We are interested in how a regular language can be modified and have the modified language still be regular.

These are called the *closure properties* of the class REG of regular languages.

For example, we claim that if $L_1, L_2$ are regular languages, then so are $L_1 \cup L_2$, $L_1L_2$, and $L_1^*$. 

To see this, use the characterization of regular languages as those specified by regular expressions. Then, as we’ve already seen, given regular expressions for $L_1$ and $L_2$, we can easily make regular expressions for their union, concatenation, and closure.
How to prove closure properties

In general, this is how to prove a closure property:

1. Choose some representation for the regular language(s) in question: as the language recognized by a DFA, NFA, or $\epsilon$-NFA, or as the language specified by a regular expression.

2. Carry out some modification to the representation to get a representation for the modified language $L$.

3. You *don’t* have to produce the same kind of representation as the one you started with. You can start with a DFA, for example, and end up with an $\epsilon$-NFA.

4. Not every choice of representation will work easily. Some are *much* better than others.
An example: closure under complement

Let us prove that if $L$ is regular, then $\overline{L} = \Sigma^* - L$ is regular.

What representation should we choose for $L$?

A regular expression is definitely *not* the right choice, because there is no obvious way to get a regular expression for $\overline{L}$ from one for $L$.

(In fact, there are examples where the smallest regular expression for $\overline{L}$ is *doubly-exponential* in the size of the regular expression for $L$! See [https://arxiv.org/abs/0802.2869](https://arxiv.org/abs/0802.2869).)

That’s a really bad blowup!
So we have to choose the right representation.

NFA is not the right choice either! Because there is no simple way to take an NFA for $L$ and convert it into an NFA for $\overline{L}$.

(There are examples known where the smallest NFA for $\overline{L}$ has size exponential in the size of the NFA for $L$.)

So what is the right choice?
The right choice here is the DFA.

Given a regular language $L$, we know there is a DFA $M = (Q, \Sigma, \delta, q_0, F)$ such that $L = L(M)$.

How can we change $M$ to get a DFA for $\overline{L}$?
Closure under complement

The answer is: change every final state to non-final, and every non-final state to final.

Formally speaking, let $M' = (Q, \Sigma, \delta, q_0, F')$, where $F' := Q - F$.

We claim that $x \in L(M')$ iff $x \in \overline{L(M)}$.

Let’s prove this.

We have: $x \in L(M')$ iff $\delta(q_0, x) \in F'$, iff $\delta(q_0, x) \notin F$, iff $x \notin L(M)$, iff $x \in L(M)$.

So we’ve proved that the class REG is closed under complement. In this case, choosing the right representation for $L$ is 95% of the game!

Why didn’t this work for NFA’s?
Let’s prove that the class REG is closed under intersection.

There are at least two methods to do this. One is using what we already know about regular languages, and the other is an automaton construction.

The first method is easiest. We want to show that if $L_1$ and $L_2$ are regular languages, then $L_1 \cap L_2$ is a regular language.

Can we write $L_1 \cap L_2$ in terms of operations we know about?
Closure under intersection

We sure can! Remember De Morgan’s laws?

We know that

\[ L_1 \cap L_2 = \overline{L_1 \cup L_2}. \]

The regular languages are closed under complement, as we already saw, so \( \overline{L_1} \) is regular. And \( \overline{L_2} \) is regular.

And \( \text{REG} \) is closed under union, as we already saw, so \( \overline{L_1} \cup \overline{L_2} \) is regular.

And finally, \( \text{REG} \) is closed under complement, so \( \overline{L_1 \cup L_2} \) is regular.

But this is \( L_1 \cap L_2 \). So the class of regular languages is closed under intersection.
Algorithm for finding DFA for intersection

This proof even gives a (terribly inefficient) algorithm for computing a DFA for \( L_1 \cap L_2 \), given DFA’s \( M_1 \) for \( L_1 \) and \( M_2 \) for \( L_2 \):

Start with the DFA for \( L_1 \). Use our construction for complement to get a DFA \( M_1' \) for \( \overline{L_1} \).

Do the same thing for \( L_2 \), getting a DFA \( M_2' \).

Now find a regular expression \( r_1 \) for \( L(M_1') \), using the state elimination algorithm we saw before.

Do the same thing for \( L(M_2') \), getting \( r_2 \).
Algorithm for finding DFA for intersection

Now form the regular expression $r_1 \cup r_2$.

Now convert this regular expression to an $\epsilon$-NFA $M_3$ using the construction we saw before.

Convert $M_3$ to an NFA $M_4$ using the algorithm we saw before.

Convert $M_4$ to a DFA $M_5$ using the algorithm we saw before.

Modify $M_5$ to get a DFA for $L(M_5)$ as we discussed earlier in this lecture.
A much better algorithm!

That method we just saw – it’s not very inefficient! It could be *doubly exponential* in the size of the original DFA’s for $L_1$ and $L_2$.

There’s a much better idea, using something called the *direct product construction*.

This construction takes DFA’s $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ and produces a DFA $M$ for $L(M_1) \cap L(M_2)$.

The idea is to have $M$ *simulate* both $M_1$ and $M_2$ *in parallel*, and accept an input $x$ iff both $M_1$ and $M_2$ accept $x$. 
The direct product construction

How can we make $M$ simulate $M_1$ and $M_2$ in parallel?

Recall that $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.

We’re going to define $M = (Q, \Sigma, \delta, q_0, F)$ for $L(M_1) \cap L(M_2)$.

What should the states of $M$ be?
The direct product construction

Since we have to simulate both $M_1$ and $M_2$, we have to take into account the possibility that our simulating DFA $M$ could be in \textit{any} element of $Q_1$ and \textit{any} element of $Q_2$.

This means we should define $Q := Q_1 \times Q_2$.

Here $\times$ is the so-called cross product. For sets $S_1, S_2$ by $S_1 \times S_2$ we mean the set of pairs $\{[p, q] : p \in S_1, \ q \in S_2\}$.

How do we define $\delta$?
We’ll define \( \delta \) so it *simulates* both \( \delta_1 \) and \( \delta_2 \).

For \( p \in Q_1 \) and \( q \in Q_2 \) we define \( \delta([p, q], a) := [\delta_1(p, a), \delta_2(q, a)] \).

What should the right initial state for \( M \) be?

It should be \( q_0 := [q_1, q_2] \).
Finally, we need to define $F$, the set of final states.

We want to accept an input $x$ if both $M_1$ and $M_2$ accept.

So the right choice is $F := F_1 \times F_2$.

This construction gives us a DFA $M = (Q, \Sigma, \delta, q_0, F)$ recognizing $L(M_1) \cap L(M_2)$, given DFA's $M_1$ and $M_2$.

The resulting DFA has $mn$ states, if $M_1$ has $m$ states and $M_2$ has $n$ states. That’s pretty efficient!
How can we *prove* that the construction works?

Think for a moment about how you would do it.
The direct product construction

The idea is to prove, by induction on $|x|$, that

$$\delta([p, q], x) = [\delta_1(p, x), \delta_2(q, x)] \quad (\ast).$$

I’m not going to carry out this proof here, but it’s a good exercise for you to do.

Once we have (\ast), we can reason as follows:

- $x \in L(M_1) \cap L(M_2)$ iff
- $\delta_1(q_1, x) \in F_1$ and $\delta_2(q_2, x) \in F_2$ iff
- $\delta([q_1, q_2], x) \in F_1 \times F_2$ iff
- $\delta([q_1, q_2], x) \in F$ iff
- $x \in L(M)$. 
The direct product construction

The same kind of construction lets you create DFA’s for $L(M_1) \cup L(M_2)$ and $L(M_1) - L(M_2)$, given DFA’s for $M_1$ and $M_2$.

There’s one more operation: symmetric difference. The symmetric difference of two languages, $L_1 \Delta L_2$, is defined to be

$$(L_1 - L_2) \cup (L_2 - L_1).$$

Can you find a direct product construction for the symmetric difference of two regular languages? What should the set of final states be?
Closure under reversal

We’re now going to look at one more closure property of the regular languages: closure under reversal. Recall that $L^R = \{ x : x^R \in L \}$.

There are two different methods to prove closure of REG under reversal: one where we start with a regular expression for $L$, and one where we start with a DFA for $L$.

Let’s do the regular expression method first.

Given a regular expression $r$ for $L$, we need to create a regular expression $r'$ for $L^R$.

How can we do this?
Let’s try it on a particular example. What’s \((1(0 \cup 01)^*)^R\)?

Convince yourself that it’s \((0 \cup 10)^*1\).

With this in mind, let’s try to create a regular expression for \(L(r)^R\), given \(r\).

The idea is, as before, induction on the number of operators in \(r\).
Closure under reversal

The base case is that $r$ has 0 operators.
If $r = \emptyset$, then its reversal is also $\emptyset$.
If $r = \epsilon$, then its reversal is also $\epsilon$.
If $r = a$, then its reversal is also $a$.
Now let's do the induction step. Either
\[ r = r_1 \cup r_2, \]
or $r = (r_1)(r_2)$,
or $r = (r_1)^*$. 

Closure under reversal

We then use the following identities:

1. \((L_1 \cup L_2)^R = L_1^R \cup L_2^R\)
2. \((L_1L_2)^R = (L_2^R)(L_1^R)\)
3. \((L_1^*)^R = (L_1^R)^*\).

Let’s prove each of these:

1. \(x \in (L_1 \cup L_2)^R\) iff \(x^R \in L_1 \cup L_2\), iff \(x^R \in L_1\) or \(x^R \in L_2\), iff \(x \in L_1^R\) or \(x \in L_2^R\).
2. \( x \in (L_1 L_2)^R \) iff \( x^R \in L_1 L_2 \), iff
\[ \exists y \in L_1, z \in L_2 \text{ such that } x^R = yz, \text{ iff} \]
\[ \exists y \in L_1, z \in L_2 \text{ such that } (x^R)^R = (yz)^R, \text{ iff} \]
\[ \exists y \in L_1, z \in L_2 \text{ such that } x = z^R y^R, \text{ iff} \]
\[ x \in L_2^R L_1^R. \]
For the last one, we first need to prove that \((L_1^i)^R = (L_1^R)^i\) for all \(i \geq 0\). Do this by induction on \(i\) (exercise).

Assuming this, we get

\[
3. \quad (L_1^*)^R = \left( \bigcup_{i \geq 0} L_1^i \right)^R = \bigcup_{i \geq 0} (L_1^i)^R = \bigcup_{i \geq 0} (L_1^R)^i = (L_1^R)^*.
\]

Now we can complete the induction proof by applying each of these rules.
Another construction starts with a DFA $M$ for $L$ and produces an automaton for $L^R$.

The idea is pretty simple: if $x \in L(M)$, then this means that there is a path from $q_0$ to some final state $q_f$ labeled with $x$.

If we now “turn around” all the transitions, so the arrows go in the reverse direction, then there is a path from $q_f$ to $q_0$ labeled with $x^R$ in this new machine.

So if we let the initial state be the final state of $M$, and the final state be the initial state of $M$, and turn around all the transitions, we get an automaton for $L^R$. 
Another reversal construction

There are two problems, however.

First is that even just “reversing the arrows” might change a DFA into an NFA. Example: start with the DFA

![Diagram](image1)

and reverse the arrows to get

![Diagram](image2)
Another reversal construction

That’s not a real problem, since there is no requirement to produce a DFA; an NFA will do just fine.

The second problem is more serious: there might be multiple final states, which then turn into multiple initial states, which is not allowed by our NFA model.

To fix this, introduce a new initial state, and put $\epsilon$ transitions from this initial state to all the final states of the old automaton.
Another reversal construction

Let’s carry out this construction on an example: We start with the following DFA:

Next, make the initial state $q_0$ final, reverse the direction of the arrows, and remove the “finality” of the two final states $q_2$ and $q_4$. 
Another reversal construction

Finally, add a new initial state $q_5$, and put $\epsilon$-transitions from $q_5$ to what were previously the final states ($q_2$ and $q_4$).
Another reversal construction

That’s the construction. To really be convincing, we should prove it formally.

Here’s the strategy. Start with a DFA $M = (Q, \Sigma, \delta, q_0, F)$.

Consider the reversed NFA $M' = (Q, \Sigma, \delta', -, \{q_0\})$ we get before we add the $\epsilon$-transitions. (Here we regard the initial state as unspecified.)

We define $\delta'(p, a) := \{q : \delta(q, a) = p\}$.

Then we need to show that $q \in \delta'(p, x)$ iff $\delta(q, x^R) = p$. We can do this by induction on $|x|$.

Exercise: try to complete the proof using this outline.
To sum up everything we saw today, the class $\text{REG}$ of regular languages is closed under the following operations:

- union
- concatenation
- Kleene closure
- intersection
- complement
- symmetric difference
- relative complement $L_1 - L_2$
- reversal