600.271 – Automata & Computation Theory

Lectures by Stephen Checkoway
Notes by David Li

Spring 2015

Introduction **2**

Lecture 1 (01/27/2015) **3**

Lecture 2 (01/29/2015) **6**

Lecture 3 (02/02/2015) **11**

Lecture 4 (02/05/2015) **14**

Lecture 5 (02/10/2015) **15**

Lecture 6 (02/12/2015) **17**

Lecture 7 (02/17/2015) **19**

Lecture 8 (02/19/2015) **21**

Lecture 9 (02/24/2015) **22**

Lecture 10 (02/26/2015) **24**

Lecture 11 (03/10/2015) **26**

Lecture 12 (03/12/2015) **27**

Introduction

600.271, or Automata & Computation Theory, is one of the required courses for a Computer Science major or minor at Johns Hopkins University. These notes are live-TEX'd, and these are my first attempt at live-TEXing notes. I used the TikZ package to draw the automata in this document.

I am responsible for all faults in this document, mathematical or otherwise; any merits of the material here should be credited to the lecturer, not to me.

Please email any corrections or suggestions to dli44@jhu.edu.

Lecture 1 (01/27/2015)

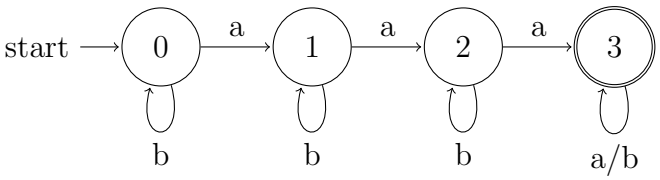
What is Automata & Computation Theory?
It's all about computation and its limits. There are three parts to this class:

1. Simple models of computation
2. What things can we compute?
3. What things can we compute efficiently?

For the most part, we will be covering decision problems, that is, problems whose solution is either yes or no.
The first model of computation we will be dealing with is one with a finite and fixed amount of memory. We represent the memory in a finite number of states.

Example. Does an input string have 3 'a's?

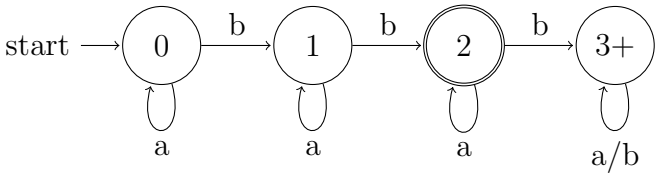
In this first model, we read input one character at a time.



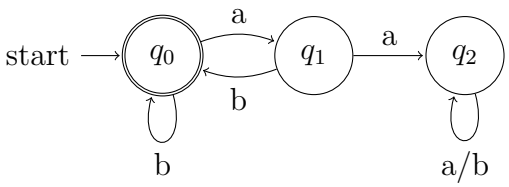
These are called **Deterministic Finite Automaton** or **DFA** for short. These have the following three properties:

- There is one start state.
- There is a set of accept states.
- For each state q and input character σ, the DFA has a unique transition from q to some state on input σ.

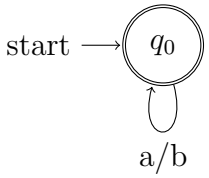
Example. Does the input string contain exactly two 'b's?



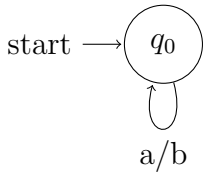
Example. Every time an 'a' appears in the input, is it immediately followed by a 'b'?



Example. A DFA that accepts every string.



Example. A DFA that rejects every string.



We will now define terms that will be important for the rest of the class.

Definition. An **alphabet** is a finite, nonempty set. We usually denote an alphabet with Σ. Some examples of alphaets are $\Sigma_1 = \{a, b, c\}$ or $\Sigma_2 = \{0, 1, 2, 3\}$.

Definition. Elements of an alphabet are called **letters** or **symbols**. a is a letter of Σ_1.

Definition. A **string** or **word** over an alphabet is a finite sequence of letters or symbols. We denote the empty string by ε.

Definition. A **concatenation** of two strings x, y is xy, the string consisting of the letters of x followed by the letters of y. We also have $x\varepsilon = \varepsilon x = x$.

Definition. Given a string x, $x^k = \underbrace{xx\cdots x}_{k \text{ times}}$ or x concatenated with itself k times. We also define $x^0 = \varepsilon$

Definition. A language over an alphabet Σ is a set of strings over Σ. Note that a language *can* be infinite.

Example. Let $\Sigma = \{0, 1\}$. Suppose L_1 is the language of all strings with length less than or equal to 2. Then,

$$L_1 = \{\varepsilon, 0, 1, 00, 01, 10, 11\}.$$

Definition. Σ^* is the language of all strings over Σ.

Example. Suppose L_2 is the language of all strings ending in a 0. Then,

$$L_2 = \{w0 \mid w \text{ is a word in } \Sigma\} = \{w0 \mid w \in \Sigma^*\}.$$

We can also create new languages from existing ones using the operations on sets.

$$L_3 = L_1 \cup L_2$$
$$L_4 = L_2 - L_2$$
$$L_5 = a^n b^n \text{ for } a, b \in \Sigma, n \geq 0 = \{\varepsilon, ab, aabb, \ldots\}$$

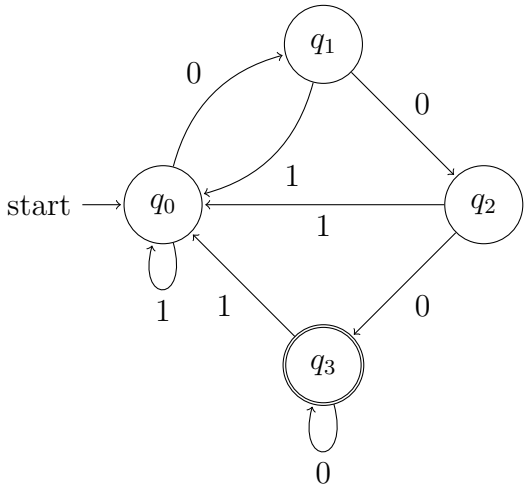
Notice that L_1 is a finite language. L_2, Σ^*, L_3, L_4, and L_5 are infinite languages. So, remember that languages can be infinite but elements of languages are *always* finite.

Definition. Given a string x, we define the length of a string, $|x|$ to be the number of characters in the string. For instance, $|\varepsilon| = 0$ and $|aabb| = 4$.

There is a connection between the DFAs and the theory of computation. Essentially, we can use DFAs to decide if a string is in a language? Can we check if a string is in L_5?
Mixed response from the students.
In fact, we cannot check if a string is in L_5. In order to do so, we would have to count the number of **a**s in the string, but since we have a fixed and finite number of states, we cannot verify strings of arbitrarily large lengths.

Lecture 2 (01/29/2015)

Today, we want to move onto a more mathematical description of DFAs rather than the informal description. Using a more mathematical model, we can describe the idea more concisely. Let's start with the following DFA:



Let's think about the strings that are accepted by this DFA. If we start at the start state, how can we get to the accept state? For instance, this machine accepts the string 000. In general, this machine accepts strings that end in three zeroes. Keep in mind that this also includes those strings that end in three or more zeroes. We say that the machine accepts any string in the set $\{w000 \mid w \in \Sigma^*\}$, where in this case, Σ^* is the binary alphabet of 0 and 1.
Earlier we talked about how we were interested in decision problems, which turns out to be equivalent to ask if a string is in a language. Recall that a language is a possibly infinite set of strings and that the strings are always finite. Note that there are several key features to every DFA:

- There is a *finite* set of states, which we call Q.
- There is an input alphabet associated with it, which we denote by Σ. Recall that this alphabet must be finite.
- There is a transition function that takes a state and an input and gives a state. We write this as $\delta : Q \times \Sigma \mapsto Q$.
- There are accept states or final states, which we denote by F. Note that $F \in Q$.

- There is a start state $q_0 \in Q$.

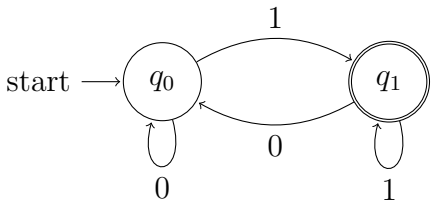
We say that a DFA M is a 5-tuple, where $M = (Q, \Sigma, \delta, q_0, F)$. If we have specified these five, then we have completely described the DFA. Let's write down the DFA from the first example.
We have

- $Q = \{A, B, C, D\}$
- $\Sigma = \{0, 1\}$.
- $\delta =$

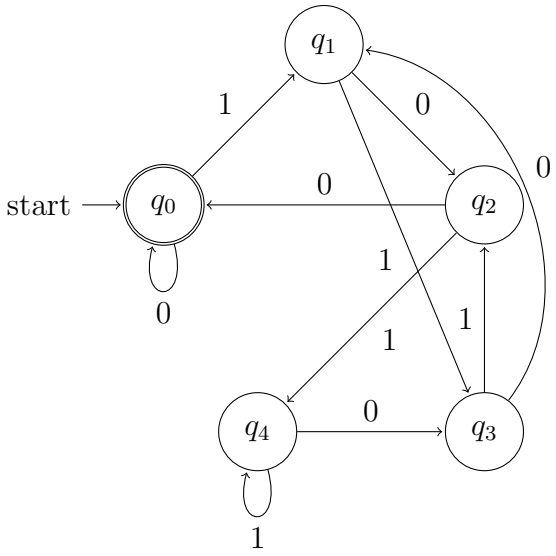
	A	B	C	D
0	B	C	D	A
1	A	A	A	A

  We can also specify it as follows: $\delta(A, 0) = B, \delta(B, 0) = C, \delta(C, 0) = D, \delta(D, 0) = D, \delta(q, 1) = A \forall q \in Q$.
- $q_0 = A$
- $F = \{D\}$.

We can also formalize the notion of a DFA accepting a string. We say that a DFA $M = (Q, \Sigma, \delta, q_0, F)$ **accepts** a string $w \in \Sigma^*$ if starting from the start state q_0 and moving between states according to the transition function δ, the machine ends in an accept state.
So, now we have the notion of a language of a machine. We call the set A of strings that a DFA accepts the **language** of M. It is written as $L(M) = A$.
We say that a machine M **recognizes** a set B if the language of M is equal to B.
If we consider our input over $\{0, 1\}$ as binary numbers, then the language of odd numbers is $\{w1 \mid w \in \Sigma_1\}$ where $\Sigma_1 = \{0, 1\}^*$. Say we want to build a DFA that recognizes this language.
We would have two states. We start in one state and the other state is the final state. If we meet a 1, we would go to our final state and otherwise, we go back to the start state.



Let's consider the language $\Sigma = \{0, 1\}$ and build a DFA that recognizes the set of strings $\{w \mid$ treating w as a binary number, w is a multiple of $5\}$.
We would have five states labeled from 0 to 4(considering everything modulo 5). 0 is our start state and our accept state. When we encounter a 0, we double the number of the state and go to that state. When we encounter a 1, we double the number of the state and add 1 and go to that state. See the following diagram:



Well, why does this work? We have exploited the fact that we can write a binary number a $b_n 2^n + b^{n-1}2^{n-1} + \cdots + b_1 2 + b_0$. When we get a new digit, we move everything over one place and input our new digit. And since we're working in modulo 5, we only need to keep track of the number modulo 5. This is kind of messy but if we write it down in a more mathematical notation, it might not be quite so bad. We have $Q = \{0, 1, 2, 3, 4\}, F = \{0\}, q_0 = 0$. We now need to specify δ. $\delta : Q \times \Sigma \mapsto Q$ is defined as $\delta(Q, b) = 2Q + b \mod 5$.
Now let's say we want to generalize this to any fixed value of n. Let L_n be a language where we have $\{w \mid w \text{ a binary number divisible by } n\}$. Here, we can't just write down a diagram because we don't have an explicit value for n. Maybe if $n = 100$ or $n = 1000000$ we can draw the DFA, but we want to solve this for the family of languages $L_1, L_2, \cdots$. We can build a DFA $M_n = (Q_n, \Sigma_n, \delta_n, q_{0_n}, F_n)$. We can specify these as follows:

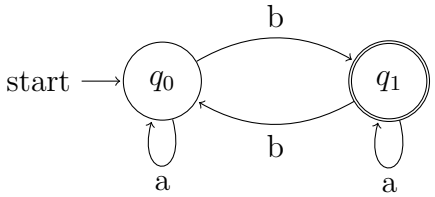
- $Q_n = \{0, 1, 2, \cdots, n\}$
- $\Sigma_n = \{0, 1\}$
- $\delta(Q, b) = 2Q + b \mod n$
- $q_{0_n} = 0$
- $F_n = \{0\}$

This is neat, since we have found the solution for this whole family. In the case L_5, note that this accepts the empty string ε. Suppose we wanted to exclude ε. What would we do to change the DFA? We would create a new start state that would have the same transitions as the start state of the old machine.
Now, we want to formalize the notion of computation. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA and let $x = x_1 x_2 \cdots x_n$ be a string with each $x_i \in \Sigma$. We say that M accepts x if there exist states $r_0, r_1, \cdots r_n \in Q$ such that three conditions hold:

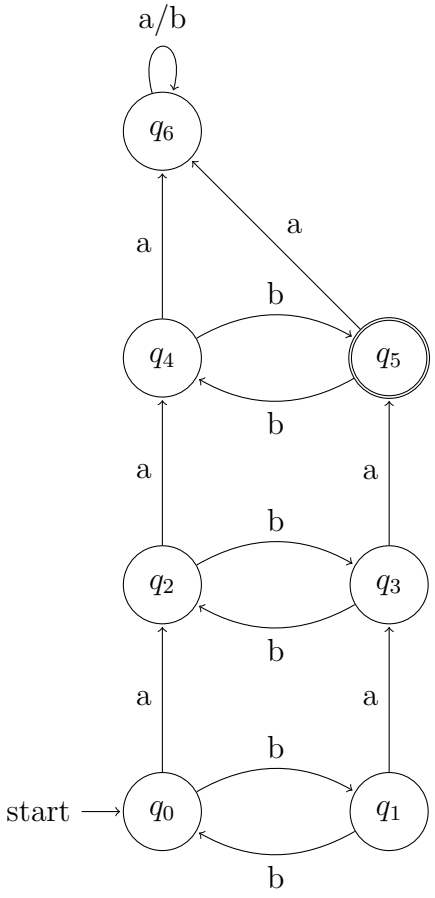
- $r_0 = q_0$

- $r_i = \delta(r_i - 1, x_i) \forall 1 \leq i \leq n$,
- $r_n \in F$.

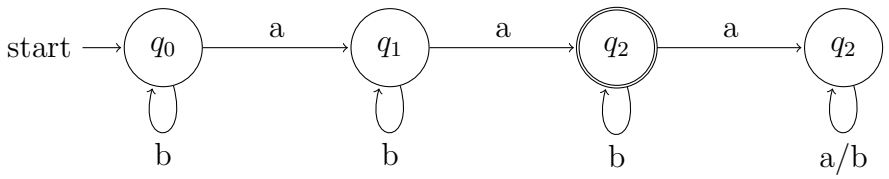
So, if we were to take a look at the DFA for L_5 again, and consider the string 1010, then it is pretty easy to see $r_0 = 0, r_1 = 1, r_2 = 2, r_3 = 0, r_4 = 0$ and we have ended up in an accept state. We call this sequence of states that a machine M goes through on an input x the **computation** of M on x. We call an accepting computation one that ends in an accept state.
One final definition, then more examples.
We call a language **regular** if it is recognized by some DFA.
We want to see what type of languages we can make from DFAs. Let's consider the alphabet $\{\texttt{a}, \texttt{b}\}$ and the language $L_1 = \{w \mid w \text{ has an odd number of } \texttt{b}\text{s}\}$.



We want to restrict this language a little bit, so we come up with a new language $L_2 = \{w \mid w \text{ has an odd number of } \texttt{b}\text{s and } \texttt{w} \text{ has exactly two } \texttt{a}\text{s}\}$.
This is a little more complex.



We've actually decomposed L_2 into two parts; the L_1 part and an L_3 part that verifies that there are two as. Note that $L_2 = L_1 \cap L_3$. If we draw the DFA for L_3, we find



that we've taken L_1 and "multiplied" it (in some sense) by L_3 and we have a new "Franken"-machine L_2. It turns out that we can do this in general. We have now come to the first theorem of the course!

Theorem 1. *If L_1 and L_2 are regular languages, then $L_1 \cap L_2$ is a regular language.*

Proof. We do a proof by construction. Since L_1 and L_2 are regular, there exist DFAs $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1), M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ such that $L(M_1) = L_1$ and $L(M_2) = L_2$. Now, we want to build a new machine $M = (Q, \Sigma, \delta, q_0, F)$ that recognizes the intersection of the languages. How do we keep track of this? We use pairs of states as our language $Q = Q_1 \times Q_2$. We want them to start in the respective start state of each machine, so this tells us $q_0 = (q_1, q_2)$. Also, we want to move each machine according to its own transition function, so we have $\delta((Q, R), x) = (\delta_1(Q, x), \delta_2(R, x))$. Finally, we determine F. We want the set of pairs (Q, R) such that $Q \in F_1, R \in F_2$, but this is the exact same as $F_1 \times F_2$. We claim that $L(M) = L_1 \cap L_2$. If we wanted to be extremely rigorous then we would need to use the accepting computations to show set equality. □

Lecture 3 (02/02/2015)

Recall the definition of a DFA – $M = (Q, \Sigma, \delta, q_0, F)$ and the definition of a regular language - a language that is recognized by some DFA.
Today, we're going to take a look at another operation. Recall that last time we showed that if languages A and B are regular, then their intersection $A \cap B$ is also regular. We'll start by taking a look at another statement of this form:

Theorem 2. *If a language A is regular, then the complement of the language, A^c is also regular.*

Proof. First we must understand what A^c means – because there must be a universe to take the complement. Then, $A^c = \{w \in \Sigma^* \mid w \notin A\}$. Then, if A is regular, then there is some DFA M such that $L(M) = A$. Our goal is to construct a new DFA, M' such that $L(M') = A^c$. Note the similarity to our proof that the intersection of two regular languages is regular. Consider $M = (Q, \Sigma, \delta, q_0, F)$ and $M' = (Q, \Sigma, \delta, q_0, F')$ where $F' = Q - F$. Now, we want to show two things: if some string $w \in L(M)$ then $w \notin L(M')$ and if some string $w \notin L(M)$ then $w \in L(M)$.
If $w \in L(M)$ then M ends in an accept state on input w, but by definition, w does not end in an accept state in M'. Conversely, if M does not end in an accept state on input w then M' will end in an accept state on input w.

□

What about this?

Theorem 3. *If A and B are regular languages, then $A \cup B$ is also regular.*

Proof. Although this *could* be done with a proof by construction, it is far easier to use the previous two theorems. We can use DeMorgan's Law to show that $A \cup B = (A^c \cap B^c)^c$. Here, we're basically done, since because A and B are regular, then A^c and B^c are regular so $A^c \cap B^c$ is regular and finally, $(A^c \cap B^c)^c$ must also be regular.

□

The book gives another one, which happens to be difficult to do with DFAs. We want to consider another operator (endswidth)
If we have a regular language L, then (endswith)(L) is some other language $\{xw \mid x \in \Sigma^*, w \in L$.
We want to show that (endswith)(L) is also regular. How do we go about proving this?
We want two types of boxes, one for L and one for Σ^*. The box for L is basically the DFA for L. We want to start going through Σ^* but at some point, we want to move over to the box for L. But when do we move over?

To answer this question, we will define a new type of machine.
A **nondeterministic finite automata** (**NFA**) is unlike a DFA in that it "gives us choices". There are multiple transitions on the same input so we can have something that looks like this:
We can have transitions on *no* input, with the transition usually denoted by ε. This means we haven't read any input but we still move from one state to another on the epsilon transition. Finally, we can have states without transitions for some or all input symbols.
Let's do some examples before going into the formal definition. We will fix our input alphabet to be $\{\mathtt{a}, \mathtt{b}\}$.
The way this works is that if you go into a state with no transitions, then that string is rejected. We say that $L(N_1) = \emptyset$. This is kinda boring, so let's try another one.
Let's try to figure out $L(N_2)$. We could go `aba`, or `bba`, `aa`. Let's call this language L_2. Now, we'll return to our (endswith) function. We didn't know when to transition from one "box" to another. With NFAs, we have an answer to this. The (endswith)(L_2) is those strings that end in `aba`, or `bba`, `aa`.
We can do this because we have the epsilon transition. Now that we have NFAs, it appears that we have more computation power. Let's try go about making this formal now. An NFA $N = (Q, \Sigma, \delta, q_0, F)$ with

- Q a finite set of states
- An alphabet Σ
- q_0 is still our start state
- $F \subseteq Q$ are our accept states
- $\delta(Q, \Sigma_\varepsilon)$ gives a set of states, so the image of δ is a subset of the power set of Q. $\Sigma_\varepsilon = \Sigma \cup \varepsilon$.

Let's try some examples. Consider the NFA N_3:
Here we have a somewhat complicated machine - $N_3 = (Q, \Sigma, \delta, q_0, F)$ with $Q = \{1, 2, 3\}, \Sigma = \{a, b\}, q_0 = 1, F = \{1\}$. Now what about δ? For δ, we have to build a table:

	`a`	`b`	ε
1	$\emptyset$	$\{2\}$	$\{3\}$
2	$\{2, 3\}$	$\{3\}$	$\emptyset$
3	$\{1\}$	$\emptyset$	$\emptyset$

Now, what does it mean for a NFA to accept a string? We say that a NFA $N = (Q, \Sigma, \delta, q_0, F)$ accepts a string w if we can write $w = x_1 x_2 \cdots x_m$ for $x_i \in \Sigma_\varepsilon$ (note that $|w| \leq m$ because some x_i could be ε) and there exist states $r_0, \cdots, r_m$ such that

1. $r_0 = q_0$
2. $r_i \in \delta(r_{i-1}, x_i)$
3. $r_m \in F$

If we were to follow this procedure for N_3 for input `baabaa`, we can follow $w =$`baaba`ε`a` in the machine
The last thing we'll want to do today is to show you that NFAs and DFAs are equivalent. There are some cases where a DFA is much harder to construct than an NFA.

Theorem 4. *For every NFA N there exists a DFA M such that $L(M) = L(N)$.*

Proof. We have our $N = (Q, \Sigma, \delta, q_0, F)$ and we are going to build a DFA $M = (Q', \Sigma, \delta', q_0', F')$. If we want to keep track of all of the possible places we could be in N, then what should our states look like in M? We should have $Q' = 2^Q$. Then where do we start? We should have $q_0' = E(\{q_0\})$. Then what should our F' be? It should be the set of subsets S of Q such that $S \cap F$ is not empty. We now need to handle δ. We need δ' So, $\delta'(S, a) = \bigcup_{q \in S} \delta(q, a)$. The last thing we need is to handle epsilon transitions.
For any state $S \in M$ (or $S \subseteq Q$), $E(S) = \{q\}$ such that q is reachable in S by following zero or more epsilon transitions. Then to handle epsilon transitions, we also need to alter our δ' so we define $\delta'(S, a) = E(\delta(S, a))$.

□

Lecture 4 (02/05/2015)

Lecture 5 (02/10/2015)

So far we've only dealt with the class of languages called regular languages, that is, those that can be recognized by a DFA or NFA. As it turns out we can build regular languages using smaller languages and operations on languages. But it's a little clumsy to define languages in terms of these operators.
Here, we have the language consisting of a single string, $\{a\}$. We can apply the Kleene star operator on this to get the set of all languages with zero or more `a`s. We can concatenate this with other singletons and use the union operators. Since regular languages are closed under the Kleene star and under concatenation and under unions, any combination of these describes a regular language. Since this is a little cumbersome, we're going to use a notation called **regular expressions**. This is nothing more than a shorthand for writing down a regular language. In some cases, it's very hard to write down a language using a regular expression but usually, it's pretty straightforward.
Let's try this on our language. Using this, in principle, we can construct any regular language. There are essentially six types of regular languages:

- $\emptyset$ - represents the empty language
- ε - represents the language consisting of the empty string
- `a` - represents the language consisting of the single string `a`
- If R_1, R_2 are regular expressions, $R_1 \cup R_2$ is a regular expression.
- If R_1, R_2 are regular expressions, $R_1 R_2$ is a regular expression.
- If R_1 is a regular expression, then $(R_1)^*$ is also a regular expression.

Then, what is the standard order of operations? The order is star, then concatenation, and finally union. As one final shorthand, we can write Σ to mean the union of all the elements in Σ.
Imagine we have the language $L_1 = \{wba \mid w \in \Sigma^*\}$, with $\Sigma = \{a, b\}$. If we were to write this as an equivalent regular expression, we could write this as $\Sigma^* ba$. What if we have $L_2 = \{w \mid w$ starts and ends with the same symbol.$\}$? We get $a\Sigma^* a \cup b\Sigma^* b \cup a \cup b \cup \varepsilon$.
I claimed earlier that regular expressions are equivalent to regular languages. We need to show that a language generated by regular expressions can be recognized by a DFA or NFA and that if a language is recognized by a DFA or NFA, then it can be generated by a regular expression.

Theorem 5. *A regular expression R can be converted to an equivalent NFA N. This means the language generated by R is equivalent to the language recognized by N.*

Proof. We're going to perform a type of induction called structural induction.
Base cases:

- $\emptyset$. We can build an NFA
- ε. We can build an NFA
- $a \in \Sigma$. We can build a NFA

For these, we have clearly built NFAs that are equivalent.
Inductive step: Assume there exists NFAs N_1, N_2 such that $L(N_1) = L(R_1)$ and $L(N_2) = L(R_1)$. Then, by definition, $L(R_1 \cup R_2) = L(R_1) \cup L(R_2)$. Similarly, $L(R_1R_2) = L(R_1)L(R_2)$ and $L(R_1^*) = L(R_1)^*$. By these equivalences, the first one is $L(N_1) \cup L(N_2)$, the second is $L(N_1)L(N_2)$ and the third is $L(N_1)^*$
Since regular languages are closed under these operations, and these are regular languages because N_1 and N_2 are NFAs. Therefore, we can build an NFA for any regular expression R. □

We can use the ideas of this proof to actually construct the NFA. We can start with a regular expression and build an NFA out of it. Let's say our regular expression is $a(ba)^* \cup b(ab)^*$. This is any alternating sequence of a and b with odd length. Let's convert this to an NFA. To do this, we need to do it a step at a time.
We can verify this using strings like `babab`. Note that this NFA isn't the smallest one that can do this, but it is one that works. We've only done one direction. The other direction has a pretty simple idea, but doing it is a hassle.

Theorem 6. *Every NFA has an equivalent regular expression.*

First, we prove the following lemma.

Lemma. Every NFA can be converted to an NFA with one accept state and no transitions to the start state and no transitions from the accept state

Proof. Add a new start state and a new accept state. Make all of the previous accept states transition to the new one with epsilon transitions, and make the new start state transition to the old start state by an epsilon transition. We could formalize this, but this is just the basic idea. □

Now we can prove the theorem.

Proof. First, we convert our NFA with a single accept state to a new type of machine, called a generalized NFA (GNFA) where the edges of the transitions are labeled with regular expressions. Next, we remove states from the GNFA one at a time producing equivalent GNFAs until we have a GNFA with one start state, one accept state, and one transition labeled by a regular expression. This gives us our equivalent regular expression. Now, you might ask how we remove states, since it's the central operation we're performing. First, just pick any state q_r different from the start and accept states. For any states q_i, q_j such that we have a transition r_1 from q_2

□

Lecture 6 (02/12/2015)

We're going to continue talking about regular languages today, and then we'll go into nonregular languages. To start, we're going to take a look at a DFA.
We want to see which string (with length > 4) are accepted by this DFA.
$w_1 = aaaa, w_2 = baab$ is in the language.
$w_3 = bbbb, w_4 = bbab, w_5 = abaabb$ is in the language.
All of these strings have this interesting property; they can be written as $xyz, x, y, z \in \Sigma^*$, where y is nonempty, but they also xy^iz is in the language for all values of i. It seems surprising at first glance, but if we let $x = \varepsilon, y = aa, z = aa$ and $xy^2z = aaaaaa$ is accepted by the DFA. Also, $xy^0z = aa \in L$. Now, what about w_2? Let $x = b, y = aa, z = b \implies xy^3z = ba^6b \in L$. For w_4, we can let $x = z = \varepsilon$ and $y = bbbb$, so that $b^{4i} \in L$ for all $i \in \mathbb{N}$.
Note that not all partitions into x, y, z will work, but for these strings, at least one will work. We might wonder if this is a property of the DFA itself or if this is just a property of the language. If we try to come up with another DFA for this language, we can get:
These DFAs have the same language, and this property still holds, even though the states and transitions have changed entirely.
We can take these special strings and partition them into three parts, and arbitrarily "blow-up" the middle part of the string and the resulting string will be in the language. We can give this property a name; a language L is **pumpable** if there exists an integer p such that for all strings $w \in L, |w| > p$, there is a partition $w = xyz$ such that

- $xy^iz \in L$ for all $i \in \mathbb{N}$
- y is nonempty.
- We can constrain the length of $xy \leq p$.

As it turns out this property is not exclusive to regular languages, but it turns out to be interesting and important to our study of regular languages.
Let's try another language $L_2 = \{a^ib^jc^k\}$ with the property that $i = 1 \implies j = k$.
Claim: L is pumpable with $p = 1$.

Proof. This actually requires five cases to deal with, but we'll just give the first part of the proof.
Case 1: $w = ab^jc^k$.
We want to be able to pump this: $x = \varepsilon, y = a, z = b^jc^k$. This satisfies the last two conditions quite easily. We check the first condition: $xy^nz = a^nb^jc^k$. Because of the previously mentioned property

of the set, $j = k$, so $a^n b^j c^j \in L$. If $n = 1$, we must have $j = k$, but this is already satisfied. If $n \neq 1$, we can let j, k be anything, and in particular, we are allowed to have $j = k$.
Case 2: $w = b^j c^k$.
$x = \varepsilon, y = b, z = b^{j-1}c^k$, assuming $j > 0$. If $j = 0$, set $y = c, z = c^{k-1}$. There are more cases to deal with, but these are increasingly *less* fun to deal with. □

So why do we care about this property? There is an important theorem called the **pumping lemma** that states regular languages are pumpable. However, this theorem doesn't say if a language is pumpable or not. The main use for this theorem is its contrapositive; if a language L is not pumpable, then L is not regular.
This is a complicated proof to write down, but the idea is quite simple. Essentially, when we traverse a DFA, we can traverse transitions until we get back to our original state; in our first drawing, we could traverse aa from state 2 to state 3 and back to state 2 any number of times (including zero times). The argument for the proof hinges on the pigeonhole principle; for our first drawing, $p = 4$. This is because if we go through 4 transitions, we need to have passed through 5 states, and since there are only four states, we must have visited at least one of the states more than once.
If a DFA Q has n states, then $p = n + 1$ is a valid pumping length. Although there is the notion of a minimal pumping length, I don't find it extremely enlightening to discuss.
We'll prove the pumping lemma now.

Theorem 7. *Every regular language is pumpable.*

Proof. Let $M = \{Q, \Sigma, \delta, q_0, F\}$ be a DFA that recognizes a language L and let p equal to the number of states in M. Let $s = s_1 s_2 \cdots s_n$ of length n where $n \geq p$. Let $r_1, \cdots, r_{n+1}$ be the states that M enters while processing s, so that $r_{i+1} = \delta(r_i, s_i)$. By the pigeonhole principle, there are distinct i, j such that $r_i = r_j$. As M is computing on s, it is moving from state to state until it reaches r_i. Let input it takes to get to r_i be our x. Therefore, we have $x = s_1 \cdots s_i$. Then, we keep computing until we reach r_i again; this next segment is our $y = s_{i+1} \cdots s_j$ and we let $z = s_j \cdots s_n$ be the rest. Since y takes r_i to r_j, and any y^i also takes r_i to r_j, and since r_{n+1} is an accept state, M must accept $xy^i z$. We know that $i \neq j$ so $|y| > 0$ and finally, $j \leq p + 1$, so $|xy| \leq p$. Thus we have satisfied all the properties of the pumping lemma. □

Since we have some time left, let's do an example to show that a language is not regular by showing that it can't be pumped. Next time, we'll also discuss other ways for showing this.
We'll start with a language we've talked about many times: $L = \{a^n b^n \mid n \in \mathbb{Z}\}$. To show L is not regular, we need to show the following:
For all pumping lengths p there exists $w \in L, |w| > p$ such that for all partitions of $w = xyz$, such that $|xy| \leq p, |y| > 0$ and there exists an i such that $xy^i z \notin L$.

Proof. Assume L is regular with pumping length p. Now we'll let $w = a^p b^p$, which has length $2p > p$. Let x, y, z be a partition of w such that y is nonempty and $|xy| \leq p$. Let's write down what x, y, z have to look like. Since xy has length at most p, xy consists of 0 or more as, so $x = a^i, y = a^j, z = a^{p-i-j}b^p$ for $j > 0$.
Consider $xy^2z = a^i a^{2j} a^{p-i-j} b^p = a^{p+j} b^p$. Since $j > 0, p + j \neq p$ so $xy^2z \notin L$ and L is not regular. □

Lecture 7 (02/17/2015)

Today we're going to go more into how we prove languages are *not* regular. If we want to show that a language is not regular, we have two ways we can go about doing this, and they're both proofs by contradiction. One way is to show that L is not pumpable, and the second is to use closure properties of regular languages. The second method requires some more insight, but the pumping lemma involves a more "brute-force" method.

Last time, we proved that $L = \{a^n b^n, n \geq 0\}$ is not a regular language. We said that if L is regular and has pumping length p, we let $w = a^p b^p$. Then by the pumping lemma, we know w can be partitioned into xyz such that $|xy| \leq p$ and $|y| > 0$. We then picked a value for i (the exponent of $xy^i z$) greater than 1, and noticed that since $|xy| \leq p$, they are composed only of **a**s, and $x = a^k, y = a^m, m > 0$ so $z = a^{p-k-m}b^p$ and then $xy^i z = a^{p+m}b^p \notin L$.
Let's consider the steps that we used to show a language is not regular using the pumping lemma.

1. We assume that our language L is regular with pumping length p.
2. Then choose some $w \in L$ where $|w| \geq p$. To do this, we choose some word in the language that depends on p in some way and makes our choice of x and y easy to deal with. For instance, we could have chosen $a^{p/2}b^{p/2}$, but this would have made x and y more difficult to deal with.
3. Then we have to consider all possible partitions $xyz = w$ such that $|xy| \leq p$ and $|y| > 0$.
4. Finally, we choose $i \neq 1$ such that $xy^i z \notin L$.

Notice that we must prove that every possible partition is not pumpable.

Let's try an example. Let $L_1 = \{a^m b^n \mid m \geq n\}$. First, we assume that L is regular with pumping length p. What w should we choose? We can pick $w = a^p b^p$. We can partition this as $x = a^k, y = a^m, z = a^{p-k-m}b^p$. Here, we consider $xy^0 z$, which is just xz. Then, we have $xy^0 z = a^{p-m}b^p$. Since $p > 0$, we have $xy^0 z \notin L$. This is an example of "pumping down", where the example before is "pumping up". In general, I find "pumping up" easier to do than "pumping down" because if you "pump down", you only left with xz instead of $xy^i z$ for some $i \geq 2$.

Let's consider $L_2 = \{a^{2^n} \mid n \in \mathbb{N}\}$. To show that this language is not regular, we can also apply the pumping lemma. First we assume L_2 is regular and has pumping length p. Now, what should we pick for p? We can pick $w = a^{2^p}$, which has length 2^p. We need to consider all possible partitions $w = xyz$ with $|xy| \leq p$ and $|y| > 0$. This gives us $x = a^m, y = a^n, z = a^{2^p - m - n}$. Here, we want to

pump up and show that what we end up with is not regular. We get $xy^2z = a^m a^{2n} a^{2^p - m - n} = a^{2^p + n}$. Since $m + n \leq p$, so $2^p + n \leq 2^p + p < 2^p + 2^p = 2^{p+1}$. Since $2^p < 2^p + n < 2^{p+1}$, $2^p + n$ is not a power of 2, and $xy^2z \notin L_2$

Consider $L_3 = \{ss \mid s \in \Sigma^*\}$. If $\Sigma = \{a\}$ for some symbol a, it's regular; $L_3 = (aa)^*$. We want to show that if Σ contains more than 1 letter, then L_3 is not regular. Let $\Sigma = \{a, b\}$, and let $w = a^p b^p a^p b^p$. Let $xyz = w$ with $|xy| \leq p$, $|y| > 0$ and $x = a^m, y = a^n, z = a^{p-m-n} b^p a^p b^p$. Then consider $xy^2z = a^m a^{2n} a^{p-m-n} b^p a^p b^p = a^{p+n} b^p a^p b^p$. Suppose $xyz^2 \in L_3$. Then, $xy^2z = ss$ for some string s. Since $|xy^2z| = 4p + n$, $|s| = 2p + \frac{1}{2}n$. Then, the first s starts at the beginning and ends somewhere in the first set of **b**s. But this means the second s must start somewhere in the **b**s. But the first s starts with a but the second s starts with a b. This is a contradiction and $xy^2z \notin L_3$.

As you can see, it's tricky and somewhat of a hassle to prove languages are not regular using the pumping lemma. Let's consider the other method, which is to use closure properties of regular languages. Let $L_4 = \{x \mid x$ has the same number of **a**s and **b**s, in any order$\}$. We could do this using the pumping lemma, not with great difficulty, as it turns out. To apply closure properties, assume L_4 is regular. Then, we apply the intersection properties; $L_4 \cap a^*b^*$ must be regular but $L_4 \cap a^*b^* = L = a^n b^n$ which is not regular. This is impossible, so our original assumption that L_4 is regular must be incorrect.

We'll try one that's a little more complicated: $L_5 = \{s \mid |s| = 2n, n \in \mathbb{N}$ with the first half is different from the second half$\}$. Here, we can consider a sequence of steps. Suppose L_5 is regular, then L_5^C must be regular. This consists of all strings w such that $|w| = 2n + 1, n \in \mathbb{N}$ and $w = ss$ for some string s. Then, so is $L_5^C \cap (\Sigma\Sigma)^*$. But this is L_3, which we proved was not regular. Then, L_5 must not be regular.

Let $L_6 = \{a^k b^n c^m$ where if $a = 1$ then $n = m\}$. We showed that this was pumpable, but claimed it was not regular. There are two ways to show that L_6 is nonregular. We can assume L_6 is regular and then consider L_6^R and then apply the pumping lemma, but that's not much fun. So, the other way we can do it is by assuming L_6 is regular and intersecting it with ab^*c^*, which we know is regular. Let f be a homomorphism such that $f(a) = \varepsilon, f(b) = a, f(c) = b$ and then $f(L_6 \cap ab^*c^*) = a^n b^n$ which is not regular. But since regular languages are closed under homomorphism, this is impossible. So, L_6 must not be regular.

Lecture 8 (02/19/2015)

Lecture 9 (02/24/2015)

Last time we were talking about Chomsky normal form. Recall that it is a grammar of the form
There is a theorem that says every grammar has an equivalent grammar in Chomsky normal form. We'll show this by walking through its construction. Consider a grammar that is composed of $S \mapsto SaSbS|SbSaB|\varepsilon$. This will generate all strings of even length with the same number of **a**s and **b**s. To change this into Chomsky normal form, we will add a new start variable $S_0 \mapsto S$. The second step is to transform the grammar so that the right-hand side of each rule has length at most two. We replace each rule of the form $A \mapsto XU$ where X is any terminal or variable and U is any string of terminals or variables, where U is a string of length ≥ 1. Basically, if we have a rule of length 3 or more, we want to condense it.

To do this, we would change $A \mapsto XU$ into $A \mapsto XB$ and $B \mapsto U$. This gives us $S \mapsto SA|SbSaS|\varepsilon$, where $A \mapsto aSbS$. We then have So all we have done is expanded out any rules with three or more terminals or variables and created rules with only two or fewer terminals or variables.

Third, we want to remove ε rules. The way we do this is by removing rules of the form $A \mapsto \varepsilon$ and by adding rules $B \mapsto$
Finally, we want to remove what we call "unit rules", rules of the form $A \mapsto B$, and add rules $A \mapsto u$ where $B \mapsto u$ is a rule.
Therefore, we can convert any grammar into a Chomsky normal form. Why do we do this? It's easier to prove things about Chomsky normal form, but also, there is an algorithm that can check if a string is generated by a grammar. It might be inefficient, but it's an algorithm that works. There's actually a faster one that Sipser mentions in other chapters, but I don't think we're going to talk about that.
(Note that Sipser's algorithm removes the ε rules second and this increases the number of rules by a lot. This order is much better.)

Now, we'll discuss closure properties of context-free languages.

Theorem 8. *Context-free languages are closed under union.*

Proof. Let $g_1 = (V_1, \Sigma, R_1, S_1)$ and $g_2 = (V_2, \Sigma, R_1, S_1)$ be context-free grammars that have disjoint V. Then, we can construct a grammar $g = (V, \Sigma, R, S)$ where S is a new start variable and $V = V_1 \cup V_2$, $R = R_1 \cup R_2 \cup S \mapsto S_1 \cup S \mapsto S_2$.
If we have a string w generated by g_1 then this can also be generated by g and if g_2 derives w then g also derives w.

□

Theorem 9. *Context-free languages are closed under concatenation.*

Proof. As with above, let $g_1 = (V_1, \Sigma, R_1, S_1)$ and $g_2 = (V_2, \Sigma, R_1, S_1)$ and construct $g = (V, \Sigma, R, S)$ with $V = V_1 \cup V_2$, $R = R_1 \cup R_2 \cup S_1 \mapsto S_2$, and S as a new start variable.
If we have S_1 derives w_1 and S_2 derives w_2, then S yields $s_1 s_2$ which derives $w_1 S_2$ and $S_1 w_1$, which is in the language of g. Conversely, if S derives some string w, then it must be the case that X yields x_1, x_2 which derives w so x_1 must derive some string w_1 and x_2 must derive some string w_2 with w_1 in the language of g_1 and w_2 in the language of g_2.

□

Theorem 10. *Context-free languages are closed under the Kleene star.*

Proof. Let $g_1 = (V_1, \Sigma, R_1, S_1)$ and build $g = (V, \Sigma, R, S)$, with $V = V_1 \cup S$, $S \mapsto S_1 S | \varepsilon$
Let's see if we can prove that this works. We want to show that if $w = w_1 \cdots w_k$ with $w_i \in L(g_1)$ then $w \in L(g)$, or that S_1 derives w_i for all i. We have that S yields $\underbrace{S_1 \cdots S_1}_{k \text{ times}} S$. Then we know that all of the w_i are derived by S_1^k
If $w \in L(g)$ then either $w = \varepsilon \in L(g_1^*)$ or S derives $\underbrace{S_1 \cdots S_1}_{k \text{ times}} S$ which derives w.

□

What about intersection? As it turns out, context-free languages are *not* closed under intersection. We can't quite prove this yet, since we don't know how to prove a language is not context-free. We can consider the languages $L_1 = \{a^i b^j c^k \mid i = j\}$ and $L_2 = \{a^i b^j c^k \mid j = k\}$. If we look at $L_1 \cap L_2$, this is not context-free.
Related to this, context-free languages are *not* closed under complementation. If it was closed under complementation, this would imply closure under intersection. But it's not, so it's not closed under complementation.
The last thing I want to do today is a useful closure property.

Theorem 11. *The intersection of a context-free language and a regular language is context-free.*

Corollary: If A is context-free and B is regular, then $A - B$ is context-free because $A - B = A \cap B^C$. There is a proof with the finer details on my website.

Lecture 10 (02/26/2015)

Today we're going to talk about a new type of pumpability. This one is called CF-pumpability and it is a weaker condition than the pumpability that we talked about before.

We say that a language L is pumpable with pumping length p if for all words w of length at least p w can be partitioned into $w = uvxyz$, $vy > 0$, $|vxy| \leq p$, and for all i, the string uv^ixy^iz is in the language L.

Let's look at a language that is CF-pumpable. Let $L = \{s \mid s\#s^R, s \in \{a, b\}^*\}$ with pumping length 3. This means all strings of length 3 or more can be pumped. Consider the string $sa\#as^R$, with $s = \{a, b\}^*$. Clearly this in the language L. Let $u = s, v = a, x = \#, y = a, z = s^R$. We can check the conditions on the partition and notice that this is valid. Then, $uv^ixy^iz = sa^i\#(sa^i)^R \in L$.

We can construct a parse tree as follows:
If we consider pumping down, then we're replacing the larger subtree (in the parse tree) with the smaller one. This looks like:
If we instead want to pump up, then we replace the small subtree with the larger subtree.
Why do we care about CF-pumpability? We can use it to show that a particular language is not context-free, in the same way that we used the pumping lemma for regular languages to show languages were non-regular. This leads us to a theorem:

Theorem 12. *Any context-free language is CF-pumpable.*

Let's try an example with a language I claimed was not context-free. Show $L = \{a^nb^nc^n\}$ is not context-free.

Proof. Suppose L is context-free with pumping length p. Choose $w = a^pb^pc^p$. Since $|vxy| \leq p$, we have three cases: vxy contains no **a**s, vxy contains no **b**s, or vxy contains no **c**s. Then, pump up with $i = 2$. Then, uv^2xy^2z splits into three cases:

1. uv^2xy^2z has more **b**s than **a**s or more **c**s than **a**s.
2. uv^2xy^2z has more **a**s than **b**s or more **c**s than **b**s.
3. uv^2xy^2z has more **b**s than **c**s or more **a**s than **c**s.

Therefore $uv^2xy^2z \notin L$ so L is not CF-pumpable and therefore is not context-free.

□

Let the language $L_1 = \{w \mid w = w^R \wedge w \text{ has the same number of } \mathbf{a}\text{s and } \mathbf{b}\text{s}\}$.

Proof. Suppose L_1 is context-free with pumping length p. Consider the string $w = a^p b^p b^p a^p$. Now, we have some possibilities. In all of these cases, if we pump down, we won't end up with a palindrome. □

Lecture 11 (03/10/2015)

Let's talk about Turing machines, invented by Alan Turing, which are the most powerful model of computation that exists. They aren't the most efficient, but we don't know of any program that we can do on a computer that we also can't do on a Turing machine.
A Turing machine consists of a finite set of states, a one-way infinite tape, a pair of "distinguished" states (q_a and q_r – whenever it enters q_a, it stops and accepts the string, and whenever it enters q_r, it stops and rejects the string), and a tape head.
In a single step of computation, the symbol under the tape head is read, and the current state it is in is evaulated, and it writes a symbol and moves to a new state and moves the head either left or right.
When we compute a string on a Turing machine, there are three things that can happen:

1. We accept the string.
2. We reject the string.
3. The machine runs forever, also known as "looping".

Let's start with the example context-free language $\{a^n b^n c^n \mid n \geq 0\}$. We want to build a Turing machine that recognizes this language.
Formally, a Turing machine is a 7-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ where Q, Σ, Γ are finite sets and

- Q is the set of states
- Σ is the input alphabet not containing the blank symbol $\sqcup$
- Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$
- $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$ is the transition function
- $q_0 \in Q$ is the start state
- $q_a \in Q$ is the accept state
- $q_r \in Q$ is the reject state, and $q_r \neq q_a$

Lecture 12 (03/12/2015)

We started talking about Turing Machines (now abbreviated by **TM**), which are formally, 7-tuples. We say that a Turing Machine M recognizes L if L is the language of M, and there is a special type of TM called a **decider** if it never loops. These are particularly nice and useful, and we will try to build deciders whenever we can (turns out this isn't always possible). If M is a decider and M recognizes L, then M **decides** L.
If we think about DFAs and NFAs, they were deciders, but here we have a slightly different situation because it's possible for a TM to loop. Last time, we gave an example of a TM, and we should try another example this time. Let $L = \{w \mid w\text{hasthesamenumberof0sand1s}\}$ with the alphabet $\Sigma = \{0, 1\}$.
Let's first think about what we want our Turing machine to do. How do we do this?

1. Start at the beginning and scan right until we find a **0**
2. Mark it, and move back to the beginning,
3. Scan right until we find a **1**
4. Mark it, and move back to step 1.

First, let's override the first symbol with a blank so that we can mark the start of the string with a blank. In order for this to work, we need to remember the symbol we overwrote. Then, we scan right until we find the opposite symbol, mark it, and scan back to the left until we reach the blank. We should worry about how we're starting and stopping, so we also need to scan past all of the marked symbols.

I mentioned last time that single-tape TMs are equivalent to multitape TMs. Formally, a multitape TM is the same 7-tuple $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ except δ is now a function $\delta : Q \times \Gamma^k \mapsto Q \times \Gamma^k \times \{L, R, S\}^k$ where there are k tapes. So, if we have $\delta(q_i, a_1, \cdots, a_k) = (q_j, b_1, \cdots, b_k, L, R, \cdots, L)$, then it moves from state q_i to q_j, and

Theorem 13. *A k-tape Turing Machine is equivalent to some 1-tape Turing Machine.*

Proof. See Sipser p.177 for diagram and proof. □