**MICHAEL SIPSER:** Hi, folks. Why don't we get started? Welcome back. Good to see you all here. So I am going to first-- well, we'll recap what we did last time and what we're going to do today. I'll talk a little bit about the problem set. And we'll also have a break, as requested, halfway through.

So why don't we jump in? What we did last time was besides introducing the course, we introduced finite automata in regular languages, which are the languages that the finite automata can recognize. We talked about these regular operations. Those allow us to build what we call our regular expressions. These are ways of describing languages.

So we have finite automata can describe languages, and regular expressions can describe languages. And one of our goals is to show that those two systems are equivalent to one another, even though they look rather different at first glance. So to move in that direction, we're going to prove closure properties for the class of regular languages over these regular operations. So we'll show that-- well, we already showed that any two regular languages have their union, also being regular. And we'll show that for the other two operations as well.

So let's just look ahead to what we're going to do today. We're going to introduce an important concept which is going to be a theme throughout the course, called nondeterminism. And having that as a tool that we can use, we'll be able to show closure under concatenation and star, finishing up what we started to do last time. And then we'll use those closure constructions to show how to convert regular expressions to finite automata. And that's going to be halfway to our goal of showing that the two systems are equivalent to one another. And the following lecture, we will show how to do the conversion in the other direction.

So I thought we would just jump in, then, and look at-- return to the material of the course. As you remember, we were looking at the closure properties for the class of regular languages. We started doing that. And if you recall, hopefully, we did closure under union. And then we tried to do closure under concatenation, which I have shown here on this slide, the proof attempt that we tried to do last time. And let's just review that quickly, because I think that's going to be helpful to see how to fix the problem that came up.

So if you remember, we're given two regular languages, A1 and A2. And we're trying to show that the concatenation language A1A2 is also regular. And so the way we go about all of these things is we assume that A1 and A2 are regular. So that means we have machines, finite automata, for A1 and A2. We'll call them M1 and M2, that recognize A1 and A2, respectively.

And then what we need to do in order to show the concatenation is regular is to make a finite automaton which recognizes the concatenation. And we tried to do that last time. So if you remember, that concatenation machine-- M, we're calling it-- what is it supposed to do? It's supposed to accept its input if it's in the concatenation language. And that means that the input can be split into two parts, x and y, where x is in the A language, and y is in the B lang-- y is accepted by M1-- and x is accepted by M1, and y is accepted by M2. Sorry I garbled that up.

So x should be in A1, and y should be in A2. if you can split w that way, then M should accept it. So M has to figure out if there's some way to split the input so that the first machine accepts the first part, the second machine accepts the second part.

And the idea that we came up with for doing that was to take these two machines, build them in to a new machine M, and then connect the accepting states for M to the start state-- connect the accepting states for M1 to the start state for M2. Because the idea would be that if M1 has accepted an initial part, well, then you want to pass control to M2 to accept the rest.

But as we observed, that doesn't quite work. Because the first place to split w after you found an initial part that's accepted by M1 may not be the right place. Because the remainder may not be accepted by M2. You might have been better off waiting until you found another place that M1 accepted, later on in the string, say, over here. And then by splitting it over there, then maybe you do get successfully find that the remainder is accepted by M2. Whereas if you tried to split it in the first place, the remainder wouldn't have been accepted by M2.

So all you need to do-- M has to know, is there some place to split the input so that you can get both parts accepted by the respective machines? The problem is that M might need to know the future in order to know where to make the split. And it doesn't have access to the future. So what do we do?

So what we're going to do is introduce a new concept that will allow us to basically get the effect of M1-- and the-- sort of being able to see the future. And that new concept is going to be very important for us throughout the term. It's called nondeterminism. And so we're going to introduce a new kind of finite automaton called a nondeterministic finite automaton. And first, we'll look at that, and then we'll see how that fits in with the previous deterministic finite automaton, that we introduced last time.

So here's an example. It's always good to start off with an example. Here is a picture of a nondeterministic finite automaton. It looks very similar, at first glance, to the former kind, the deterministic finite automaton. But if you look a little carefully, you see that there are some key differences.

The most important difference is that in state q1, for example, whereas in the machines that we introduced last time, there had to be exactly one way to go on each possible input symbol so you knew how to follow along through the machine it's computing, here there are two ways to go. In q1, you can either stay in a1, or you can go to q2. That's the essence of nondeterminism. There could be many ways to proceed.

And furthermore, on q1, if you get a b, then there's nowhere to go. So that's also possible within nondeterminism. So let's just start looking at these features.

There are multiple paths forward-- multiple paths possible. You might be able to have one, as we had before, or many ways to go at each step, or maybe 0 ways to go at each step. Those are all legitimate for a nondeterministic machine, which is doing a nondeterministic computation.

Another difference, if you look carefully, is that we're allowing here the empty string to appear on a transition. That's perhaps a little less essential to the spirit of nondeterminism. But it's going to turn out to be a convenience when we actually apply nondeterminism to build machines, as you'll see very shortly.

Now, if there are many different ways to go-- and some of those ways to go might have different outcomes. As we remember from before, we accepted the input, if you end up in an accept state, and we rejected the input, if you end up not in an accept state, in a non-accepting state. Then you reject.

But now there might be several different ways to go. And we'll do an example in a minute. But there might be several different ways to go. And they might disagree. Some of them might accept. Other ones might reject.

So then what do you do? Well, in that case, acceptance always overrules rejection. That's the essence of nondeterminism the way we're setting it up. You may ask why that is. And the spirit of that will become clear in due course. But right now, just take it as a rule.

When we're having a nondeterministic machine, acceptance overrules rejection. So as long as there is-- one of the possible ways to go ends up at an accept, we say the whole thing is accepted. The only way we can possibly reject-- if all of the possible ways to go end up at rejection, end up at a non-accept state. So we'll see example of-- I think we're going to do an example right now, yeah.

So if we take, for example, this machine N1 now on an input ab-- and we're going to process the symbols one by one, just like we did before. But now, to follow along, there might be several different ways to go. So if we take the first symbol a, and we run the machine-- so when the machine, it starts at the star state, as before-- but now an a comes in. And there might be two-- now there are two different ways to go.

So we're going to keep track of both of them. After the machine reads an a, you can think of it as being in two states now simultaneously. It can be in state q1, and it can be in state q2. So those are two different possible places it could be at this moment. OK?

Now we read the next symbol, the b. And from a b, you take each of the places where the machine could be at the end of the previous symbol, and you then apply reading a b, the next symbol, from each of those states where the machine could be in from the previous symbol. So the machine could be in q1 and q2 after reading an a. Now we apply a b.

Well, q1 on a b goes nowhere. So you think of that branch, if you will, of the computation as just dying off. It has nowhere to go. It just vanishes. However, the other possibility, which was state q2 on a b, does allow, does have a place. So the machine is now going to go from q2 to q3 on that branch of the computation, which reading-- on reading a b.

And then it has, coming out of q3, there are two symbols. There's an a and an empty string symbol. Now, on an a, the machine would have to read an a in order to transition along that arrow. But when there's an empty symbol on the arrow, that means the machine can go along that arrow for free without even reading anything. As long as it gets to q3, it can automatically jump the q4.

And so once it has read a b and gone to q3, now it can either stay in q3, or it can go along the empty transition and go to q4. So again, it is going to be a nondeterministic step at this point. The essence of having a empty transition is that there is going to be nondeterminism. That's why we didn't introduce that for deterministic automata, because you don't have to transition along an empty string transition. You can stay where you are, or you can go along the empty string transition without reading any input and go over to the next state, which, in this case, is q4.

So let's just see where we are. After reading an a, we're in states q1, q2. But now after reading a b, we're in states q3 and q4 as possibilities. And now we're at the end of the input, and we look and see what we got.

If any one of the states as possibilities that we are right now at the end of the string is an accept state, then we say, overall, the machine has accepted. So that corresponds to what we said over here before-- accept the input if some path leads to an accept. So if any way of proceeding through these nondeterministic choices will lead you to an accept, then you will say, we're going to accept the input. OK? So this input here is accepted.

Let's do another example. Suppose we have the input aa instead of ab. So aa, after the first day, as before, we're in states q1 and q2 as possibilities. Now we read an a again. Now, the one that's on state q1, that possibility, q1 possibility, after reading an a, it again branches to q1 and q2. So we know after reading the second a, we're going to be in at least q1 and q2.

Now how about the state that had been on q2 on reading an a, the one from before? After reading the first a, you were in q2. Now reading the second a, there's nowhere to go. So that one just gets removed.

So after reading aa, we remain in states q1 and q2 as possibilities. Neither of those are accept states. So therefore, on input aa, the machine rejects. OK? Let's just do a couple more, and then I'll ask you to do one.

So we have aba as an input. Let's see what happens then. So remember, after reading ab, the machine is in the two states q3, q4 as possibilities. That's what we have from the first example. So after reading ab, we're in states q3, q4.

Now we read another a. The q4 on an a has nowhere to go. In fact, it has nowhere to go in any input. So no matter what comes in after you're in state q4, that branch dies.

But on q3, which is another one of the possibilities reading an a, it can follow along, just transition. Because that's one of the labels on that transition, is a. So you can follow along just in transition on reading an a, which is the last symbol in the string. And so now after aba, you are in only state q4 as a possibility. But that happens to be an accept state, so the machine accepts. OK.

And now lastly, let's take our final example. What happens if we have abb? So as we remember before, after reading ab-- that was the first example-- we were in states q3, q4 as possibilities. Now we read a b. Well, neither of those states have anywhere to go on a b.

So now all threads, all branches, of the computation die off. And at this point, the machine is totally dead. It has no active possibilities left. So certainly, it's going to reject this input, because none of the active states-- there are no active states or accepting states.

And in fact, if you looked at anything that came later, anything that extended the string abb would also be rejected. Because once the machine had all-- all possibilities have died off, there's no way for them to come back to life on any extensions. So with that-- oh, here's an important point before I'm going to jump to a check-in on you. But I think one thing that might be on your mind about this nondeterminism is how does that correspond to reality? Well, it doesn't.

We're not intending for nondeterminism as we're defining it to correspond to a physical device. But nevertheless, as you'll see, it's a very mathematically useful concept, this nondeterminism. And it's going to be playing a big role throughout the subject as we'll experience it during the rest of the term.

So with that, I'm going to have a little check-in. I'm going to ask you to consider what happens on one of the inputs. So here we go. What does it do on input aab? So here's the machine. You can look at it. And suppose, hopefully, there's a poll here for me to give to you so you can give me your input. So what does the machine do on input aab? Most of you have answered.

Again, you're not going to be penalized for getting the wrong answer. But hopefully, you'll get the right answer. Anyway, let's just take a look here. So time is up. Let's end the polling and share the results.

So the majority of you, majority have gotten the correct answer, which is a. The machine does accept aab. Because when you have a-- so I'll show you the path that corresponds to accepting. You go a, a, b, and then empty string. And so that sequence of steps is one of the nondeterministic possibilities that the machine can follow. And that shows that the machine does accept the input aab.

You can think about it, the way we did it before also. If you read an a, it's in the two possibilities q1, q2. You read a second a, again, in the possibilities q1, q2. Now you read a b. It's in the possibilities q3, q4. And that's it, aab. So now you read the b. It's in possibility q3, q4. q4 is an accepting state. That overrules the non-accepting state. And so the machine accepts.

You have to understand this. So if you didn't get it right, go back and think about where you slipped up. OK? Because this is just getting-- we're just getting warmed up here. It's going to get a lot harder. OK, so stop sharing the results. And so let's continue.

So just as we did last time, we can formally define a nondeterministic finite automata. Here's the picture again. OK. So it looks a lot like the case we had before, the Deterministic Finite Automata, or DFA, as we'll call them. It's a 5 tuple. So I've written down little reminders for what those components of that 5 tuple are, that list of five components.

So they're all the same as before-- states, alphabet, transition function, start state, and accepting states. So that the formal definition looks exactly the same except the structure of the transition function. So now, before, if you remember, you had a state and an input symbol, and you got back a state.

Now we have something more complicated-looking. We have a state and an input symbol, but instead of just sigma, it's sigma sub epsilon. And that that's a shorthand for sigma union epsilon. And that's a way-- my way of saying that you're allowed to have on your transition arrows either an input symbol or an empty string. So the transition function has to tell you what to do when you have an empty string coming in as well. So that would be part of your table for the transition function.

Now, over here, what's going on over here? Well, now, instead of just producing a single state, when you've read, for example, an a from q1, there's a whole set of possibilities. So here we have what's called a power set. That's the set of subsets of the collection Q. So here we're going to produce an entire subset of states. Instead of just one state coming out, there might be a subset of possible states that you can go to.

So the power set of Q is a set of subsets of Q. So that's what this notation means. Again, this is something that I'm, hopefully, presenting to you as a bit of a reminder. You've seen this somewhere else before. But please make sure you understand the notation, going forward, because we'll be doing less hand-holding as we start moving forward. OK.

So just let's take a look. In the N1 example here, just to illustrate what's going on, when you're in state q1 reading an a, now you get a whole set of possibilities, which, in this case, is q1 and q2. Whereas, if you're reading a b, what would be that set? Coming out of q1, what's the set of possible successor states? Well, there are none. So it's the empty set. OK? So hopefully, you're understanding the notation here.

So now here's, I think, really important. How do we understand nondeterminism, intuitively speaking? And there are multiple different ways, which each has their value under different circumstances.

So one way is thinking about nondeterminism as a kind of parallelism. So every time the machine has a nondeterministic choice to make, where there's more than one outcome, you think of the machine as a branching, forking, new threads of the parallel computation at that stage, where it makes an entire copy of itself when there's a choice of possibilities. And then each of those independently proceeds to read the rest of the input as separate threads of the computation.

So if you're familiar with parallel computing, this should be reasonably familiar to you. The only key thing to remember is that as this thing forks a number of possibilities, the acceptance rule is, that if any one of those possibilities gets to an accept at the end of the input, it raises a flag and says, accept. And that overrules everybody else. So acceptance dominates.

So another way of looking at it is the mathematical view, where you can imagine-- and we're going to use all these. So you really need to understand them all. The mathematical view is you can think of the computation as kind of a tree of possibilities. So you start off at the very beginning at the root of the computation, which is when it really begins. But every time there's a nondeterministic branching that occurs, that node of the tree has multiple children coming out of that node. And so the different threads of the computation correspond to different branches of that tree.

And now you're going to accept if any one of those branches leads to an accepting state-- OK, obviously, somewhat similar to what we had before. But I think it's a little bit of a different perspective on how to think about nondeterminism. And the last one is going to sound a little weird.

But actually, I think for people who are in the business, it's the one they use the most. And that's the magical way of thinking about nondeterminism. And that is, when the machine has nondeterministic choices to make, you think of the machine as magically guessing the correct one at every stage, and the correct one being the one that will eventually lead it to accept. OK?

So you can think of the machine as guessing which is the right way to go. And if there is some way right way to go, it always guesses right. Of course, if the machine ends up rejecting, because there is no right way to go, then it doesn't matter. There is no good guess. But if there is some good guess, we'll think of the machine as taking that good guess and going that way. OK.

So now here is a very important thing. We introduced this new model, the Nondeterministic Finite Automaton, NFA. It turns out, even though it looks more powerful, because it has this nondeterminism, it isn't any more powerful. It can do exactly the same class of languages, the regular languages. And we'll show that with this theorem here, that if an NFA recognizes a, then a is regular.

So we'll prove that by showing how to convert an NFA to an equivalent DFA, which does the same language. So we can take an NFA that has the nondeterminism and find another DFA which doesn't have nondeterminism, but does the same language. It accepts exactly the same strength, even though it lacks that nondeterministic capability.

This is going to be extremely useful, by the way, and for example, in showing that closure under concatenation. OK, so in this presentation here, I'm going to ignore the epsilon transitions. Because once you get the idea for how to do this, you could figure out how to incorporate them. They just make things a little more complicated. So let's just focus on the key aspect of nondeterminism, which is that the machine could have several ways to go at any point in time. There could be several next states on an input. OK?

Now the idea for the construction-- so we're going to start with a nondeterministic machine M, and we're going to build a deterministic machine M prime, which does exactly the same thing. And the way M prime works is it's going to do what you would do if you were simulating M. What would you do? This is what we were doing as I was explaining it to you.

If you were simulating M, every time you get an input symbol, you just keep track of what is the set of possible states at that point in time. That's what the DFA is going to do. it's going to have to keep track of which possible set of states the NFA could be in at the point on that input where we are right now.

And then as you get to the next symbol, the DFA is going to have to update things to keep track of the next set of states the NFA could be in at this point, just like you would do. OK? And so here's a kind of a picture. And how do we implement that?

So here's the NFA that we're starting with, M, and we're going to make here the DFA. But in order to remember which set of states that DFA could be in at a given point-- so maybe it's in the set of states that M could be in. Did I say it wrong? Which set of states the NFA could be in a given time-- so maybe M, the NFA, could be in, at some point, state q3 and q7. The way the DFA keeps track of that, it's going to have a state for every possible subset of states of the NFA. That's how it remembers which subset of states the NFA is in.

That's the way DFAs work. They have a separate state for each possibility that they need to keep track of. And the possibilities here are the different subsets of states that the NFA could be in at a given point. OK? So corresponding to this subset, to these two possibilities q3, q7, the DFA is going to have a state with the subset q3, q7. And it's going to, for every possible subset here, there's going to be a different state of M prime. So M prime is going to be bigger.

OK. So quickly, the construction of M, the states of M prime now, q prime, are going to be the power set, the set of subsets of states from the original machine M. And now we have to look at how the transition function of the DFA, when you made the primed machines of the DFAs, the DFA machine. So these are the deterministic components.

So delta prime, when it has a subset, something like this, has one of its states, which corresponds to a subset of states of M, and it reads an input symbol, you just have to do the updating the way you would naturally do. You're going to look at every state in R, look at where that can go under a-- so there's a bunch of sets there. And look at all the possible states that could be in one of those subsets, and that's the set of states that you could be.

That's going to be the new set of states, and that's going to be in the new state of M prime. OK? So it's going to be the subset corresponding to all of the states that could be in, when you apply the transition function of the nondeterministic machine, to one of the states in the subset of states that the nondeterministic machine could be in. OK?

It's a little bit of a mouthful. I suggest you look at this, if you didn't quite get it, after the fact. Good to understand.

The starting stage for the NFA-- for the DFA-- I'm sorry-- is going to be which subset now we're going to start off with. It's going to be the subset corresponding to just the start state of M. And the accepting states are going to be-- of the deterministic machine are going to be all of the subsets that have at least one accepting state from the NFA. OK?

So I hope you got that. Because I'm going to give you another little check-in here. Which is I'm going to ask you, how big is M prime? How many states does M prime have? I told you what those states are. So just go think about that. So check-in two-- if M has n states, how many states does M prime have by this construction? OK, so let's launch the next poll. OK, five seconds-- and I think we're almost done here. good.

All right, share results-- I don't know if sharing results is a good thing. I'm not trying to make you, if you didn't get the right answer-- because most of the people did get the right answer-- but if you didn't get the right answer, trying to make you feel bad. But it's a little bit of suggestion that you need to review some basic concepts.

So the basic concept here is if you have a collection-- you have a set of states, how many subsets are there? And the number of subsets is going to be exponential. So if you have a collection of n elements, the number of subsets of those n elements is 2 to the n. That's the fact we're using here. And that's why M prime has 2 to the n states, if M had n states. And you should make sure you understand why that is.

All right, so with that, as requested, we're going to have a little break. And that break is going to last us exactly five minutes. So we will return in five minutes. I'm going to be prompt. So I gave you a little timer here. So please, I'm going to begin it right when this is over.

OK, almost ready. I hope you're all refreshed and ready for the second half. So now that we have nondeterminism, we're going to use that as a tool to prove the closure properties that we were aiming for, starting from last lecture. OK.

So remember, let's look at closure under a union. Now, we already did that, but I'm going to do it again, but this time, using nondeterminism. And you'll see how powerful nondeterminism is. Because it's going to allow us to do it almost with no effort.

We'll start off the way we did before. I'm going to start off with two DFAs. But actually, these could be NFAs even. But let's say we started with the two DFAs for the two languages A1 and A2. And now we're going to construct an NFA, recognizing the union. And that's good enough, because we already know that we can convert NFAs to DFAs. And therefore, they do regular languages, too. OK.

So now here are the two DFAs that do the languages A1 and A2. And what I'm going to do is I'm going to put them together into a bag of states, which is going to be M, the NFA that's going to do the union language. So remember-- what does M supposed to do? M is supposed to accept its input, if either M1 or M2 accept.

So how is it going to do that? What it's going to do, we're going to add a new state to M, which is going to branch under epsilon transitions. And now you can start to see how useful these epsilon transitions are going to be for us. Going to branch under epsilon transitions to the two original start states of M1 and M2.

And we're done. Why? Well, now, nondeterministically, as we get an input, w coming in to M-- and at the very beginning, even just right after it gets going, the very first thing that happens is it's going to branch to M1 and also branch to M2 nondeterministically as two possibilities. And then inside M1 and M2, it's going to actually start reading the input. And each one is going to be now following along as it would have originally the states corresponding to reading those input symbols.

And M, as a combination of M1 and M2, is going to have a possibility for one state in M1 and one state in M2. And so M is going to have those combined into one package. And now at the end of the input, if either of these end up at an accepting state, then M is going to accept as a nondeterministic finite automaton. Because that's how nondeterminism works.

You accept if either-- if any one of the branches ended up accepting-- which is just what you need for union. So when we're doing union, you want either one of these to be accepting. And the nondeterminism just is built conveniently to allow us to do the union almost for free.

So you can again, thinking about nondeterminism as terms of parallelism, you could think of the nondeterministic machine as running in parallel M1 and M2 on the input. And if either one of them ends up accepting, M will accept. Or you can think about it in terms of that guessing that I referred to before, which means that as M is getting-- when it's just about to read the first symbols of its input, it guesses whether that's going to be an input accepted by M1 or an input accepted by M2.

And the magic of nondeterminism is that it always guesses right. So that input happens to be an input that's going to be accepted by M2. M is going to guess that M2 is the right way to follow. And it's going to go in the M2 direction. Because nondeterminism, the magic is you always guess right. I wish that was true in real life. It would make exams a lot easier.

Anyway, so now let's see how we can use that to do closure under concatenation. OK, so now we're going to actually have a picture of very similar to the one we had originally. But now using nondeterminism, we can make it work. So here we have the two machines doing the two languages, A1 and A2. And we're going to combine them into one bigger machine M, as shown.

Remember, what M is supposed to do is accept its input, if there's some way of splitting that input, such that the first half is accepted by the M1, and the second part is accepted by M2. The way we're going to get that effect is by putting in a transit empty-- empty transitions, epsilon transitions, going from the accept states of M1 to the start state of M2, just as I've shown in this diagram.

So these were the original accepting states of M1. And now they're going to be declassified as accepting states. But they're going to have new transitions, empty transitions, attached to them, which allow them to branch to M2 without reading any input.

And so intuitively speaking, this is going to do the right thing. Because once M1 has accepted some part of w, then you can nondeterministically branch to M2. And you're going to be start processing inside M2.

And the point is-- I jumped ahead of myself-- is that the reason why it fixes the problem we had before is that the epsilon transitions don't-- the machine does not have to take that. It can stay where it is as one nondeterministic option, or it can move along the epsilon transition, without reading any input, as another nondeterministic option. So it's using this nondeterminism now to both stay in M1 to continue reading more of the input and to jump into M2 to start processing what might be the second half or the second part of the input which M2 accepts.

And you can think of it in terms of the guessing as that the machine is guessing where to make that split. Once it found an initial part that's accepted by M1, it guesses that this is the right split point. And that passes to M2. But there might be other guesses that it could make corresponding to other possibilities.

And so with nondeterminism, it always guesses right. If there is some way to split the string into two parts accepted by M1 and M2, the machine will make that good guess. And then M1 will accept the first part, and M2 will accept with the second part. And we'll get M accepting that whole string altogether.

And so that is the solution to our puzzle for how do we do closure under concatenation. OK, I hope that came through. Because we're just getting going with nondeterminism. We're going to be using nondeterminism a lot, and you're going to need to get very comfortable with it. OK?

Now let's do closure under star. And closure under star works very similarly, but now we're just going to have a single language. If A is regular, so is A star. So they're not a pair of languages, because a star is a unary operation applying to just a single language. So if we have a DFA recognizing A, in order to show that A star is regular, we have to construct a machine that recognizes A star. And the machine we're going to construct is as before and then an NFA. OK?

So here is M, the DFA for A. And we're going to build an NFA M prime that recognizes A star. And let's think now, what does it mean to recognize A star? So if I'm going to give you an input, when is it in the star language? What does M prime have to do?

So remember what star is. Star means you can take as many copies of you lot as you like of strings in the original language, and that's in the star language. So to determine if something is in the star language, you have to see, can I break it up into pieces which are all in the original language?

So you want to see, can I take my input w and cut it up into a bunch of pieces-- four, in this case-- where each of those pieces are members of A, the members of the original language? So that's what M prime's job is. It has its input and wants to know, can I cut that input up into pieces, each of which are accepted by the original machine M? That's what M prime does.

And if you think about it a little bit, really what's happening is that as soon as M-- so M prime is going to be simulating M. That's the way I like to think about this, as having M inside. So if you were going to be doing this yourself, you're going to take w. You're going to run it for a while. You'll see, oh, M is accepted. Now I have to start him over again to see if it accepts the next segment.

So every time M accepts, you're going to restart M to see if it accepts another segment. And so by doing that, you're going to be cutting w up into different segments, each of which is accepted by M. Of course, it's never totally clear whether you should, for any given segment, you should cut it there or you should wait a little longer and find another, a later place to cut. But that's exactly the same problem that we had before with concatenation. And we solved it using nondeterminism, and we're going to solve it again using nondeterminism.

So the way we're going to get that effect of starting the machine over again, once it's accepted, is by adding in epsilon transitions that go from the start states back to-- from the accept state back to the start state. So now every time M has accepted, it has an option-- not a requirement, but has an option. It can either stay continuing to process, or it could restart, making a cut at that point and trying to see if there's yet a second, another segment of the input that it's going to accept.

And this is basically the whole thing, with one little problem that we need to deal with. And that is we need to make sure that M prime accepts the empty string. Because remember, the empty string is always a member of the star language. And as it's written right now, we're going to be requiring there to be at least one copy of at least one segment. We're not taking into account the possibility of no segments, which is the empty string.

And the way we're going to get that is-- well, I mean, one thing, one way to get to add-- so we're missing the empty string right now. So how do we fix it? Basically, we're just going to take the construction we have on the screen, and we're going to adjust it to add in the empty string. Because it's possibly missing.

One way to do that, which is tempting, but wrong, is to make the start state of M an accepting state for M prime. So we could have made this an accepting state, too. And now M prime is also going to accept the empty string. That's the good news.

The problem is that the start date might be playing some other role in M besides just being the start. There might be times when M comes back to the start state later on. And if we make the start state the an accept state, it's going to suddenly start accepting a bunch of other things too, which might not be intended. So it's a bad idea to make the start state an accept state.

Instead, we'll take the simple solution alternative of adding a new start state, which will never be returned to under any circumstances, and make that a new start-- an accept state as well. So here, we'll have to make this additional modification. So as I'm saying, this is what we need to do. And the way we'll do that is by adding a new start state, which is also an accept state, to make sure it accepts the empty string. And then that also can branch to start off M as before, if the string that's input is not the empty string. And so then M prime is actually going to have to do some work to see if it can be cut off, as it was doing before.

So that's the proof of closure under star. I'm not going to do it anything beyond what I've just described. These proofs by picture are convincing enough, I hope. And if not, they are explained in somewhat more detail, somewhat more formally, in the textbook. But for the lecture, this is where I'm going to stop, with these two arguments.

And so now-- oh, we have one quick check-in on this. So if M has n states, how many states does M prime have by this construction? So I'm not intending these to be very hard, more just to keep you awake. So how many states does M prime have? OK, maybe a little too easy even for a check-in. Yeah, everybody is getting this one.

Because all you did was-- we added one new state. So the answer is as you have-- I think pretty much everybody is observing that it's number b. So I'm going to end the polling, and I'm going to share the results. And everybody got that one. And so let's continue on.

And so the very last thing we're going to do today is show you how to convert regular expressions to NFAs, thereby showing that every language that you can describe with a regular expression is a regular language. On Tuesday, we'll show how to do the conversion in the other direction and so thereby showing that these two methods of describing languages are equivalent to one another.

So here's our theorem. If R is a regular expression, and A is the language-- a set of strings that that regular expression describes, then A is regular. OK? So we're going to show how to convert. The strategy is to convert R to an equivalent NFA M. And so we have to think about, remember, these regular expressions that we introduced last time. These are these expressions that look like ab union b star, something like that-- so built up out of the regular operations from the primitive regular expressions that don't have any operations, that we're calling atomic.

So if R is an atomic regular expression, it just looks like either just a single symbol or an empty string symbol or an empty language symbol. Or R can be a composite regular expression-- whoops. We're having a little-- yeah, so we have two possibilities here. R is either atomic or composite. And so let's look at what the equivalent expression is in each case.

So if R is just the single letter regular expression-- that's a totally legitimate regular expression, just a regular expression 1. So that just describes the language of the string 1. So we have to make an NFA which accepts-- which recognizes just that language, accepts only the string 1. So it's a very simple NFA. It just starts in the start state.

And on that single symbol, it branches to an accept state. And there were no other transitions allowed. So if you get anything else coming in besides that one, that string, which is just that one symbol, the NFA is going to reject it. If it's too long, if it gets aa coming in, well, there's nowhere to go from this accepting state on an A. So the machine is just going to die. It has to be in an accept state at the end of the input.

Now, I want you think for yourself for a minute, how do we make an NFA which accepts only the empty string and no other strings? You can do that with just one state with an NFA, just this one here. The machine is going to start off in the start state, which is also immediately an accept state. So it accepts the empty string. But if anything else comes in, there's nowhere to go when the machine dies.

So this machine accepts just the empty string. Or its language is the language with one element, the empty string. How about the empty language? Well, here's an NFA which has no accepting state, so it can't be accepting anything.

Now, if we have a composite regular expression, we're already finished. Because we showed how to build up-- we showed constructions which give us closure under union, concatenation, and star. And those constructions are going to enable us to build up the NFAs that do the language of these more complex regular expressions built up out of the NFAs that do the individual parts.

So if we already have NFAs that do R1 and R2, then the closure under union construction gives us an NFA that does R1 union R2 as a language. So I hope that's clear, but I'm going to do an example which will hopefully illustrate it. And it's going to show you-- basically, what I'm giving you is an automatic procedure for converting a regular expression into an equivalent NFA.

So let's just see that procedure in action, which is really just following this recipe that I described for you. So here is a regular expression a union ab star. So this is a regular expression. It's some language. Whatever it is, I don't care. But I want to make an NFA which recognizes that same language.

And the way I'm going to do that is first build NFA for the components, the subexpressions of this regular expression, and then combine them, using our closure instructions, to be NFAs for larger and larger subexpressions, until I get the NFA that's the equivalent of the entire expression. So let's just see how that goes.

So the very most primitive parts, the smallest subexpressions here, are just the expressions for a and for b. So here's the one just for a. So this is the NFA which recognizes the language, which is just the one string a. Here is the NFA whose language is just the one string b.

And now I want an NFA which accepts only the string ab. Now, of course, you could just do that by hand yourself. It's simple enough. But what I'm arguing is that we can do this automatically, using the closure construction for concatenation. Because really there's a hidden concatenation symbol. This is a concatenate b.

So now for ab, I'm going to take the thing from a and the part from b-- so these two things that I had from before, and use the concatenation construction to combine them. You see that? So now I have automatically an NFA which does the language whose string is just ab, just the ab string. And it's not the simplest NFA. You can make a simpler one, but the virtue of this one is that I got it automatically just by following the closure construction.

So now I'm going to do a more complex one, just the inside here, a union ab. So the way I'm going to build that is from the two parts, the a part and the ab part, the a part and the ab part. So here is the a part. Here's the ab part. I've already got those from before. It's really kind of a proof by induction here. But I think it's simple enough, we don't have to use that language.

So we have the a part, the ab part. And now we are going to apply the closure under union construction to combine those into one machine. And remember how that worked. We had a new symbol here, which branches under empty string to the previous-- we're adding a new start state, which branches to the original start states under empty transition. And now this is an NFA for this language, a union ab.

And lastly, now we're one step away from getting the star of this. And how are we going to do that? We're going to take this thing here and apply the construction for the star closure. And that's going to be an NFA which does a union ab star, which is what we wanted in the first place.

So first, we're going to bring that one down. Because we've already built that one. And now remember how we built the closure under star. We made the accepting states return back to the start state, and we added a new start state to make sure we got the empty string in there that transitioned to the original start state under epsilon. OK?

So that's all I wanted to say for today's lecture. Let's do a quick review. Very important concept, nondeterminism and nondeterministic finite automata-- we proved they were equivalent in power, showed the class of regular languages closed under concatenation in star. We showed how to do conversion of regular expressions to NFAs.

So I think that is it for today's lecture. And thank you, all, for being here. I'll try to answer a few of these.

"Why does concatenation have order?" Well, because it's an ordered construction. Is there a simple way to prove closure under concatenation without using nondeterminism? No.

"Why are the empty strings at the accept state? Can't they be at any state? Doesn't star make copies of any part of the input?" No, it's only-- you have to think about what's going on. You have to branch back to the beginning only on an accept. Because that means you found a piece that's in the original language.

"Is there an automaton that can add some or subtract memory automata?" Well, depends on what you mean by all that. But certainly, there are more powerful machines that we're going to study than finite automata. But yes, there is. And even finite automata can add and subtract, if you present the input in the right way. I would refer you to the first problem on the homework. So I think I'm going to check out then. Take care, everybody. Bye-bye.