$ denote the language of all strings accepted by G.

Given a clean ε-NFA $N = (Q, \Sigma, \delta, q_0, \{f\})$, we first convert it into an equivalent GFA $G_0 = (Q, \Sigma, \delta_0, q_0, \{f\})$ by "consolidating edges" as follows: For every pair of states $q, r \in Q$, let $\{a_1, \ldots, a_k\}$ be the set of all elements a of $\Sigma \cup \{\varepsilon\}$ such that $r \in \delta(q, a)$. Then define

$$\delta_0(q, r) := a_1 + \cdots + a_k \ .$$

(If the set is empty, then set $\delta_0(q, r) := \emptyset$.) Thus several edges of N from q to r turn into one edge labeled with the union of the labels from N. If there are no edges, then we have an edge labeled with $\emptyset$. One can prove by induction on the length of a string that N and G_0 are equivalent, i.e., $L(N) = L(G_0)$.

G_0, is the first of a sequence of equivalent GFAs $G_0, G_1, \ldots, G_\ell$ where we obtain G_{i+1} from G_i by (i) removing and bypassing an intermediate state of G_i (i.e., a state that is not the start state or the final state), then (ii) consolidating edges. Formally, for each $0 \leq i < \ell$, if $G_i = (Q_i, \Sigma, \delta_i, q_0, \{f\})$ has an intermediate state, then we choose such a state $q \in Q_i - \{q_0, f\}$ (it doesn't matter which) and define $G_{i+1} := (Q_{i+1}, \Sigma, \delta_{i+1}, q_0, \{f\})$, where

- $Q_{i+1} = Q_i - \{q\}$ and
- for all states $r, s \in Q_{i+1}$, letting $R := \delta_i(r, q)$, $S := \delta_i(q, q)$, $T := \delta_i(q, s)$, and $U := \delta_i(r, s)$, define

$$\delta_{i+1}(r, s) := U + RS^*T \ .$$

The regexp U allows you to traverse the existing edge in G_i directly from r to s, and RS^*T allows you to move directly from r to s reading a string that would have taken you through q (which is no longer there). The RS^*T results from bypassing q, and the union with U is the edge consolidation.

NOTE: you are allowed to simplify any expressions you build above, i.e., replace them with simpler, equivalent regexps. For example, if there is "no" self-loop at q (that is, $S = \emptyset$), then

$$U + RS^*T = U + R\emptyset^*T = U + R\varepsilon T = U + RT \ ,$$

and so you can set $\delta_{i+1}(r, s) := U + RT$. Similarly, if $U = S = \emptyset$, then you can set $\delta_{i+1}(r, s) := RT$.

Iterate the $G_i \mapsto G_{i+1}$ step above until you get a GFA G_ℓ with no intermediate states. Then since N was clean and we never introduced any edges into q_0 or out of f, the table for G_ℓ looks like

	q_0	f
q_0	$\emptyset$	E
f	$\emptyset$	$\emptyset$

where E is some regexp over Σ [draw the transition diagram]. Clearly, $L(G_\ell) = L(E)$, and so

$$L(N) = L(G_0) = L(G_1) = \cdots = L(G_\ell) = L(E)\,,$$

making E equivalent to N.

Notice how we could choose an intermediate state arbitrarily going from G_i to G_{i+1}. Different choices of intermediate states may lead to syntactically different final regexps, but these regexps are all equivalent to each other, since they are all equivalent to N.

Theorem 13.1.3. *Let L be any language over an alphabet Σ. The following are equivalent:*

1. *L is denoted by some regular expression.*
2. *L is recognized by some GFA.*
3. *L is recognized by some ε-NFA.*
4. *L is recognized by some clean ε-NFA*
5. *L is recognized by some NFA.*
6. *L is recognized by some DFA.*

If any (all) of these cases hold, we say that L is a *regular* language. (There are even more equivalent ways of characterizing regular languages, including grammars.)

We've shown all the nontrivial cases of the theorem. The trivial ones are DFA $\mapsto$ NFA $\mapsto$ ε-NFA, clean ε-NFA $\mapsto$ ε-NFA, and regexp $\mapsto$ GFA. You should teach yourself how these trivial transformations work.

Corollary 13.1.4. *For any two regular expressions R and S over an alphabet Σ, there exist regular expressions over Σ for the complement $\overline{L(R)}$ of $L(R)$ and for the intersection $L(R) \cap L(S)$.*

Proof. For the complement, convert R into an equivalent DFA A (via an ε-NFA and/or an NFA), then build the complementary DFA $\neg A$ (swapping final and non-final states), then convert $\neg A$ back into an equivalent regular expression. For the intersection, convert R and S into equivalent DFAs A and B, respectively, then use the product construction to build the DFA $A \wedge B$ for the intersection, then convert $A \wedge B$ back into an equivalent regular expression. □

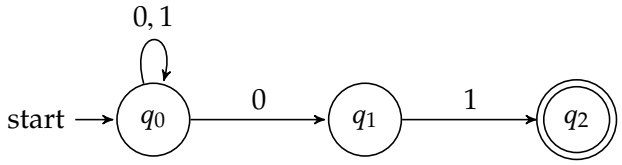
These constructions for the complement and intersection may not be very concise. The regexps you get as a result may be significantly more complicated than the originals.

Example

Let's do an example of transforming an NFA into a RE.

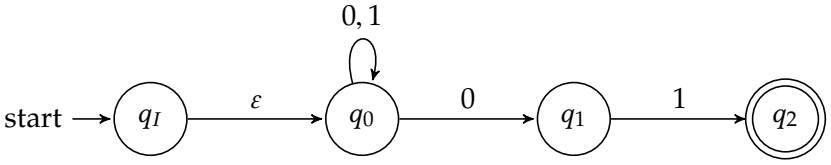
Consider the NFA accepting all strings that end in 01.

An NFA accepting all strings that end in 01:



We can make this into a clean ε-NFA.

A clean ε-NFA accepting all strings that end in 01:



Now we construct G_0. For this we will build the transition function δ_0. (We will leave blank the boxes in all the following tables if what would go there is the empty set $\emptyset$, so as not to clutter up the presentation with things that don't matter.) We have in the sets that prepare us for defining δ_0 the following:

	I	0	1	2
I		ε		
0		$\{0, 1\}$	$\{0\}$	
1				$\{1\}$
2				

and thus

δ_0	I	0	1	2
I		ε		
0		0 + 1	0	
1				1
2				

We now need to remove states q_0 and q_1. Let's do the second one first.
We get $Q_1 = \{I, 0, 2\}$ and let's table the U, R, S, and T, and the final result:

r, s	$r \to s$ U	$r \to q_1$ R	$q_1 \to q_1$ S	$q_1 \to s$ T	Result
I,I I,0 I,2	ε			 1	ε
0,I 0,0 0,2	0 + 1	0 0 0		 1	 0 + 1 01
2,I 2,0 2,2				 1	

This gives a transition table

δ_1	I	0	2
I		ε	
0		0 + 1	01
2			

Now we remove q_0.

r, s	$r \to s$ U	$r \to q_0$ R	$q_0 \to q_0$ S	$q_0 \to s$ T	Result
I,I I,2		ε ε	0 + 1 0 + 1	 01	 $(0 + 1)^*01$
2,I 2,2			0 + 1 0 + 1	 01	

This gives a final transition table

δ_2	I	2
I		$(0 + 1)^*01$
2		

and this is the regular expression we actually might expect.

Lecture 14

14.1 Grammars, Type 3 grammars, and regular languages

We have now dealt with DFAs, NFAs, ε-NFAs, and regular expressions, and we have shown that they were all really the same thing, in that the languages accepted by one of these will be accepted by some instance of every other one of these.

It's time to finish that argument with one more variation on the same theme.

Definition 14.1.1. A *g*rammar G is a quadruple

$$G = (V_N, V_T, P, S)$$

where

- V_N is a finite set of *variable* symbols
- V_T is a finite set of *terminal* symbols
- P is a finite set of *productions*
- S is a *start symbol* and is an element of V_N

We assume that $V_N \cap V_T = \emptyset$.
We write $V_N \cup V_T = V$.
The productions are expressions of the form

$$\alpha \to \beta$$

where α is a string in V^+ and β is a string in V^*.

If we have a production

$$\alpha \to \beta$$

and we have strings γ and δ, then we can write

$$\gamma\alpha\delta \overset{G}{\Longrightarrow} \gamma\beta\delta$$

to indicate that $\gamma\beta\delta$ can be directly derived from $\gamma\alpha\delta$ by the application of a single production of G.

We can (in the usual and now obvious way) extend this to a notation

$$\alpha_1 \overset{G^*}{\Longrightarrow} \alpha_n$$

if we have a sequence of direct derivations

$$\alpha_1 \overset{G}{\Longrightarrow} \alpha_2 \overset{G}{\Longrightarrow} \ldots \overset{G}{\Longrightarrow} \alpha_n$$

Definition 14.1.2. The *language generated by G*, written $L(G)$, is

$$L(G) = \{w | w \in V_T^* \quad and \quad S \overset{G^*}{\Longrightarrow} w\}$$

Example

Consider the grammar with

1. $V_N = \{S\}$
2. $V_T = \{0, 1\}$
3. $P = \{S \to 0S1, S \to 01\}$

We have only one variable, S. We have only two terminals, 0 and 1.
We have only the two productions.
This means that the only things we can generate are

$$S \to 0S1 \to 00S11 \to \ldots \to 0^{n-1}S1^{n-1} \to 0^n1^n$$

by applying the first production $n-1$ times and then the second production once. So that's what $L(G)$ is:

$$L(G) = \{0^n1^n\}$$

Example

Consider the grammar with

1. $V_N = \{S, B, C\}$
2. $V_T = \{a, b, c\}$
3. Productions
   a) $S \to aSBC$
   b) $S \to aBC$

c) $CB \to BC$

d) $aB \to ab$

e) $bB \to bb$

f) $bC \to bc$

g) $cC \to cc$

It is not that hard to see that for $n \geq 1$, any string $a^n b^n c^n$ is in the language generated by this grammar.

It's somewhat harder (but it can be done) to show that these are all the strings in the language.

Definition 14.1.3. A grammar as defined above is a *Type 0 grammar*.

Definition 14.1.4. Given a grammar G, if it is the case that for every production $\alpha \to \beta$ in P we have $|\alpha| \leq |\beta|$, then we say that the grammar is a *Type 1* or *context sensitive grammar*.

Definition 14.1.5. Given a grammar G, if it is the case that for every production $\alpha \to \beta$ in P we have that

- α is a single variable;
- β is any string other than ε.

then we say that the grammar is a *Type 2* or *context free grammar*.

Definition 14.1.6. Given a grammar G, if it is the case that every production in P is of the form

- $A \to aB$ for a a terminal and variables A, B
- or $A \to a$ for a a terminal and A a variable

then we say that the grammar is a *Type 3* or *regular grammar*.

Before we go to the next major theorem, we need to fudge a little and extend our definition, but in order to show that we haven't really changed the languages that are generated, we will need the following.

Theorem 14.1.7. *Let $G = (V_N, V_T, P, S)$ be a Type 1, 2, or 3 grammar. Then there exists a Type 1, 2, or 3 grammar (as appropriate) G_1, for which $L(G) = L(G_1)$, and such that the start symbol of G_1 does not appear on the right hand side of any production of G_1.*

Proof. We choose a new symbol S_1 not in V_N or in V_T. We let $G = (V_N \cup \{S_1\}, V_T, P_1, S_1)$. And we let P_1 be the set of all productions in P plus productions $S_1 \to \alpha$ for any production $S \to \alpha$ that is in P.

We claim that $L(G) = L(G_1)$.

Suppose that we have a derivation in G

$$S \stackrel{G^*}{\Longrightarrow} w$$

and wlog assume the first production used in this derivation is

$$S \to \alpha$$

Then we really have

$$S \to \alpha \stackrel{G^*}{\Longrightarrow} w$$

By the construction of G_1, we have a production

$$S_1 \to \alpha$$

in G_1, which means that we have

$$S_1 \stackrel{G_1}{\Longrightarrow} \alpha$$

And since all the productions from G are also in G_1, we have a derivation

$$S_1 \stackrel{G_1}{\Longrightarrow} \alpha \stackrel{G_1^*}{\Longrightarrow} w$$

so $w \in L(G_1)$.

This shows that $L(G_1) \subseteq L(G)$.

To go the other way around, we assume we have some $w \in L(G_1)$. Then we have a derivation

$$S_1 \stackrel{G_1^*}{\Longrightarrow} w$$

There has to be a first production

$$S_1 \to \alpha$$

used in this derivation, so we have

$$S_1 \stackrel{G_1}{\Longrightarrow} \alpha \stackrel{G_1^*}{\Longrightarrow} w$$

We just argue in the other direction. That first production in G_1 comes from a production

$$S \to \alpha$$

in G. And since these are the ONLY productions in G_1 that involve S_1, the second part of this derivation

$$\alpha \overset{G_1^*}{\Longrightarrow} w$$

cannot involve any productions that use the symbol S_1, which means that all the productions correspond to productions in G and that we have

$$\alpha \overset{G^*}{\Longrightarrow} w$$

We piece these two together to get

$$S \overset{G}{\Longrightarrow} \alpha \overset{G^*}{\Longrightarrow} w$$

and thus that $w \in L(G)$.

So $L(G) \subseteq L(G_1)$.

The containment now works both ways, so the languages are the same set.

The Type 1-ness, Type 2-ness, and Type 3-ness, as appropriate, is obviously maintained. □

Now we can expand the definition of Type 1, 2, and 3 grammars to permit productions of the form

$$S \to \varepsilon$$

PROVIDED that S does not appear on the RHS of any production.

If we do this, then we can argue that if a language L is generated by a Type 1, 2, or 3 grammar, then there is a grammar of the appropriate type that generates $L \cup \{\varepsilon\}$ and there is a grammar of the appropriate type that generates $L - \{\varepsilon\}$, and this is found by applying the previous theorem if necessary and then either adding or deleting a production $S \to \varepsilon$.

The use of the term "regular" should be a major warning indicator that the next theorem is coming ...

Theorem 14.1.8. *1. Let $G = (V_N, V_T, P, S)$ be a regular grammar. Then there exists an NFA M such that $L(G) = L(M)$.*

2. Let $D = (Q, \Sigma, q_0,)$ be a DFA. Then there exists a regular grammar G such that $L(D) = L(G)$.

Proof. We're going to do part (1) with an NFA, but we'll do part (2) with a DFA, and since we know the languages accepted are the same, this will finish off a proof that

$$DFA \iff NFA \iff \varepsilon - NFA \iff RegularExpression \iff RegularGrammar$$

The intuitive idea behind the proof should be fairly obvious: A finite automaton consumes symbols and makes state transitions. A formal regular grammar emits symbols and makes variable symbol transitions. And if we start with a regular grammar (as opposed to a grammar that is only Type 0, 1, or 2, with fewer conditions), then we can pair productions in the grammar with transitions in the automaton.

Proof of Part 1

Given a regular grammar G, we will construct an NFA M.

The states of M are the variables V_N together with an additional state A. The initial state of M is S. If the set of productions P contains the production $S \rightarrow \varepsilon$, then the final states of M are $\{S, A\}$; otherwise the final state of M is $\{A\}$;

We note that S will not appear on the right hand side of a production if the production $S \rightarrow \varepsilon$ is in P. (If there is such a production, then we can create another variable S_1 and replace productions with S on the RHS with productions that go to S_1 instead.)

Given a state B and any terminal symbol $a \in V_T$, we include in the set of states $\delta(B, a)$ all the variable symbols C for which we have productions $B \rightarrow aC$.

If there is a production $B \rightarrow a$, then we also include the final state A in the set of states for $\delta(B, a)$

We have $\delta(A, a) = \emptyset$ for all terminal symbols a.

So let's assume that G generates a string $w = a_1 ... a_n$. Then there is a sequence of variables A_i such that we have the derivation

$$S \implies a_1 A_1 \implies a_1 a_2 A_2 \implies a_1 ... a_n$$

By our definition of δ, we have that $A_1 \in \delta(S, a_1)$, $A_2 \in \delta(A_1, a_2)$, and so forth, and the last step takes us to the final state A.

So w is accepted by the automaton M.

The empty string is a special case, but we have that covered as well.

In the other direction, if we have a string w in the language $L(M)$, then we have a sequence of states S, A_1, A_2, ... A through which M travels as it reads w. But each of these transitions is essentially exactly a production in G, so there is a derivation in G that corresponds exactly to the state transitions of M.

And we have the empty string covered as well.

Proof of Part 2

This is actually somewhat easier. We start with a DFA $D = (Q, \Sigma, \delta, q_0, F)$ and we create a grammar G. The grammar G has the states Q as the variables V_N, the alphabet Σ as the terminal symbols V_T, the start state q_0 as the start symbol S, and we only need to specify the productions P.

We have

1. a production $B \to aC$ if we have the transition $\delta(B, a) = C$
2. a production $B \to a$ if we have the transition $\delta(B, a) = C$ for C a final state.

The proof that $w \in L(D)$ if and only if $w \in L(G)$ is almost identical to the proof of part 1.o

Now, we have $\varepsilon \in L(D)$ if and only if q_0 is a final state. If not, then ε is in neither language, and we are done.

If q_0 is a final state, and thus $\varepsilon \in L(D)$, then we need to build a modified grammar to account for ε.

We do this by adding the production $S \to \varepsilon$.

□

Lecture 15

15.1 Proving languages not regular

Definition 15.1.1. We say that a language L is *pumpable* iff

there exists an integer $p > 0$ such that
for all strings $s \in L$ with $|s| \geq p$,
there exist strings x, y, z with $xyz = s$ and $|xy| \leq p$ and $|y| > 0$ such that
for every integer $i \geq 0$,
$xy^i z \in L$.

Lemma 15.1.2 (Pumping Lemma for Regular Languages)**.** *For any language L, if L is regular, then L is pumpable.*

Proof.

Suppose L is a (nontrivial, infinite) regular language. Then by definition $L = L(A)$ for some DFA $A = \langle Q, \Sigma, \delta, q_0, F \rangle$. Suppose $|Q| = p$, in other words A has p states.

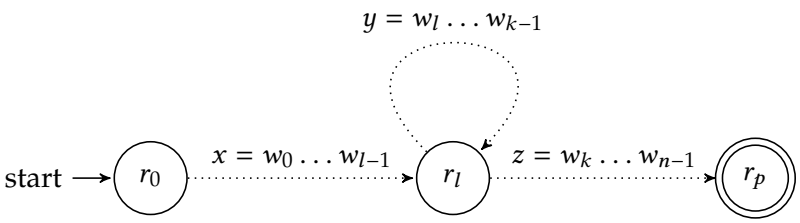
Take $w \in L(A)$ such that $|w| > p$. In other words, $w = w_0, \ldots, w_{n-1}$ where $n > p, w_i \in \Sigma$. Define

$$r_i \coloneqq \hat{\delta}(q_0, w_0 \ldots w_{i-1})$$

I.e. r_i is the state reached after reading in the first i symbols of w

Now consider the sets Q and $\{r_i \mid 0 \leq i \leq n\}$. In particular, $|Q| = p < n = |\{r_i\}|$. By the pigeonhole principle, any function mapping the r_i's to states in Q can't be injective. So we can pick two integers l and k s.t. $0 \leq l < k \leq p$ and $r_l = r_k$. Now we can break up the string like so:

1. $x = w_0 \ldots w_{l-1}$
2. $y = w_l \ldots w_{k-1}$
3. $z = w_k \ldots w_{n-1}$



□

Here is the contrapositive, which is an equivalent statement:

Lemma 15.1.3 (Pumping Lemma (contrapositive form))**.** *For any language L, if L is not pumpable, then L is not regular.*

We will use the contrapositive form to prove that certain languages are not regular by showing that they are not pumpable. By definition, a language L is *not* pumpable iff

for any integer $p > 0$,
there exists a string $s \in L$ with $|s| \geq p$ such that
for all strings x, y, z with $xyz = s$ and $|xy| \leq p$ and $|y| > 0$,
there exists an integer $i \geq 0$ such that
$xy^i z \notin L$.

The value of p above is called the *pumping length*.

Lecture 16

16.1 Template for Pumping Lemma Proofs

Here is a template for a proof that a language L is not pumpable (and hence not regular). Parts in brackets are to be filled in with specifics for any given proof.

> Given any $p > 0$,
> let $s =$ [describe some string in L with length $\geq p$].
> Now for any x, y, z with $xyz = s$ and $|xy| \leq p$ and $|y| > 0$,
> let $i =$ [give some integer ≥ 0 which might depend on p, s, x, y, and z].
> Then we have $xy^iz \notin L$ because [give some reason/explanation].

Note:

- We cannot choose p. The value of p could be any positive integer, and we have to deal with whatever value of p is given to us.
- We *can* and *do* choose the string s, which may differ depending on the given value of p (so the description of s uses p somehow). We must choose s to be in L and with length $\geq p$, however.
- We cannot choose x, y, or z. These are given to us and could be any strings, except we know that they must satisfy $xyz = s$, $|xy| \leq p$, and $|y| > 0$.
- We get to choose $i \geq 0$ based on all the previous values.
- The text is enamored of showing that when we write $w = xyz \in L$ and apply the pumping lemma to conclude that $xy^iz \in L$ for all i, we can conclude that $xy^0z = xz \in L$ and get a contradiction. We can equally well use $xyyz \in L$, $xyyyz \in L$, $xyyyyz \in L$, and so forth, to get the contradiction.

Example 16.1.1. Let $L := \{0^n1^n \mid n \geq 0\}$. We show that L is not pumpable using the template:

Given any $p > 0$,
let $s := 0^p 1^p$. (Clearly, $s \in L$ and $|s| \geq p$.)
Now for any x, y, z with $xyz = s$ and $|xy| \leq p$ and $|y| > 0$,
let $i := 0$.
Then we have $xy^i z = xy^0 z = xz \notin L$, which can be seen as follows: Since $|xy| \leq p$ it must be that x and y consist entirely of 0's, and so $y = 0^m$ for some m, and we further have $m \geq 1$ because $|y| > 0$. But then $xz = 0^{p-m}1^p$, and so because $p - m < p$, the string xz is *not* of the form $0^n 1^n$ for any n, and thus $xz \notin L$.

The next three examples are minor variations of each other.

Example 16.1.2. Let

$$L := \left\{ w \in \{0,1\}^* \mid w \text{ has the same number of 0's as 1's} \right\}.$$

We show that L is not pumpable using the template:

Given any $p > 0$,
let $s := 0^p 1^p$. (Clearly, $s \in L$ and $|s| \geq p$.)
Now for any x, y, z with $xyz = s$ and $|xy| \leq p$ and $|y| > 0$,
let $i := 0$.
Then we have $xy^i z = xy^0 z = xz \notin L$, which can be seen as follows: Since $|xy| \leq p$ it must be that x and y consist entirely of 0's, and so $y = 0^m$ for some m, and we further have $m \geq 1$ because $|y| \geq 1$. But then $xz = 0^{p-m}1^p$, and so because $p - m \neq p$, the string xz does *not* have the same number of 0's and 1's, and thus $xz \notin L$. [Notice that picking any $i \neq 1$ will work.]

Example 16.1.3. Let

$$L := \left\{ w \in \{0,1\}^* \mid w \text{ has more 0's than 1's} \right\}.$$

We show that L is not pumpable using the template:

Given any $p > 0$,
let $s := 0^p 1^{p-1}$. (Clearly, $s \in L$ and $|s| \geq p$.)
Now for any x, y, z with $xyz = s$ and $|xy| \leq p$ and $|y| > 0$,
let $i := 0$.
Then we have $xy^i z = xy^0 z = xz \notin L$, which can be seen as follows: Since $|xy| \leq p$ it must be that x and y consist entirely of 0's, and so $y = 0^m$ for some m, and we further have $m \geq 1$ because $|y| > 0$. But then $xz = 0^{p-m}1^{p-1}$, and so because $p - m \leq p - 1$, the string xz does *not* have more 0's than 1's, and thus $xz \notin L$. [Notice that $i := 0$ is the only choice that works.]

Example 16.1.4. Let

$$L := \left\{ w \in \{0,1\}^* \mid w \text{ has fewer 0's than 1's} \right\} .$$

We show that L is not pumpable using the template:

> Given any $p > 0$,
> let $s := 0^p 1^{p+1}$. (Clearly, $s \in L$ and $|s| \geq p$.)
> Now for any x, y, z with $xyz = s$ and $|xy| \leq p$ and $|y| > 0$,
> let $i := 2$.
> Then we have $xy^i z = xy^2 z = xyyz \notin L$, which can be seen as follows: Since $|xy| \leq p$ it must be that x and y consist entirely of 0's, and so $y = 0^m$ for some m, and we further have $m \geq 1$ because $|y| > 0$. But then $xyyz = 0^{p+m} 1^{p+1}$, and so because $p + m \geq p + 1$, the string $xyyz$ does *not* have fewer 0's than 1's, and thus $xyyz \notin L$. [Notice that picking any $i \geq 2$ will work.]

Example 16.1.5. Let

$$L := \left\{ w \in \{0,1\}^* \mid w = 1^P \text{ and } P \text{ is a prime number} \right\} .$$

We show that L is not pumpable using the template:

> Given any $p > 0$,
> then for any w and x, y, z with $w = xyz$, $|w| = P$ a prime, and $|xy| \leq p$ and $|y| > 0$,
> let $|x| = k$ and $|y| = \ell$.
> We know that $\ell > 0$ and that $xz, xyz, xyyz, \ldots$, are all in L.
> If ℓ is an odd integer, then some string among $xy^0 z, xy^2 z, xy^4 z, \ldots$ is of even length greater than or equal to four, and thus not of prime length.
> If ℓ is an even integer, then some string among $xz, xyz, xyyz, \ldots$, of lengths $P - 2n, P, P + 2n, P + 4n, \ldots$ is of length greater than 3 and divisible by 3 and thus not prime.

We can view use of the pumping lemma as a game with four turns, based on a language L:

1. Your opponent chooses any positive integer p.
2. You respond with some string $s \in L$ such that $|s| \geq p$.
3. Your opponent chooses three strings x, y, and z satisfying
   a) $xyz = s$,

b) $|xy| \leq p$, and

c) $|y| > 0$.

4. You conclude the game by choosing a natural number i.

You win the game if $xy^iz \notin L$. Otherwise, your opponent wins. Proving a language L is not pumpable amounts to describing a winning strategy for yourself in this game.

Lecture 17

17.1 Closure properties of regular languages.

We show that several constructions on regular languages yield regular languages.

We've proved this already:

Proposition 17.1.1. *If L and M are regular languages, then so is $L \cup M$.*

Proof. If r is a regular expression for L and s is a regular expression for M, then $r + s$ is a regular expression for $L \cup M$, by definition of the "+" operator. □

The same idea proves

Proposition 17.1.2. *If L and M are regular languages, then so are LM and L^*.*

We've proved this, too:

Proposition 17.1.3. *If L is regular, then $\overline{L}$ is regular.*

Proof. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA for L. Let $B = (Q, \Sigma, \delta, q_0, Q - F)$. Then we can see that B is a DFA for $\overline{L}$ as follows: for every string $w \in \Sigma^*$,

$$\begin{aligned} w \in \overline{L} &\iff A \text{ rejects } w \\ &\iff \hat{\delta}(q_0, w) \notin F \\ &\iff \hat{\delta}(q_0, w) \in Q - F \\ &\iff B \text{ accepts } w. \end{aligned}$$

Thus $L(B) = \overline{L}$, and so $\overline{L}$ is regular. □

In the proofs of Propositions 17.1.1 and 17.1.2, we transformed regular expressions to show that the new language is regular. In the second proof, we transformed a DFA. Often, one or the other way works best. One may also be convenient to transform an NFA or ε-NFA.

To illustrate these techniques, we'll prove the next closure property in two ways—transforming a regular expression *and* transforming an ε-NFA. Both techniques are useful.

Recall that w^R is the reversal of string w. If L is a language, we define

$$L^R := \{w^R \mid w \text{ is in } L\}.$$

So L^R just contains the reversals of strings in L. For example, if $L = \{aab, bca, aaa, \varepsilon\}$, then $L^R = \{baa, acb, aaa, \varepsilon\}$. Notice that $(w^R)^R = w$ for any string w, and thus $(L^R)^R = L$ for any language L.

Next, we show closure under intersection. We've already seen this explicitly with the product construction on DFAs. There is another, much easier proof, as it turns out.

Proposition 17.1.4. *If L and M are regular, then so is $L \cap M$.*

Proof. Let L and M be regular. By one of De Morgan's laws,

$$L \cap M = \overline{\overline{L} \cup \overline{M}}.$$

Since regularity is preserved under complements and unions, the right-hand side is regular, and so $L \cap M$ is regular. □

Corollary 17.1.5. *If L and M are regular (and over the same alphabet), then $L - M$ is regular.*

Proof. Notice that $L - M = L \cap \overline{M}$, and the right-hand side is regular because complementation and intersection both preserve regularity. □

Lecture 18

Proposition 18.0.1. *If L is regular, then so is L^R.*

For our first proof of Proposition 18.0.1, we give an explicit way to transform any regular expression r for a language L into a new regular expression r^R for the reversal language L^R. To justify the transformation we use the following lemma:

Lemma 18.0.2. *Fix an alphabet Σ.*

1. $\emptyset^R = \emptyset$.
2. $\{\varepsilon\}^R = \{\varepsilon\}$,
3. *For any symbol $a \in \Sigma$,* $\{a\}^R = \left\{a^R\right\} = \{a\}$.

For any two languages L and M over Σ,

3. $(L \cup M)^R = L^R \cup M^R$,
4. $(LM)^R = M^R L^R$,
5. $(L^*)^R = (L^R)^*$.

Proof. Facts (1) and (2) are obvious. In particular, any string of length 1 is its own reversal.

Facts (3)–(5) maybe less so. Let's verify (3): let w be any string.

$$\begin{aligned} w \in (L \cup M)^R &\iff w^R \in L \cup M \\ &\iff w^R \in L \text{ or } w^R \in M \\ &\iff w \in L^R \text{ or } w \in M^R \\ &\iff w \in L^R \cup M^R. \end{aligned}$$

Thus (3) is true.

For (4), let w be any string. First, suppose $w \in (LM)^R$. Then $w^R \in LM$, and thus there exist strings $x \in L$ and $y \in M$ such that $w^R = xy$. But notice that $(xy)^R = y^R x^R$. So

$$w = (w^R)^R = (xy)^R = y^R x^R \in M^R L^R.$$

Conversely, suppose $w \in M^R L^R$. Then $w = uv$ for some $u \in M^R$ and $v \in L^R$. Thus $u^R \in M$ and $v^R \in L$, which means that $v^R u^R \in LM$, and so

$$w^R = (uv)^R = v^R u^R \in LM,$$

which implies that $w \in (LM)^R$.

Finally (5): let w be any string in $(L^*)^R$. Then w^R is in L^*, and so $w^R = x_1 \cdots x_k$ for some $k \geq 0$ and strings $x_i \in L$ for all $1 \leq i \leq k$. Then,

$$w = (w^R)^R = (x_1 \cdots x_k)^R = x_k^R \cdots x_1^R \in (L^R)^*,$$

because each x_i^R is in L^R. Conversely, if w is in $(L^R)^*$, then $w = z_1 \cdots z_k$ for some k and each $z_i \in L^R$, which means $z_i^R \in L$. Then

$$w^R = (z_1 \cdots z_k)^R = z_k^R \cdots z_1^R \in L^*,$$

and so $w \in (L^*)^R$. □

We'll now use this lemma to recursively transform any regular expression r into r^R.

First proof of Proposition 18.0.1. We transform r into r^R by the following rules, which are justified by Facts (1)–(5) of Lemma 18.0.2 above.

1. If $r = \emptyset$, then define $r^R = \emptyset^R := \emptyset$.
2. If $r = \varepsilon$, then define $r^R := \varepsilon$.
3. If $r = a$ for some $a \in \Sigma$, then define $r^R = a^R := a$.
4. If $r = s + t$ for some regular expressions s and t, then define $r^R = (s + t)^R := s^R + t^R$ (use recursion to find s^R and t^R).
5. If $r = st$ for some regular expressions s and t, then define $r^R = (st)^R := t^R s^R$ (note the reversal).
6. If $r := s^*$ for some regular expression s, then define $r^R = (s^*)^R := (s^R)^*$.

By facts (1)–(5) above, this procedure correctly produces an regular expression for L^R given one for L. More formally, we have the following claim, which suffices to prove the proposition:

Claim 18.0.3. $L(r^R) = L(r)^R$ for any regexp r over Σ.

Proof of the claim. The proof is by induction on the length of r. We have two base cases and three inductive cases, and these mirror the five rules for building regexps as well as the five facts of Lemma 18.0.2:

Case 1: $r = \emptyset$. We have

$$L(\emptyset^R) = L(\emptyset) = \emptyset = \emptyset^R = L(\emptyset)^R \ .$$

(The first equality is by definition, i.e., $\emptyset^R := \emptyset$; the second follows from how we defined regexp semantics (particularly, the regexp $\emptyset$ does not match any strings); the third is Fact (1) of Lemma 18.0.2; the last is again by regexp semantics.)

Case 2: $r = a$ for some $a \in \Sigma$. We have

$$\begin{aligned}
L(a^R) &= L(a) && \text{(definition of } a^R\text{)} \\
&= \{a\} && \text{(regexp semantics)} \\
&= \left\{a^R\right\} && \text{(Fact (2) of Lemma 18.0.2)} \\
&= \{a\}^R && \text{(definition of the reversal of a language)} \\
&= (a)^R && \text{(regexp semantics again)}
\end{aligned}$$

Case 3: $r = s + t$ for regexps s, t. Since s and t are both shorter than r, we can assume by the inductive hypothesis that the claim holds for s and t, that is, $L(s^R) = L(s)^R$ and $L(t^R) = L(t)^R$. Then

$$\begin{aligned}
L((s+t)^R) &= L(s^R + t^R) && \text{(definition of } (s+t)^R\text{)} \\
&= L(s^R) \cup L(t^R) && \text{(regexp semantics)} \\
&= L(s)^R \cup L(t)^R && \text{(inductive hypothesis)} \\
&= (L(s) \cup L(t))^R && \text{(Fact (3) of Lemma 18.0.2)} \\
&= L(s+t)^R && \text{(regexp semantics)}
\end{aligned}$$

Case 4: $r = st$ for regexps s, t. The inductive hypothesis applies to s and t, so we have

$$\begin{aligned}
L((st)^R) &= L(t^R s^R) && \text{(definition of } (st)^R\text{)} \\
&= L(t^R)L(s^R) && \text{(regexp semantics)} \\
&= L(t)^R L(s)^R && \text{(inductive hypothesis)} \\
&= (L(s)L(t))^R && \text{(Fact (4) of Lemma 18.0.2)} \\
&= L(st)^R && \text{(regexp semantics)}
\end{aligned}$$

Case 5: $r = s^*$ for regexp s. The inductive hypothesis applies to s, so we have

$$\begin{aligned} L((s^*)^R) &= L((s^R)^*) && \text{(definition of } (s^*)^R) \\ &= L(s^R)^* && \text{(regexp semantics)} \\ &= (L(s)^R)^* && \text{(inductive hypothesis)} \\ &= (L(s)^*)^R && \text{(Fact (5) of Lemma 18.0.2)} \\ &= L(s^*)^R && \text{(regexp semantics)} \end{aligned}$$

This proves the claim. □

Now Proposition 18.0.1 follows immediately from the claim: If L is regular, then $L = L(r)$ for some regular expression r. But then $L^R = L(r)^R = L(r^R)$ by the claim, and so L^R is regular, being denoted by the regexp r^R. This proves Proposition 18.0.1. □

The key to the whole proof above is the inductive definition of r^R given at the beginning. The rest of the proof is just verifying that the transformation works as advertised.

For example, let's use the rules to find r^R where $r = \mathbf{b}(\mathbf{a} + \mathbf{b}\mathbf{c}^*)^*$.

$$\begin{aligned} (\mathbf{b}(\mathbf{a} + \mathbf{b}\mathbf{c}^*)^*)^R &= ((\mathbf{a} + \mathbf{b}\mathbf{c}^*)^*)^R\mathbf{b}^R \\ &= ((\mathbf{a} + \mathbf{b}\mathbf{c}^*)^R)^*\mathbf{b} \\ &= (\mathbf{a}^R + (\mathbf{b}\mathbf{c}^*)^R)^*\mathbf{b} \\ &= (\mathbf{a} + (\mathbf{c}^*)^R\mathbf{b}^R)^*\mathbf{b} \\ &= (\mathbf{a} + (\mathbf{c}^R)^*\mathbf{b})^*\mathbf{b} \\ &= (\mathbf{a} + \mathbf{c}^*\mathbf{b})^*\mathbf{b}. \end{aligned}$$

The only real change in going from r to r^R is that concatenations are reversed. So you can write down r^R quickly by just reversing all the concatenations in r and leaving the other operations intact.

Instead of transforming regular expressions, another way to prove Proposition 18.0.1 is to transform an ε-NFA.

Second proof of Proposition 18.0.1. Let A be an ε-NFA recognizing L. We can assume that A has only one final state (say, by making A clean). Let B be the ε-NFA constructed from A as follows:

- Make the state set and alphabet of B the same as that of A.
- Make the start state of B to be the final state of A.
- Make the only final state of B to be the start state of A.

- Reverse the arrows on all the transitions of A to get the transitions of B, i.e., if $q \xrightarrow{a} r$ is a transition from state q to state r reading symbol a (or ε), then make $q \xleftarrow{a} r$ the corresponding transition of B.

Now it is clear that A accepts a string w just when there is a path from A's start state to its final state reading w. But this is true if and only if there is a path from B's start state (A's final state) to B's final state (A's start state) reading w^R. This is just the path in A followed in reverse. So A accepts w iff B accepts w^R. Hence B recognizes L^R, and so L^R is regular. □

Just for brevity's sake, we left out formal details in the second proof. A good exercise for you is to supply those formal details, i.e., define B formally as a 5-tuple from a given 5-tuple for A, then prove formally by induction on the length of a string w that B accepts w if and only if A accepts w^R, hence concluding that $L(B) = L(A)^R$.

Lecture 19

19.1 String Homomorphisms

Next we consider images and inverse images under *string homomorphisms*

Definition 19.1.1. Let Σ and T be alphabets. A *string homomorphism* (or just a *homomorphism*) from Σ^* to T^* is a function h that takes any string $w \in \Sigma^*$ and produces a string in T^* (that is, if $w \in \Sigma^*$, then $h(w) \in T^*$) such that h preserves concatenation, i.e., if w and x are any strings in Σ^*, then $h(wx) = h(w)h(x)$.

In this definition, it may or may not be the case that $\Sigma = T$.

A string $w \in \Sigma$ is the concatenation of its individual symbols: $w = w_1 w_2 \cdots w_n$ for some $n \geq 0$. And so if h is a homomorphism,

$$h(w) = h(w_1 w_2 \cdots w_n) = h(w_1)h(w_2 \cdots w_n) = \cdots = h(w_1)h(w_2) \cdots h(w_n)$$

is the concatenation of all the strings $h(w_i)$ for $1 \leq i \leq n$. This means that to completely specify a homomorphism h, we only need to say what string $h(a)$ is for each symbol $a \in \Sigma$.

What if $w = \varepsilon$? It is always the case that $h(\varepsilon) = \varepsilon$ for any homomorphism h. We can see this by noticing that $\varepsilon = \varepsilon\varepsilon$ and so $h(\varepsilon) = h(\varepsilon\varepsilon) = h(\varepsilon)h(\varepsilon)$, that last equation because h is a homomorphism. If we let $w := h(\varepsilon)$, then we just showed that $w = ww$. But the only string w that satisfies this equation is ε, and thus $h(\varepsilon) = \varepsilon$.

For example, let $\Sigma = \{a, b, c\}$ and let $T = \{0, 1\}$. Define the homomorphism h by $h(a) = 01$, $h(b) = 110$, and $h(c) = \varepsilon$. Then $h(abaccab) = (01)(110)(01)(\varepsilon)(\varepsilon)(01)(110) = 011100101110$.

Definition 19.1.2. Let Σ and T be alphabets, and let h be a homomorphism from Σ^* to T^*.

1. For any language $L \subseteq \Sigma^*$, we define the language $h(L) \subseteq T^*$ as

$$h(L) = \{h(w) \mid w \text{ is in } L\}.$$

We say that $h(L)$ is the *image* of L under h.

2. For any language $M \subseteq T^*$, we define the language $h^{-1}(M) \subseteq \Sigma^*$ as

$$h^{-1}(M) = \{w \in \Sigma^* \mid h(w) \text{ is in } M\}.$$

We say that $h^{-1}(M)$ is the *inverse image* of M under h.

Regularity is preserved under taking images and inverse images of a homomorphism.

Proposition 19.1.3. *Let h, L, and M be as in the definition above.*

1. *If L is regular, then so is $h(L)$.*
2. *If M is regular, then so is $h^{-1}(M)$.*

We'll prove (1) by transforming regular expressions and we'll prove (2) by transforming DFAs.

Proof of (1). Let r be any regular expression. We show how to convert r into another regular expression, which we denote $h(r)$, such that $L(h(r)) = h(L(r))$. Thus if $L = L(r)$, then $h(L) = L(h(r))$ and hence $h(L)$ is regular, because $h(r)$ is a regular expression.

The (recursive) transformation rules are derived in a way similar to the proof for reversals, by noting how applying h interacts with the operators used to build regular expressions. The following five facts are easy to see, and we won't bother to prove them:

1. $h(\emptyset) = \emptyset$.
2. $h(\{\varepsilon\}) = \{\varepsilon\}$. (Note that we always have $h(\varepsilon) = \varepsilon$ for any homomorphism h.)
3. $h(\{a\}) = \{h(a)\}$ for any $a \in \Sigma$.
4. For any $L, M \subseteq \Sigma^*$, $h(L \cup M) = h(L) \cup h(M)$.
5. For any $L, M \subseteq \Sigma^*$, $h(LM) = h(L)h(M)$.
6. For any $L \subseteq \Sigma^*$, $h(L^*) = h(L)^*$.

Facts (1)–(5) tell us how to transform any regular expression r for a regular language L into the regular expression $h(r)$ for $h(L)$:

1. If $r = \emptyset$, then define $h(r) := \emptyset$.
2. If $r = \varepsilon$, then define $h(r) := \varepsilon$.
3. If $r = a$ for any $a \in \Sigma$, then define $h(r) := h(a)$ (that is, the regular expression which is the concatenation of the symbols forming the string $h(a)$ and which denotes the language $\{h(a)\}$).

4. If $r = s + t$ for some regular expressions s and t, then define $h(r) := h(s) + h(t)$. (The regular expressions $h(s)$ and $h(t)$ are computed recursively using these rules.)

5. If $r = st$ for some s and t, then define $h(r) := h(s)h(t)$.

6. If $r = s^*$ for some s, then define $h(r) = h(s)^*$.

Facts (1)–(5) imply (by induction on r) that this construction works as advertised. □

Using the h of the last example, let's compute $h(r)$, where $r = \mathbf{b}(\mathbf{a} + \mathbf{bc}^*)^*$.

$$\begin{aligned} h(\mathbf{b}(\mathbf{a} + \mathbf{bc}^*)^*) &= h(\mathbf{b})h((\mathbf{a} + \mathbf{bc}^*)^*) \\ &= h(\mathbf{b})(h(\mathbf{a} + \mathbf{bc}^*))^* \\ &= h(\mathbf{b})(h(\mathbf{a}) + h(\mathbf{bc}^*))^* \\ &= h(\mathbf{b})(h(\mathbf{a}) + h(\mathbf{b})h(\mathbf{c}^*))^* \\ &= h(\mathbf{b})(h(\mathbf{a}) + h(\mathbf{b})h(\mathbf{c})^*)^* \\ &= 110(01 + 110(\varepsilon^*))^* \\ &= 110(01 + 110)^* \ . \end{aligned}$$

Thus if L is given by $\mathbf{b}(\mathbf{a} + \mathbf{bc}^*)^*$, then $h(L)$ is given by $110(01 + 110)^*$.

Proof of (2). Let $A = (Q, T, \delta, q_0, F)$ be a DFA recognizing M. From A we build a DFA $B = (Q, \Sigma, \gamma, q_0, F)$ as follows:

- The state set, start state, and set of final states are the same in B as in A.
- The alphabet of B is Σ.
- The transition function γ for B is defined as follows for every state $q \in Q$ and $a \in \Sigma$:

$$\gamma(q, a) := \hat{\delta}(q, h(a)).$$

The idea is that to compute $\gamma(q, a)$ for some $q \in Q$ and $a \in \Sigma$, we look in the DFA B to see where we would go from q by reading $h(a)$. We then make a single edge transition on a from q to this new state.

To show that this construction is correct, we show that $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$ for any $w \in \Sigma^*$. Since both automata A and B share the same state set, start state, and final states, this equality implies B accepts w if and only if A accepts $h(w)$ (and thus

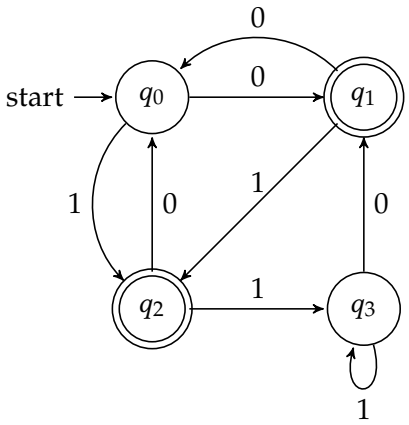
$L(B) = h^{-1}(M)$, and thus $h^{-1}(M)$ is regular). Given any string $w = w_1 w_2 \cdots w_n \in \Sigma^*$, we have

$$\begin{aligned}\hat{\gamma}(q_0, w) &= \gamma(\cdots \gamma(\gamma(q_0, w_1), w_2) \cdots, w_n) \\ &= \hat{\delta}(\cdots \hat{\delta}(\hat{\delta}(q_0, h(w_1)), h(w_2)) \cdots, h(w_n)) \\ &= \hat{\delta}(q_0, h(w_1)h(w_2) \cdots h(w_n)) \\ &= \hat{\delta}(q_0, h(w_1 w_2 \cdots w_n)) = \hat{\delta}(q_0, h(w)) \ .\end{aligned}$$

□

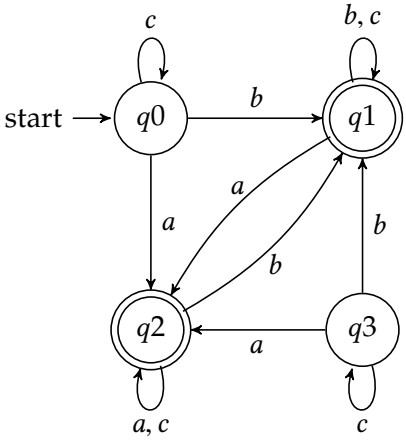
Remark. That does it. Alternatively, there is an inductive (on $|w|$) proof that avoids ellipses. I'll leave it to you to come up with it.

For example, supppose A is the DFA below:



We have $h(a) = 01$. Following 01 from q_0 in A, we get $q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2$, so we draw an edge $q_0 \xrightarrow{a} q_2$ in B. Similarly, $h(b) = 110$, and reading 110 from q_0 gives the path $q_0 \xrightarrow{1} q_2 \xrightarrow{1} q_3 \xrightarrow{0} q_1$, so we draw an edge $q_0 \xrightarrow{b} q_1$ in B. Now $h(c) = \varepsilon$, which does not take us anywhere from q_0, so we draw a self-loop $q_0 \xrightarrow{c} q_0$.

We do the same computation for states q_1, q_2, q_3, obtaining the DFA B:



Note that q_3 is unreachable from q_0, and so it can be removed. B accepts all strings that contain at least one symbol other than c. That is,

$$L(B) = \{w \in \{a, b, c\}^* \mid w \text{ has at least one symbol other than } c\}.$$

B is not the simplest DFA that recognizes this language. In fact, we can collapse the two final states into one, getting an equivalent DFA with only two states. Later, we will see a systematic way to find the simplest DFA (i.e., fewest states) for any regular language.

19.2 Using closure properties to show nonregularity

The pumping lemma is a good tool to show that a language is not regular, but it doesn't always suffice. For example, the language $\{0^n1^m \mid n \neq m\}$ is not regular, and it is difficult (but not impossible) to show this using the pumping lemma alone. This is where closure properties can be useful when combined with the pumping lemma. A proof that a language L is not regular might take the form of a proof by contradiction:

> Suppose L is regular. Then by such-and-such a closure property of regular languages, we know that such-and-such other language L' is also regular. But L' cannot be regular because it is not pumpable [insert use of pumping lemma here for L']. Contradiction.

So proving L not regular reduces to proving L' not regular. Although we may not be able to apply the pumping lemma to L directly, we may be able to apply it to L' instead. Even if we can apply the pumping lemma to L directly, it may still be easier to use closure properties.

Let's apply this idea to the language $L := \{0^n 1^m \mid n \neq m\}$. This language is actually not pumpable, that is, there is a direct proof via the pumping lemma that L is not regular. Can you find it? However, we now give a much easier proof using closure properties.

Proposition 19.2.1. *The language $L := \{0^n 1^m \mid n \neq m\}$ over the binary alphabet $\Sigma = \{0, 1\}$ is not regular.*

Proof. Suppose L is regular. Then since the class of regular languages is closed under complements, it follows that the language $L_1 := \overline{L}$ is also regular. The language $L_2 := \{0^n 1^m \mid m, n \geq 0\}$ is also regular, because L_2 is just $L(0^*1^*)$. Then the language $L_3 := L_1 \cap L_2$ is also regular, because the class of regular languages is closed under intersection. But L_3 is exactly the language $\{0^n 1^m \mid n = m\} = \{0^n 1^n \mid n \geq 0\}$, which as we have already seen is not pumpable (this was our first example of using the pumping lemma, above) and thus not regular. Contradiction. Thus L is not regular. □

Next, we apply the technique to a language that *is* pumpable (so we cannot use the pumping lemma directly). The language L in question is the union of two languages D and E over the four-letter alphabet $\{a, b, c, d\}$, where E is the set of all strings with the same number of b's as c's, and D is the set of all strings that contain a "close duplicate", that is, two occurrences of the same symbol with at most one other symbol in between. More formally, letting $s := (a + b + c + d + \varepsilon)$, the language D is the regular language given by the regular expression

$$D := L(s^*(asa + bsb + csc + dsd)s^*) \ .$$

We show below that the language $L := D \cup E$ is not regular, but we cannot use the pumping lemma directly to do this, because L is actually pumpable. The way to see that L is pumpable is by using the usual pumping lemma template but instead describing a winning strategy for *our opponent*:

> Let $p := 5$. Clearly, $p > 0$.
> Let $s = w_1 w_2 \ldots w_n$ be any string in L of length $n \geq 5$. Since the first five symbols $w_1, \ldots, w_5$ are chosen from a four-letter alphabet, by the pigeonhole principle there must be a duplicate, i.e., there exist $1 \leq j < k \leq 5$ such that $w_j = w_k$.
> Now choose x, y, z as follows:
>
> 1. If $k = j + 1$ or $k = j + 2$, then choose any $\ell \in \{1, 2, 3, 4, 5\}$ such that either $\ell < j$ or $\ell > k$, and pump on w_ℓ, i.e., set $y := w_\ell$, $x := w_1 \cdots w_{\ell-1}$, and $z := w_{\ell+1} \cdots w_n$.
> 2. Otherwise, either $k = j+3$ or $k = j+4$. Pump on $y := w_{j+1} w_{j+2}$ with $x := w_1 \cdots w_j$ and $z := w_{j+3} \cdots w_n$.

In either case, one checks for all $i \neq 1$ that xy^iz contains a close duplicate, whence $xy^iz \in D$: In case (1), w_j and w_j form a close duplicate, and this is unaffected by pumping y. In case (2), if $i = 0$ ("pumping down"), then the original w_j are w_k are made close; if $i \geq 2$ ("pumping up"), then yy contains a close duplicate.
Thus $xy^iz \in L$ for all $i \in \mathbb{N}$: if $i \neq 1$, then $xy^iz \in D \subseteq L$, and if $i = 1$, then $xy^iz = xyz = s \in L$.

Proposition 19.2.2. *The language $L := D \cup E$ described above is not regular.*

Proof. Suppose for the sake of contradiction that L is regular. Let $h : \{0,1\}^* \to \{a,b,c,d\}^*$ be the homomorphism given by

$$h(0) = abd$$
$$h(1) = acd$$

Letting $L' := h^{-1}(L)$, we have that L' is also regular by one of the closure properties of regular languages. Now let $w \in \{0,1\}^*$ be any binary string, and notice that $h(w)$ has no close duplicates, i.e., $h(w) \notin D$. It follows that $h(w) \in L \iff h(w) \in E$ for any w, and thus $L' = h^{-1}(L) = h^{-1}(E)$. Also notice that the number of 0's in w equals the number of b's in $h(w)$, and the number of 1's in w equals the number of c's in $h(w)$, and thus

$$L' := h^{-1}(E) = \left\{w \in \{0,1\}^* \mid w \text{ has the same number of 0's as 1's}\right\}.$$

But we already know that this language is not pumpable (one of our first examples of using the pumping lemma), hence not regular. Contradiction. Thus L must not be regular. □

Lecture 20

20.1 DFA minimization

We say that a DFA is *minimal* if there is no equivalent DFA with fewer states.

We will show (the Myhill-Nerode theorem) that for any regular language L there is a *unique* minimal DFA recognizing L. We will also describe how to construct such a DFA, given any other DFA recognizing L. By uniqueness, we mean that any two minimal DFAs recognizing L are actually the same DFA, up to relabeling of the states. (In technical terms, the two DFAs are *isomorphic*.)

Example 20.1.1. Consider this 7-state DFA that accepts a binary string iff its second to last symbol is 1:

	0	1
$\rightarrow q_\varepsilon$	q_0	q_1
q_0	q_{00}	q_{01}
q_1	q_{10}	q_{11}
q_{00}	q_{00}	q_{01}
q_{01}	q_{10}	q_{11}
$*q_{10}$	q_{00}	q_{01}
$*q_{11}$	q_{10}	q_{11}

The states record in their labels the most recent two characters read. This DFA is not minimal; in fact, there is an equivalent DFA with only four states.

[Example] To find the minimal equivalent DFA, we find pairs of states that are indistinguishable and collapse them into one state.

Definition 20.1.2. Let $N = \langle Q, \Sigma, \delta, q_0, F\rangle$ be any DFA.

1. We say that N is *sane* iff every state in Q is reachable from the start state q_0. That is, N is sane if and only if, for every $q \in Q$, there exists $w \in \Sigma^*$ such that $q = \hat{\delta}(q_0, w)$.

2. For any state $q \in Q$, define $N_q := \langle Q, \Sigma, \delta, q, F\rangle$, the DFA obtained from N by moving the start state to q. (Of course, $N_{q_0} = N$.)

Note:

- For every DFA N there is an equivalent sane DFA with as many or fewer states: simply remove the states of N (if any) that are unreachable from the start state. The removed states clearly have no effect on whether a string is accepted or not.
- Thus every minimal DFA must be sane. We'll restrict our attention then to sane DFAs.

At this point, depending on time, we may skip the following and go straight to Lecture 21.1.

Definition 20.1.3. Let L be any language over alphabet Σ.

1. For any $w \in \Sigma^*$, define $L_w := \{x \mid wx \in L\}$.
2. Define $C_L := \{L_w \mid w \in \Sigma^*\}$.

Notice that we always have $L = L_\varepsilon$.
Lemmas 20.1.4 and 21.0.2 below imply the Myhill-Nerode theorem.

Lemma 20.1.4. *Let $N = \langle Q, \Sigma, \delta, q_0, F\rangle$ be any sane DFA, and let $L = L(N)$. Fix any $w \in \Sigma^*$, and let $q = \hat{\delta}(q_0, w)$. Then*

$$L_w = L(N_q)\ . \tag{20.1}$$

It follows that $C_L = \{L(N_q) \mid q \in Q\}$, and so $\|C_L\| \leq \|Q\|$.

Proof. For any string $x \in \Sigma^*$,

$$\begin{aligned} x \in L_w &\iff wx \in L \\ &\iff \hat{\delta}(q_0, wx) \in F \\ &\iff \hat{\delta}(\hat{\delta}(q_0, w), x) \in F \\ &\iff \hat{\delta}(q, x) \in F \\ &\iff x \in L(N_q). \end{aligned}$$

This shows that $L_w = L(N_q)$, from which it follows immediately that $C_L \subseteq \{L(N_q) \mid q \in Q\}$. The fact that $\{L(N_q) \mid q \in Q\} \subseteq C_L$ comes from fact that, since N is sane, for every $q \in Q$ there exists $w \in \Sigma^*$ such that $q = \hat{\delta}(q_0, w)$ (and thus $L_w = L(N_q)$). □

Corollary 20.1.5. *If L is regular, then C_L is finite.*

Lecture 21

Lemma 21.0.2 below is essentially the converse of Lemma 20.1.4. First we need to prove:

Lemma 21.0.1. *Let L be any language over Σ, let w and w' be any strings in Σ^*, and let a be any symbol in Σ. Then if $L_w = L_{w'}$, then $L_{wa} = L_{w'a}$.*

Proof. We'll show that if $L_w \subseteq L_{w'}$, then $L_{wa} \subseteq L_{w'a}$. This is enough, because to get equality we just run the same argument with w and w' swapped.

Suppose $L_w \subseteq L_{w'}$ and let x be any string in Σ^*. Then

$$\begin{aligned} x \in L_{wa} &\implies wax \in L \\ &\implies ax \in L_w \\ &\implies ax \in L_{w'} \\ &\implies w'ax \in L \\ &\implies x \in L_{w'a}. \end{aligned}$$

Thus $L_{wa} \subseteq L_{w'a}$. □

Lemma 21.0.2. *Let $L \subseteq \Sigma^*$ be any language over Σ. If C_L is finite, then L is recognized by the following minimal DFA:*

$$N_{\min} := \langle C_L, \Sigma, \delta_{\min}, q_{0,\min}, F_{\min} \rangle,$$

where

- $q_{0,\min} := L_\varepsilon = L$,
- $\delta_{\min}(L_w, a) := L_{wa}$ *for all* $w \in \Sigma^*$ *and* $a \in \Sigma$, *and*
- $F_{\min} := \{L' \in C_L \mid \varepsilon \text{ is in } L'\}$.

Note that the transition function $\delta_{\min}$ is well-defined because of Lemma 21.0.1. The output state L_{wa} only depends on the language L_w, and does not change if we substitute another string w' such that $L_w = L_{w'}$.

Proof of Lemma 21.0.2. Fix a string $w \in \Sigma^*$. First we prove that

$$L_w = \hat{\delta}_{\min}(q_{0,\min}, w) \ . \tag{21.1}$$

This may be obvious, based on how we defined $\delta_{\min}$ but we'll prove it anyway by induction on $|w|$.

Base case: $|w| = 0$**.** In this case, $w = \varepsilon$, and we have

$$\begin{aligned} L_w = L_\varepsilon &= L \\ &= q_{0,\min} \\ &= \hat{\delta}_{\min}(q_{0,\min}, \varepsilon) \\ &= \hat{\delta}_{\min}(q_{0,\min}, w). \end{aligned}$$

Inductive case: $|w| > 0$**.** Then $w = xa$ for some $a \in \Sigma$ and some $x \in \Sigma^*$ with $|x| = |w| - 1$. Assuming (the inductive hypothesis) that Equation (21.1) holds for x instead of w (that is, assuming that $L_x = \hat{\delta}_{\min}(q_{0,\min}, x)$), we get

$$\begin{aligned} L_w = L_{xa} &= \delta_{\min}(L_x, a) \\ &= \delta_{\min}(\hat{\delta}_{\min}(q_{0,\min}, x), a) \\ &= \hat{\delta}_{\min}(q_{0,\min}, xa) \\ &= \hat{\delta}_{\min}(q_{0,\min}, w). \end{aligned}$$

Now we can show that $L = L(N_{\min})$:

$$\begin{aligned} w \in L &\iff w\varepsilon \in L \\ &\iff \varepsilon \in L_w \\ &\iff L_w \in F_{\min} \\ &\iff \hat{\delta}_{\min}(q_{0,\min}, w) \in F_{\min} \\ &\iff w \in L(N_{\min}). \end{aligned}$$

Finally, $N_{\min}$ is a minimal DFA by Lemma 20.1.4. □

Corollary 21.0.3. *If C_L is finite, then L is regular.*

Theorem 21.0.4 (Myhill-Nerode)**.** *A language L is regular iff C_L is finite. If such is the case, the size of C_L equals the number of states of the unique minimal DFA recognizing L.*

Proof. We've proved most of this already. The first sentence of the theorem is clear by Corollaries 20.1.5 and 21.0.3. For the second sentence, we already constructed a minimal DFA $N_{\min}$ recognizing L with state set C_L in Lemma 21.0.2. The only thing left to show is that $N_{\min}$ is *unique* among minimal DFAs recognizing L.

To that end, we first show that the map $q \mapsto L(N_q)$ of Lemma 20.1.4 preserves the structure of the DFA. As in Lemma 20.1.4, let $N = \langle Q, \Sigma, \delta, q_0, F\rangle$ be any sane DFA (not necessarily minimal) recognizing L. Recall that $C_L = \{L(N_q) \mid q \in Q\}$ by Lemma 20.1.4. The correspondence $q \mapsto L(N_q)$ mapping Q (the state set of N) onto C_L (the state set of the DFA $N_{\min}$ constructed in the proof of Lemma 21.0.2) may or may not be one-to-one, depending on whether or not Q has the same size as C_L. But in any case, the mapping preserves all the structure of the DFA N:

1. We have $L(N_{q_0}) = L(N) = L = L_\varepsilon = q_{0,\min}$, and so the start state q_0 of N is mapped to the start state $q_{0,\min}$ of $N_{\min}$.

2. Given any $q \in Q$ and $a \in \Sigma$, let $r = \delta(q, a)$. Fix some (any) string $w \in \Sigma^*$ such that $q = \hat{\delta}(q_0, w)$. (N is sane because it is minimal, therefore w exists.) Now using Equation (20.1) of Lemma 20.1.4 twice—first for q then for r—we get

$$\delta_{\min}(L(N_q), a) = \delta_{\min}(L_w, a) = L_{wa} = L(N_r),$$

   the last equality holding because $r = \delta(q, a) = \delta(\hat{\delta}(q_0, w), a) = \hat{\delta}(q_0, wa)$. This shows that an a-transition $q \xrightarrow{a} r$ in N corresponds to an a-transition $L(N_q) \xrightarrow{a} L(N_r)$ between the corresponding states in $N_{\min}$.

3. For any $q \in Q$,

$$q \in F \iff \varepsilon \in L(N_q) \iff L(N_q) \in F_{\min}.h$$

   Thus the accepting states of N map to accepting states of $N_{\min}$, and the rejecting states of N map to rejecting states of $N_{\min}$.

Now suppose that N is minimal. Since N and $N_{\min}$ are both minimal and equivalent, they have the same number of states: $\|Q\| = \|C_L\|$. Then by the Pigeonhole Principle we must have $L(N_q) \neq L(N_r)$ for all $q, r \in Q$ with $q \neq r$, because the two sets have the same size. So the mapping $q \mapsto L(N_q)$ is a natural one-to-one correspondence between Q and C_L.

The preservation of the structure of N under this correspondence makes it clear that N and $N_{\min}$ are the same DFA, via the relabeling $q \leftrightarrow L(N_q)$. □

21.1 Constructing the minimal DFA

The proof of Theorem 21.0.4 holds the seeds of an algorithm for converting a sane DFA N into its minimal equivalent DFA $N_{\min}$.

Definition 21.1.1. Let $N = \langle Q, \Sigma, \delta, q_0, F\rangle$ be any DFA. For any states $q, r \in Q$ and $x \in \Sigma^*$, we say that *q and r are distinguished by string x* iff x is in one of the languages $L(N_q)$ and $L(N_r)$ but not both. We say that q and r are *distinguishable* if there exists some string that distinguishes them; otherwise, they are *indistinguishable*.

This fact is obvious based on the definition above.

Fact 21.1.2. *Two states q and r of N are indistinguishable iff $L(N_q) = L(N_r)$.*

Thus indistinguishable states of N are those that are mapped to the same state of $N_{\min}$. We now give a method for finding pairs of indistinguishable states of N. By merging groups of mutually indistinguishable states of N into single states, we effectively convert N into $N_{\min}$.

The idea of the algorithm is to record pairs of states that *are* distinguishable, until we can't find any more of those. Then any pairs left over must be indistinguishable. Here is the algorithm.

Algorithm: DFA Minimization Algorithm — Table of Distinguishabilites

Input : A DFA $N = \langle Q, \Sigma, \delta, q_0, F \rangle$
Output: A two dimensional array T

```
1  T ← Array2D(|Q|, |Q|,' ') // init. a 2D array with all blanks
2  for p, q ∈ Q do
3  |  if (p ∈ F ∧ q ∉ F) ∨ (p ∉ F ∧ q ∈ F) then
4  |  |  T[p, q] = T[q, p] ='X' // These states are distinguished by
   |  |       ε
5  |  end
6  end
7  repeat
8  |  if [(∃p, q ∈ Q) T[p, q] =' '] ∧ [(∃a ∈ Σ) T[δ(p, a), δ(q, a)] ='X'] then
9  |  |  T[p, q] = T[q, p] ='X'
10 |  end
11 until T does not change
```

After this algorithm finishes, the remaining blank entries of T are exactly the pairs of indistinguishable states.

The minimal DFA will then result from merging groups of indistinguishable states into single states. (Note that the algorithm still can be run even if N is not sane, but then the collapsed DFA may not be sane.)

[Running the algorithm on the DFA of Exercise 4.4.1 and drawing the resulting DFA]

An Example

So let's do the example of Exercise 4.4.1 of the text. We have a transition table

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

The first thing we are going to observe is that we don't need H at all. There are no transitions into H, so it is unreachable. That simplifies some things.

We write the matrix T, putting dots instead of blanks just to make things a little more visible.

	A	B	C	D	E	F	G
A	.	.	.	.	.	.	.
B	.	.	.	.	.	.	.
C	.	.	.	.	.	.	.
D	.	.	.	.	.	.	.
E	.	.	.	.	.	.	.
F	.	.	.	.	.	.	.
G	.	.	.	.	.	.	.

The first loop, lines 2-5 of the algorithm, have us flagging states as distinguishable if one is final and the other is not. This produces

	A	B	C	D	E	F	G
A	.	.	.	X	.	.	.
B	.	.	.	X	.	.	.
C	.	.	.	X	.	.	.
D	X	X	X	.	X	X	X
E	.	.	.	X	.	.	.
F	.	.	.	X	.	.	.
G	.	.	.	X	.	.	.

Now we iterate the second loop until nothing changes. If $\delta(p, 0) = p'$ and $\delta(q, 0) = q'$ and p' is already distinguishable from q', we flag p and q.

$\delta(A, 0) = B$, $\delta(C, 0) = D$, and B is dist from D, so A and C are dist.

$\delta(A, 0) = B$, $\delta(E, 0) = D$, and B is dist from D, so A and E are dist.

$\delta(A, 1) = A$, $\delta(H, 1) = D$, and A is dist from D, so A and H are dist.

This produces

	A	B	C	D	E	F	G
A	.	.	X	X	X	.	.
B	.	.	.	X	.	.	.
C	X	.	.	X	.	.	.
D	X	X	X	.	X	X	X
E	X	.	.	X	.	.	.
F	.	.	.	X	.	.	.
G	.	.	.	X	.	.	.

We iterate.
$\delta(A, 1) = A$, $\delta(B, 1) = C$, and A is dist from C, so A and B are dist.
$\delta(A, 1) = A$, $\delta(F, 1) = E$, and A is dist from E, so A and F are dist.
This produces

	A	B	C	D	E	F	G
A	.	X	X	X	X	X	.
B	X	.	.	X	.	.	.
C	X	.	.	X	.	.	.
D	X	X	X	.	X	X	X
E	X	.	.	X	.	.	.
F	X	.	.	X	.	.	.
G	.	.	.	X	.	.	.

We iterate.
$\delta(B, 0) = A$, $\delta(C, 0) = D$, and A is dist from D, so B and C are dist.
$\delta(B, 0) = A$, $\delta(E, 0) = D$, and A is dist from D, so B and E are dist.
$\delta(B, 0) = A$, $\delta(G, 0) = F$, and A is dist from F, so B and G are dist.
Thus

	A	B	C	D	E	F	G
A	.	X	X	X	X	X	.
B	X	.	X	X	X	.	X
C	X	X	.	X	.	.	.
D	X	X	X	.	X	X	X
E	X	X	.	X	.	.	.
F	X	.	.	X	.	.	.
G	.	X	.	X	.	.	.

$\delta(C, 0) = D$, $\delta(F, 0) = G$, and E is dist from G, so C and F are dist.
$\delta(C, 1) = B$, $\delta(G, 1) = G$, and B is dist from G, so C and G are dist.
Thus

	A	B	C	D	E	F	G
A	.	X	X	X	X	X	.
B	X	.	X	X	X	.	X
C	X	X	.	X	.	X	X
D	X	X	X	.	X	X	X
E	X	X	.	X	.	.	.
F	X	.	X	X	.	.	.
G	.	X	X	X	.	.	.

$\delta(E, 0) = D$, $\delta(F, 0) = G$, and D is dist from G, so E and F are dist.
$\delta(E, 0) = D$, $\delta(G, 0) = F$, and D is dist from F, so E and G are dist.

	A	B	C	D	E	F	G
A	.	X	X	X	X	X	.
B	X	.	X	X	X	.	X
C	X	X	.	X	.	X	X
D	X	X	X	.	X	X	X
E	X	X	.	X	.	X	X
F	X	.	X	X	X	.	X
G	.	X	X	X	X	X	.

It would seem at this point that we are done. We might have A and G indistinguishable, B and F indistinguishable, and C and E indistinguishable.

If we rewrite the transition table, this would be conversion of

	0	1
$\rightarrow A$	B	A
B	A	C
C	D	B
$*D$	D	A
E	D	F
F	G	E
G	F	G
H	G	D

into

	0	1
$\rightarrow A, G$	B, F	A, G
B, F	A, G	C, E
C, E	D	B, F
$*D$	D	A, G
C, E	D	B, F
B, F	A, G	C, E
A, G	B, F	A, G

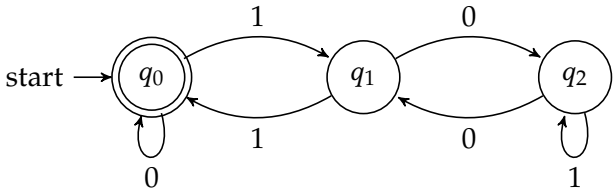
and a four-state DFA.

Lecture 22

22.1 Another Language Representation

So far, we've dealt with languages using a couple representations. In general, languages are sets, but we've seen we can represent regular languages using various types of automata as well as regular expressions. We also saw that extending automata to gain power is not a trivial task (e.g. adding nondeterminism does not give DFAs more power). However, there is another representation that may be easier to extend: a grammar.

Consider the following DFA $D = \langle Q, \Sigma, \delta, q_0, F \rangle$ recognizing multiples of 3 in binary:



We can represent each edge in the following way:

$$\begin{aligned} A &\to \alpha B \quad (A, B \in Q), (\alpha \in \Sigma) \\ C &\to \varepsilon \quad (C \in F) \end{aligned}$$

Resulting in the following 'rules'[1]:

$$
\begin{aligned}
q_0 &\to 0q_0 \\
q_0 &\to 1q_1 \\
q_0 &\to \varepsilon \\
q_1 &\to 0q_2 \\
q_1 &\to 1q_0 \\
q_2 &\to 0q_1 \\
q_2 &\to 1q_2
\end{aligned}
$$

This form of representing a language is called a grammar. In order to see whether or not a grammar 'recognizes' a particular string, you start with the start symbol (in this case q_0) and use the rules above as 'replacement rules', where you are allowed to replace something on the left of an arrow with whatever comes to the right. For example, to derive the string '110' we would use this series of replacements:

$$q_0 \Rightarrow 1q_1 \Rightarrow 11q_0 \Rightarrow 110q_0 \Rightarrow 110$$

Note how the last step replaces q_0 with ε, corresponding to the rules above.

Definition 22.1.1. Formally, a *context-free grammar* G is a 4-tuple: $G = \langle V, T, P, S \rangle$:

- V is a finite set of symbols called *variables* or *nonterminals*. These are things that appear to the left of an arrow. In our example above, $V = \{q_0, q_1, q_2\}$
- T is a finite set of symbols called *terminals* or *terminal symbols*. These are all symbols that appear in any rule but never appear on the left of a rule (this is what makes it 'context-free'). Again, per our example, $T = \{0, 1, \varepsilon\}$
- P is the (finite) set of rules themselves, often called (somewhat boringly) *rules* or *productions*.
- $S \in V$ is the *start symbol*; for our case $S = q_0$.

It should be easy to see that using the above method of converting NFAs to grammars, one will always end up with productions yielding exactly one nonterminal followed by one terminal symbol. However, this is a representation of a language that is easily modified, which begs the question: can we use it to recognize more than just regular languages? It turns out, by allowing nonterminals to

[1] sometimes we abbreviate multiple rules with the same left-hand side with '|' representing 'or', e.g. $q_0 \to 0q_0 \mid 1q_1 \mid \varepsilon$

appear to the right of terminal symbols (in the body of a production) we can do just that. Consider the grammar below:

$$S \to 0S1 \mid 01$$

It is immediately clear that any string (and only strings) from the language $L = \{0^n 1^n \mid n \in \mathbb{N}\}$ can be derived from this grammar. For example: $S \Rightarrow 0S1 \Rightarrow 00S11 \Rightarrow 0011$

We've been using '$\Rightarrow$' informally, so we should define what it actually means

Definition 22.1.2. Let $G = \langle V, T, P, S\rangle$ be a grammar. Let $\alpha A \beta \in (V \cup T)^*$, where $A \in V$, and let $A \to \gamma$ be a production. Then we say $\alpha A \beta \underset{G}{\Rightarrow} \alpha\gamma\beta$ or $\alpha A \beta \Rightarrow \alpha\gamma\beta$ if G is understood. '$\Rightarrow$' is read as *derives*.

Analogous to defining the extended transition function, we want to extend the derives relationship to account for multiple steps.

Definition 22.1.3. Let G be a grammar as above.

Basis For any string $\alpha \in (T \cup V)^*$, we say $\alpha \overset{*}{\underset{G}{\Rightarrow}} \alpha$.

Induction If $\alpha \overset{*}{\underset{G}{\Rightarrow}} \beta$ and $\beta \underset{G}{\Rightarrow} \gamma$ then $\alpha \overset{*}{\underset{G}{\Rightarrow}} \gamma$.

Again, if G is understood we would write $\alpha \overset{*}{\Rightarrow} \gamma$

Given the above, we can now define the language of a grammar:

Definition 22.1.4. Let $G = \langle V, T, P, S\rangle$ be a grammar. The *language* of G is written $L(G)$ and is defined as

$$L(G) = \left\{ w \in T^* \mid S \overset{*}{\underset{G}{\Rightarrow}} w \right\}$$

22.2 (*) Converting Regular Languages to Grammars

We can also now formalize how to convert a regular language to a grammar.

Theorem 22.2.1. *Given a DFA $D = \langle Q, \Sigma, \delta, q_0, F\rangle$, we can construct a grammar $G = \langle Q, \Sigma, P, q_0\rangle$ in the following way:*

- *Notice: $V = Q$, $T = \Sigma$, and the start symbol $S = q_0$*
- *For all $q, r \in Q$ and $a \in \Sigma$ such that $\delta(q, a) = r$, add the production $q \to ar$ to P*
- *For all $f \in F$, add the production $f \to \varepsilon$ to P*

Then $L(D) = L(G)$.

Proof. It suffices to show that the extended transition function and the grammar agree on all strings derivable from the start state. Symbolically:

$$\hat{\delta}(q_0, w) = r \iff q_0 \overset{*}{\underset{G}{\Rightarrow}} wr$$

Both directions proceed by induction on $|w|$. Notice the base case ($w = \varepsilon$) is trivially true by definition:

$$\hat{\delta}(q_0, \varepsilon) = q_0 \iff q_0 \overset{*}{\underset{G}{\Rightarrow}} q_0$$

Let $w = xa$ where $x \in \Sigma^*$ and $a \in \Sigma$

($\Rightarrow$) Letting $\hat{\delta}(q_0, x) =: p$, our inductive hypothesis is $\hat{\delta}(q_0, x) = p \Rightarrow q_0 \overset{*}{\underset{G}{\Rightarrow}} xp$

$$r = \hat{\delta}(q_0, xa) = \delta(\hat{\delta}(q_0, x), a) = \delta(p, a)$$

Since we have $\delta(p, a) = r$, then by construction of G, we have '$p \to ar$' $\in P$. So

$$q_0 \overset{*}{\underset{G}{\Rightarrow}} xp \underset{G}{\Rightarrow} xar$$
$$\therefore q_0 \overset{*}{\underset{G}{\Rightarrow}} wr$$

Note: if $r \in F$, then '$r \to \varepsilon$' $\in P$ so $\hat{\delta}(q_0, w) \in F \Rightarrow q_0 \overset{*}{\underset{G}{\Rightarrow}} w$

($\Leftarrow$) Here, our inductive hypothesis is for any $p \in V$:

$$q_0 \overset{*}{\underset{G}{\Rightarrow}} xp \Rightarrow \hat{\delta}(q_0, x) = p$$

Given $q_0 \overset{*}{\underset{G}{\Rightarrow}} xar$, there must be an intermediate step such that

$$q_0 \overset{*}{\underset{G}{\Rightarrow}} xp \underset{G}{\Rightarrow} xar$$

for some variable p, which implies the rule '$p \to ar$' $\in P$. Given the construction of G, this means the corresponding state p has a transition $\delta(p, a) = r$. So

$$\begin{aligned} \hat{\delta}(q, w) = \hat{\delta}(q, xa) &= \delta(\hat{\delta}(q, x), a) \\ &= \delta(p, a) \\ &= r \end{aligned}$$

□

Lecture 23

23.1 Sentential Forms

It is useful to talk about any strings derivable from a grammar, including those with nonterminal symbols. These strings have a special name:

Definition 23.1.1. Let $G = \langle V, T, P, S \rangle$ be a context-free grammar. Then any string $\alpha \in (V \cup T)^*$ such that $S \overset{*}{\Rightarrow} \alpha$ is a *sentential form*.

We can also differentiate between *left-sentential forms* (where $S \underset{lm}{\overset{*}{\Rightarrow}} \alpha$) and *right-sentential forms* (where $S \underset{rm}{\overset{*}{\Rightarrow}} \alpha$).

Notice that $L(G) = \{w \in T^* | w \text{ is a sentential form of } G\}$

Sentential forms, the $\Rightarrow$ and $\overset{*}{\Rightarrow}$ operators. Leftmost and rightmost derivations.

23.2 Parse Trees

Parse trees, yield of a parse tree is the concatenation of leaves (which may or may not be terminal). We're primarily interested in those parse trees which start at S and yield a string in $L(G)$

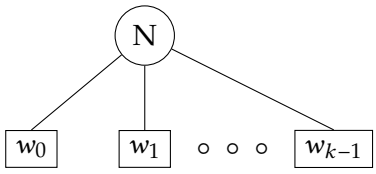
Equivalence with Derivations.

Theorem 23.2.1. *Let $G = \langle V, T, P, S \rangle$ be a context free grammar, and suppose we have a parse tree with root labeled by variable A and with yield w, where $w \in T^*$. Then there is a leftmost derivation $A \underset{lm}{\overset{*}{\Rightarrow}} w$ in grammar G*

Proof. The proof is by induction on the height of the parse tree.

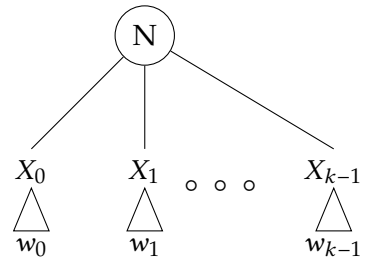
Basis:

We start the base case at a height of 1, so the tree must be as below:



Since this is a parse tree, there must be a production $A \to w$. So $A \underset{lm}{\overset{*}{\Rightarrow}} w$ is a leftmost production.

Induction: If the height of the tree is $n > 1$, then the tree must be of the following form:



The X's are either terminals or variables.

1. If X_i is a terminal, define w_i to be the string consisting of X_i alone.
2. If X_i is a variable, then it must be the root of some subtree with a yield of terminals, which we shall call w_i. Note that in this case, the subtree is of height less than n, so the inductive hypothesis applies to it. That is, there is a leftmost derivation $X_i \underset{lm}{\overset{*}{\Rightarrow}} w_i$

First note that $w = w_0 w_1 \ldots w_{k-1}$

We construct the leftmost-derivation as follows. First, start with the derivation $A \underset{lm}{\Rightarrow} X_0 X_1 \ldots X_{k-1}$. Then for each $i = 0, 1, \ldots, k-1$ we show that $A \underset{lm}{\overset{*}{\Rightarrow}} w_0 w_1 \ldots w_{i-1} X_i X_{i+2} \ldots X_{k-1}$. To do so, we need another induction on i.

Sub-Basis: For $i = 0$, we already know $A \underset{lm}{\overset{*}{\Rightarrow}} X_0 X_1 \ldots X_{k-1}$

Sub-Induction: Our inductive hypothesis is $A \underset{lm}{\overset{*}{\Rightarrow}} w_0 w_1 \ldots w_{i-2} X_{i-1} X_i \ldots X_{k-1}$.

1. If X_{i-1} is a terminal, do nothing. However, we mentally replace X_{i-1} as the terminal string w_{i-1}. Therefore we have

$$A \underset{lm}{\overset{*}{\Rightarrow}} w_0 w_1 \ldots w_{i-2} w_{i-1} X_i \ldots X_{k-1}$$

2. If X_{i-1} is a variable, continue with a derivation of w_{i-1} from X_{i-1}, in the context of the derivation being constructed. That is, if this derivation is

$$X_{i-1} \overset{*}{\underset{lm}{\Rightarrow}} \alpha_0 \overset{*}{\underset{lm}{\Rightarrow}} \alpha_1 \cdots \overset{*}{\underset{lm}{\Rightarrow}} w_{i-1}$$

then we proceed with

$$\begin{aligned} w_0 w_1 \cdots w_{i-2} X_{i-1} X_i \cdots X_{k-1} & \\ & \overset{*}{\underset{lm}{\Rightarrow}} w_0 w_1 \cdots w_{i-2} \alpha_0 X_i \cdots X_{k-1} \\ & \overset{*}{\underset{lm}{\Rightarrow}} w_0 w_1 \cdots w_{i-2} \alpha_1 X_i \cdots X_{k-1} \\ & \vdots \\ & \overset{*}{\underset{lm}{\Rightarrow}} w_0 w_1 \cdots w_{i-2} w_{i-1} X_i \cdots X_{k-1} \end{aligned}$$

which is a derivation $A \overset{*}{\underset{lm}{\Rightarrow}} w_0 w_1 \cdots w_{i-1} X_i X_{i+1} \cdots X_{k-1}$.

□

The language $L(G)$ of a grammar G.

Originally devised by Noam Chomsky and others to study natural language. This did not succeed very well, but they found heavy use in programming language syntax and parsing.

Example 23.2.2. $L = \{a^n b^m c^n \mid m, n \in \mathbb{N}\}$

$$\begin{aligned} S &\to aBc \mid ac \\ B &\to bB \mid b \end{aligned}$$

Example 23.2.3. $L = \left\{a^i b^j c^k \mid i \leq j\right\}$
Here it is easier to consider a subset of the language first: $L = \left\{a^i b^j \mid i \leq j\right\}$

$$\begin{aligned} A &\to aAb \mid B \\ B &\to Bb \mid ab \mid b \end{aligned}$$

Now, we can augment it by adding any number of c's to the right:

$$\begin{aligned} S &\to A \mid AC \\ C &\to cC \mid c \end{aligned}$$

A grammar for expressions in arithmetic:

$$E \to E + E$$
$$E \to E - E$$
$$E \to E * E$$
$$E \to E/E$$
$$E \to (E)$$
$$E \to c$$
$$E \to v$$

Parse tree for $v + c - v * (v + c)$.

Conventions and shorthand: head of first production is start symbol, can collapse productions with same head with the | separator, etc.

Lecture 24

Ambiguity. Example: two parse trees for $c + c * c$. One is "better" than the other, because it more closely resembles the intended evaluation order given by the precedence and associativity rules (operators applied to left and right siblings only). Removing ambiguity is a good thing to eliminate "bad" parse trees, if it is possible (it is not always possible).

Recall the grammar for arithmetic expressions from before:

$$E \to E + E \mid E - E \mid E * E \mid E/E \mid (E) \mid c \mid v$$

We can build an equivalent, unambiguous grammar whose parse trees properly reflect the order of evaluation. Idea: define a hierarchy of three syntactic categories (variables): E (expression), T (term), and F (factor), based on the three precedence levels: $+, -$ (lowest), $*, /$ (middle), and atomic and parenthesized expressions (highest), respectively. Each category generates just those expressions whose top-level operator has at least the corresponding precedence (E for any operator, T for $*, /$ and above, and F for only the highest). So the equivalent, unambiguous grammar is

$$\begin{aligned} E &\to E + T \mid E - T \mid T \\ T &\to T * F \mid T/F \mid F \\ F &\to c \mid v \mid (E) \end{aligned}$$

So, for example: $E \overset{*}{\Rightarrow} T \pm T \pm T \pm \cdots \pm T$, and T generates a series of factors separated by $*$ and $/$, etc. Note that instead of $E \to E + T \mid E - T \mid T$, we could have used the equivalent $E \to T + E \mid T - E \mid T$. We didn't, however, because the latter productions, while generating the same sentential forms, do not correctly reflect the *left-to-right associativity* of the $+$ and $-$ operators: the last operator applied is the rightmost.

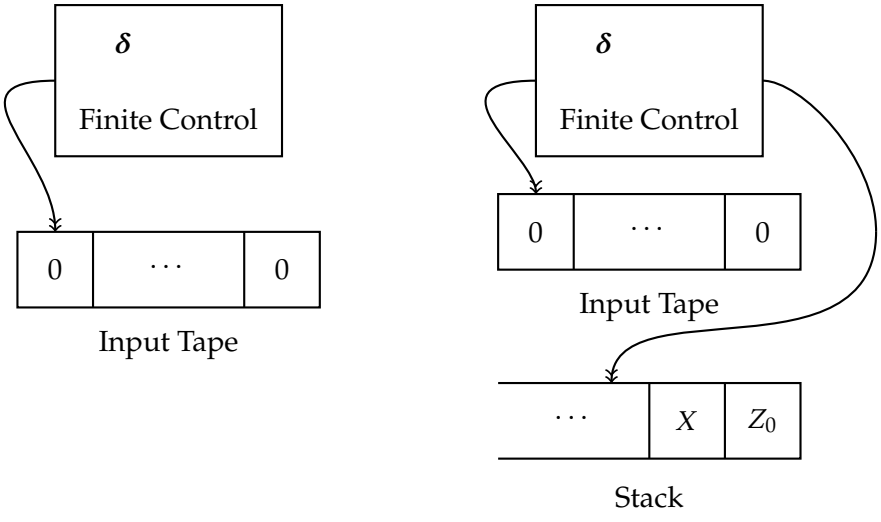
Example: parse tree for $c + c * c * (c + c)$, etc.

Lecture 25

25.1 Pushdown Automata

We've seen that we can use grammars to derive more powerful languages, but the process of 'deriving' strings from a grammar seems somewhat distinct from using an automaton to 'accept' a string. It's as if grammars require user intervention to choose a parse tree while automata run on their own. In fact, this distinction is superficial and we can construct a new type of automaton that recognizes context-free languages. We call these automata *pushdown*[1] *automata*.

A pushdown automaton is essentially an ε-NFA with one addition: it has a stack that it can use for side calculations.

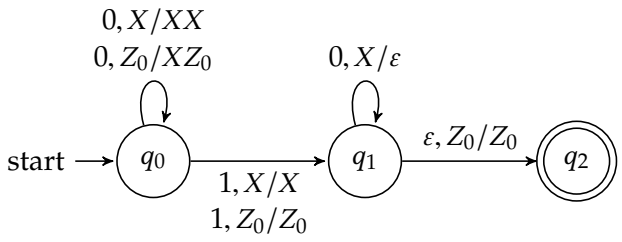


(a) A Finite Automaton

(b) A Pushdown Automaton

[1] pushdown comes from the British-English name for a stack — a pushdown store

Example 25.1.1 ($L = \{0^n 1 0^n \mid n \in \mathbb{N}\}$)**.**



A transition labeled as '$a, x/Y$' means that on reading symbol a, replace x at the top of the stack with Y

We can formalize this just as we did for other types of automata, however now there are more 'moving parts', so there are more elements in the tuple for PDAs. Things to notice from the above:

- The stack can have different symbols from the input (e.g. $\{Z_0, X\}$)
- The stack needs to have some start symbol (e.g. Z_0)
- The transition function has more information to account for pushing/popping the stack

25.2 Formal Definition

Definition 25.2.1 (Pushdown Automata)**.** A *pushdown automaton* P is a seven-tuple $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ where each component is as follows:

- Q is a finite *set of states*
- Σ is the *input alphabet*
- Γ is the *stack alphabet*
- $\delta : Q \times \Sigma \times \Gamma \to \mathcal{P}(Q \times \Gamma^*)$ is the *transition function*. Notice that now, δ takes as input a triple — one state, one input symbol, and one stack symbol — and returns a set of state/stack string pairs. The input stack symbol is popped off the stack, while the output string is pushed onto the stack.
- q_0 is the *start state*
- Z_0 is the *start stack symbol*
- F is the *set of accepting states*.

25.3 Instantaneous Descriptions

With the addition of a stack, tracing the computation of a PDA through its transition function is significantly more complex. To simplify this, we use the idea of a PDA transitioning from configuration to configuration, rather than merely from state to state. In order to represent the configuration, we need all the information about the PDA at any given time. The following three items suffice:

- $q \in Q$, the state that the PDA is in.
- $w \in \Sigma^*$, the unconsumed portion of the input.
- $\gamma \in \Gamma^*$, the contents of the stack. By convention, the topmost element of the stack is shown to the left.

Definition 25.3.1. An *instantaneous description* of a PDA is a 3-tuple consisting of the three elements listed above.

To represent chaining instantaneous descriptions as in the process of a computation, we create the turnstile relation '$\vdash$'

Definition 25.3.2. Let $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ be a PDA; $q \in Q$ be states; $a \in \Sigma$ be an input symbol; and $X \in \Gamma$ be a stack symbol. If $\delta(q, a, X)$ contains the pair (p, α), then we say $(q, aw, X\beta) \underset{P}{\vdash} (p, w, \alpha\beta)$ or just $(q, aw, X\beta) \vdash (p, w, \alpha\beta)$ if P is understood.

This is meant to reflect the idea that we consume a single symbol a, replace X on the stack with α, and move from state q to state p

Now we can define our analogue of the extended transition function: turnstar.

Definition 25.3.3.

Basis: $I \overset{*}{\vdash} I$ for any ID I

Induction: If $I \overset{*}{\vdash} J$ and $J \vdash K$ then $I \overset{*}{\vdash} K$.

Example 25.3.4. Consider PDA from Example 25.1.1 on input 00100. We can trace the computation with the following sequence of ID's:

$$\begin{aligned}(q_0, 00100, Z_0) &\vdash (q_0, 0100, XZ_0) \\ &\vdash (q_0, 100, XXZ_0) \\ &\vdash (q_1, 00, XXZ_0) \\ &\vdash (q_1, 0, XZ_0) \\ &\vdash (q_1, \varepsilon, Z_0) \\ &\vdash (q_2, \varepsilon, Z_0)\end{aligned}$$

25.4 Acceptance Criteria

Acceptance by final state

Definition 25.4.1. Let $P = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F \rangle$ be a PDA. Then define the *language accepted by P by final state* $L(P)$ as follows:

$$L(P) := \left\{ w \mid (q_0, w, Z_0) \overset{*}{\vdash} (q, \varepsilon, \alpha) \text{ where } q \in F \right\}$$

Acceptance by empty stack

Definition 25.4.2. Let P be a PDA as above. Then define *the language accepted by P by empty stack* $N(P)$ as follows:

$$N(P) := \left\{ w \mid (q_0, w, Z_0) \overset{*}{\vdash} (q, \varepsilon, \varepsilon) \right\}$$

Theorem 25.4.3. *Let L be any language. The following are equivalent:*

1. $L = L(P)$ *for some PDA P.*
2. $L = N(P)$ *for some PDA P.*
3. $L = L(G)$ *for some CFG G.*

Proof. We only prove (1) $\iff$ (2) for now.

($N(P) \Rightarrow L(P')$) The main idea is to start with a PDA $P_S = \langle Q, \Sigma, \Gamma, \delta_S, q_0, Z_0 \rangle$, which recognizes by empty stack, and add the transition '$\varepsilon, Z_0/\varepsilon$' from every state to a new accepting state to form P_A such that $L(P_A) = N(P_S)$. However, notice that there might already be transitions of this form, so naïvely adding in these transitions could result in a PDA that recognizes a different language. In order to deal with this ambiguity, we can change the start stack symbol and add a new start state to immediately push the old start stack symbol onto the stack. This is illustrated in figure 25.2.

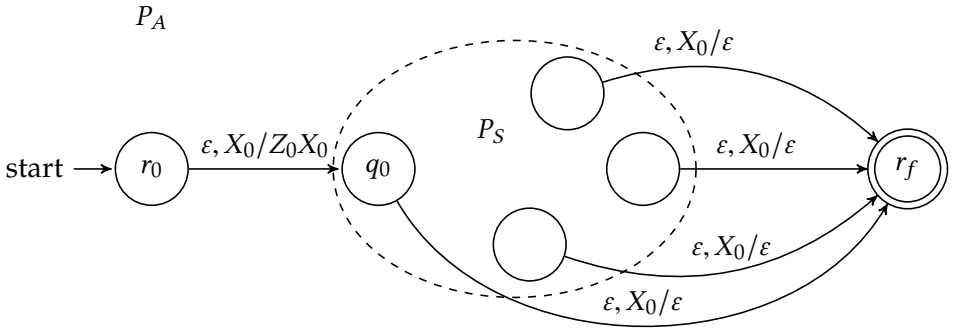


Figure 25.2

To show this works, we must define P_A formally:

$$P_A := \langle Q \cup \{r_0, r_f\}, \Sigma, \Gamma \cup \{X_0\}, \delta_A, r_0, X_0, \{r_f\} \rangle$$

where we construct δ_A as an extension of δ_S with the additional rules:

- $\delta_A(r_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}$.
- For all states $q \in Q$, inputs $a \in \Sigma \cup \{\varepsilon\}$ and stack symbols $Y \in \Gamma$, $\delta_A(q, a, Y) = \delta_S(q, a, Y) \cup (r_f, \varepsilon)$.

Now we must show for arbitrary $w \in \Sigma^*$, $w \in N(P_S) \iff w \in L(P_A)$.

(⇒) We assume that $(q_0, w, Z_0) \overset{*}{\underset{P_S}{\vdash}} (q, \varepsilon, \varepsilon)$ for some state q.

First notice that given an arbitrary PDA P, if $(q, x, \alpha) \overset{*}{\underset{P}{\vdash}} (r, y, \beta)$, it is also true that $(q, xw, \alpha\gamma) \overset{*}{\underset{P}{\vdash}} (r, yw, \beta\gamma)$ (this can be easily shown by induction). So, we can insert the new start stack symbol into the assumed computation: $(q_0, w, Z_0X_0) \overset{*}{\underset{P_N}{\vdash}} (q, \varepsilon, X_0)$. Combining this with the construction for P_A we have

$$(r_0, w, X_0) \underset{P_A}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_A}{\vdash}} (q, \varepsilon, X_0) \overset{*}{\underset{P_A}{\vdash}} (r_f, \varepsilon, \varepsilon)$$

So P_A accepts w by final state.

(⇐) This is clear from the fact that the only way to reach the accepting state is through a transition where the stack would have been empty in P_S.

($L(P) \Rightarrow N(P')$) We'll save this for next time.

□

Lecture 26

($L(P) \Rightarrow N(P')$) The basic idea is to add a 'drain' state that is reachable from all previous accepting states, whose only purpose is to empty the stack. This is illustrated in Figure 26.1

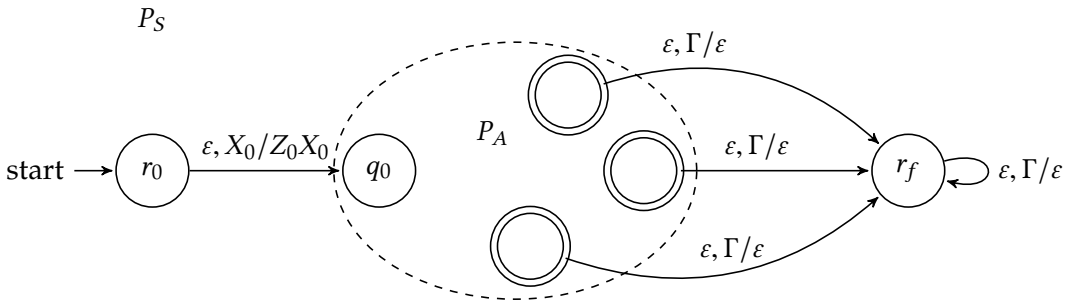


Figure 26.1

Notice that we need to add a fresh stack symbol X_0 to avoid P_A having an empty stack where it should not accept.

Theorem 26.0.1. *Let $L = L(P_A)$ for some PDA $P_A = \langle Q, \Sigma, \Gamma, \delta_A, q_0, Z_0, F\rangle$. Then there exists a PDA P_S such that $L = N(P_S)$*

Proof. The proof is just a formal treatment of Figure 26.1. Let

$$P_S = \langle Q \cup \{p_0, p\}, \Sigma, \Gamma \cup \{X_0\}, \delta_S, r_0, X_0, \emptyset\rangle$$

where we δ_S is an extension of δ_A with the following additions:

1. We start by pushing the old start stack symbol onto the stack, so

$$\delta_S(r_0, \varepsilon, X_0) = \{(q_0, Z_0X_0)\}.$$

2. All accepting states can enter the 'drain state' r_f if there is no input left:

$$(\forall q \in Q)(\forall Y \in \Gamma \cup \{X_0\}),$$
$$\delta_S(q, \varepsilon, Y) = \delta_A(q, \varepsilon, Y) \cup (r_f, \varepsilon).$$

3. The 'drain' state r_f empties the stack:

$$(\forall Y \in \Gamma \cup \{X_0\}), \delta_S(r_f, \varepsilon, Y) = \{(r_f, \varepsilon)\}.$$

To show the construction works, we must show for $w \in \Sigma^*$, $w \in N(P_S) \iff w \in L(P_A)$.

(⇒) Since we added a fresh stack symbol X_0 to the bottom of the stack, the only way to remove this symbol is to enter the drain state r_f, since any state 'inside' P_A can't reference X_0 by construction. So every accepting computation in P_S must be of the form:

$$(r_0, w, X_0) \underset{P_S}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_S}{\vdash}} (q, \varepsilon, \alpha X_0) \overset{*}{\underset{P_S}{\vdash}} (r_f, \varepsilon, \varepsilon)$$

for $q \in F$. But this must follow an accepting path in P_A, namely

$$(q_0, w, Z_0X_0) \overset{*}{\underset{P_S}{\vdash}} (q, \varepsilon, \alpha X_0)$$
$$\Rightarrow (q_0, w, Z_0) \overset{*}{\underset{P_A}{\vdash}} (q, \varepsilon, \alpha)$$

(⇐) We assume for some $q \in F, \alpha \in \Gamma^*$ that $(q_0, w, Z_0) \overset{*}{\underset{P_A}{\vdash}} (q, \varepsilon, \alpha)$. Since $\delta_A \subseteq \delta_F$ we can add an arbitrary stack string to the bottom of the stack on both sides:

$$(q_0, w, Z_0X_0) \overset{*}{\underset{P_S}{\vdash}} (q, \varepsilon, \alpha X_0)$$

Adding in the new start and 'drain' states gives

$$(r_0, w, X_0) \underset{P_S}{\vdash} (q_0, w, Z_0X_0) \overset{*}{\underset{P_S}{\vdash}} (q, \varepsilon, \alpha X_0) \overset{*}{\underset{P_S}{\vdash}} (r_f, \varepsilon, \varepsilon)$$

□

26.1 Pushdown Automata From Grammars

Now we aim to show that any grammar can be converted to an equivalent pushdown automata. The general idea is to simulate the derivations entirely in the stack of the PDA, so ultimately our PDA will have only one state. We use leftmost derivations out of convenience, since PDAs read their input left-to-right. Also, since the PDA has only one state, it is more convenient to use empty-stack as the acceptance criterion.

Definition 26.1.1. Consider a grammar $G = \langle V, T, P, S \rangle$. An arbitrary left-sentential form of this grammar looks like $xA\alpha$ for $x \in T^*$, $A \in V$, and $\alpha \in (V \cup T)^*$. Then we call $A\alpha$ the *tail* of this (left-)sentential form. If the sentential form is only terminals, its tail is ε.

Each step of the computation of the PDA should directly follow a step in the leftmost derivation of the input string. To do so, we want to guarantee that if the current sentential form of the derivation looks like $xA\alpha$ as above, then the PDA will have consumed x as input while the stack will consist of the tail of the derivation (with A at the top). Symbolically, if we use $w \in T^*, w = xy$ as input, then the sentential form $xA\alpha$ will correspond to the ID $(q, y, A\alpha)$.

Definition 26.1.2. Let $G = \langle V, T, Q, S \rangle$. Then construct the PDA

$$P = \langle \{q\}, T, V \cup T, \delta, q, S, \emptyset \rangle$$

where the transition function is constructed by the following rules:

1. For each $A \in V$

   $$\delta(q, \varepsilon, A) = \{(q, \beta) \mid `A \to \beta' \in Q\}$$

   Note that $\beta \in (V \cup T)^*$. This ensures the tails of sentential forms are the only strings to appear on the stack.

2. For each $a \in T$

   $$\delta(q, a, a) = \{(q, \varepsilon)\}$$

   This ensures we only pop terminal symbols off the stack.

Claim 26.1.3. For G and P as defined above, $L(G) = N(P)$.

Before we prove the claim, first it is helpful to see an example:

Example 26.1.4. Lets use a simplified version of the expression grammar we have used previously:

$$G = \langle \{E\}, \{v, c\}, Q, E \rangle$$

where Q consists of the single production

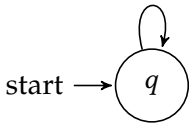
$$E \quad \to \quad E + E \mid E * E \mid (E) \mid v \mid c$$

So, we construct $P = \langle \{q\}, \{v, c\}, \{v, c, E\}, \delta, q, E, \emptyset \rangle$ with δ given by

$$\begin{aligned} \delta(q, \varepsilon, E) &= \{(q, E + E), (q, E * E), (q, (E)), (q, v), (q, c)\} \\ \delta(q, v, v) &= \{q, \varepsilon\} \\ \delta(q, c, c) &= \{q, \varepsilon\} \end{aligned}$$

Written graphically gives the following transition diagram:

$$\varepsilon, E/E + E$$
$$\varepsilon, E/E * E$$
$$\varepsilon, E/(E)$$
$$\varepsilon, E/v$$
$$\varepsilon, E/c$$
$$v, v/\varepsilon$$
$$c, c/\varepsilon$$



Proof of Claim 26.1.3. We aim to show $w \in L(G) \iff w \in N(P)$

(⇒) If $w \in L(G)$ then there is a leftmost derivation

$$\gamma_0 \underset{\ell m}{\Rightarrow} \gamma_1 \underset{\ell m}{\Rightarrow} \cdots \underset{\ell m}{\Rightarrow} \gamma_{n-1}$$

Where $S = \gamma_0$ and $w = \gamma_{n-1}$

Each γ_i is a left-sentential form, so we can break it into its head and tail:

$$\gamma_i = x_i \alpha_i$$

Letting y_i be the string such that $w = x_i y_i$, we aim to show for any $i \in \mathbb{N}, 0 \leq i < n$

$$(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$$

Basis: For $i = 0$, $\gamma_0 = S$, so $x_0 = \varepsilon$ and $y_i = w$. So we are trying to show that $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, w, S)$. But this is true by definition.

Induction: The inductive hypothesis is $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_i, \alpha_i)$, and we aim to show $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, y_{i+1}, \alpha_{i+1})$

Notice that α_i is a tail, so it begins with some stack symbol A. In addition, the derivation step $\gamma_i \underset{\ell m}{\Rightarrow} \gamma_{i+1}$ involves replacing A by the body of one of its productions, say β.

By construction of δ, we can replace A at the top of the stack by β, then consume any terminals on the stack by reading in their corresponding

input symbols. This produces the ID $(q, y_{i+1}, \alpha_{i+1})$, which corresponds to the next sentential form γ_{i+1}.

Finally, note $\alpha_{n-1} = \varepsilon$, since $\gamma_{n-1} = w$ and therefore the tail of γ_{n-1} is empty.

Therefore $(q, w, S) \overset{*}{\underset{P}{\vdash}} (q, \varepsilon, \varepsilon)$

(⇐) We will prove something more general, which will in turn prove this direction as a corollary.

We seek to show that if $(q, x, A) \overset{*}{\underset{P}{\Rightarrow}} (q, \varepsilon, \varepsilon)$, then $A \overset{*}{\underset{G}{\Rightarrow}} x$. That is, if reading in x while A is on the stack gives an accepting computation, then A derives x in the grammar. Again, we show this by induction (this time on the number of steps in the computation).

Basis: If the PDA reaches an accepting computation on one move, then the transition function must have $\delta(q, \varepsilon, A) = (q, \varepsilon)$ (since in δ moves on variables can only read ε). But this rule will only exist in the PDA if $A \to \varepsilon$ is a production in G, so clearly it is true that $A \Rightarrow \varepsilon$.

Induction: Here we assume P takes $n > 1$ moves. Since the nonterminal A is at the top of the stack, the first move must be of the form $\delta(q, \varepsilon, A) = (q, \gamma)$ where γ is some string in the stack alphabet.

Suppose $\gamma = Y_0 Y_1 \ldots Y_{k-1}$ (i.e in the grammar we used the production $A \to Y_0 Y_1 \ldots Y_{k-1}$) for each $Y_i \in (V \cup T)$. The rest of the moves of the PDA must consume x and ultimately pop each Y_i off the stack. Break up x into k strings so that $x = x_0 x_1 \ldots x_{k-1}$ where x_0 is the portion of input consumed until Y_0 is popped, then x_1 is the remaining input until Y_1 is popped, etc. Notice that if Y_i is a terminal, then $x_i = Y_i$.

From these definitions, we know the following:

$$(q, x_i, Y_i) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)$$

which by necessity takes a smaller number of steps than the computation we are using for the proof, so by the inductive hypothesis we conclude

$$Y_i \overset{*}{\Rightarrow} x_i$$

Now we have the following derivation

$$\begin{aligned} A &\overset{*}{\underset{\ell m}{\Rightarrow}} Y_0 Y_1 \dots Y_{k-1} \\ &\overset{*}{\underset{\ell m}{\Rightarrow}} x_0 Y_1 \dots Y_{k-1} \\ &\vdots \\ &\overset{*}{\underset{\ell m}{\Rightarrow}} x_0 x_1 \dots x_{k-1} \end{aligned}$$

i.e.

$$A \overset{*}{\Rightarrow} x$$

Letting $A = S$ and $x = w$, by assumption we have $(q, w, S) \overset{*}{\vdash} (q, \varepsilon, \varepsilon)$ and by the above we have $S \overset{*}{\Rightarrow} w$.

□

26.2 An Alternative Proof

Let's do an alternative proof that PDAs and CFGs are the same thing.

Theorem 26.2.1. *Let L be a context-free-language. Then there exists a PDA M such that $L = N(M)$, the language accepted by M by empty stack.*

Proof. Let $G = (V_N, V_T, P, S)$ be a CFG. We will assume that ε is not in G (extending the proof to account for this case is not hard and we will skip it). We will assume that the productions P of G are all in *Greibach normal form* (GNF). This means that all productions are of the form

$$A \to aB_1 \dots B_n$$

or

$$S \to \varepsilon$$

where A and the B_i are variable symbols and a is a terminal symbol. We will not prove that any CFG can be converted to a CFG in GNF. This also isn't actually hard.

We define the PDA to be

$$M = (\{q_1\}, V_T, V_N, \delta, q_1, S, \emptyset)$$

where $\delta(q_1, a, A)$ contains (q_1, γ) whenever $A \to a\gamma$ is a production in P.

(Note that the PDA only needs one state.)

What this does, obviously, is set up a PDA that mimics the derivation of strings in the grammar.

Now, consider a production

$$A \to a\alpha$$

where α is a sequence of variable symbols as mentioned above as the requirement for being in Greibach normal form.

Then we have, for any strings x and β, that the use of this production, namely

$$xA\beta \underset{G}{\to} xa\alpha\beta$$

corresponds exactly with transitions in M

$$a : (q_1, A\beta) \vdash (q_1, \alpha\beta)$$

We can now induct on the number of steps in a derivation in the grammar that, for $x, y \in V_T^*$, $A \in V_N$, $\alpha, \beta \in V_N^*$, a multi-step derivation

$$xA\beta \overset{*}{\to} xy\alpha$$

happens if and only if we can mimic it exactly in the PDA:

$$y : (q_1, A\beta) \overset{*}{\vdash} (q_1, \alpha)$$

This means that a string $x \in V_T^*$ is derived in G from the start state if and only if the PDA reads x and empties the stack. □

And now we do the other direction.

Theorem 26.2.2. *If L is the language accepted by empty stack by some PDA M, the $L = L(G)$ for some context free grammar.*

Proof. Let

$$M = (K, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset)$$

be the PDA and let and

$$G = (V_N, \Sigma, P, S)$$

be the context free grammar. We let V_N be the set of objects of the form

$$[q, A, p]$$

where q and p are states in K and A is in Γ, plus the new symbol S. The productions in P are

1. $S \to [q_0, Z_0, q]$ for each $q \in K$.

2.
$$[q, A, p] \rightarrow a[q_1, B_1, q_2]...[q_m, B_m, q_{m+1}]$$
for every sequence of states $q, q_1, ..., q_{m+1}$ in K, where $p = q_{m+1}$, each $a \in \Sigma \cup \varepsilon$, and $A, B_1, ..., B_m \in \Gamma$ such that
$$(q_1, B_1...B_m) \in \delta(q, a, A)$$
In the case that $m = 0$, we have $p = q_1$, $(p, \varepsilon \in \delta(q, a, A)$, and the production is just
$$[q, A, p] \rightarrow a$$

Intuitively, the productions have been constructed so that a derivation from left to right (of variable symbols) in G of a sentence x is a simulation of the PDA M when presented with the input string x.

The idea is that the terminal symbols generated by the grammar are exactly the symbols read by the PDA, and we use the stack of the PDA to store up the variable symbols as they are produced by the grammar.

We prove the $L(G) = N(M)$ by (what else?) induction on the number of steps in the derivation in G, which is equivalent to the number of moves using δ in M. We need to prove that
$$[q, A, p] \stackrel{*}{\Rightarrow} x \text{ if and only if } x : (q, A) \stackrel{*}{\vdash} (p, \varepsilon)$$

If we have that $x \in L(G)$, then we have productions
$$S \stackrel{*}{\Rightarrow} [q_0, Z_0, q] \stackrel{*}{\Rightarrow} x$$
for some state q.

But that means that we can continue in the PDA
$$x : (q_0, Z_0) \stackrel{*}{\vdash} (q, \varepsilon)$$
which means that $x \in N(M)$. Going the other direction is almost the same. If we have $x \in N(M)$ then we have
$$x : (q_0, Z_0) \stackrel{*}{\vdash} (q, \varepsilon)$$
This means we can prepend the start symbol production and have
$$S \stackrel{*}{\Rightarrow} [q_0, Z_0, q] \stackrel{*}{\Rightarrow} x$$
so $x \in L(G)$.

□

Lecture 27

Give an example using the unambiguous arithmetic expression grammar, giving an accepting execution trace for the expression $c * (c + c)$.

For (2) $\implies$ (3), we make a modification to the book: a *restricted PDA* is one that can only push or pop a single symbol on every transition.

Definition 27.0.1. A *restricted PDA* is a PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ such that, for every $q \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, and $X \in \Gamma$, the only elements of $\delta(q, a, X)$ are of the following two forms:

1. (r, YX) for some $r \in Q$ and $Y \in \Gamma$, or
2. (r, ε) for some $r \in Q$.

A transition of form (1.) we call *push* Y and abbreviate it $(r, \text{push } Y)$. A transition of form (2.) we call *pop* and abbreviate it (r, pop).

This does not decrease the power of a PDA. Restricted PDAs can recognize the same languages as general PDAs.

Lemma 27.0.2. *For every PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, there is a restricted PDA P' with the same input alphabet Σ such that $L(P') = L(P)$ and $N(P') = N(P)$.*

Proof sketch. In this proof (and more generally), the adjective "fresh" refers to an object that has not appeared before or been mentioned before. The stack alphabet of P' is $\Gamma' := \Gamma \cup \{X_0\}$, where X_0 is a fresh symbol (i.e., $X_0 \notin \Gamma \cup \Sigma$) that is also the bottom stack marker used by P'. The state set Q' of P' includes all the states in Q together with a fresh state $p_0 \notin Q$ used as the start state of P' and another fresh state e, as well as other fresh states described below. The final states of P' are those of P. Thus $P' := (Q', \Sigma, \Gamma', \delta', p_0, X_0, F)$, where the transitions of δ' are of the following types:

1. $\delta'(p_0, \varepsilon, X_0) := \{(q_0, \text{push } Z_0)\}$;
2. for all $q \in Q$, $\delta'(q, \varepsilon, X_0) := \{(e, \text{pop})\}$;

3. for every transition $(r, \gamma) \in \delta(q, a, X)$, where $q, r \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, $X \in \Gamma$, and $\varepsilon \neq \gamma = Y_k \cdots Y_1$ for some $k \geq 1$ and $Y_1, \ldots, Y_k \in \Gamma$, we replace this transition in δ' as follows: introduce fresh states $s_0, \ldots, s_{k-1}$, and, setting $s_k := r$, let $\delta'(q, a, X) := \{(s_0, \text{pop})\}$. In addition, for all $1 \leq i \leq k$ and all $Y \in \Gamma'$, include the transition $\delta'(s_{i-1}, \varepsilon, Y) := \{(s_i, \text{push } Y_i)\}$.

4. All other sets $\delta'(q, a, X)$ are empty.

The idea in (3.) is that instead of replacing X by γ on the stack all at once, we cycle through some new intermediate states, first popping X then pushing on γ one symbol at a time, eventually arriving at state r. Note that if $\gamma = \varepsilon$, then the existing transition is already a pop and need not be replaced. Having X_0 always on the bottom of the stack (and nowhere else) ensures that we don't empty the stack by popping X. The only way of getting X_0 itself popped is by making a transition to state e, after which one cannot move.

It is not horrendously difficult to prove by induction on the number of steps of the trace that

$$(q, w, \alpha X_0) \vdash_{P'}^* (r, \varepsilon, \beta X_0) \iff (q, w, \alpha) \vdash_P^* (r, \varepsilon, \beta) \tag{27.1}$$

for all $q, r \in Q$, $w \in \Sigma^*$, and $\alpha, \beta \in \Gamma$. It follows from this that, for all $w \in \Sigma^*$,

$$\begin{aligned} w \in L(P') &\iff (\exists r \in F)(\exists \gamma \in \Gamma^*)[(p_0, w, X_0) \vdash_{P'}^* (r, \varepsilon, \gamma X_0)] \\ &\iff (\exists r \in F)(\exists \gamma \in \Gamma^*)[(q_0, w, Z_0 X_0) \vdash_{P'}^* (r, \varepsilon, \gamma X_0)] \\ &\iff (\exists r \in F)(\exists \gamma \in \Gamma^*)[(q_0, w, Z_0) \vdash_P^* (r, \varepsilon, \gamma)] \\ &\iff w \in L(P) \ . \end{aligned}$$

The first equivalence follows from the definition of final-state acceptance in P' (remember that X_0 remains on the bottom of the stack in all states except e). The second equivalence takes into account the initial transition from p_0 to q_0 pushing Z_0. The third equivalence is just (27.1) above, and the last equivalence is the definition of final-state acceptance in P.

Similarly,

$$\begin{aligned} w \in N(P') &\iff (\exists r \in Q')[(p_0, w, X_0) \vdash_{P'}^* (r, \varepsilon, \varepsilon)] \\ &\iff (p_0, w, X_0) \vdash_{P'}^* (e, \varepsilon, \varepsilon) \\ &\iff (\exists r \in Q)[(p_0, w, X_0) \vdash_{P'}^* (r, \varepsilon, X_0)] \\ &\iff (\exists r \in Q)[(q_0, w, Z_0 X_0) \vdash_{P'}^* (r, \varepsilon, X_0)] \\ &\iff (\exists r \in Q)[(q_0, w, Z_0) \vdash_P^* (r, \varepsilon, \varepsilon)] \\ &\iff w \in N(P) \ . \end{aligned}$$

The first equivalence is the definition of empty-stack acceptance in P'. The second follows from the fact that e is the only state of P' at which the stack can be empty.

The third follows from the fact that all transitions to e pop X_0 (and this can happen from any state in Q). The fourth takes into account the initial transition from p_0 to q_0 pushing Z_0. The fifth equivalence uses (27.1) again, and the last is the definition of empty-stack acceptance in P.

So we have $L(P') = L(P)$ and $N(P') = N(P)$. □

Now back to showing (2) $\implies$ (3) in Theorem 25.4.3. By Lemma 27.0.2, it suffices to define a grammar equivalent to a given restricted PDA using empty-stack acceptance. Suppose we are given a restricted PDA $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0)$ (the final states are irrelevant). Our grammar $G_P = (V, T, P, S)$ has the following ingredients:

- a special start symbol S,
- terminal set $T := \Sigma$,
- variables (other than S) of the form $[pXq]$ for all states $p, q \in Q$ and stack symbols X (note that we treat this as a single variable symbol),
- The following productions:

  1. for every state $r \in Q$, the production

     $$S \to [q_0 Z_0 r]$$

     (these are the only productions with head S),

  2. for every transition $(r, \text{pop}) \in \delta(q, a, X)$, where $q, r \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, and $X \in \Gamma$, the production

     $$[qXr] \to a$$

     and

  3. for every transition $(r, \text{push } Y) \in \delta(q, a, X)$, where $q, r \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, and $X, Y \in \Gamma$, the productions

     $$[qXt] \to a[rYs][sXt]$$

     for all states $s, t \in Q$.

The idea of the variable $[pXq]$ is to generate exactly those strings in Σ^* that the PDA can read going from state p to state q, where the net effect on the stack is having the single symbol X popped off at the end. That is, we want the following equivalence for all states $p, q \in Q$, stack symbols X, and strings $w \in \Sigma^*$:

$$[pXq] \stackrel{*}{\Rightarrow} w \iff (p, w, X) \vdash^* (q, \varepsilon, \varepsilon) \,. \tag{27.2}$$

This can be proved by induction, and it follows from this and the S-productions that

$$\begin{aligned} w \in L(G_P) &\iff S \overset{*}{\Rightarrow} w \\ &\iff (\exists r \in Q)[[q_0 Z_0 r] \overset{*}{\Rightarrow} w] \\ &\iff (\exists r \in Q)[(q_0, w, Z_0) \vdash^* (r, \varepsilon, \varepsilon)] \\ &\iff w \in N(P) \ . \end{aligned}$$

So $L(G_P) = N(P)$ as desired.

We'll start with a simple PDA as an example of this construction. Let

$$P = (\{q, p\}, \{0, 1\}, \{X, Z_0\}, \delta, q, Z_0) \ ,$$

where

1. $\delta(q, 0, Z_0) = \{(q, \text{push } X)\}$.
2. $\delta(q, 0, X) = \{(q, \text{push } X)\}$.
3. $\delta(q, 1, X) = \{(p, \text{pop})\}$.
4. $\delta(p, 1, X) = \{(p, \text{pop})\}$.
5. $\delta(p, \varepsilon, Z_0) = \{(p, \text{pop})\}$.

One can check that $N(P) = \{0^n 1^n \mid n \geq 1\}$. The grammar G_P then has the following productions:

$$\begin{aligned} S &\to [qZ_0q] \mid [qZ_0p] \\ [qXp] &\to 1 \\ [pXp] &\to 1 \\ [pZ_0p] &\to \varepsilon \\ [qZ_0q] &\to 0[qXq][qZ_0q] \mid 0[qXp][pZ_0q] \\ [qZ_0p] &\to 0[qXq][qZ_0p] \mid 0[qXp][pZ_0p] \\ [qXq] &\to 0[qXq][qXq] \mid 0[qXp][pXq] \\ [qXp] &\to 0[qXq][qXp] \mid 0[qXp][pXp] \end{aligned}$$

It will be easier to read if we rename the variables by single letters: $A = [qXp]$, $B = [pXp]$, $C = [pZ_0p]$, $D = [qZ_0q]$, $E = [qZ_0p]$, $F = [qXq]$, $G = [pZ_0q]$, and $H = [pXq]$:

$$S \to D \mid E$$
$$A \to 1 \mid 0FA \mid 0AB$$
$$B \to 1$$
$$C \to \varepsilon$$
$$D \to 0FD \mid 0AG$$
$$E \to 0FE \mid 0AC$$
$$F \to 0FF \mid 0AH$$

This grammar can be simplified a lot. Notice that there are no G- or H-productions; this means that if either G or H show up in any sentential form, they can never disappear, and so no string of all terminals can be derived. This means that the second D-production and the second F-production are useless and can be removed. Also, since B only derives 1 and C only derives ε, we can bypass these two productions, substituting 1 and ε directly for B and C respectively in the bodies of the other productions:

$$S \to D \mid E$$
$$A \to 1 \mid 0FA \mid 0A1$$
$$D \to 0FD$$
$$E \to 0FE \mid 0A$$
$$F \to 0FF$$

Now notice that if F ever shows up in any sentential form, it can never disappear. Thus any productions involving F are useless and can be removed:

$$S \to D \mid E$$
$$A \to 1 \mid 0A1$$
$$E \to 0A$$

Removing F eliminated the only remaining D-production, and so any productions involving D are useless and can be removed:

$$S \to E$$
$$A \to 1 \mid 0A1$$
$$E \to 0A$$

Finally, the only places where E occurs are in the two productions $S \to E$ and $E \to 0A$, and so we can bypass the E-production entirely:

$$S \to 0A$$
$$A \to 1 \mid 0A1$$

Now it should be evident that the language of this grammar is indeed $N(P) = \{0^n 1^n \mid n \geq 1\}$.

Lecture 28

28.1 Pumping Lemma For CFGs

The pumping lemma for context-free languages: proof and applications

Example 28.1.1. $L = \{a^m b^n c^m d^n \mid m, n \geq 0\}$

Example 28.1.2. $L = \{a^n b^n c^n \mid n \geq 0\}$

Example 28.1.3. $L = \left\{a^j b^k c^\ell \mid 0 \leq j \leq k \leq \ell\right\}$

Lemma 28.1.4 (Pumping Lemma for CFLs)**.** *Let L be any context-free language. There exists $p > 0$ such that, for any string $s \in L$ with $|s| \geq p$, there exist strings v, w, x, y, z such that: (i) $s = vwxyz$, (ii) $|wxy| \leq p$, (iii) $|wy| > 0$ (i.e., $wy \neq \varepsilon$); and for all $i \geq 0$, $vw^i xy^i z \in L$.*

Proof. Since L is context-free, there exists a CFG G such that $L = L(G)$. Let n be the number of nonterminals of G, and let d be the maximum of 2 and the body length of any production of G. Note that parse trees of G have branching at most d, and so any parse tree of depth $\leq n$ has $\leq d^n$ many leaves.

Let $p := d^{n+1}$. Given any string $s \in L$ such that $|s| \geq p$, let T be a minimum-size parse tree of G yielding s. Since $|s| \geq p > d^n$, T must have depth $\geq n + 1$. Let q be a maximum-length path in T from the root to a leaf. Since q has maximum length, the internal nodes of q, starting at the bottom, have heights $1, 2, 3, \ldots$, that is, there are no skips in the heights; the height of a node along q is given by the length of q below that node. Thus the first $n + 1$ internal nodes along q, counting up from the leaf, all have height $\leq n + 1$. By the pigeonhole principle, some nonterminal A of G is repeated among the internal nodes of heights $\leq n + 1$ along q. Let A_1 and A_2 be two such nodes both labeled A, of heights h_1 and h_2, respectively, and assume that $h_1 < h_2$ (and we know that $h_2 \leq n + 1$).

Now define v, w, x, y, z to be the following strings:

- v is the portion of T's yield that lies to the left of the yield of (the subtree rooted at) A_2.

- w is the portion of A_2's yield that lies to the left of the yield of A_1.
- x is the yield of A_1.
- y is the portion of A_2's yield that lies to the right of the yield of A_1.
- z is the portion of T's yield that lies to the right of the yield of A_2.

Then clearly, $vwxyz = s$, which is the yield of T. Moreover, wxy is the yield of A_2, and because A_2's tree has depth h_2, it follows that $|wxy| \leq d^{h_2} \leq d^{n+1} = p$. We save the verification that $|wy| > 0$ for last.

Let W be the "wedge" obtained from the tree at A_2 by pruning at A_1. W has yield wy. Let T_0 be the tree obtained from T by removing W and grafting the tree at A_1 onto A_2. Then T_0 is a parse tree of G yielding $vxz = vw^0xy^0z$. This shows that $vw^0xy^0z \in L$. For any $i > 0$, let T_i be the tree obtained from T_0 by inserting i many copies of W, one on top of another, starting at A_2, and grafting on A_1's tree to the bottommost copy of W. Then T_i is a parse tree of G yielding vw^ixy^iz, and hence the latter string is also in L. This shows that $vw^ixy^iz \in L$ for all $i \geq 0$.

Finally we verify that $|wy| > 0$. Suppose $|wy| = 0$. Then $w = y = \varepsilon$, and so $s = vxz$, which is the yield of T_0. But T_0 is strictly smaller than T, which contradicts the choice of T as a minimum size tree yielding s. Thus $|wy| > 0$. □

Working arithmetic expression evaluator in C?

Lecture 29

29.1 Turing Machines

What we now refer to as a Turing Machine was invented for the purpose of analyzing "that which is considered to be 'computation' ". The Turing Machine quite literally *is* the abstract notion of a "computer", as summarized by what we call the *Church-Turing Thesis*, after Alonzo Church and Alan Turing:

> Every action that we would intuitively think to be a "computation" can be computed by a Turing Machine.

Definition 29.1.1. A TM is a tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- Q is a finite set (the *state set*),
- Σ is an alphabet (the *input alphabet*),
- Γ is an alphabet (the *tape alphabet*), and we have $\Sigma \subseteq \Gamma$ (by relabeling if necessary, we also can assume that $\Gamma \cap Q = \emptyset$),
- δ is the *transition function*, a partial function $Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$,
- $q_0 \in Q$ is the *start state*,
- $B \in \Gamma - \Sigma$ is the *blank symbol*, and
- $F \subseteq Q$ is the set of *accepting, or final, states*.

This is similar to, but more general than, previous notions of automata and of abstract versions of "computing devices". The TM has a tape head positioned over some symbol on the tape, which extends infinitely to the left and write. From any given state, reading the symbol over which the tape head is positioned, the TM writes a symbol back onto the tape (this could be the same symbol already there), moves to the right or to the left, and changes state (which could be the same state).

As we have done throughout this course, we will be interested in a computational process that reads an input string and possibly transitions eventually to an accepting/final state. If this happens, the TM will be said to *halt and accept the input string*. If the TM reaches an accepting state, it won't read any more symbols, and it won't care if it has read all the symbols or if there are any symbols still on the tape.

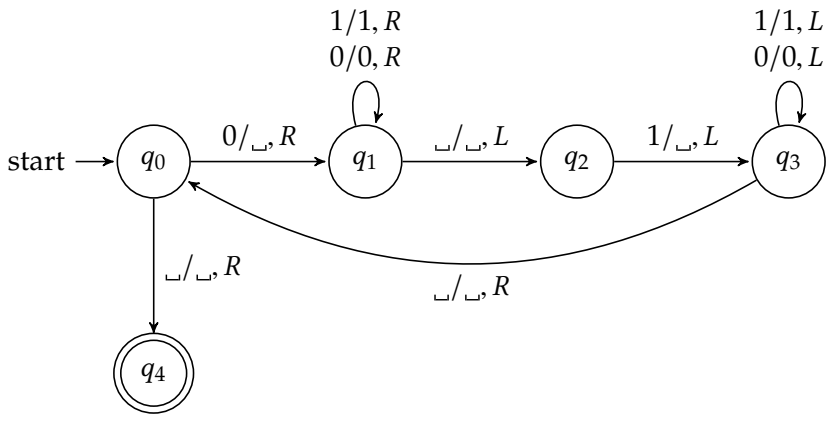
We note that the actions of TMs that halt on given inputs are *finite but unbounded*. That is, since the TM halts, it will halt after a finite number of steps, and thus it will have read only finitely many symbols from the tape. But there is no integer K that bounds the number of steps or the number of symbols that are read; given any fixed K, we can describe a TM and an input string such that the TM reads more than K distinct symbols and takes more than K steps before it halts.

29.2 Examples

Given the Church-Turing thesis, we would expect that for anything we would think of as a computation, we should be able to devise a TM that performs that computation. Since we have said above that a TM is a more general notion of computation than what we have seen before, we should be able to compute things with a TM that we could not compute will less general devices.

Example computations: recognizing $\{0^n1^n \mid n \geq 0\}$, recognizing palindromes, etc. Basic ops: moving a block down the tape (to make room), copying a string, reversing a string, binary increment/decrement, converting unary to binary and vice versa, unary and binary addition, unary multiplication, etc., proper subtraction, monus, etc. (spill over to next lecture)

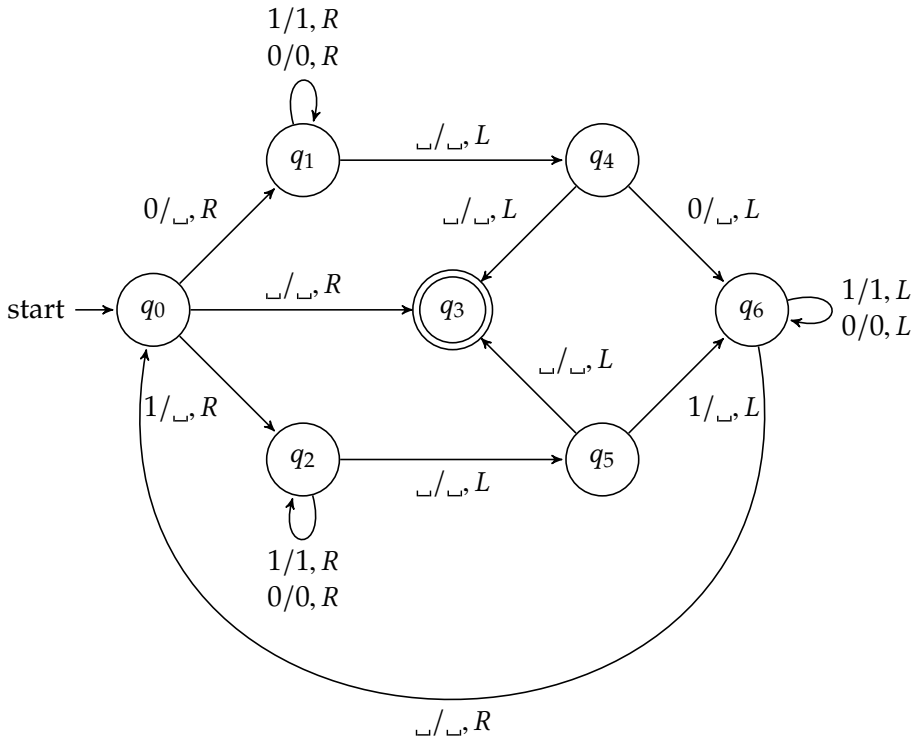
- Recognizing $\{0^n1^n | n \geq 0\}$



  - In q_0, reading 0, write B, change to q_1, and move R. (We have now read and blanked the leftmost 0.)

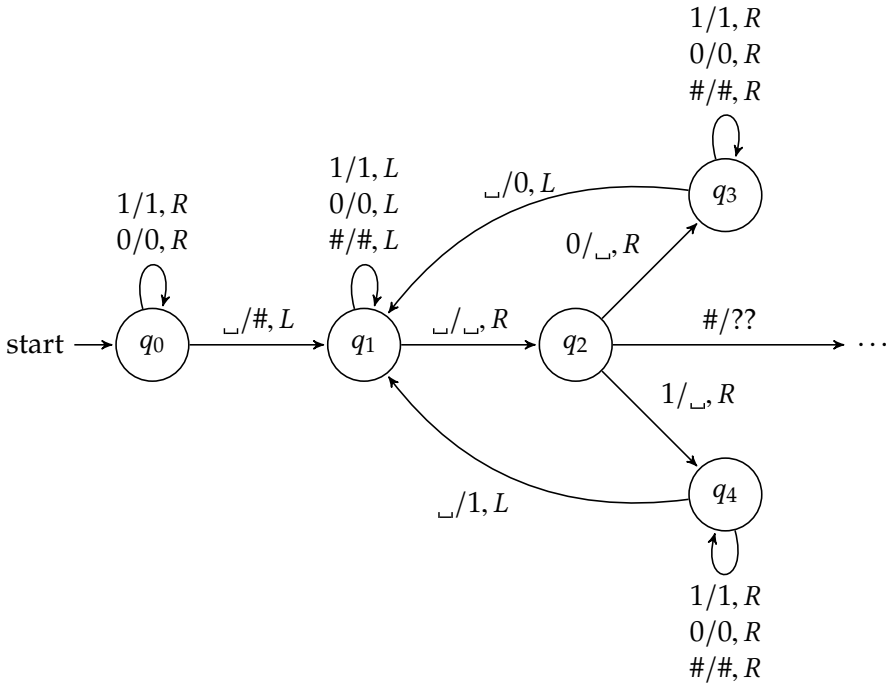
- In q_1, reading 0 or 1, write that back, stay in q_1, and move R. (Read and move past all the zeros and ones until we move right to the blank that is just beyond the input data.)
- In q_1, reading B, write B, change to q_2, and move L. (We are at the blank just past the rightmost non-blank. Change state to reflect that fact and move to the position of the rightmost non-blank.)
- In q_2, reading 1, write B, change to q_3, and move L. (We have now read and blanked the rightmost 1.)
- In q_3, reading 0 or 1, write that back, stay in q_3, and move L. (Read and move past all the zeros and ones until we move left to the blank that is just before the ((now modified)) input data.)
- In q_3, reading B, write B, change to q_0, and move R. (Go back to the top of the loop.)
- In q_0, reading B, write B, change to q_4, and move R. (q_4 is the one final state; we're done.)

- Recognizing Palindromes



Very similar to the previous example. We need to read the leftmost symbol, write the blank, and retain the knowledge (by moving to one of two different states) of whether we read a 0 or a 1. We then move all the way to the blank just past the (possibly modified) input data. If the rightmost symbol matches the state we are in, that has retained the knowledge of what we read as the leftmost symbol, we blank the rightmost symbol, change to a "moving left" state, and start moving left. We match up all symbols (meaning we had a palindrome) if and only if we blank the entire input string, and if so, we move to a final state.

- Moving a block



Note: copying and reversing a string is very similar to the above.

- Copying a string
  - In q_0, reading 0, write X, change to q_{X1}, and move R. (Remember what we have read, write something we can recognize later, and move right.)
  - In q_0, reading 1, write Y, change to q_{Y1}, and move R. (Same as just above.)
  - In q_0, reading B, write B, change to q_F, and move R. (We will finish with the tape head positioned over the first symbol of the copied string.)

  - In q_{X1} or q_{Y1}, reading 0 or 1, write that back, stay in the same state, and move R. (Move past zeros and ones looking for the end of the original string.)
  - In q_{X1}, reading B, write that back, change to q_{X2}, and move R.
  - In q_{Y1}, reading B, write that back, change to q_{Y2}, and move R.
  - In q_{X2} or q_{Y2}, reading 0 or 1, write that back, stay in the same state, and move R. (Move past zeros and ones looking for the end of the partially copied string.)
  - In q_{X2}, reading B, write 0, change to q_Z, and move L.
  - In q_{Y2}, reading B, write 1, change to q_Z, and move L.
  - In q_Z, reading 0 or 1, write that back, stay in the same state, and move L. (Move past zeros and ones in the copied string, looking for the blank that separates the copied from the original.)
  - In q_Z, reading B, write that back, change to q_W, and move L. (Now we're going to read left in the original string.)
  - In q_W, reading 0 or 1, write that back, stay in the same state, and move L. (Move past zeros and ones in the original string, looking for the last symbol that we copied.)
  - In q_W, reading X, write 0, change to q_0, and move R. (Put back the symbol we remembered, and get ready to iterate.)
  - In q_W, reading Y, write 1, change to q_0, and move R. (ditto)

- Unary addition

- Binary increment on $\mathbb{N}$ (i.e., adding 1 to a binary number)

  We write the number from left to right and not right to left so the possible last carry makes the string of bits longer to the right.

  - In q_0, reading 0, write 1, change to q_D, and move L. (We have incremented by 1, there is no carry, and we are Done. Note that this should only happen if the initial input is just a single 0.)
  - In q_0, reading 1, write 0, change to q_C, and move R. (We have incremented by 1, and there is a Carry.)
  - In q_C, reading 0, write 1, change to q_D, and move L. (The carry does not propagate any further than here, so we are Done.)
  - In q_C, reading 1, write 0, stay in q_C, and move R. (The carry propagates further.)

  - In q_C, reading B, write 1, change to q_D, and move L. (The carry has propagated to make the binary number one more digit in length, but we are now Done.)
  - In q_D, reading 0 or 1, write that back, stay in q_D, and move L. (Read and move past all the zeros and ones until we move left to the blank that is just before the ((now modified)) input data.)
  - In q_D, reading B, write that back, change to q_F, and move R. (This puts us back to the initial position at the ones digit.)

  Note that this increments by one and then accepts. If we wanted this to be a subprogram, we could change in the last step to q_0 instead of q_F so as to set up for doing it again.

- Binary decrement on $(\mathbb{N} - 0)$ (ones complement, then add one, then ones complement)

Lecture 30

30.1 Instantaneous Descriptions (IDs) of a TM computation

State diagrams and/or transition tables are cumbersome for TM's, and they don't include the tape. We need some other notation for describing the current state of the machine.

Notice: even though TMs may run for an infinitely long time, if we ask how many cells were visited after n steps this must be some finite number. Therefore, at any given step in the computation there are only a finite number of (non-blank) symbols on the tape. To fully describe the TM state then, we need:

- The current state q
- The position the TM is currently scanning
- The full contents of the tape

We represent this as $X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_m$ where the TM is in state q and each X_j is a symbol on the tape listed in order, where the TM will scan symbol X_i to determine the next transition.

To represent transitions of a TM M to a new configuration, we use the $\vdash_M$ operator. If the TM we are talking about is obvious, we just use $\vdash$ For example, suppose for the previous configuration $\delta(q, X_i) = (p, Y, L)$. We can represent this as the *instantaneous description* (ID)

$$X_1X_2\cdots X_{i-1}qX_iX_{i+1}\cdots X_m \vdash X_1X_2\cdots X_{i-2}pX_{i-1}YX_{i+1}\cdots X_m$$

Be careful of the edge cases:

- If the above TM was scanning the leftmost symbol, then we would have to "add" a blank to the new configuration:

$$qX_1X_2\cdots X_m \vdash qBYX_2\cdots X_m$$

- If we were scanning the rightmost symbol and $Y = B$, then the new blank would be "absorbed" by the rest of the tape

$$X_1 \cdots X_{m-1} q X_m \vdash X_1 \cdots q X_{m-1}$$

NOTE:$A \vdash B$ is commonly read as "A turnstile B", "A yields B", "B is derivable from A" or "B is provable by A"

Similar to the extended transition function, we define $\overset{*}{\vdash}$ to represent zero or moves of the TM

Definition 30.1.1. Instantaneous descriptions are defined inductively

Basis: $I \overset{*}{\vdash} I$ for any ID I

Induction: $I \overset{*}{\vdash} J$ if there exists some ID K such that $I \vdash K$ and $K \overset{*}{\vdash} J$

Definition 30.1.2. Given a Turing machine $M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle$, the language it recognizes is defined as

$$L(M) \coloneqq \left\{ w \in \Sigma^* \mid q_0 w \overset{*}{\vdash} \alpha p \beta; \alpha, \beta \in \Gamma^*; p \in F \right\}$$

In other words, beginning in the start state the TM M reaches a final state after reading in w. What is left on the tape is irrelevant.

Definition 30.1.3. A language is *Turing recognizable* if some Turing machine recognizes it.

Because we defined rejection to be either "halt and reject" or loop, we may not be able to guarantee that the machine halts on strings outside of the language.

Definition 30.1.4. A *decider* is a Turing machine that halts on all possible inputs. Languages recognized by deciders are called *decidable*

TMs can also act as transducers, where instead of acceptance/rejection they compute functions (this was how Turing originally envisioned them).

Definition 30.1.5. A Turing machine $M = \langle Q, \Sigma, \Gamma, \delta, q_0, B, F \rangle$ defines a (possibly partial) function $f : \Sigma^* \to \Gamma^*$. For any string $w \in \Sigma^*$, if p is a "halting state" then

$$f(w) \coloneqq \alpha\beta \qquad \text{where } q_0 \overset{*}{\vdash} \alpha p \beta$$

TM tricks: addition, proper subtraction (monus), multiplication? Maintaining lists, moving strings around, etc. Marking with symbols (example comparing two binary numbers), remembering data in the state, etc.

Examples: Converting between unary and binary (requires binary increment and decrement). Simulating a two-way infinite tape with a one-way infinite tape (with end marker). Comparisons (binary and unary).

Church-Turing thesis: TMs capture our intuitive notion of computation. Anything conceivable as "computation" can be done by a TM, and vice versa.

Lecture 31

TEXT TO BE REPLACED BY NEW TEXT:

> Encoding problem inputs as strings. Any finite object can be encoded as a string, including numbers, graphs, finite lists of finite objects, strings over another, perhaps bigger, alphabet, etc., even descriptions of finite automata and TMs, themselves. For any finite object O, let $\langle O \rangle$ be a string encoding O in some reasonable way (varying with the type of object). Example: encoding a TM as a string. Thus TMs can be inputs (and outputs) of TMs!
>
> Universal TMs: served as the inspiration for stored-program electronic computers. Your computer's hardware is "essentially" a universal TM.
>
> The diagonal halting problem (language)
>
> $$A_D := \{\langle M \rangle \mid M \text{ is a TM that eventually accepts on input } \langle M \rangle\}$$
>
> **Theorem 31.0.1.** *A_D is undecidable.*
>
> (The proof uses Cantor-style diagonalization.)
>
> **Corollary 31.0.2.** *$A_T := \{\langle M, w \rangle \mid M \text{ accepts on input } w\}$ is undecidable*

31.1 Universal Turing Machines

Definition 31.1.1 (Informal?)**.** A *U*niversal Turing Machine is a TM U that, when presented with an encoding of another TM T and an input string w, will simulate the action of T on input w.

Rather than go into abstract detail at the outset, let's do an example of what an encoding would look like. Consider a TM $(Q, \{0,1\}, \{0,1,B\}, \delta, q_0, B, F)$ whose transition function is:

	B	0	1
1	–	–	$2, 0, R$
2	$3, 1, L$	$3, 1, L$	$2, 1, R$
3	$4, 0, R$	$4, 0, R$	$3, 1, L$
4	–	–	–

We are going to encode this TM as a string using an additional symbol c (that is sometimes repeated) as a marker, and we'll write this on several lines so as to make things more clear.

$$\begin{array}{cccccc} ccc & 0 & c & 0 & c & 11R0 \\ cc & 111L1 & c & 111L1 & c & 11R1 \\ cc & 1111R0 & c & 1111R0 & c & 111L1 \\ cc & 0 & c & 0 & c & 0 \\ ccc & & & & & \end{array}$$

The encoding should be intuitively clear. We have ccc as the demarcation of the encoding of the TM (and thus can separate the encoding from the input string). We separate each line of the transition function with a cc marker, and we separate the transition on each symbol by a c. States are encoded with an appropriate number of strings of 1s. Left and Right are encoded with their symbols, as is the writing of 0s and 1s.

So, for example, in state 1 (the initial row), reading a 1 (the last column), we transition to state 2 (which is encoded as 11), write a 0, and move right. That's what $11R0$ means.

We will assume that any state with no transitions from it is an accepting state. In this case, that's state 4.

Note that we don't actually have to pre-allocate an arbitrary number of states (this wouldn't be possible). We can achieve the effect of pre-allocation by recomputing everything every time. That is, we copy, for example, the 1111 for a transition to state 4, and then we run a subprogram that walks through all the cc markers, checking off each of the 1 digits in the 1111 string. When the 1111 is completely checked off, we know to stop with that cc-marked input.

This has an analog in a higher level language. We normally think of setting up arrays and then doing direct lookup with subscripts. But if we had a linked list and not an array, we would find the k-th entry by walking the array one item at a time, incrementing a counter once for every step we take. When our counter value matches k, we stop walking and access that value.

We take as an accepting state of the TM any state that has no moves at all.

Now, how is the simulation of this TM by a UTM going to happen?

We would like to do this with a *t*wo-tape Turing machine, so we need to describe what that is.

Definition 31.1.2. A *multi-tape* Turing machine has k tapes and k tape heads, with each tape infinite in both directions. On a single move, depending on the state and the symbols read by all the tape heads, the TM can:

1. change state;
2. print a new symbol on each of the cells under each of the tape heads;
3. move each of the tape heads independently one cell to the left or to the right, or not move at all.

Theorem 31.1.3. *If a language L is accepted by a multitape TM, then it is accepted by a single-tape TM.*

Proof. We won't do this formally, but think about it. Since the theory of computing is mostly concerned about the existence of the ability to compute something, not about the efficiency of that computation, we can be sloppy about performance and yet still get existence.

Consider a TM T with two tapes and two tape heads. We'll build a TM V that has one tape and one head and simulates T.

Instead of laying out the input tapes for T in two pieces t_1 and t_2, lay them out for V one after the other, with a change-of-tape marker X in between them, so the tape for V is

$$t_1 X t_2.$$

Consider, instead of a single state s of T, that each state splits into two states s_1 and s_2, depending on whether one is simulating the upper tape head action or the lower tape head action of T.

Your initial objection to doing this should be that you are constraining the original first tape t_1 to be of fixed length, since it's followed on the V tape by the second tape t_2 from T. But remember that if we have to, if we run out of cells to hold what would be t_1, we can run a "subprogram" that goes all the way to the end of V's tape and shifts the t_2 part one cell to the right. This would be horribly inefficient as an algorithm, but we don't care about efficiency.

We will also need a marker M_1 and a marker M_2 on the single tape for V that indicates where the tape heads are positioned. And we'll need the little subprogram that flips the order of two cells, so we can move the M_i markers right or left as needed.

Now, we simply sequentialize the action of the two heads of T.

□

Our simulation of a TM T by a universal Turing Machine U goes as follows.

We put the encoding of T, followed by the input data for T, on the second tape and we use the first tape to hold the cells we write with U. These will either be

m, a marker, or B, the blank. There will initially be two m cells, one that lies over the last c symbol before the encoding for the transition for state 1, and one that lies over the leftmost input symbol on the second tape. This second m simulates the position of the tape head for T.

Now, using state changes as appropriate, U moves to the right, to the second marker m, reads that first symbol (and exchanges m with that symbol), and changes state to record what symbol has been read.

U now moves left to the first of the m markers. It now moves right to find the appropriate block encoding for that column in the tableau. That is, if U has read a blank, then it looks for the first column. If U has read a zero, it looks for the second column. If U has read a one, it looks for the third column. And it knows about columns because they are separated by a single c marker in the tableau.

With state changes, U can now determine what T would do, and make that transition.

Definition 31.1.4. We define a *procedure* to be a TM that may or may not halt on all the inputs presented to it.

Definition 31.1.5. We define an *algorithm* to be a TM that will halt, either accepting or not accepting, on all the inputs presented to it.

Definition 31.1.6 (The Halting Problem)**.** Does there exist an algorithm (i.e., a TM U that always halts) which, when presented with the encoding of an arbitrary TM T and an arbitrary input configuration, will halt and accept exactly when T halts and accepts on that input?

Theorem 31.1.7 (The Halting Problem)**.** *No.*

Proof. The set of TMs can be encoded as strings over

$$\{0, 1, c, L, R\}$$

and we can order them by length. Not all strings will be legal TMs, but a UTM reading such an encoding would simply halt without accepting.

We can similarly encode all possible inputs as strings over

$$\{0, 1\}$$

and we can order them by length.

We can thus enumerate until we have the i-th TM T_i and the i-th string x_i.

We then define a language

$$L_1 = \{x_i | x_i \text{ is not accepted by } T_i\}$$

Claim 31.1.8. We claim that L_1 is not a language accepted by any TM.

Proof. (of the claim)
If some T_j were to accept L_1, we would have that

$$x_j \in L_1$$

and thus that

$$x_j \notin L(T_j)$$

But since $L_1 \subseteq L(T_j)$ we have that

$$x_j \in L(T_j).$$

And that's a contradiction. □

Now, let's assume that we have a UTM U that always halts when given the encoding of any TM M and an input string w for that TM, and it halts and accepts if M halts on w, and halts and rejects if M halts and rejects w.

We are going to use U to build a TM T that accepts L_1, and that's going to be the contradiction, because we just showed that no such TM exists.

1. T enumerates sentences until x_i is enumerated.
2. T generates an encoding of the T_i that is the i-th TM in our enumeration of TMs.
3. T turns control over to U.
4. If U determines that T_i does not halt on x_i, then T halts and accepts x_i.
5. If U determines that T_i does halt on x_i, then U simulates T_i on x_i. If T_i halts in the simulation, then U halts and determines if T_i accepts or rejects.
    a) If T_i rejects x_i, then T accepts x_i.
    b) If T_i accepts x_i, then T rejects x_i.

So, if we had a UTM U that would determine whether an arbitrary TM M, presented with an arbitrary input string w, halted or not, we could use U to build a TM T that accepts L_1.

This is a contradiction, so no such U can exist.

□

Techno-Culture

This theorem is A Very Big Deal. What this says is that we can prove that there is no algorithmic way to determine if a program has an infinite loop. We would wish the world to be otherwise, but it isn't.

Lecture 32

Recall that languages can be viewed as *problems*:

Definition 32.0.1. Given a problem (or language) L over an alphabet Σ, an *instance* is some string $w \in \Sigma^*$ together with a *solution*, where the solution as a boolean value that is indicative of whether $w \in L$

Notice: Finding our first undecidable language (A_D) was hard, but finding our second (A_T) was trivial: since all instances of A_D are instances of A_T, it's clear that A_D is *no harder* than A_T and we represent this as $A_D \leq_m A_T$

Definition 32.0.2. Given languages $A \in \Sigma^*, B \in T^*$, we say A is *many-one reducible* or *m-reducible* to B if there exists some computable function $f : \Sigma^* \to T^*$ with the property that given a string w, $w \in A \iff f(w) \in B$. We write this as $A \leq_m B$.

In other words, $A \leq_m B$ if and only if there is some Turing machine that converts instances of A into instances of B.

This allows us to leverage languages that we know to be undecidable to prove the undecidability of other languages. Generally, to prove a language B is undecidable we use one of two strategies:

Undecidability Proof Strategies

1. Show that given a decider for B, we can decide A for some undecidable language (proof by contradiction).
2. Show $A \leq_m B$ for some language A which is known to be undecidable. We show this works by using Strategy 1 above.

Theorem 32.0.3. *For two languages $A \in \Sigma^*, B \in T^*$ If A is undecidable, and $A \leq_m B$, then B is also undecidable.*

Proof. Notice that because $A \leq_m B$, we have some TM that converts yes instances of A to yes instances of B and no instances of A to no instances of B. Call this TM $M_{A \to B}$.

Assume for the sake of contradiction that we have a decision procedure for B. That is, assume we have some TM D_B that decides B. Then we can construct a decider for A as follows:

Algorithm: D_A decider for A

Input : A string $w \in \Sigma^*$
Output: yes/no for $w \in A$
1 Run $M_{A \to B}$ on input w to obtain w', an instance of B
2 Run D_B on input w' to determine to obtain the boolean b
3 return b

$$b \text{ true} \iff w' \in B \iff w \in A$$

But this is impossible, since A is undecidable. Therefore, M_B cannot exist (B is also undecidable). □

Theorem 32.0.4. *H_T is undecidable, where $H_T := \{\langle M, w\rangle \mid M \text{ halts on input } w\}$*

Proof. Here we show that $A_T \leq_m H_T$

Algorithm: $M_{A_T \to H_T}$ A reduction from A_T to H_T

Input : $\langle M, w\rangle$ where M is a TM and $w \in \{0,1\}^*$
Output: $\langle R, x\rangle$ where R is a TM and $x \in \{0,1\}^*$ such that
M accepts $w \Leftrightarrow R$ halts on x

```
1  CreateTM R such that
2      Begin to simulate M on input x
3      while simulating do
4          if M accepts then
5              reject and halt
6          else if M halts and rejects then
7              loop
8          end
9      end
10 end
11 return ⟨R, w⟩ // A description of R above, plus the input
       string w
```

Notice that $M_{A_T \to H_T}$ does not accept/rejects: all it has to do is augment the description of M by turning its accepting state to a rejecting state, and making all other possible "halting" states into potential infinite loops.

Notice too that the new machine R never accepts any input. It only rejects (and halts) where M accepted, and loops otherwise. Therefore, R halts iff M accepts, exactly the behavior we wanted.

Because $M_{A_T \to H_T}$ is constructible, $A_T \leq_m H_T$ □

One more example:

Theorem 32.0.5. *$L_{ne} := \{M \text{ is a TM} \mid L(M) \neq \emptyset\}$ is undecidable. (The language of all Turing machines that do not accept anything).*

Proof. We show $A_T \leq_m L_{ne}$

Algorithm: $M_{A_T \to L_{ne}}$ A reduction from A_T to L_{ne}

```
Input  : ⟨M, w⟩ where M is a TM and w ∈ {0,1}*
Output: ⟨R⟩ where R is a TM that accepts something
CreateTM R taking input x such that
    Ignore input and simulate M on input w
    while simulating do
        if R accepts then
            accept
        else if M halts and rejects then
            loop
        else
            keep looping
        end
    end
end
return ⟨R⟩ // A description of R
```

Notice that R only accepts strings if M accepts w (and then it accepts all strings). □

Definition 32.0.6. A *property* is a set of Turing recognizable languages.

A property is *trivial* if it is empty, or it is all Turing recognizable languages.

Theorem 32.0.7 (Rice's Theorem)**.** *Every nontrivial property of Turing recognizable languages is undecidable.*

Proof. Let $\mathcal{P}$ be a nontrivial property.

Case $\emptyset \notin \mathcal{P}$)

Since $\mathcal{P}$ nontrivial, there is some language $L_{\mathcal{P}} \in \mathcal{P}$. Let M_L be a recognizer for $L_{\mathcal{P}}$. We will show $A_T \leq_m L_{\mathcal{P}}$

Algorithm: $M_{A_T \to L_\mathcal{P}}$ A reduction from A_T to $L_\mathcal{P}$

Input : $\langle M, w \rangle$ where M is a TM and $w \in \{0,1\}^*$
Output: $\langle R \rangle$ where $L(R) = L_\mathcal{P}$ if M accepts w, otherwise $L(R) = \emptyset$

```
1  CreateTM R taking input x such that
2      Simulate M on input w
3      while simulating do
4          if R accepts then
5              simulate M_L on input x
6          else if M halts and rejects then
7              loop
8          else
9              keep looping
10         end
11     end
12 end
13 return ⟨R⟩ // A description of R
```

Case $\emptyset \in \mathcal{P}$) TBD

□

Lecture 33

Other undecidable problems:

$$H \coloneqq \{\langle M, w\rangle \mid M \text{ is a TM that eventually halts on input}$$

$$H_\varepsilon \coloneqq \{\langle M\rangle \mid M \text{ is a TM that eventually halts on input } \varepsilon\}$$

$$I_G \coloneqq \{\langle G_1, G_2\rangle \mid G_1 \text{ and } G_2 \text{ are CFGs and } L(G_1) \cap L(G_2) \neq$$

$$E_G \coloneqq \{\langle G\rangle \mid G \text{ is a CFG that yields all strings over its inp}$$

We can prove these undecidable by leveraging the fact that H_D is undecidable. A typical proof goes like: Suppose there is a decision procedure for L, then we can use this procedure to build a decision procedure for (some previously known undecidable problem). This is impossible, hence L is undecidable.

Theorem 33.0.1. *H_ε is undecidable.*

Proof. Suppose that H_ε is decided by some decider D. Given an a TM $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ and a string $w \in \Sigma^*$, we can then use D to decide (algorithmically) whether M halts on w, thus contradicting the fact that H is undecidable. This decision algorithm works as follows: Given M and w as above, we first algorithmically construct a TM R, based on M and w, which acts as follows on any input string x: simulate M on input w, and do whatever M does. Note that R ignores its own input string x entirely. After constructing $\langle R\rangle$, we then simulate D on input $\langle R\rangle$. If D accepts, then we accept; otherwise D rejects (because D halts), and so we reject in this case.

The algorithm described above then decides whether M halts on input w, for the following reasons:

- If M halts on w then R halts on all its input strings, including ε. Thus D accepts $\langle R\rangle$ and so we accept.
- If M loops on input w, then R loops on all its input strings, including ε. Thus D rejects $\langle R\rangle$, and so we reject.

□

Theorem 33.0.2. *The I_G is undecidable.*

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be an arbitary TM. By adding a new start state and transition if necessary, we can assume that $\delta(q_0, \_)$ is defined, so that M takes at least one step before halting. This change to M can be done algorithmically in a way that does not alter the eventual halting vs. nonhalting behavior of M. Let \$ be a symbol not in $Q \cup \Gamma$. We start by recalling the languages L_1 and L_2 from the last assignment:

$$L_1 := \left\{ w\$x^R \mid w \text{ and } x \text{ are IDs of } M \text{ and } w \vdash x \right\}$$
$$L_2 := \left\{ w^R\$x \mid w \text{ and } x \text{ are IDs of } M \text{ and } w \vdash x \right\}$$

Here is a grammar F_1 for L_1:

$$\begin{aligned}
S_1 &\to _S_1 \mid S_1_ \mid O_1 \\
O_1 &\to aO_1a && \text{(for each } a \in \Gamma) \\
O_1 &\to T_1 \\
T_1 &\to qacIcrb && \text{(for each transition } (q,a) \to (r,b,R) \text{ and } c \in \Gamma) \\
T_1 &\to cqaIbcr && \text{(for each transition } (q,a) \to (r,b,L) \text{ and } c \in \Gamma) \\
I &\to aIa && \text{(for all } a \in \Gamma) \\
I &\to B \\
B &\to _B \mid B_ \mid \$
\end{aligned}$$

Similarly, a grammar F_2 for L_2:

$$\begin{aligned}
S_2 &\to _S_2 \mid S_2_ \mid O_2 \\
O_2 &\to aO_2a && \text{(for each } a \in \Gamma) \\
O_2 &\to T_2 \\
T_2 &\to caqIbrc && \text{(for all } (q,a) \to (r,b,R) \text{ and } c \in \Gamma) \\
T_2 &\to aqcIrcb && \text{(for all } (q,a) \to (r,b,L) \text{ and } c \in \Gamma) \\
I &\to aIa && \text{(for all } a \in \Gamma) \\
I &\to B \\
B &\to _B \mid B_ \mid \$
\end{aligned}$$

(F_1 and F_2 "share" the nonterminals I and B and their productions.) It is easy for an algorithm to construct F_1 and F_2 given a description of M as input.

Here is the idea. M halts on input ε if and only if there is a finite sequence of IDs

$$q_0\_ \vdash w_1 \vdash w_2 \vdash \cdots \vdash w_{n-1} \vdash w_n \,,$$

where n is the number of steps taken and w_n is a halting ID of M (the transition function is undefined for w_n). Consider the string obtained by reversing every other ID in the sequence, then ending each ID with\$. If n is even, then we get the string

$$s := q_0\_\$w_1^R\$w_2\$w_3^R\$\cdots\$(w_{n-1})^R\$w_n\$\,,$$

and if n is odd, we get the string

$$s' := q_0\_\$w_1^R\$w_2\$w_3^R\$\cdots\$(w_{n-1})\$w_n^R\$\,.$$

In either case, we want to make both G_1 and G_2 generate this string, but if no such string exists (i.e., M does not halt), then we want $L(G_1)$ and $L(G_2)$ to be disjoint. Suppose M halts in an even number of steps. (The case of an odd number of steps is handled similarly.) Then G_1 will generate s as follows:

$$\underbrace{q_0\_\$w_1^R}_{S_1}\,\$\,\underbrace{w_2\$w_3^R}_{S_1}\,\$\cdots\cdots\$\underbrace{(w_{n-1})^R}_{S_1}\,\$w_n\$$$

by generating a string of S_1's separated by dollar signs, followed by a halting ID and final \$. Notice that the S_1's ensure that $q_0\_ \vdash w_1$, $w_2 \vdash w_3$, etc. G_2 will generate the same string s in a different way:

$$q_0\_\$\,\underbrace{w_1^R\$w_2}_{S_2}\,\$\,\underbrace{w_3^R\$}_{S_2}\cdots\cdots\$\,\underbrace{(w_{n-1})^R\$w_n}_{S_2}\,\$$$

by generating $q_0\_$ followed by a string of S_2's terminated by dollar signs. Notice that the S_2's ensure that $w_1 \vdash w_2$, $w_3 \vdash w_4$, etc. Thus if both grammars generate the same string, that string must look like either s or s', and so we must have $q_0\_ \vdash w_1 \vdash w_2 \vdash \cdots \vdash w_n$ and w_n is a halting configuration, whence M halts on ε.

Now the formal details. Let A be a new nonterminal generating all strings over Γ, that is, A has productions $A \to \varepsilon$ and $A \to aA$ for each $a \in \Gamma$. Let H and H^R be new nonterminals with productions $H \to AqaA$ and $H^R \to AaqA$ for all $q \in Q$ and $a \in \Gamma$ such that $\delta(q, a)$ is *undefined*. Then H generates all halting configurations of M, and H^R generates the reversals of all halting configurations of M.

Now let G_1 be the grammar with start symbol R_1 obtained from F_1 by adding the three productions:

$$R_1 \to S_1\$R_1 \mid H \mid \varepsilon\,.$$

Similarly, let G_2 have start symbol R_2 and be obtained from F_2 by adding a new nonterminal C and the three productions

$$\begin{aligned} R_2 &\to q_0\_C \\ C &\to S_1\$C \mid H^R \mid \varepsilon \end{aligned}$$

Then G_1 and G_2 are as desired. □

Lecture 34

34.1 PCP and Undecidability

Having proved that the Halting Problem cannot be solved, we can use this result to prove that other important problems are also unsolvable.

We remember that we can view a language as a problem. For example, we know how we could encode a graph as a string of symbols. A graph consists of nodes and arcs, so we can encode a graph as

$$XXXNodeCountXXi_1Xj_1XXetc.XXX$$

in much the same way we encode a Turing Machine for a UTM. The node count could be a binary number, and then we have for the arcs pairs of numbers that represent an arc from the i node to the j node. Unlike with a finite automaton, for which this won't work (since we can't have an arbitrary number of states), we could design a TM that would use an auxiliary tape to count up and keep track of node numbers. This input string could be read by a program in a higher level language, and the only difference between your program and the TM is that the programming language and the computer on which it ran would have built-in features for storing numbers, hardware for doing addition, and such, which for the TM you would have to construct as subprograms.

Consider L, then, the language of all possible strings representing all possible graphs with a finite number of nodes. We can now ask some of the obvious questions we ask about graphs, for example:

> Is there a Turing Machine which, when presented with an arbitrary string from L, will always halt and will accept the string exactly when the string represents a graph with only one connected component?

In the case of this problem, the answer is a clear "yes" and one could create the appropriate TM by creating a TM that ran a breadth-first search on the graph of the input string. This would be tedious as a TM problem, but it's not intellectually hard.

Definition 34.1.1. A problem, expressed as a language L, is *solvable* if there is a Turing Machine that will take as input any of the strings of L, always halt, and accept exactly the strings for which the answer to the problem is "yes".

One of the most important problems in this part of theoretical computer science is *Post's Correspondence Problem* (PCP), named for the logician Emil Post (1897-1954).

We have a finite alphabet Σ, and two lists of strings in Σ^*:

$$A = \{w_1, ..., w_k\}$$
$$B = \{x_1, ..., x_k\}$$

We note that the lists have the same length k. This instance of PCP *has a solution* if there is a sequence of indices $i_1, ..., i_m$ for $m \geq 1$ such that

$$w_{i_1} ... w_{i_m} = x_{i_1} ... x_{i_m}$$

We can refer to the pairs (w_j, x_j) as *corresponding pairs* of strings.

Example 34.1.2. Let $\Sigma = \{0, 1\}$ and consider the lists

	List A	List B
i	w_i	x_i
1	1	111
2	10111	10
3	10	0

This instance of PCP has a solution with $(i_1, i_2, i_3, i_4) = (2, 1, 1, 3)$, where we can see that

$$10111|1|1|10 = 10|111|111|0$$

with the stroke character inserted for readability.

Example 34.1.3. On the other hand, there are instances of PCP with no solution. Let $\Sigma = \{0, 1\}$ and consider the lists

	List A	List B
i	w_i	x_i
1	10	101
2	011	11
3	101	011

Clearly, if we are to have a solution, then the first pair of strings must be index 1, since the other two pairs don't start with the same symbol. That gives us

$$\begin{array}{c} 10 \\ 101 \end{array}$$

and the next index must be either 1 or 3, since the A list string must start with a 1. But 1 won't work; it would produce 1010 from the A list and 101101 from the B list. So our second index is 3, and we get

$$\begin{array}{c} 10101 \\ 101011 \end{array}$$

At this point we are locked in: all subsequent index values must be 3, and the strings will never be of the same length.

MPCP and Undecidability

In order to simplify things somewhat, we work with the *modified Post's correspondence problem* (MPCP). In the MPCP, we insist that the first index be 1. As with many things in this course, the change in constraints makes things easier to prove but doesn't change what it is that we are proving things about.

We won't prove the following, but these are some of the major first theorems in this part of theoretical computer science.

Theorem 34.1.4. *If PCP were solvable, then MPCP would be solvable.*

Theorem 34.1.5. *PCP is not solvable.*

(We prove Theorem 34.1.5 in the obvious way, by contradiction, showing that MPCP is not solvable. And we prove that MPCP is not solvable using the one sledgehammer we have in fact just proved: If MPCP were solvable, then the Halting Problem would be solvable.)

We can now use this to prove that a number of problems involving context-free languages are not solvable. Recall that a grammar is context-free if all productions have a single variable symbol on the left and any string other than the empty string on the right. We didn't prove it, but it's in the notes: The languages generated by context-free grammars (cfgs) are exactly the languages accepted by some pushdown automaton.

In dealing with regular languages, we had several closure properties of union, intersection, and complement. Backing off from Type 3 (regular) grammars to Type 2 (context-free) grammars, things change.

The reduction of a question about cfgs to PCP proceeds as follows. Let A and B be the two sets of strings in Σ^+, and choose k additional symbols $K = \{a_1, ..., a_k\}$ not in Σ. We define two grammars

$$G_A = (\{S_A, V_T, P_A, S_A\}$$
$$G_B = (\{S_B, V_T, P_B, S_B\}$$

where $V_T = \Sigma \cup K$ and the productions P_A and P_B are defined as follows: For each i, $1 \leq i \leq k$, P_A has productions

$$S_A \rightarrow w_i S_A a_i$$
$$S_A \rightarrow w_i a_i$$

and P_B has productions

$$S_B \rightarrow x_i S_B a_i$$
$$S_B \rightarrow x_i a_i$$

Then we have

$$L_A = L(G_A) = \{w_{i_1} ... w_{i_m} a_{i_m} ... a_{i_1}\}$$
$$L_B = L(G_B) = \{x_{i_1} ... x_{i_m} a_{i_m} ... a_{i_1}\}$$

Theorem 34.1.6. *It is unsolvable whether the intersection of the languages generated by two arbitrary cfgs is empty.*

Proof. We will do this proof, because it's easy.

The intersection $L_A \cap L_B$ is empty if and only if PCP with the lists A and B has no solution.

So if we could solve emptiness of the intersection, we could solve PCP.

□

We will state but not prove the following that shows that context-free and regular are really different things. We let G_1 and G_2 be arbitrary context-free grammars, and we let R be a regular language.

Theorem 34.1.7. *The following are all unsolvable.*

1. *Determining that G_1 generates all strings over its terminal alphabet.*
2. *Determining that $L(G_1)$ is the particular regular set R.*
3. *Determining that $L(G_1)$ is a regular set.*
4. $L(G_1) \supseteq R$.
5. $\overline{L(G_1)} = \emptyset$.
6. *Determining that $L(G_1) \cap L(G_2)$ is a context-free language.*
7. *Determining that $\overline{L(G_1)}$ is a context-free language.*