Abstract

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

The date above is the date of this document’s most recent modification.
## Contents

1 Lecture 1  
1.0.1 Biconditionals ........................................ 6  
1.1 Methods of proof ........................................ 7  
1.1.1 Direct proofs .......................................... 7  

2 Lecture 2  
2.0.1 Proof by cases ........................................ 8  
2.0.2 Proof by induction ................................... 8  
2.0.3 Proof by contradiction ................................ 11  

3 Lecture 3  
3.1 Strong induction and the well-ordering principle ........... 12  
3.1.1 * Equivalence of strong induction and the well-ordering principle ........... 12  
3.2 Proof that $\sqrt{2}$ is irrational .......................... 13  

4 Lecture 4  
4.1 Describing sets ........................................... 14  
4.1.1 Listing the elements of a set ......................... 14  
4.1.2 Set formers ........................................... 15  
4.1.3 Don’t confuse a set with its members! ................ 16  
4.2 Subsets and the empty set ................................ 16  
4.2.1 Proving two sets equal ............................... 17  
4.3 Boolean set operations ................................... 17  
4.4 Sets of sets, ordered pairs, Cartesian product .......... 19  
4.4.1 * Ordered pairs as sets .............................. 20  
4.5 Relations and functions .................................. 21  
4.6 The pigeonhole principle ................................ 21  

5 Lecture 5  
5.1 Alphabets, strings, and languages ......................... 22  
5.1.1 The concatenation operator ........................... 23  
5.1.2 The empty string ..................................... 23  
5.1.3 Languages ............................................ 23  
5.1.4 Languages as decision problems ..................... 24  

6 Lecture 6  
6.1 Finite automata .......................................... 25  

7 Lecture 7  

8 Lecture 8  
8.1 Nondeterministic finite automata (NFAs) ................. 27  
8.2 $\epsilon$-transitions ...................................... 27
9 Lecture 9
  9.1 The subset construction ........................................... 28

10 Lecture 10
  10.1 Proof that the subset construction is correct ................. 28

11 Lecture 11
  11.1 Eliminating $\epsilon$-transitions from an NFA ................. 29
    11.1.1 Method 1 .................................................. 29
    11.1.2 Method 2 .................................................. 29
  11.2 Regular expressions .............................................. 30
    11.2.1 Regular expression syntax ................................ 31
    11.2.2 Regular expression semantics .............................. 31

12 Lecture 12 ............................................................. 32

13 Lecture 13
  13.1 Transforming regular expressions into $\epsilon$-NFAs ......... 32

14 Lecture 14
  14.1 Transforming $\epsilon$-NFAs into regular expressions .......... 32

15 Lecture 15
  15.1 Proving languages not regular .................................. 35

16 Lecture 16 ............................................................. 35

17 Lecture 17
  17.1 Closure properties of regular languages ...................... 37

18 Lecture 18
  18.1 Using closure properties to show nonregularity ............. 42

19 Lecture 19
  19.1 DFA minimization .............................................. 46

20 Lecture 20
  20.1 Constructing the minimal DFA .................................. 48

21 Lecture 21 ............................................................. 51

22 Lecture 22 ............................................................. 51

23 Lecture 23 ............................................................. 51

24 Lecture 24 ............................................................. 52

25 Lecture 25 ............................................................. 52
<table>
<thead>
<tr>
<th>Lecture</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>26 Lecture 26</td>
<td>52</td>
</tr>
<tr>
<td>27 Lecture 27</td>
<td>57</td>
</tr>
<tr>
<td>28 Lecture 28</td>
<td>58</td>
</tr>
<tr>
<td>29 Lecture 29</td>
<td>58</td>
</tr>
<tr>
<td>30 Lecture 30</td>
<td>59</td>
</tr>
<tr>
<td>31 Lecture 31</td>
<td>59</td>
</tr>
</tbody>
</table>
1 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 1.1. A natural number is any whole number that is at least zero. We can list the natural numbers as 0, 1, 2, 3, …. 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 1.2. 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 1.3. 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 1.4. 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.

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 \( H \rightarrow C \). \( H \) 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 (\( \pi \), 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.

1.0.1 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 \( H \rightarrow C \) and \( C \rightarrow H \).
1.1 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.

1.1.1 Direct proofs

Theorem 1.5. 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. \( \square \)

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.

\(^1\)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.
2 Lecture 2

Continuing with examples of proofs.

2.0.1 Proof by cases

Theorem 2.1. 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

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

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.

2.0.2 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 $S(n)$ is true for every natural number $n$. The argument is in two stages:

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

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

- $S(0)$ is true (this is the base case)
- $S(1)$ is true (this applies the inductive step with $n = 0$ and the fact that we’ve already established $S(0)$)
• $S(2)$ is true (by the inductive step again, this time with $n = 1$, as well as the previous proof of $S(1)$)
• $S(3)$
• etc.

The point is that once we’ve established $S(n)$ for some value of $n$, then we can conclude $S(n + 1)$ by the inductive step. So if we prove both the base case and the inductive step for general $n$, we must conclude that $S(n)$ 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 $S(1)$ and the induction step is to prove $S(n) \rightarrow S(n + 1)$ 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 arbitrary example, you can prove $S(17)$ as the base case then prove $S(k) \rightarrow S(k + 1)$ for all integers $k \geq 17$. Conclude that $S(n)$ 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 ??.

Proof of Theorem ?? by induction. We let $S(n)$ be the statement that $2^n \geq n^2$, then we wish to prove $S(n)$ 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 $S(4)$ is true.

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

We can do this by a direct chain of inequalities:

\[
\begin{align*}
2^{n+1} &= 2(2^n) \\
&\geq 2n^2 \\
&= n^2 + n^2 \\
&\geq n^2 + 4n \\
&= n^2 + 2n + 2n \\
&\geq n^2 + 2n + 1 \\
&= (n + 1)^2.
\end{align*}
\]

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

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

**Base case:** To see that $S(1)$ 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 $S(n)$ holds then $S(n + 1)$ holds. Assume that $S(n)$ holds, i.e., that either $n$ is even or $n - 1$ is even (this is the inductive hypothesis), and consider the statement $S(n + 1)$: “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 $S(n + 1)$ holds, and so we are done.

**Case 2:** $n$ is odd, i.e., $n$ is not even. Since the inductive hypothesis $S(n)$ (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 $S(n + 1)$ holds in this case as well.

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

We can now conclude by induction that $S(n)$ holds for all integers $n \geq 1$. \[\square\]

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 2.4.** 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 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 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 $\ell$. 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. \[\square\]
2.0.3 Proof by contradiction

The next theorem’s proof uses a new proof technique: *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 2.5.** 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 ??, 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 $\ell$ (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 $\ell$ 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,\(^2\) 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 2.6.** 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 ?? (forward direction) we can write $a = 2k + 1$ and $b = 2\ell + 1$ for some integers $k$ and $\ell$. Then

$$ab = (2k + 1)(2\ell + 1) = 4k\ell + 2k + 2\ell + 1 = 2(2k\ell + k + \ell) + 1 = 2m + 1,$$

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

\(^2\)A contradiction is often indicated symbolically by $\Rightarrow \Leftarrow$.  

11
3 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 \( S(n) \) holds for all integers \( n \geq c \), it suffices to prove that \( S(n) \) follows from \( S(c), S(c+1), S(c+2), \ldots, S(n-1) \). In other words, it is enough to prove \( S(n) \) assuming that \( S(k) \) holds for all integers \( k \) with \( c \leq k \leq n-1 \). This still requires proving \( S(c) \) outright with no assumptions, but then you can establish \( S(c+1) \) given \( S(c) \), because you’ve just proved \( S(c) \). Then you can establish \( S(c+2) \) assuming both \( S(c) \) and \( S(c+1) \) 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 \( S(n) \) to prove \( S(n+1) \), 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 \( S(n) \) is not enough to directly prove \( S(n+1) \), 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.

3.1.1 * 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 \( S(n) \) be the property that \( n \not\in X \), i.e., that \( n \) is not an element of \( X \). We now prove by strong induction that \( S(n) \) is true for every natural number \( n \), hence \( X \) must be empty, which contradicts the fact that \( X \) is nonempty.

Base case: If \( S(0) \) 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 \( S(0) \) must be true. (See how this is a mini-proof by contradiction.)

Inductive step: Fix any natural number \( n \) and assume (inductive hypothesis) that \( S(m) \) is true for all natural numbers \( m \leq n \). This means that \( m \not\in X \) for all natural numbers \( m \leq n \). Then \( S(n+1) \) must also be true, for if \( S(n+1) \) were false, then \( n+1 \) would be the least element of \( X \). Again, a contradiction. Thus \( S(n+1) \) is true.

To reiterate: by strong induction, we have that \( S(n) \) (equivalently, \( n \not\in X \)) is true for all natural numbers \( n \), and hence \( X \) must be empty. This finishes the proof of the well-ordering principle. \( \square \)
Proof of strong induction using the well-ordering principle. Let $S$ be any property of numbers. Suppose that $S(0)$ is true, and for any natural number $n$, we know that if $S(0), \ldots, S(n)$ are all true then $S(n+1)$ must also be true. Then we use the well-ordering principle and proof-by-contradiction to show that $S(n)$ is true for all natural numbers $n$. Let $X$ be the set of all natural numbers $n$ such that $S(n)$ is false, i.e.,

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

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 $S(n_0)$ 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 $S(0), \ldots, S(n)$ are all true, but $S(n+1) \iff S(n_0)$, 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 3.1.** 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 ??—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 $\ell$ 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. 

$\square$
4 Lecture 4

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 4.1 (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.

4.1 Describing sets

4.1.1 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,

\[ \{1, 4, 9, 16, 25\} \]

(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 (??), above:

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

In some cases—only when it is intuitively clear—the listing can omit some elements and use an ellipsis (\( \ldots \)) 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.,

\[ N = \{0, 1, 2, \ldots\}, \]
\[ Z = \{\ldots, -2, -1, 0, 1, 2, \ldots\} .\]

**Definition 4.2.** 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, \( \|\{1, 4, 9, 16, 25\}\| = 5 \).

### 4.1.2 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, \( \langle \text{expression} \rangle \) is some arbitrary expression, usually involving one or more variable names, e.g.,
\( x, y, z, \ldots \), and \( \langle \text{property} \rangle \) is some statement about the variables used in the \( \langle \text{expression} \rangle \). 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 (\( \mid \)) can be read as “such
that,” and the set former itself can be read as, “the set of all \( \langle \text{expression} \rangle \) such that \( \langle \text{property} \rangle \).”

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

\[ \{x^2 \mid x \in Z \land 1 \leq x \land x \leq 5\} . \]

Informally, this is the set of all squares of integers in the range 1 to 5, inclusive.\(^3\) The two inequalities
involving \( x \) can be contracted to the shorthand, “\( 1 \leq x \leq 5 \),” so the set former can be written,

\[ \{x^2 \mid x \in Z \land 1 \leq x \leq 5\} . \]

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

\[ \{x^2 \mid x \in N \land 0 < x < 6\} \]

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:

\[ \{y^2 \mid y \in Z \land 1 \leq y \leq 5\} . \]

---

\(^3\)We will use the wedge symbol (\( \land \)) to mean “and” (conjunction), the vee symbol (\( \lor \)) 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 \( P \lor Q \) 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.
Here is another example using two dummy variables to denote the set of rational numbers:

\[ \mathbb{Q} = \left\{ \frac{a}{b} \mid a, b \in \mathbb{Z} \land b \neq 0 \right\} . \]

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

\[ \mathbb{Q} = \left\{ \frac{x}{y} \mid x, y \in \mathbb{Z} \land y \neq 0 \right\} . \]

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.

4.1.3 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 be 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.

4.2 Subsets and the empty set

Definition 4.3. 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 \( A \nsubseteq B \) 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:

\[ \emptyset = \{ \} = \{ x \mid x \in \mathbb{Z} \land x \notin \mathbb{Z} \} = \{ x \mid 0 = 1 \} . \]

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 4.4. 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).
4.2.1 Proving two sets equal

The last item in Fact ?? (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.

4.3 Boolean set operations

Definition 4.5. Let \( A \) and \( B \) be any sets. We define

\[
A \cup B := \{ z \mid z \in A \lor z \in B \},
\]

\[
A \cap B := \{ z \mid z \in A \land z \in B \},
\]

\[
A - B := \{ z \mid z \in A \land z \notin B \}.
\]

\( A \cup B \) 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 4.6. 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 \( A - B = \{ x \mid x \in A \land x \notin B \} \), 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 \( A - B = \{ x \mid x \in A \land x \notin B \} \), 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):

\[
z \in A - (A - B) \iff z \in A \land z \notin (A - B) \\
\iff z \in A \land \neg (z \in (A - B)) \\
\iff z \in A \land \neg (z \in A \land z \notin B) \\
\iff z \in A \land (z \notin A \lor z \in B) \\
\iff (z \in A \land z \notin A) \lor (z \in A \land z \in B) \\
\iff 0 \lor (z \in A \land z \in B) \\
\iff z \in A \land z \in B \\
\iff z \in A \cap B.
\]

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 4.7.** 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 4.8** (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 \( z \in (A \cap B) \cup (\text{anything}) \)).

**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, \( z \in A \cap (B \cup C) \). \( \square \)

### 4.4 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 4.9.** *For any ordered pairs \((a, b)\) and \((c, d)\),*

\[
(a, b) = (c, d) \iff (a = c \land b = d).
\]
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 4.10.** Let $A$ and $B$ be any sets. We define the Cartesian product of $A$ and $B$ as follows:

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

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 4.11.** 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 4.12** (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.

### 4.4.1 * Ordered pairs as sets*

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

**Definition 4.13.** 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 4.11, and so it is a legitimate way to implement ordered pairs as sets. There are other ways, but all correct implementations must satisfy Fact 4.11.
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).

4.5 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 “\((x, y) \leq\)” 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).

4.6 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 4.14** (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 4.15.** 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, \( d(V) \subseteq \{0, 1, \ldots, n-2\} \). 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 \( d(u) = d(v) \).

\( \Box \)

There is a stronger version of the pigeonhole principle:

**Theorem 4.16 (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 4.17.** 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 \( a \in A \).

5 Lecture 5

5.1 Alphabets, strings, and languages

**Definition 5.1.** Let \( \Sigma \) be any nonempty, finite set. A string \( w \) over \( \Sigma \) is any finite sequence \( w_1w_2\cdots w_n \), 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 \( \Sigma \) we sometimes call the alphabet, and the elements of \( \Sigma \) 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 \( \Sigma \) themselves to be strings of length 1.
5.1.1 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, however. 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.

5.1.2 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 \( \epsilon \) (the Greek letter epsilon).\(^4\) The symbol \( \epsilon \) 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 \( \epsilon \) as a placeholder instead.

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

\[ w\epsilon = \epsilon w = w. \]

\( \epsilon \) is the only string with this property; when part of a concatenation, it simply disappears.

5.1.3 Languages

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

---

\(^4\)Some books and papers use \( \lambda \) (lambda) to denote the empty string.
5.1.4 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 $\Sigma$.

A decision problem, then, just asks a yes/no question about every string in $\Sigma^*$. Given any decision problem, the yes-instances of the problem form subset of $\Sigma^*$, i.e., a language over $\Sigma$. Conversely given any language $L$ over $\Sigma^*$, 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 $\Sigma^*$.

Put in very general, somewhat vague terms, a computational device $A$ **recognizes** a language $L$ over $\Sigma$ 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 $w$.  

24
6 Lecture 6

6.1 Finite automata

The first computational device we consider is a very simple (and very weak) one: the deterministic finite automaton, or DFA for short. A DFA has a finite number of states, with a preset collection of allowed transitions between the states labeled 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 auxiliary 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: [Give DFA recognizing binary strings that contain at least one 1.]

Several examples of automata today:

- checking that the last symbol of a binary string is 1
- checking for an even number of 0’s in a binary string
- product construction for even 0’s and odd 1’s
- complementary automata

Transition diagrams for automata.

7 Lecture 7

Automata more formally as mathematical objects. Definition of a Deterministic Finite Automaton (DFA).

Expanding the transition function \( \delta \) to \( \hat{\delta} \) for all strings in \( \Sigma^* \).

**Definition 7.1.** Let \( A = (Q, \Sigma, \delta, q_0, F) \) 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 \( \hat{\delta}(q, \epsilon) := q \);

**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.** Check that \( \hat{\delta} \) agrees with \( \delta \) on individual symbols, i.e., strings of length 1.

Defining computation, acceptance, language recognition.

**Definition 7.3.** Let \( A = (Q, \Sigma, \delta, q_0, F) \) be a DFA, and let \( w \in \Sigma^* \) be a string. We say that \( A \) accepts \( w \) iff \( \hat{\delta}(q_0, w) \in F \). Otherwise, we say that \( A \) rejects \( w \). The language recognized by \( A \) is the language

\[ L(A) := \{ w \in \Sigma^* \mid A \text{ accepts } w \} . \]

\(^5\)“Automaton” is the singular form of the noun. The plural is “automata.”
More examples:

- nonempty binary strings that start and end with the same symbol
- binary strings of length $\geq 2$ whose penultimate symbol is 1
- binary strings with a multiple of 5 many 0s
- binary representations of multiples of 3

DFAs given in tabular form.

Example: finding a search string in text.

Proofs that certain automata recognize certain languages.

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

1. We define the *product* of $A$ and $B$ as the following DFA:
   
   $A \land B := (Q \times R, \Sigma, \eta, (q_0, r_0), F \times G)$,

   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 $\Sigma$ denote the common alphabet of the automata.

**Theorem 7.5.** For any DFA $A$, $L(\neg A) = \overline{L(A)}$.

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

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

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

**Theorem 7.6.** For any DFAs $A$ and $B$, $L(A \land B) = L(A) \cap L(B)$.

*Proof.* Let $A$, $B$, and $A \land B$ 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:** $\hat{\eta}((q_0, r_0), \epsilon) = (q_0, r_0) = (\hat{\delta}(q_0, \epsilon), \hat{\zeta}(r_0, \epsilon))$. 

26
Inductive case: Let $x$ be a string over $\Sigma$ and let $a$ be a symbol in $\Sigma$. 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 $xa$:

$$\hat{\eta}((q_0, r_0), xa) = \eta(\hat{\eta}((q_0, r_0), x), a)$$  

(definition of $\hat{\eta}$)

$$= \eta(\hat{\delta}(q_0, x), \hat{\zeta}(r_0, x)), a)$$  

(inductive hypothesis)

$$= (\hat{\delta}(\hat{\delta}(q_0, x), a), \hat{\zeta}(\hat{\zeta}(r_0, x), a))$$  

(definition of $\eta$)

$$= (\hat{\delta}(q_0, xa), \hat{\zeta}(r_0, xa))$$  

(definitions of $\hat{\delta}$ and $\hat{\zeta}$)

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

$$w \in L(A \land B) \iff \hat{\eta}((q_0, r_0), w) \in F \times G$$  

(definition of acceptance for $A \land B$)

$$\iff (\hat{\delta}(q_0, w), \hat{\zeta}(r_0, w)) \in F \times G$$  

(the equation we just proved inductively)

$$\iff \hat{\delta}(q_0, w) \in F \text{ and } \hat{\zeta}(r_0, w) \in G$$  

(definition of Cartesian product)

$$\iff w \in L(A) \text{ and } w \in L(B)$$  

(definitions of acceptance for $A$ and $B$)

$$\iff w \in L(A) \cap L(B)$$  

(definition of set intersection)

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

8 Lecture 8

8.1 Nondeterministic finite automata (NFAs)

Examples. Compare with DFAs. Every DFA is essentially an NFA. Formal definition of NFA. Formal definition of acceptance. [Define a path in an automaton with label $w$.]

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.

[Define equivalence of automata.]

This suggests the question: are there languages that are recognized by NFAs but not DFAs? Surprisingly, no. We’ll prove that for any DFA $N$, 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 $\epsilon$-transitions

An $\epsilon$-NFA (or an NFA with $\epsilon$-transitions, or $\epsilon$-moves), is an NFA with an additional type of allowed transition: an edge labeled with $\epsilon$. When this edge is followed, no symbol from the input is read,
i.e., the input pointer is not advanced. These $\epsilon$-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 $\epsilon$-NFA, but even $\epsilon$-NFAs are no more powerful at recognizing languages than DFAs.

**Definition 8.1.** For alphabet $\Sigma$, let $\Sigma_\epsilon$ denote $\Sigma \cup \{\epsilon\}$, that is the set of all strings over $\Sigma$ of length 0 or 1.

**Definition 8.2.** An $\epsilon$-NFA is a tuple $(Q, \Sigma, \delta, q_0, F)$, where $Q$, $\Sigma$, $q_0$, and $F$ are exactly as in the case of a DFA, but $\delta : Q \times \Sigma_\epsilon \rightarrow 2^Q$.

**Definition 8.3.** Let $N := (Q, \Sigma, \delta, q_0, F)$ be an $\epsilon$-NFA and let $w$ be any string over $\Sigma$. We say that $N$ accepts $w$ if there exist $n \geq 0$, states $s_0, \ldots, s_n \in Q$, and strings $w_1, \ldots, w_n \in \Sigma_\epsilon$ such that

- $w = w_1 \cdots w_n$ (string concatenation),
- $s_0 = q_0$ (the start state),
- $s_n \in F$ (an accepting state), and
- for all $1 \leq i \leq n$, $s_i \in \delta(s_{i-1}, w_i)$.

In this case, we say that $\langle s_0, \ldots, s_n \rangle$ is an accepting path of $N$ on input $w$. Otherwise, $N$ rejects $w$.

The difference between this definition and acceptance for an NFA is that now some of $w_i$ may be $\epsilon$ (and so it is possible that $n > |w|$).

**9 Lecture 9**

**9.1 The subset construction**

Here we prove that for any NFA $N$ there is an equivalent DFA $D$. The proof will explicitly construct $D$ from a formal description of $N$. This is called the subset construction of a DFA from an NFA.

General idea: A state of $D$ corresponds to a set of states of $N$, and records the set of possible states that one could arrive at in $N$ by reading a prefix of the input.

[Define $\epsilon$-closed set and $\epsilon$-closure.]

[Formal construction and proof]

[Example. Optimize by only building states reachable from the start state.]

**10 Lecture 10**

**10.1 Proof that the subset construction is correct**

Formal proof that the accepting path criterion is equivalent to the extended transition function criterion for NFA acceptance.
11 Lecture 11

Formally define $\epsilon$-NFA and acceptance via the accepting path criterion.

Example: search for “colo[u]r”

Define $\text{eclose}(q)$ for a state $q$, the $\epsilon$-closure of $q$. Define $\text{eclose}(S)$ for a set of states $S$.

11.1 Eliminating $\epsilon$-transitions from an NFA

Here we show how to eliminate $\epsilon$-transitions from an $\epsilon$-NFA to get an equivalent NFA without $\epsilon$-transitions.

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

11.1.1 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 $F' := \{ q \in Q | \text{eclose}(q) \cap F \neq \emptyset \}$.

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 $L(N') = L(N)$.

11.1.2 Method 2

We construct $N'$ via the algorithm below. In the algorithm, the $\epsilon$-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 $r \in \delta(q,\epsilon)$ 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 $r \in \delta(q,\epsilon)$ 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 $\delta(q,\epsilon) := \emptyset$ (that is, remove all $\epsilon$-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 $\epsilon$-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 $q \xrightarrow{a} s$ can be rerouted to use the previously existing transitions $q \xrightarrow{\epsilon} r \xrightarrow{a} s$ 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 $\Sigma_\epsilon$ and $p := \langle s_0, \ldots, s_n \rangle$ is a corresponding accepting path, as in Definition ???. We can remove all the $\epsilon$-transitions from $p$ to obtain an accepting path of $w$ with no $\epsilon$-transitions as follows:
  - If the last symbol $w_n = \epsilon$, then we can just remove that transition: since $s_n \in F$ and $s_n \in \delta(s_{n-1}, \epsilon)$, 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 $\epsilon$-transitions at the end of $p$.
  - After the above, if there are still any $\epsilon$-transitions in $p$, then there must be one that is immediately followed by a non-$\epsilon$-transition. That is, $w_i = \epsilon$ and $w_{i+1} = a$ for some $i \geq 1$ and $a \in \Sigma$. Then $\langle s_0, \ldots, s_{i-1}, s_{i+1}, \ldots, s_n \rangle$ is an accepting path for $w = w_1 \cdots w_{i-1} w_{i+1} \cdots w_n$, because $s_i \in \delta(s_{i-1}, \epsilon)$ 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 $\epsilon$-transitions in $p$.

11.2 Regular expressions

Used to denote (specify) languages. Syntax. Example: Same as the $\epsilon$-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.
11.2.1 Regular expression syntax

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

- ∅
- a (for any symbol a ∈ Σ),
- 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 · (juxtaposition) operator is called *concatenation*. These are both binary infix operators and are associative. The unary postfix * operator 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).

11.2.2 Regular expression semantics

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

- The regexp ∅ does not match any string.
- Any regexp a (where a ∈ Σ) 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 w₁ · · · wₙ, where n is a natural number and each wᵢ 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 = ε. This means that R* *always* matches ε, regardless of R. In particular, the regexp ∅* matches the empty string ε and nothing else. It is thus natural to use ε as shorthand for the regexp ∅*, and pretend that this is another atomic regexp.

**Definition 11.1.** 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.
12 Lecture 12

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

13 Lecture 13

13.1 Transforming regular expressions into $\epsilon$-NFAs

Definition 13.1. We will say that an $\epsilon$-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 $a \in \Sigma \cup \{\epsilon\}$), 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 $a \in \Sigma \cup \{\epsilon\}$).

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

1. Add a new start state $q'_0 \notin Q$ with a single $\epsilon$-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 $\epsilon$-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 $\epsilon$-NFA.

Theorem 13.2. For every regular expression $R$ there exists an $\epsilon$-NFA $N$ such that $L(N) = L(R)$.

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

14 Lecture 14

14.1 Transforming $\epsilon$-NFAs into regular expressions

Note that the book goes from DFAs to regexps. Starting with $\epsilon$-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 14.1. Given an alphabet $\Sigma$, let $\text{REG}_\Sigma$ be the set of all regular expressions over $\Sigma$. A generalized finite automaton (GFA) with alphabet $\Sigma$ is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- $Q$ is a nonempty, finite set (the state set),
- $\delta$ is a function mapping ordered pairs of states to regular expressions over $\Sigma$, that is, $\delta : Q \times Q \to \text{REG}_\Sigma$, 

• $q_0$ 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 14.2.** 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 $\epsilon$-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 $\Sigma \cup \{\epsilon\}$ 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 $G_i = (Q_i, \Sigma_i, \delta_i, q_0, \{f\})$ has an intermediate state, then we choose such a state $q \in Q_i - \{q_0, f\}$ (it doesn’t matter which) and define $G_{i+1} := (Q_{i+1}, \Sigma_i, \delta_{i+1}, q_0, \{f\})$, 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

$$U + RS^*T = U + R\emptyset^*T = U + R\epsilon T = U + RT,$$
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 $G_\ell$ 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 $G_\ell$ looks like

\[
\begin{array}{c|cc}
 & q_0 & f \\
\hline
q_0 & \emptyset & E \\
f & \emptyset & \emptyset
\end{array}
\]

where $E$ is some regexp over $\Sigma$ [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 14.3.** Let $L$ be any language over an alphabet $\Sigma$. 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 $\epsilon$-NFA.
4. $L$ is recognized by some clean $\epsilon$-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 $\mapsto$ $\epsilon$-NFA, clean $\epsilon$-NFA $\mapsto$ $\epsilon$-NFA, and regexp $\mapsto$ GFA. You should teach yourself how these trivial transformations work.

**Corollary 14.4.** For any two regular expressions $R$ and $S$ over an alphabet $\Sigma$, there exist regular expressions over $\Sigma$ for the complement $L(R)$ 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 $\epsilon$-NFA and/or an NFA), then build the complementary DFA $\neg A$ (swapping final and nonfinal 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 $A \land B$ for the intersection, then convert $A \land B$ back into an equivalent regular expression. \(\square\)

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.
15 Lecture 15

15.1 Proving languages not regular

Definition 15.1. 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 $i \geq 0$,

$xy^iz \in L$.

Lemma 15.2 (Pumping Lemma for Regular Languages). For any language $L$, if $L$ is regular, then $L$ is pumpable.

[Proof]
Here is the contrapositive, which is an equivalent statement:

Lemma 15.3 (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

$xy^iz \notin L$.

The value of $p$ above is called the pumping length.

16 Lecture 16

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 $\geq 0$ 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 ≥ 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| ≤ p, and |y| > 0.

• We get to choose i ≥ 0 based on all the previous values.

**Example:** Let \( L := \{0^p1^n \mid n \geq 0 \} \). We show that L is not pumpable using the template:

Given any \( p > 0 \),
let \( s := 0^p1^p \). (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 \( xy^iz = xy^0z = xz \notin L \), 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 \( p - m \neq p \), the string \( xz \) is not of the form \( 0^n1^m \) for any \( n \), and thus \( xz \notin L \).

The next three examples are minor variations of each other.

**Example:** Let

\[ L := \{ w \in \{0,1\}^* \mid w \text{ has the same number of 0's as 1's} \} \, . \]

We show that L is not pumpable using the template:

Given any \( p > 0 \),
let \( s := 0^p1^p \). (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 \( xy^iz = xy^0z = xz \notin L \), 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:** Let

\[ L := \{ w \in \{0,1\}^* \mid w \text{ has more 0's than 1's} \} \, . \]

We show that L is not pumpable using the template:

Given any \( p > 0 \),
let \( s := 0^p1^{p-1} \). (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 \( xy^iz = xy^0z = xz \notin L \), 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: Let
\[ L := \{ w \in \{0, 1\}^* \mid w \text{ has fewer 0's than 1's} \} . \]
We show that \( L \) is not pumpable using the template:

Given any \( p > 0 \),
let \( s := 0^p1^{p+1} \). (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 \( xy^i z = xy^2z = xyyz \notin L \), 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.]

We can view use of the pumping lemma as a game with four turns (and full disclosure), 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 \( x, y, z \) 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.

17 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.** 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. \( \square \)

The same idea proves

**Proposition 17.2.** If \( L \) and \( M \) are regular languages, then so are \( LM \) and \( L^* \).

We’ve proved this, too:
Proposition 17.3. 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 \( w \in \Sigma^* \),

\[
\begin{align*}
& w \in \overline{L} \iff A \text{ rejects } w \\
& \iff \hat{\delta}(q_0, w) \notin F \\
& \iff \hat{\delta}(q_0, w) \in Q - F \\
& \iff B \text{ accepts } w.
\end{align*}
\]

Thus \( L(B) = \overline{L} \), and so \( \overline{L} \) is regular.

In the proofs of Propositions ?? and ??, 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 \( \epsilon \)-NFA.

To illustrate these techniques, we’ll prove the next closure property in two ways—transforming a regular expression and transforming an \( \epsilon \)-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 \( L = \{aab, bca, aaa, \epsilon\} \), then \( L^R = \{baa, cba, aaa, \epsilon\} \). Notice that \( (w^R)^R = w \) for any string \( w \), and thus \( (L^R)^R = L \) for any language \( L \).

Proposition 17.4. If \( L \) is regular, then so is \( L^R \).

For our first proof of Proposition ??, 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 17.5. Fix an alphabet \( \Sigma \).

1. \( \emptyset^R = \emptyset \).

2. For any symbol \( a \in \Sigma \), \( \{a\}^R = \{a^R\} = \{a\} \).

For any two languages \( L \) and \( M \) over \( \Sigma \),

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.

\[
\begin{align*}
& w \in (L \cup M)^R \\
& \iff w^R \in L \cup M \\
& \iff w^R \in L \text{ or } w^R \in M \\
& \iff w \in L^R \text{ or } w \in M^R \\
& \iff w \in L^R \cup M^R.
\end{align*}
\]
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 \((xy)^R = y^Rx^R\). So

\[
w = (w^R)^R = (xy)^R = y^Rx^R \in M^R L^R.
\]

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 17.14.* We transform \( r \) into \( r^R \) by the following rules, which are justified by Facts (1)–(5) of Lemma 17.6 above.

1. If \( r = \emptyset \), then define \( r^R = \emptyset^R := \emptyset \).
2. If \( r = a \) for some \( a \in \Sigma \), then define \( r^R = a^R := a \).
3. 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 \( t^R \)).
4. If \( r = st \) for some regular expressions \( s \) and \( t \), then define \( r^R = (st)^R := t^Rs^R \) (note the reversal).
5. 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 17.6.** \( L(r^R) = L(r)^R \) for any regexp \( r \) over \( \Sigma \).

*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 17.6:
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 ??; the last is again by regexp semantics.)

Case 2: \( r = a \) for some \( a \in \Sigma \). We have
\[
L(a^R) = L(a) \quad \text{(definition of } a^R) \\
= \{a\} \quad \text{(regexp semantics)} \\
= \{a^R\} \quad \text{(Fact (2) of Lemma ??)} \\
= \{a\}^R \quad \text{(definition of the reversal of a language)} \\
= L(a)^R \quad \text{(regexp semantics again)}
\]

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
\[
L((s + t)^R) = L(s^R + t^R) \quad \text{(definition of } (s + t)^R) \\
= L(s^R) \cup L(t^R) \quad \text{(regexp semantics)} \\
= L(s)^R \cup L(t)^R \quad \text{(inductive hypothesis)} \\
= (L(s) \cup L(t))^R \quad \text{(Fact (3) of Lemma ??)} \\
= (L(s + t))^R \quad \text{(regexp semantics)}
\]

Case 4: \( r = st \) for regexps \( s, t \). The inductive hypothesis applies to \( s \) and \( t \), so we have
\[
L((st)^R) = L(t^R s^R) \quad \text{(definition of } (st)^R) \\
= L(t^R) L(s^R) \quad \text{(regexp semantics)} \\
= L(t)^R L(s)^R \quad \text{(inductive hypothesis)} \\
= (L(s) L(t))^R \quad \text{(Fact (4) of Lemma ??)} \\
= L(st)^R \quad \text{(regexp semantics)}
\]

Case 5: \( r = s^* \) for regexp \( s \). The inductive hypothesis applies to \( s \), so we have
\[
L((s^*)^R) = L((s^R)^*) \quad \text{(definition of } (s^*)^R) \\
= L(s^R)^* \quad \text{(regexp semantics)} \\
= (L(s)^R)^* \quad \text{(inductive hypothesis)} \\
= (L(s^*)^R) \quad \text{(Fact (5) of Lemma ??)} \\
= L(s^*)^R \quad \text{(regexp semantics)}
\]

This proves the claim.
Now Proposition ?? 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 ??.

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 \( r = b(a + bc^*)^* \).

\[
(b(a + bc^*)^*)^R = ((a + bc^*)^R)^R b = (a + (bc^*)^R)^* b
\]

\[
= (a + (c^R b^R)^* b = (a + (c^R)^* b^R) = (a + c^* b)^* b.
\]

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 ?? is to transform an \( \epsilon \)-NFA.

**Second proof of Proposition ??**. Let \( A \) be an \( \epsilon \)-NFA recognizing \( L \). We can assume that \( A \) has only one final state (say, by making \( A \) clean). Let \( B \) be the \( \epsilon \)-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 \( \epsilon \)), 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 \( L(B) = L(A)^R \).

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 17.7.** 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,

\[
L \cap M = \overline{L} \cup \overline{M}.
\]

Since regularity is preserved under complements and unions, the right-hand side is regular, and so \( L \cap M \) is regular.
Corollary 17.8. 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. \qed

18 Lecture 18

Next we consider images and inverse images under string homomorphisms

Definition 18.1. Let $\Sigma$ and $T$ be alphabets. A string homomorphism (or just a homomorphism) from $\Sigma^*$ 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 $\Sigma^*$, 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: $w = w_1w_2\cdots w_n$ for some $n \geq 0$. And so if $h$ is a homomorphism,

$$h(w) = h(w_1w_2\cdots w_n) = h(w_1)h(w_2)\cdots h(w_n)$$

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 $\varepsilon$, 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 $h(abaccab) = (01)(110)(01)(\varepsilon)(01)(110) = 01110101110$.

Definition 18.2. Let $\Sigma$ and $T$ be alphabets, and let $h$ be a homomorphism from $\Sigma^*$ 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 18.3. 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 $h^{-1}(M)$.  

42
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. \( h(\{a\}) = \{h(a)\} \) for any \( a \in \Sigma \).
3. For any \( L, M \subseteq \Sigma^* \), \( h(L \cup M) = h(L) \cup h(M) \).
4. For any \( L, M \subseteq \Sigma^* \), \( h(LM) = h(L)h(M) \).
5. 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 \( r = a \) 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 \( \{h(a)\} \)).
3. 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.)
4. If \( r = st \) for some \( s \) and \( t \), then define \( h(r) := h(s)h(t) \).
5. 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 \( r = b(a + bc^*)^* \).

\[
\begin{align*}
    h(b(a + bc^*)^*) &= h(b)h((a + bc^*)^*) = h(b)(h(a + bc^*))^* = h(b)(h(a) + h(bc^*))^* \\
    &= h(b)(h(a) + h(b)h(c^*))^* = h(b)(h(a) + h(b)h(c))^* \\
    &= 110(01 + 110(e^*))^* = 110(01 + 110)^* .
\end{align*}
\]

Thus if \( L \) is given by \( b(a + bc^*)^* \), 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 \( \Sigma \).
The transition function $\gamma$ 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

$$\hat{\gamma}(q_0, w) = \gamma(\cdots \gamma(\gamma(q_0, w_1), w_2) \cdots , w_n) = \hat{\delta}(\cdots \hat{\delta}(\delta(q_0, h(w_1)), h(w_2)) \cdots , h(w_n))$$

$$= \hat{\delta}(q_0, h(w_1)h(w_2) \cdots h(w_n)) = \hat{\delta}(q_0, h(w_1 w_2 \cdots w_n)) = \hat{\delta}(q_0, h(w)).$$

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:

![DFA](image)

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 $h(c) = \epsilon$, 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 $B$:

![DFA](image)
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.

### 18.1 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. There are languages that are not regular and yet are pumpable (we will see an example below), and so we can’t prove that they are not regular directly using the pump 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 $L := \{0^n1^m \mid n \neq m \}$. 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 18.4.** The language $L := \{0^n1^m \mid n \neq m \}$ 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 $L_2 := \{0^n1^m \mid m, n \geq 0 \}$ 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 $\{0^n1^m \mid n = m \} = \{0^n1^n \mid n \geq 0 \}$, 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 $s := (a + b + c + d + \epsilon)$, the language $D$ is the regular language given by the regular expression

$$D := L(s^*(asa + bsb + csc + dsd)s^*).$$

45
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 $s = w_1w_2 \ldots w_n$ 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 $w_\ell$, 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 $w_k$ 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 18.5.** 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 $h(w)$ as 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

\[
L' := h^{-1}(E) = \{w \in \{0, 1\}^* | w \text{ has the same number of 0's as 1’s} \}.
\]

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.

---

**19 Lecture 19**

**19.1 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:** Consider this 7-state DFA that accepts a binary string iff its second to last symbol is 1:

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$q_0$</td>
<td>$q_0$</td>
<td>$q_1$</td>
</tr>
<tr>
<td>$q_1$</td>
<td>$q_{10}$</td>
<td>$q_{11}$</td>
</tr>
<tr>
<td>$q_{00}$</td>
<td>$q_{00}$</td>
<td>$q_{01}$</td>
</tr>
<tr>
<td>$q_{01}$</td>
<td>$q_{10}$</td>
<td>$q_{11}$</td>
</tr>
<tr>
<td>*$q_{10}$</td>
<td>$q_{00}$</td>
<td>$q_{01}$</td>
</tr>
<tr>
<td>*$q_{11}$</td>
<td>$q_{10}$</td>
<td>$q_{11}$</td>
</tr>
</tbody>
</table>

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 19.1.** Let $N = (Q, \Sigma, \delta, q_0, F)$ 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 $N_q := (Q, \Sigma, \delta, q, F)$, 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 Section ??.

**Definition 19.2.** Let $L = \langle Q, \Sigma, \delta, q_0, F \rangle$ be any language over alphabet $\Sigma$.

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 $L = L_\epsilon$.

**Lemma 19.3.** 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

$$L_w = L(N_q).$$

(2)

It follows that $C_L = \{ L(N_q) \mid q \in Q \}$, and so $\|C_L\| \leq \|Q\|$. 

47
Proof. For any string \( x \in \Sigma^* \),
\[
x \in L_w \iff wx \in L \iff \hat{\delta}(q_0, wx) \in F \iff \hat{\delta}(\hat{\delta}(q_0, w), x) \in F \iff \hat{\delta}(q, x) \in F \iff x \in L(N_q).
\]
This shows that \( L_w = L(N_q) \), from which it follows immediately that \( \mathcal{C}_L \subseteq \{L(N_q) \mid q \in Q\} \). The fact that \( \{L(N_q) \mid q \in Q\} \subseteq \mathcal{C}_L \) 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 19.4.** If \( L \) is regular, then \( \mathcal{C}_L \) is finite.

## 20 Lecture 20

Lemma ?? below is essentially the converse of Lemma ??.

**Lemma 20.1.** Let \( L \) be any language over \( \Sigma \), let \( w \) and \( w' \) be any strings in \( \Sigma^* \), and let \( a \) be any symbol in \( \Sigma \). 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 \( \Sigma^* \). Then
\[
x \in L_{wa} \implies wx \in L \implies ax \in L_w \implies ax \in L_{w'} \implies w'ax \in L \implies x \in L_{w'a}.
\]
Thus \( L_{wa} \subseteq L_{w'a} \).

**Lemma 20.2.** Let \( L \subseteq \Sigma^* \) be any language over \( \Sigma \). If \( \mathcal{C}_L \) is finite, then \( L \) is recognized by the following minimal DFA:
\[
N_{\text{min}} := (\mathcal{C}_L, \Sigma, \delta_{\text{min}}, q_{0,\text{min}}, F_{\text{min}}),
\]
where
- \( q_{0,\text{min}} := L_\epsilon = L \),
- \( \delta_{\text{min}}(L_w, a) := L_{wa} \) for all \( w \in \Sigma^* \) and \( a \in \Sigma \), and
- \( F_{\text{min}} := \{L' \in \mathcal{C}_L \mid \epsilon \text{ is in } L'\} \).

Note that the transition function \( \delta_{\text{min}} \) is well-defined because of Lemma ??.

The output state \( L_{wa} \) only depends on the language \( L_w \), and does not change if we substitute another string \( w' \) such that \( L_w = L_{w'a} \).

**Proof of Lemma ??**. Fix a string \( w \in \Sigma^* \). First we prove that
\[
L_w = \hat{\delta}_{\text{min}}(q_{0,\text{min}}, w).
\]
This may be obvious, based on how we defined \( \delta_{\text{min}} \) but we’ll prove it anyway by induction on \(|w|\).

**Base case:** \(|w| = 0\). In this case, \( w = \epsilon \), and we have
\[
L_w = L_\epsilon = q_{0,\text{min}} = \hat{\delta}_{\text{min}}(q_{0,\text{min}}, \epsilon) = \hat{\delta}_{\text{min}}(q_{0,\text{min}}, w).
\]
**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 (20.3) holds for $x$ instead of $w$ (that is, assuming that $L_x = \hat{\delta}_{\text{min}}(q_{0,\text{min}}, x)$), we get

$$L_w = L_{xa} = \delta_{\text{min}}(L_x, a) = \delta_{\text{min}}(\hat{\delta}_{\text{min}}(q_{0,\text{min}}, x), a) = \hat{\delta}_{\text{min}}(q_{0,\text{min}}, xa) = \hat{\delta}_{\text{min}}(q_{0,\text{min}}, w).$$

Now we can show that $L = L(N_{\text{min}})$:

$$w \in L \iff we \in L_{qa} \iff e \in L_w \iff L_w \in F_{\text{min}} \iff \hat{\delta}_{\text{min}}(q_{0,\text{min}}, w) \in F_{\text{min}} \iff w \in L(N_{\text{min}}).$$

Finally, $N_{\text{min}}$ is a minimal DFA by Lemma 20.3.

**Corollary 20.3.** If $C_L$ is finite, then $L$ is regular.

**Theorem 20.4 (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 20.2 and 20.3. For the second sentence, we already constructed a minimal DFA $N_{\text{min}}$ recognizing $L$ with state set $C_L$ in Lemma 20.2. The only thing left to show is that $N_{\text{min}}$ is unique among minimal DFAs recognizing $L$.

To that end, we first show that the map $q \mapsto L(N_q)$ of Lemma 20.2 preserves the structure of the DFA. As in Lemma 20.2, 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 20.2. The correspondence $q \mapsto L(N_q)$ mapping $Q$ (the state set of $N$) onto $C_L$ (the state set of the DFA $N_{\text{min}}$ constructed in the proof of Lemma 20.2) 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 $L(N_{q_0}) = L(N) = L = L_{q_0} = q_{0,\text{min}}$, and so the start state $q_0$ of $N$ is mapped to the start state $q_{0,\text{min}}$ of $N_{\text{min}}$.

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 (20.3) of Lemma 20.2 twice—first for $q$ then for $r$—we get

$$\delta_{\text{min}}(L(N_q), a) = \delta_{\text{min}}(L_w, a) = L_{wa} = L(N_r),$$

the last equality holding because $r = \delta(q, a) = \hat{\delta}(q_0, w, a) = \hat{\delta}(q_{0,\text{min}}, wa)$. 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 $N_{\text{min}}$.

3. For any $q \in Q$,

$$q \in F \iff e \in L(N_q) \iff L(N_q) \in F_{\text{min}}.$$

Thus the accepting states of $N$ map to accepting states of $N_{\text{min}}$, and the rejecting states of $N$ map to rejecting states of $N_{\text{min}}$. 

49
Now suppose that $N$ is minimal. Since $N$ and $N_{\text{min}}$ 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 $N_{\text{min}}$ are the same DFA, via the relabeling $q \leftrightarrow L(N_q)$.

### 20.1 Constructing the minimal DFA

The proof of Theorem ?? holds the seeds of an algorithm for converting a sane DFA $N$ into its minimal equivalent DFA $N_{\text{min}}$.

**Definition 20.5.** 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 20.6.** 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 $N_{\text{min}}$. 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 $N_{\text{min}}$.

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.

**Input:** a DFA $N = \langle Q, \Sigma, \delta, q_0, F \rangle$.

1. Initialize a two-dimensional array $T[p, q]$, where $p, q \in Q$ so that all its entries are blank. As we find a pair of states to be distinguishable, we will mark the corresponding entry of $T$ with an $X$. (Invariant: $T[p, q] = T[q, p]$ and $T[p, p]$ is always blank, for all $p, q \in Q$.)

2. **Step 0:** For every pair of states $(p, q)$ of $N$ such that one of $p$ and $q$ is in $F$ but not both, mark $T[p, q] := T[q, p] := X$. (States $p$ and $q$ are distinguished by $\epsilon$.)

3. **Step $k = 1, 2, 3, \ldots$** If there exists a pair of states $(p, q)$ such that
   - $T[p, q]$ is blank and
   - there exists $a \in \Sigma$ such that $T[\delta(p, a), \delta(q, a)] = X$,

   then mark $T[p, q] := T[q, p] := X$ and repeat. (If $\delta(p, a)$ and $\delta(q, a)$ are distinguished by some string $w$, then $p$ and $q$ are distinguished by $aw$.)

4. Otherwise stop.

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]
21 Lecture 21

Context-free languages and grammars (Chapter 5). What is a context-free grammar? It is a way of denoting a language. Productions and derivations. Variables, nonterminals, or “syntactic categories.” Examples: \{0^n1^n \mid n \geq 0\}, palindromes. All regular languages can be denoted by grammars, but grammars can also denote nonregular languages.

22 Lecture 22

Sentential forms, the ⇒ and ⋆⇒ operators. Leftmost and rightmost derivations.

Parse trees, yield of a parse tree. Equivalence with derivations. 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.

More examples: \{a^m b^m c^n \mid m, n \geq 0\}, \{a^i b^j c^k \mid i \leq j\}, etc.

A grammar for expressions in arithmetic:

\[
E \rightarrow E + E \\
E \rightarrow E - E \\
E \rightarrow E \ast E \\
E \rightarrow E/E \\
E \rightarrow (E) \\
E \rightarrow c \\
E \rightarrow v
\]

Parse tree for \(v + c - v \ast (v + c)\).

Conventions and shorthand: head of first production is start symbol, can collapse productions with same head with the \(|\) separator, etc.

23 Lecture 23

Ambiguity. Example: two parse trees for \(c + c \ast c\). 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:

\[
E \rightarrow E + E \mid E - E \mid E \ast E \mid E/E \mid (E) \mid c \mid v
\]

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

\[ E \to E + T \mid E - T \mid T \]
\[ T \to T * F \mid T/F \mid F \]
\[ F \to c \mid v \mid (E) \]

So, for example: \( E \Rightarrow T \pm T \pm \cdots \pm T \), 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.

24 Lecture 24

Push-down automata (PDAs). Basically, an \( \epsilon \)-NFA with a stack. Informal example recognizing \{0^n1^n | n ≥ 0\}. Formal definition, IDs, and the turnstile relation. Example of an execution trace.

Equivalence between final-state and empty-stack acceptance criteria: \( L(P) \) versus \( N(P) \).

**Theorem 24.1.** 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 \( G \).

Do (1) \( \iff \) (2).

25 Lecture 25

Do (3) \( \implies \) (2) today. Get a 1-state PDA (top-down parser). Proof of correctness idea: For the steps in a computation, the strings \( \alpha \beta \), where \( \alpha \) is the input string read so far and \( \beta \) is the current stack contents, trace out a leftmost derivation of the input string \( w \) (and vice versa). This is shown by induction on the number of transitions taken so far.

26 Lecture 26

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 26.1.** A restricted PDA is a PDA \( P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F) \) such that, for every \( q \in Q \), \( a \in \Sigma \cup \{\epsilon\} \), 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. \((r,\epsilon)\) for some \(r \in Q\).

A transition of form (1.) we call \textit{push} \(Y\) and abbreviate it \((r,\text{push } Y)\). A transition of form (2.) we call \textit{pop} and abbreviate it \((r,\text{pop})\).

This does not decrease the power of a PDA. Restricted PDAs can recognize the same languages as general PDAs.

**Lemma 26.2.** For every PDA \(P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)\), there is a restricted PDA \(P'\) with the same input alphabet \(\Sigma\) 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 \(\delta'\) are of the following types:

1. \(\delta'(p_0, \epsilon, X_0) := \{(q_0, \text{push } Z_0)\}\);

2. for all \(q \in Q\), \(\delta'(q, \epsilon, X_0) := \{(e, \text{pop})\}\);

3. for every transition \((r, \gamma) \in \delta(q,a,X)\), where \(q,r \in Q\), \(a \in \Sigma \cup \{\epsilon\}\), \(X \in \Gamma\), and \(\epsilon \neq \gamma = Y_k \cdots Y_1\) for some \(k \geq 1\) and \(Y_1, \ldots, Y_k \in \Gamma\), we replace this transition in \(\delta'\) 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 \(\delta'(s_{i-1}, \epsilon, Y) := \{(s_i, \text{push } Y_i)\}\).

4. All other sets \(\delta'(q,a,X)\) are empty.

The idea in (3.) is that instead of replacing \(X\) by \(\gamma\) on the stack all at once, we cycle through some new intermediate states, first popping \(X\) then pushing on \(\gamma\) one symbol at a time, eventually arriving at state \(r\). Note that if \(\gamma = \epsilon\), 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

\[
(q,w,\alpha X_0) \vdash_{P'}^* (r,\epsilon,\beta X_0) \iff (q,w,\alpha) \vdash_{P}^* (r,\epsilon,\beta)
\]  

(3)

for all \(q,r \in Q\), \(w \in \Sigma^*\), and \(\alpha, \beta \in \Gamma\). It follows from this that, for all \(w \in \Sigma^*\),

\[
w \in L(P') \iff (\exists r \in F)(\exists \gamma \in \Gamma^*)[(q_0, w, X_0) \vdash_{P'}^* (r, \epsilon, \gamma X_0)]
\]

\[
\iff (\exists \gamma \in \Gamma^*)[(q_0, w, Z_0 X_0) \vdash_{P'}^* (r, \epsilon, \gamma X_0)]
\]

\[
\iff (q_0, w, Z_0) \vdash_{P}^* (r, \epsilon, \gamma)
\]

\[
\iff w \in L(P).
\]

53
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 (??) above, and the last equivalence is the definition of final-state acceptance in \( P \).

Similarly,

\[
\begin{align*}
w \in N(P') & \iff (\exists r \in Q')[(p_0, w, X_0) \vdash^*_{P'} (r, \epsilon, \epsilon)] \\
& \iff (p_0, w, X_0) \vdash^*_{P'} (e, \epsilon, \epsilon) \\
& \iff (\exists r \in Q')[(q_0, w, Z_0X_0) \vdash^*_{P'} (r, \epsilon, X_0)] \\
& \iff (\exists r \in Q')[(q_0, w, X_0) \vdash^*_{P'} (r, \epsilon, \epsilon)] \\
& \iff w \in N(P).
\end{align*}
\]

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 (??) 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 ???. By Lemma ???, 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
     
     \[ S \to [q_0Z_0r] \]
     
     (these are the only productions with head \( S \)),
  
  2. for every transition \((r, \text{pop}) \in \delta(q, a, X)\), where \( q, r \in Q \), \( a \in \Sigma \cup \{\epsilon\} \), and \( X \in \Gamma \), the production
     
     \[ [qXr] \to a \]
     
     and
  
  3. for every transition \((r, \text{push} Y) \in \delta(q, a, X)\), where \( q, r \in Q \), \( a \in \Sigma \cup \{\epsilon\} \), 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 \(\Sigma^*\) 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 \(w \in \Sigma^*\):

\[
[pXq] \Rightarrow w \iff (p, w, X) \vdash^* (q, \epsilon, \epsilon).
\] (4)

This can be proved by induction, and it follows from this and the \(S\)-productions that

\[
w \in L(G_P) \iff S \Rightarrow^* w
\]

\[
\iff (\exists r \in Q)[[q_0 Z_0 r] \Rightarrow w]
\]

\[
\iff (\exists r \in Q)[(q_0, w, Z_0) \vdash^* (r, \epsilon, \epsilon)]
\]

\[
\iff w \in N(P).
\]

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. \(\delta(p, \epsilon, Z_0) = \{(p, \text{pop})\}\).

One can check that \(N(P) = \{0^n 1^n \mid n \geq 1\}\). The grammar \(G_P\) then has the following productions:

\[
S \rightarrow [qZ_0 q] \mid [qZ_0 p]
\]

\[
[qXp] \rightarrow 1
\]

\[
[pXp] \rightarrow 1
\]

\[
[pZ_0 p] \rightarrow \epsilon
\]

\[
[qZ_0 q] \rightarrow 0[qXq][qZ_0 q] \mid 0[qXp][pZ_0 q]
\]

\[
[qZ_0 p] \rightarrow 0[qXq][qZ_0 p] \mid 0[qXp][pZ_0 p]
\]

\[
[qXq] \rightarrow 0[qXq][qXq] \mid 0[qXp][pXq]
\]

\[
[qXp] \rightarrow 0[qXq][qXp] \mid 0[qXp][pXp]
\]

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 $H = [pXq]$:

- $S \rightarrow D \mid E$
- $A \rightarrow 1 \mid 0FA \mid 0AB$
- $B \rightarrow 1$
- $C \rightarrow \epsilon$
- $D \rightarrow 0FD \mid 0AG$
- $E \rightarrow 0FE \mid 0AC$
- $F \rightarrow 0FF \mid 0AH$

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 $\epsilon$, we can bypass these two productions, substituting 1 and $\epsilon$ directly for $B$ and $C$ respectively in the bodies of the other productions:

- $S \rightarrow D \mid E$
- $A \rightarrow 1 \mid 0FA \mid 0A1$
- $D \rightarrow 0FD$
- $E \rightarrow 0FE \mid 0A$
- $F \rightarrow 0FF$

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:

- $S \rightarrow D \mid E$
- $A \rightarrow 1 \mid 0A1$
- $E \rightarrow 0A$

Removing $F$ eliminated the only remaining $D$-production, and so any productions involving $D$ are useless and can be removed:

- $S \rightarrow E$
- $A \rightarrow 1 \mid 0A1$
- $E \rightarrow 0A$

Finally, the only places where $E$ occurs are in the two productions $S \rightarrow E$ and $E \rightarrow 0A$, and so we can bypass the $E$-production entirely:

- $S \rightarrow 0A$
- $A \rightarrow 1 \mid 0A1$

Now it should be evident that the language of this grammar is indeed $N(P) = \{0^n1^n \mid n \geq 1\}$. An even simpler equivalent grammar is

- $S \rightarrow 0S1 \mid 01$
27 Lecture 27

The pumping lemma for context-free languages: proof and applications (\(L = \{a^m b^n c^n d^n | m, n \geq 0\}, \ L = \{a^i b^j c^k | 0 \leq j \leq k \leq \ell\}\), etc.).

Lemma 27.1 (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., \(wy \neq \epsilon\)); and for all \(i > 0\), \(vw^i xy^i z \in L\).

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 \(vxz = vw^0 xy^0 z\). This shows that \(vw^0 xy^0 z \in L\). 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 \(vw^i xy^i z\), and hence the latter string is also in \(L\). This shows that \(vw^i xy^i z \in L\) for all \(i \geq 0\).

Finally we verify that \(|wy| > 0\). Suppose \(|wy| = 0\). Then \(w = y = \epsilon\), 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?
28 Lecture 28

Introduction to Turing machines (TMs). Idea: a clerk (or human computer) with a finite number of mental states inspects symbols on a tape, one at a time; equipped with a pencil and eraser, the clerk can change a symbol, change mental state, and move left or right one square.

Definition is similar to a PDA, but now everything is on the tape, including input, arbitrary back-and-forth steps can be made, and symbols can be overwritten. The tape is infinite, so the computation has no a priori space or time limits, but at any time during the computation, only finitely many cells are nonblank. We will assume that TMs are deterministic (one can define a nondeterministic TM analogously to an NFA or PDA).

Definition of TM as a tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$, where

- $Q$ is a finite set (the state set),
- $\Sigma$ is an alphabet (the input alphabet),
- $\Gamma$ is an alphabet (the tape alphabet), and we have $\Sigma \subseteq \Gamma$ (by relabeling if necessary, we also can assume that $\Gamma \cap Q = \emptyset$),
- $\delta$ is the transition function, a partial function $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$,
- $q_0 \in Q$ is the start state,
- $B \in \Gamma - \Sigma$ is the blank symbol, and
- $F \subseteq Q$ is the set of accepting states.

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)

29 Lecture 29

Instantaneous descriptions (IDs) of a TM computation. Formal definition of computation: initial conditions, single moves (turnstile relation), halting configurations, the $\vdash$ operator. The language recognized by a TM. Deciders and decidable languages.

TMs also compute functions. Formal definition.

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.
30 Lecture 30

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)

\[ H_D := \{ \langle M \rangle \mid M \text{ is a TM that eventually halts on input } \langle M \rangle \} \]

**Theorem 30.1.** \( H_D \) is undecidable.

(The proof uses Cantor-style diagonalization.)

31 Lecture 31

Other undecidable problems:

\[ H := \{ \langle M, w \rangle \mid M \text{ is a TM that eventually halts on input } w \} \]
\[ H_\epsilon := \{ \langle M \rangle \mid M \text{ is a TM that eventually halts on input } \epsilon \} \]
\[ I_G := \{ \langle G_1, G_2 \rangle \mid G_1 \text{ and } G_2 \text{ are CFGs and } L(G_1) \cap L(G_2) \neq \emptyset \} \]
\[ E_G := \{ \langle G \rangle \mid G \text{ is a CFG that yields all strings over its input alphabet} \} \]

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 31.1.** \( H_\epsilon \) is undecidable.

**Proof.** Suppose that \( H_\epsilon \) 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 \( \epsilon \). 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 \( \epsilon \). Thus \( D \) rejects \( \langle R \rangle \), and so we reject.
Theorem 31.2. The $I_G$ is undecidable.

Proof. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$ be an arbitrary 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:

$$L_1 := \{ w x^R | w \text{ and } x \text{ are IDs of } M \text{ and } w \vdash x \}$$

$$L_2 := \{ w^R \$ x | w \text{ and } x \text{ are IDs of } M \text{ and } w \vdash x \}$$

Here is a grammar $F_1$ for $L_1$:

$$S_1 \rightarrow S_1 | S_1 - | O_1$$
$$O_1 \rightarrow aO_1a$$
(for each $a \in \Gamma$)
$$O_1 \rightarrow T_1$$
$$T_1 \rightarrow qacIcrb$$
(for each transition $(q, a) \rightarrow (r, b, R)$ and $c \in \Gamma$)
$$T_1 \rightarrow eqaIcbcr$$
(for each transition $(q, a) \rightarrow (r, b, L)$ and $c \in \Gamma$)
$$I \rightarrow aIa$$
(for all $a \in \Gamma$)
$$I \rightarrow B$$
$$B \rightarrow B | B - | \$$

Similarly, a grammar $F_2$ for $L_2$:

$$S_2 \rightarrow S_2 | S_2 - | O_2$$
$$O_2 \rightarrow aO_2a$$
(for each $a \in \Gamma$)
$$O_2 \rightarrow T_2$$
$$T_2 \rightarrow eqaIcrb$$
(for all $(q, a) \rightarrow (r, b, R)$ and $c \in \Gamma$)
$$T_2 \rightarrow qacIcbcr$$
(for all $(q, a) \rightarrow (r, b, L)$ and $c \in \Gamma$)
$$I \rightarrow aIa$$
(for all $a \in \Gamma$)
$$I \rightarrow B$$
$$B \rightarrow B | B - | \$$

($F_1$ 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 $\epsilon$ 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
\[
s' := q_0 . w_1^R w_2^R w_3^R \cdots (w_{n-1})^R w_n^R .
\]
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:
\[
q_0 . w_1^R w_2^R \cdots (w_{n-1})^R w_n^R .
\]
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:
\[
q_0 . w_1^R w_2^R \cdots (w_{n-1})^R w_n^R .
\]
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 \( \epsilon \).

Now the formal details. Let \( A \) be a new nonterminal generating all strings over \( \Gamma \), that is, \( A \) has productions \( A \to \epsilon \) 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:
\[
R_1 \to S_1 R_1 \mid H \mid \epsilon .
\]
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
\[
R_2 \to q_0 . C \\
C \to S_1 C \mid H^R \mid \epsilon
\]
Then \( G_1 \) and \( G_2 \) are as desired.

Acknowledgment

I’d like to thank several students, including Daniel Padé, for pointing out errors in these notes and suggesting improvements.