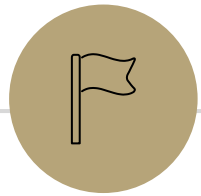


# NFAs, Relating Models of Computation

CSE 311 Summer 2025  
Lecture 20

# Announcements

- Midterm retakes are happening today and tomorrow
- HW7 is out and due this Friday, 8/15
- HW5 solutions are at the front of the class
- Final logistics are posted on the Exams page of the course website



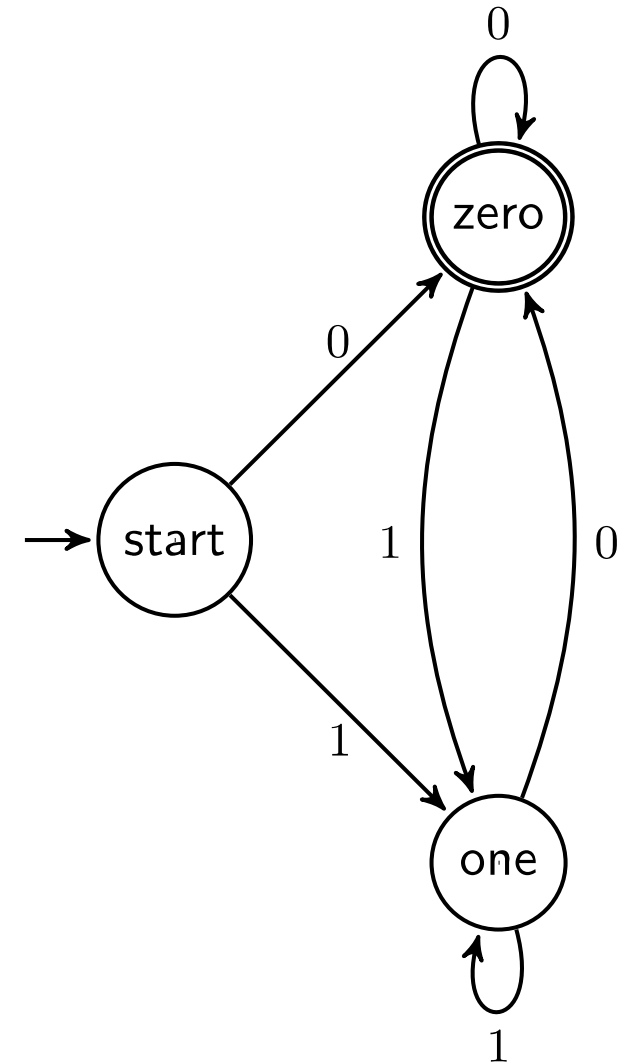
# Review



# Deterministic Finite Automaton

Our machine is going to get a string as input. It will read one character at a time and update "its state." At every step, the machine thinks of itself as in one of the (finite number) vertices. When it reads the character it follows the arrow labeled with that character to its next state.

Start at the "start state" (unlabeled, incoming arrow). After you've read the last character, accept the string if and only if you're in a "final state" (double circle).



# Deterministic Finite Automata

Some more requirements:

Every machine is defined with respect to an alphabet  $\Sigma$

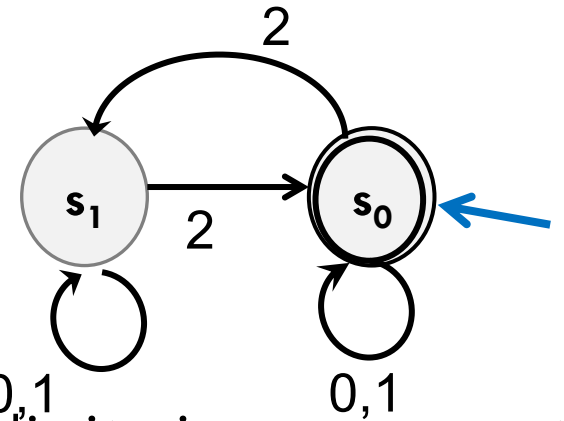
Every state has exactly one outgoing edge for every character in  $\Sigma$ .

There is exactly one start state; can have as many accept states (aka final states) as you want – including none.

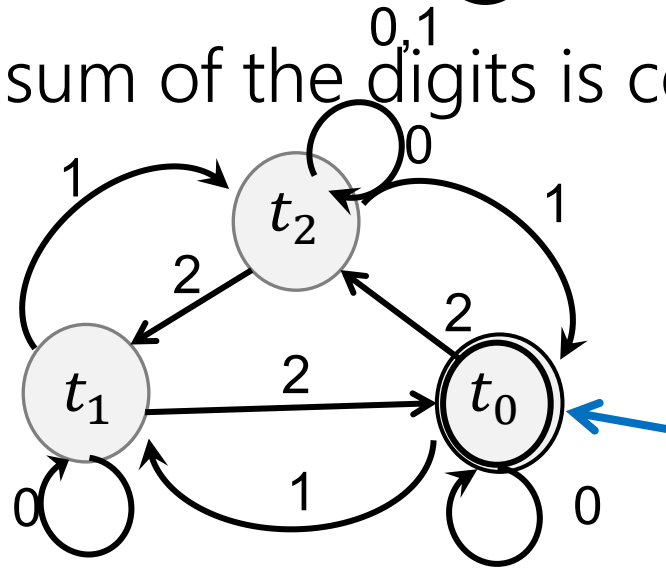
# Design some DFAs

Let  $\Sigma = \{0,1,2\}$

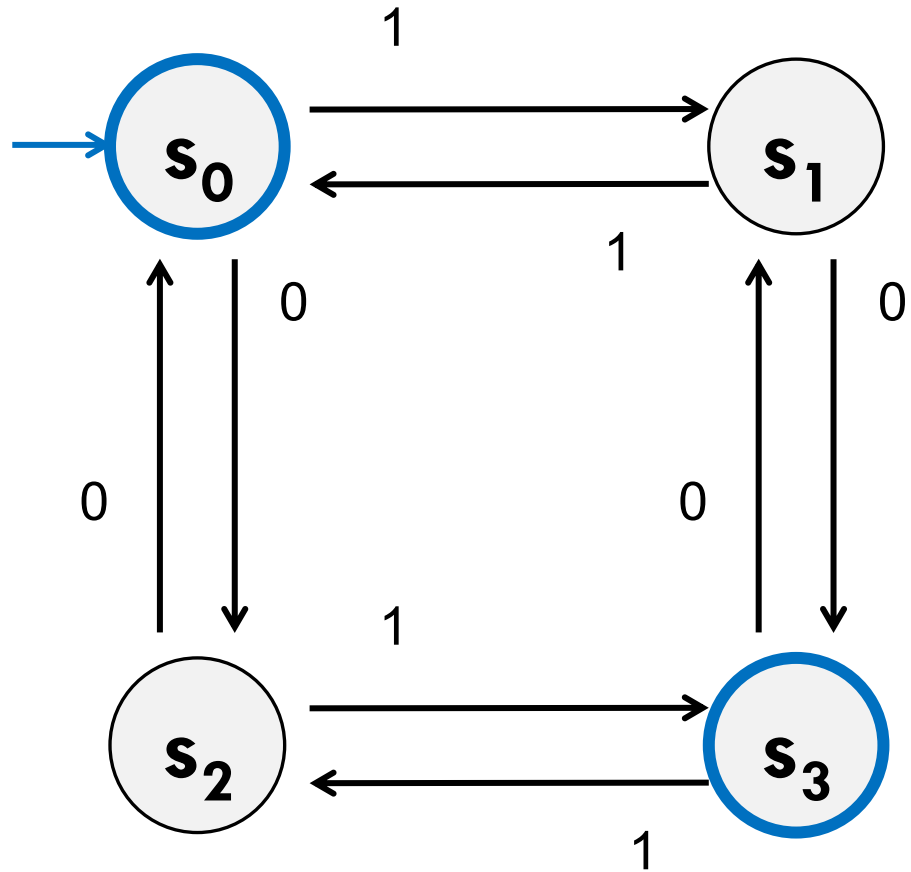
$M_1$  should recognize "strings with an even number of 2's."



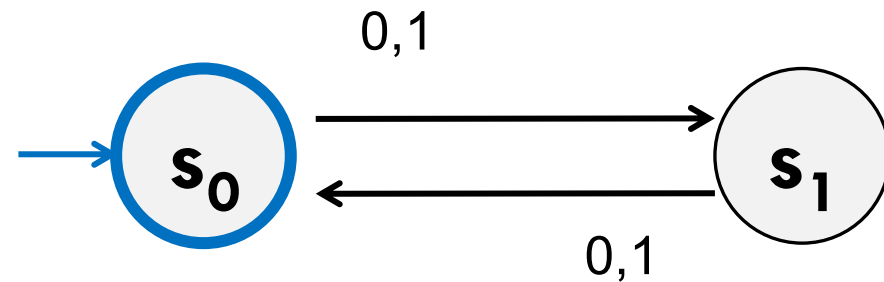
$M_2$  should recognize "strings where the sum of the digits is congruent to 0 (mod 3)"



# What language does this machine recognize?



That's all binary strings of even length.



# DFA Minimization

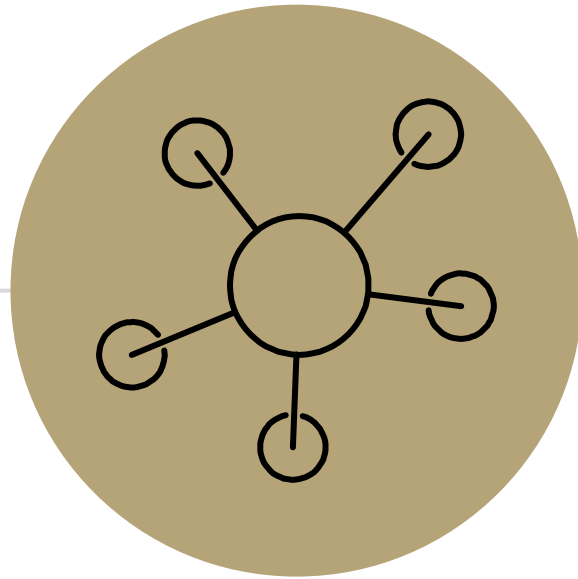
We can know!

Fun fact: there is a **unique** minimum DFA for every language (up to renaming the states)

High level idea – final states and non-final states must be different.

Otherwise, hope that states can be the same, and iteratively separate when they have to go to different spots.

In some quarters, we cover it in detail. But...we ran out of time.



NFAs, Power of  
machines

# Let's try to make our more powerful automata

We're going to get rid of some of the restrictions on DFAs, to see if we can get more powerful machines (i.e. can recognize more languages).

From a given state, we'll allow any number of outgoing edges labeled with a given character. The machine can follow any of them.

We'll have edges labeled with " $\epsilon$ " – the machine (optionally) can follow one of those without reading another character from the input.

If we "get stuck" i.e. the next character is  $a$  and there's no transition leaving our state labeled  $a$ , the computation dies.

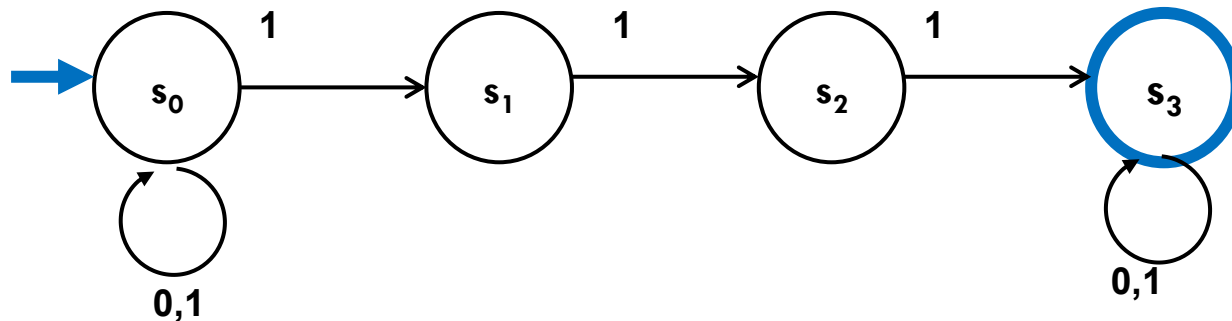
# Nondeterministic Finite Automata

An NFA:

Still has exactly one start state and any number of final states.

The NFA accepts  $x$  if there is some path from a start state to a final state labeled with  $x$ .

From a state, you can have 0,1, or many outgoing arrows labeled with a single character. You can choose any of them to build the required path.



Wait a second...

But...how does it know?

Is this realistic?

# Three ways to think about NFAs

**“Outside Observer”**: is there a path labeled by  $x$  from the start state, to the final state (if we know the input in advance can we tell the NFA which decisions to make)

**“Perfect Guesser”**: The NFA has input  $x$ , and whenever there is a choice of what to do, it **magically** guesses a transition that will eventually lead to acceptance (if one exists)

**“Parallel exploration”**: The NFA computation runs all possible computations on  $x$  in parallel (updating each possible one at every step)

# So...magic guessing doesn't exist

I know.

The parallel computation view is realistic.

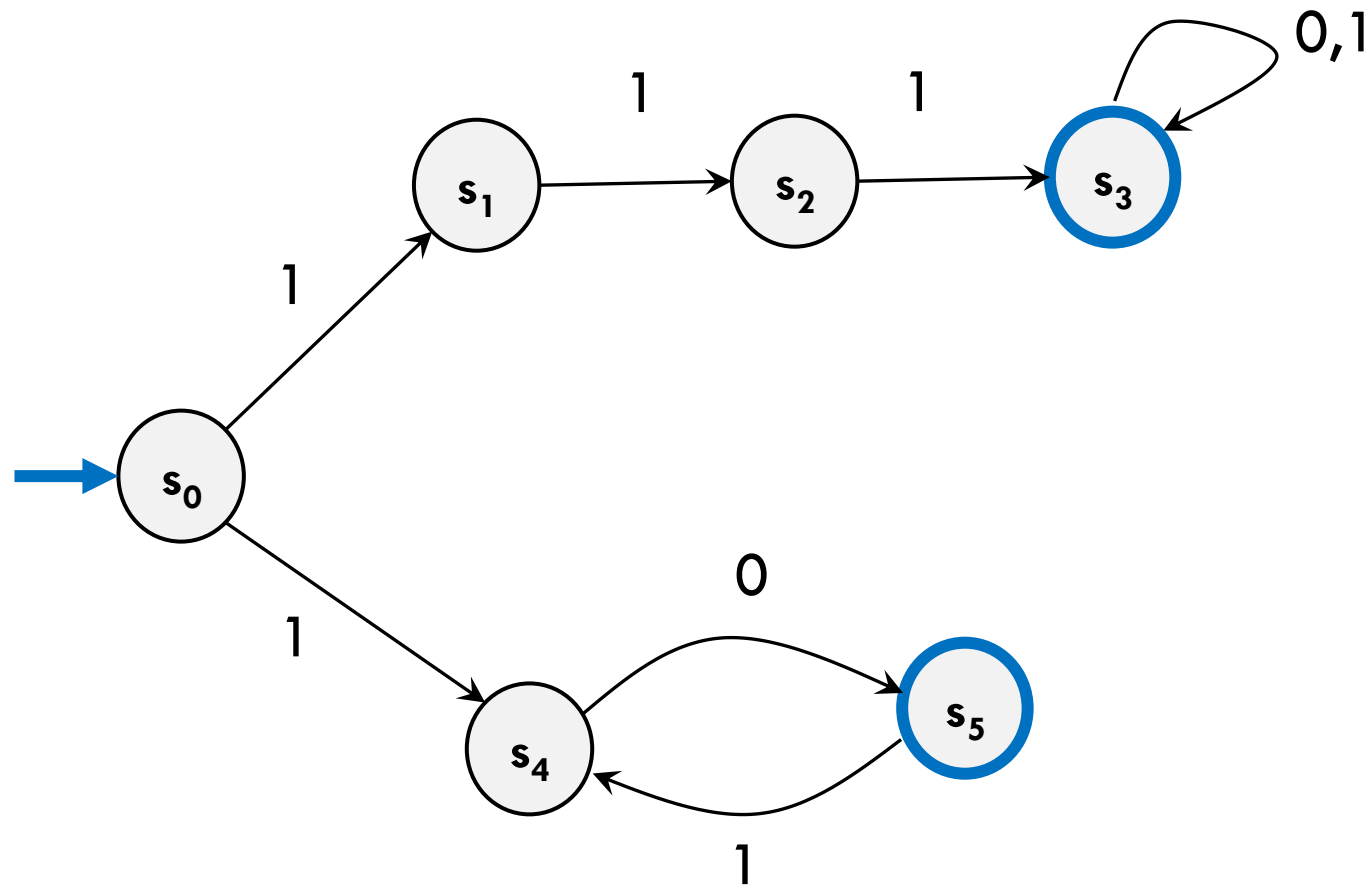
Lets us give simpler descriptions of complicated objects.

This notion of "nondeterminism" is also really useful in more advanced CS theory (you'll see it again in 421 or 431 if not sooner).

Source of the P vs. NP problem.

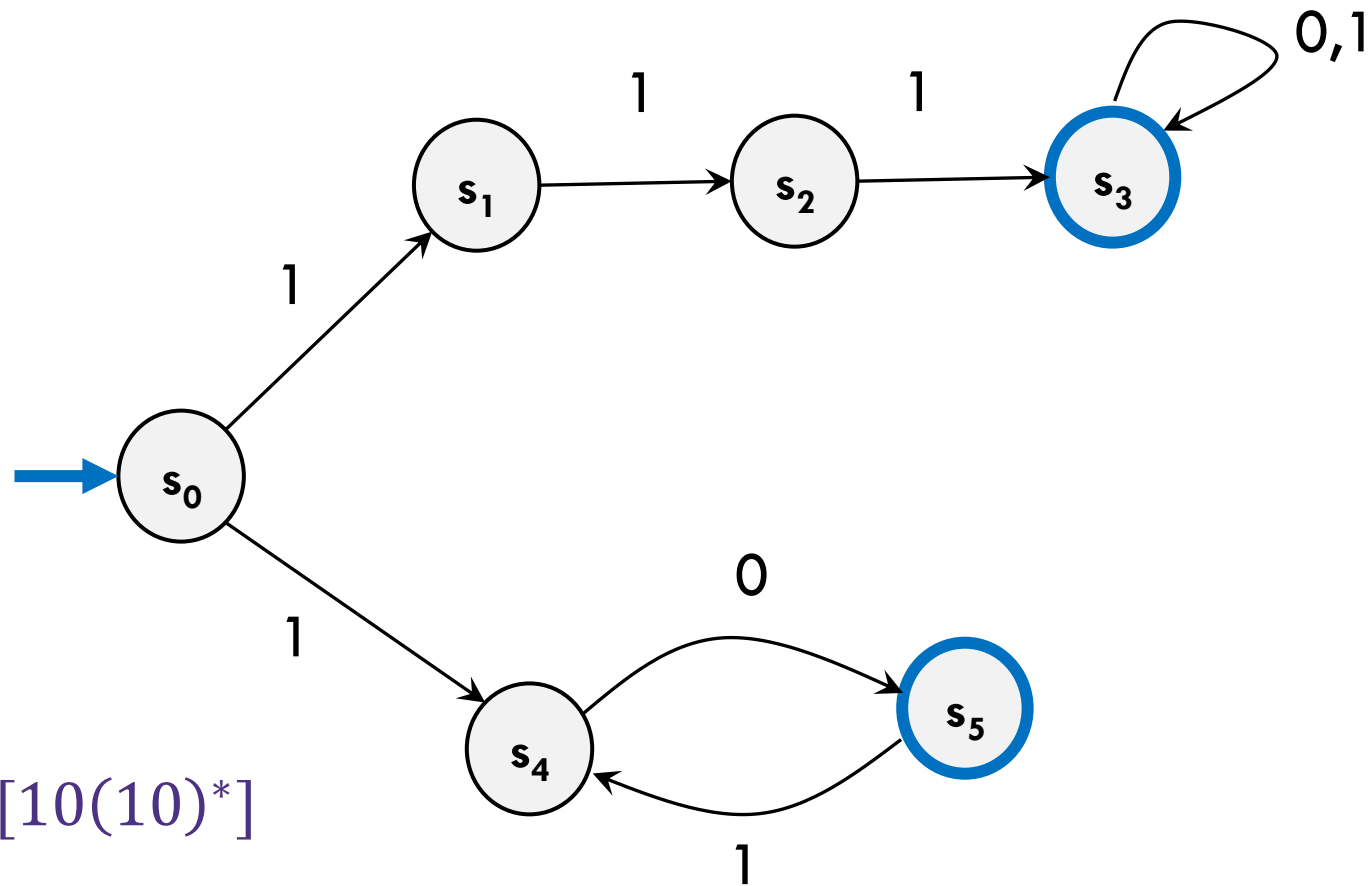
# NFA practice

What is the language of this NFA?



# NFA practice

What is the language of this NFA?

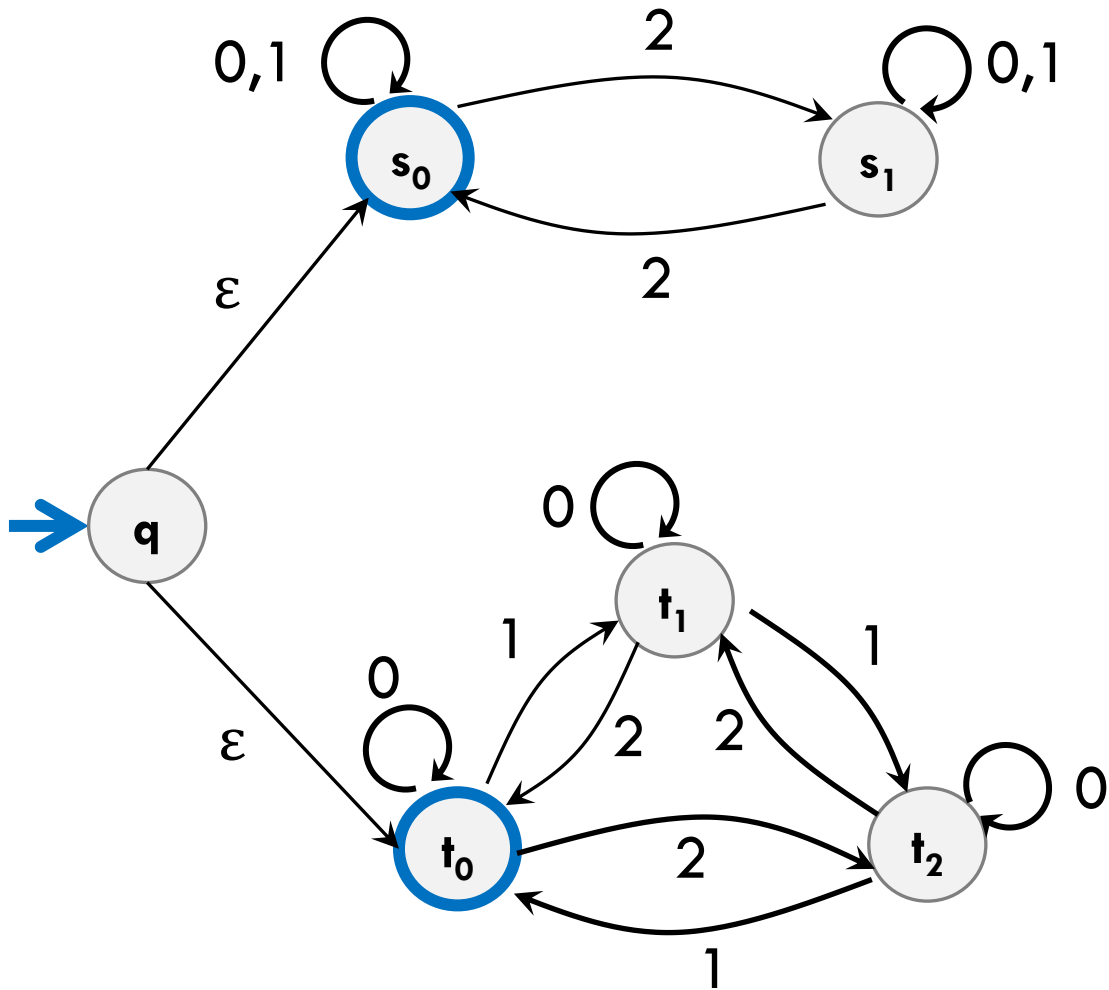


$111(0 \cup 1)^*$

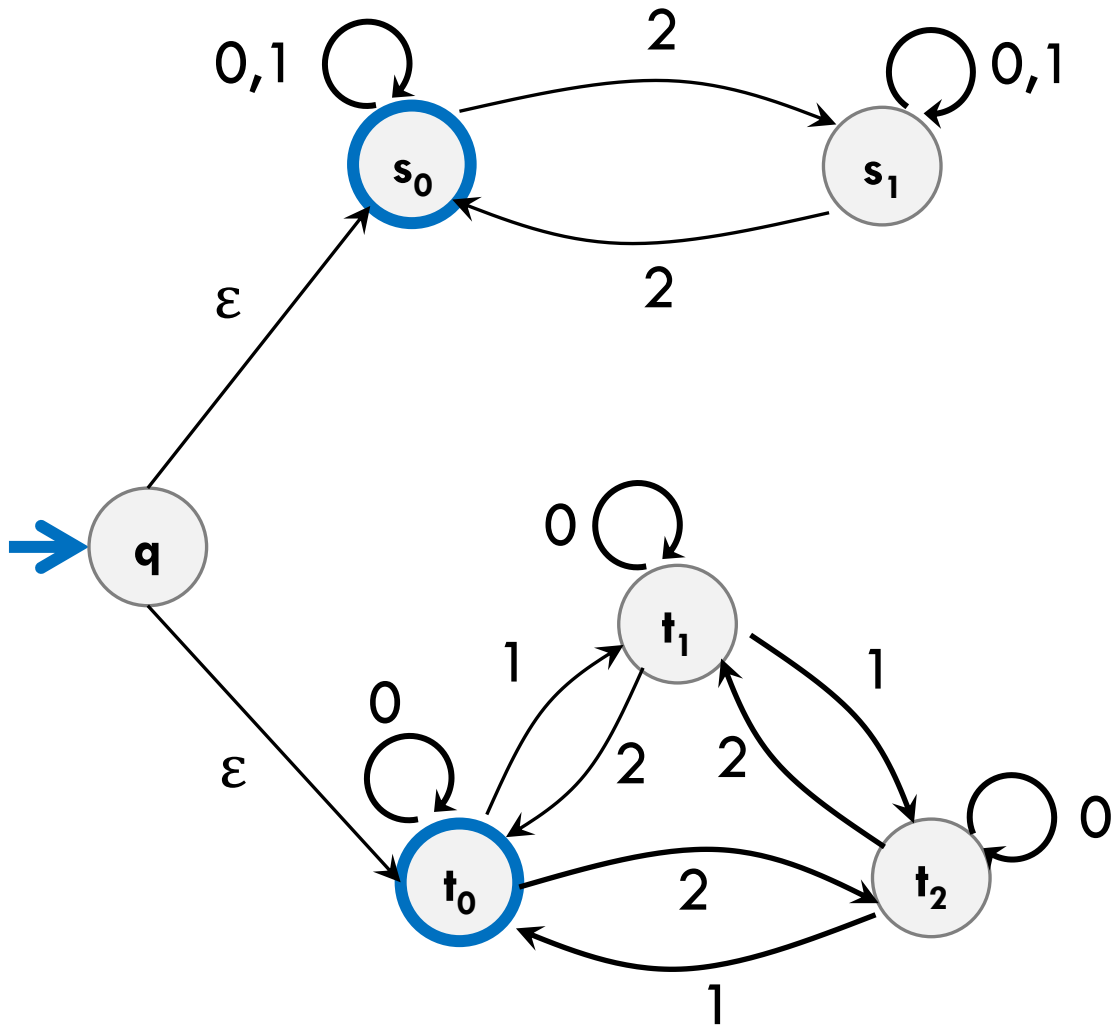
$10(10)^*$

Overall  
 $[111(0 \cup 1)^*] \cup [10(10)^*]$

# What about those $\varepsilon$ -transitions?



# What about those $\varepsilon$ -transitions?



The set of strings over  $\{0,1,2\}$  with an even number of 2's or the sum  $\%3 = 0$ .

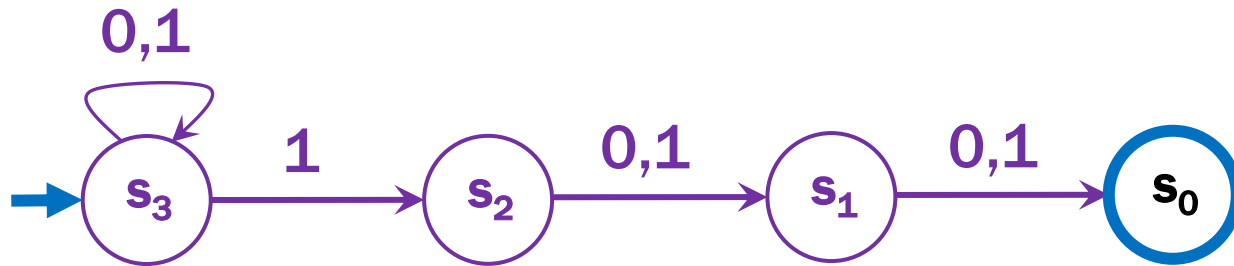
# NFA that recognizes "binary strings with a 1 in the third position from the end"

"**Perfect Guesser**": The NFA has input  $x$ , and whenever there is a choice of what to do, it **magically** guesses a transition that will eventually lead to acceptance (if one exists)

Perfect guesser view makes this easier.

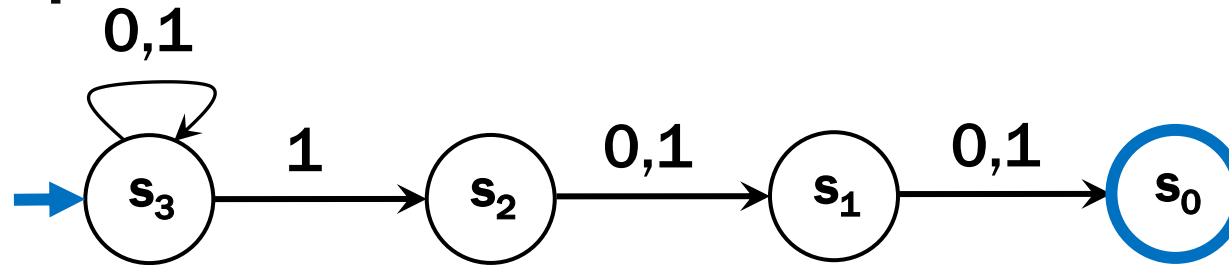
Design an NFA for the language in the title.

NFA that recognizes "binary strings with a 1 in the third position from the end"

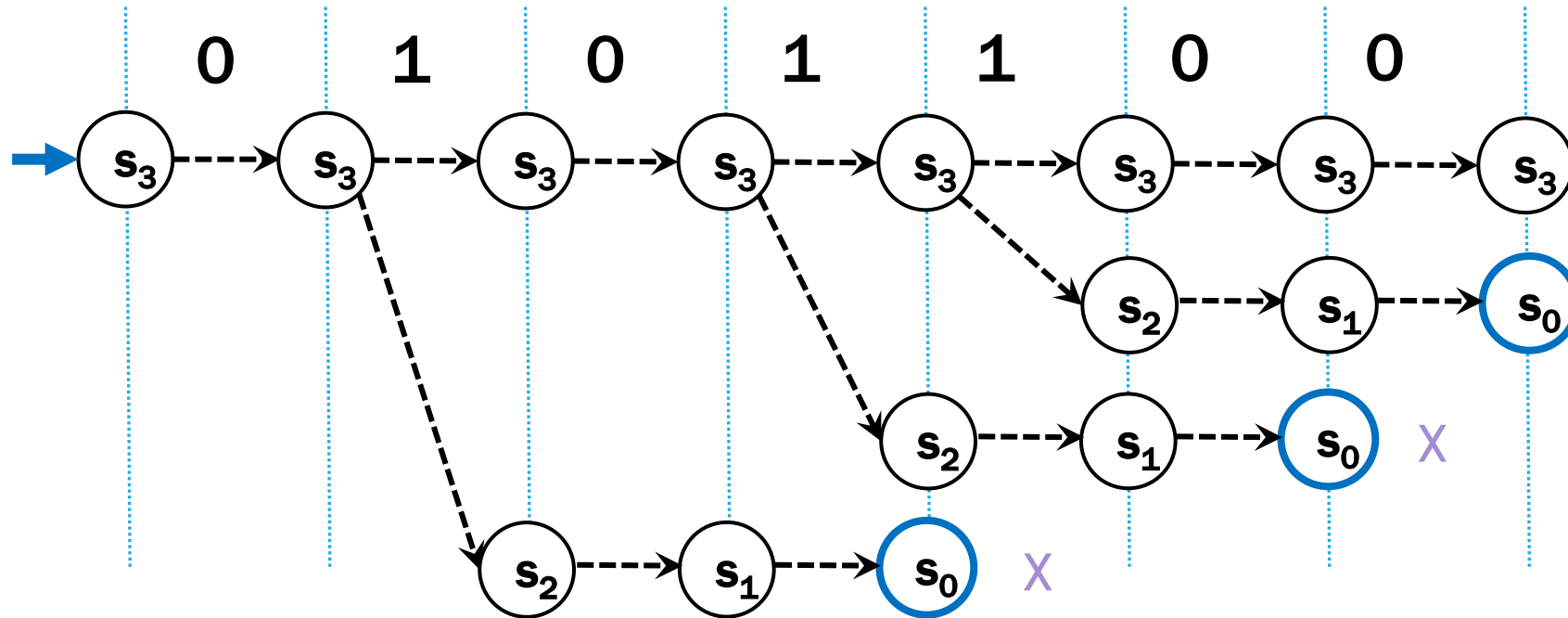


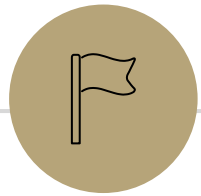
That's WAY easier than the DFA...

# Parallel Exploration view of an NFA



Input string 0101100





# Relating Models of Computation

# Regularity

So NFAs/DFAs what can and can't they do?

Can NFAs do more than DFAs?

How do they relate to context-free-grammars? Regular expressions?

i.e. is there a language  $L$  such that  $L$  is the language of an NFA but not a DFA? Or vice versa?

What about CFGs/regexes?

# Regularity

So NFAs/DFAs what can and can't they do?

Can NFAs do more than DFAs?

How do they relate to context-free-grammars? Regular expressions?

## Kleene's Theorem

For every language  $L$ :

$L$  is the language of a regular expression if and only if

$L$  is the language of a DFA if and only if

$L$  is the language of an NFA

# Regularity

So NFAs, DFAs, and regular expressions are all “equally powerful”

Every language either can be expressed with any of them or none of them.

A set of strings that is recognized by a DFA (equivalently, recognized by an NFA; equivalently, the language of a regular expression) is called a **regular language**.

So to show a language is “regular” you just need to show one of these and prove it works. There are some “irregular” languages (that don’t have a corresponding NFA/DFA/regex).

CFGs are “more powerful” (every regular language can also be represented with a CFG, but some languages with CFGs have not NFA/DFA/regex).

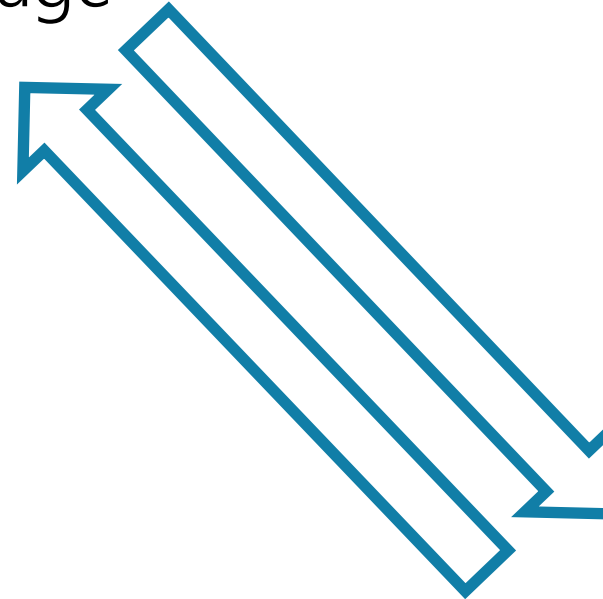
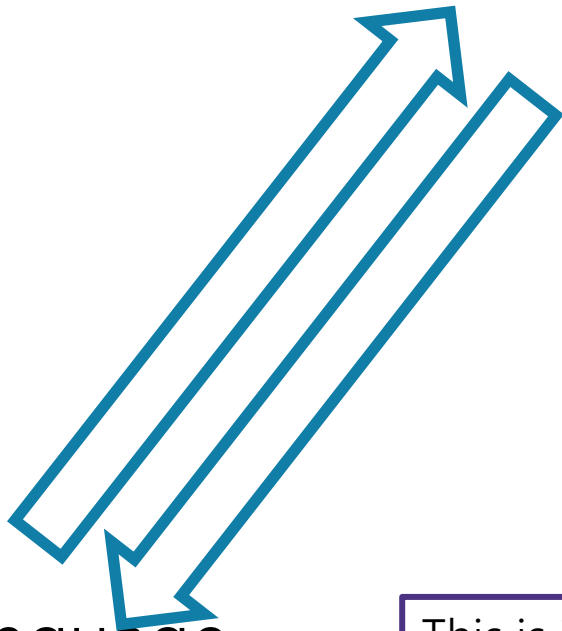
# Proof [sketch]

$L$  is the language  
of an NFA.

$L$  is the language  
of a regular  
expression.

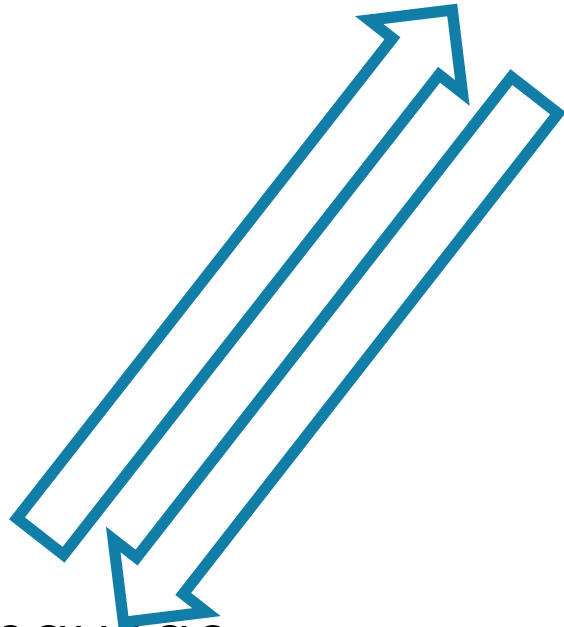
This is just a “sketch” of the proof. We want you to get the intuition for why this is true, we’ll go very quickly for some cases.

$L$  is the language  
of a DFA.



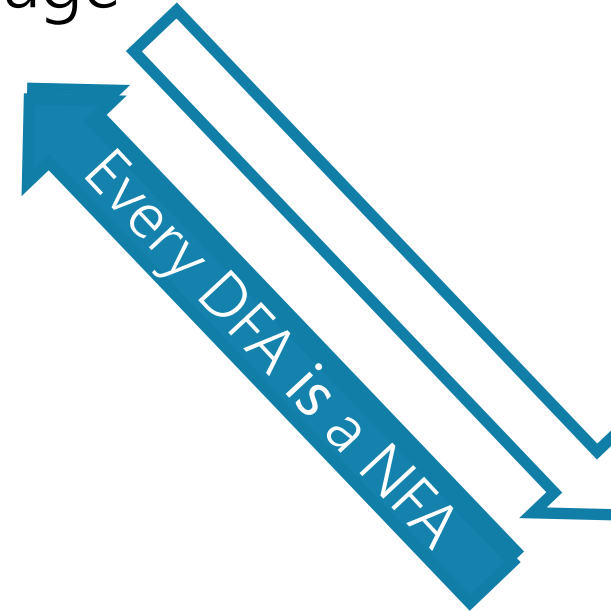
# Proof [sketch]

$L$  is the language of a regular expression.



$L$  is the language of an NFA.

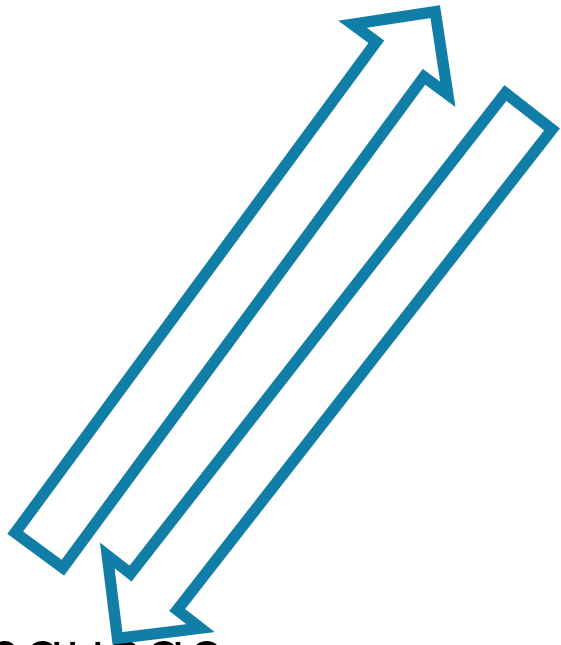
Suppose  $L$  is the language of some DFA  $M$ .  $M$  also satisfies the requirements for an NFA, so  $L$  is also the language of an NFA.



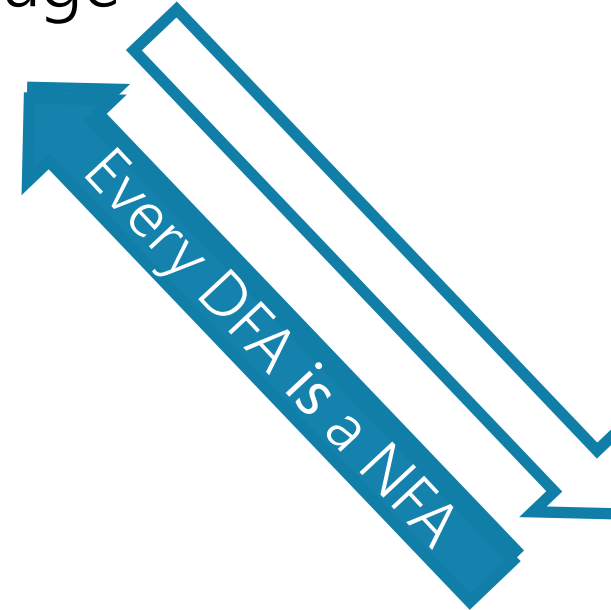
$L$  is the language of a DFA.

# Proof [sketch]

$L$  is the language  
of a regular  
expression.



$L$  is the language  
of an NFA.



$L$  is the language  
of a DFA.

# Can we convert an NFA to a DFA?

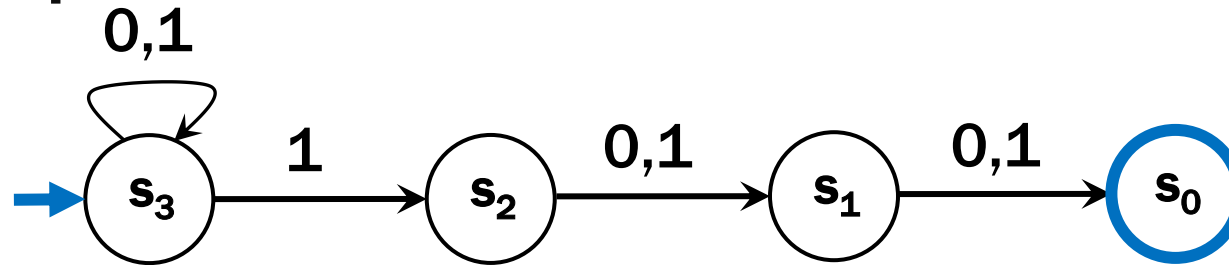
NFAs are magic though! DFAs can't guess...

**Parallel exploration:** The NFA computation runs all possible computations on  $x$  step-by-step at the same time in parallel

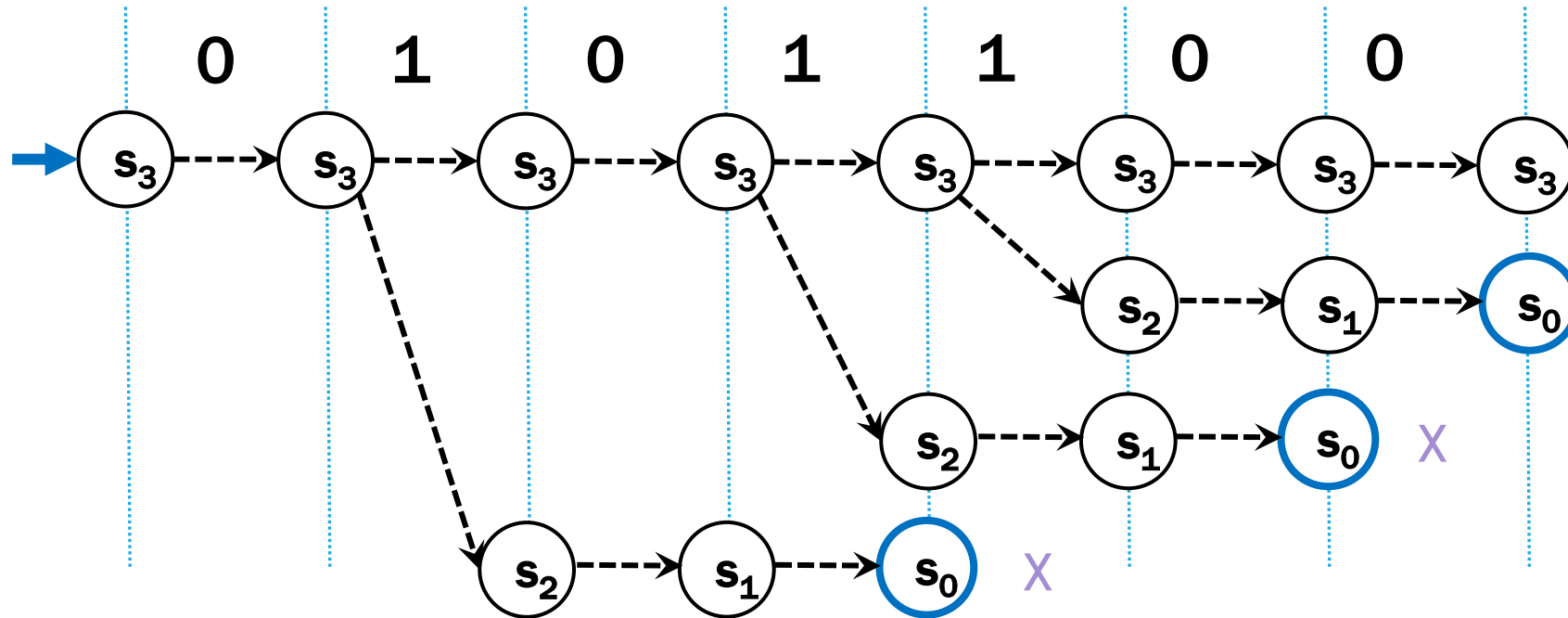
At any step, the set of all possible states we could be in is fixed!

And the update steps are deterministic if we just check all possibilities!

# Parallel Exploration view of an NFA



Input string 0101100



# Can we convert an NFA to a DFA?

NFAs are magic though! DFAs can't guess...

**Parallel exploration:** The NFA computation runs all possible computations on  $x$  step-by-step at the same time in parallel

At any step, the set of all possible states we could be in is fixed!

And the update steps are deterministic if we just check all possibilities!

# Converting from an NFA to a DFA

Let  $N$  be an NFA with a set of states  $S$ .

Need to define a DFA  $D$  that recognizes the same language.

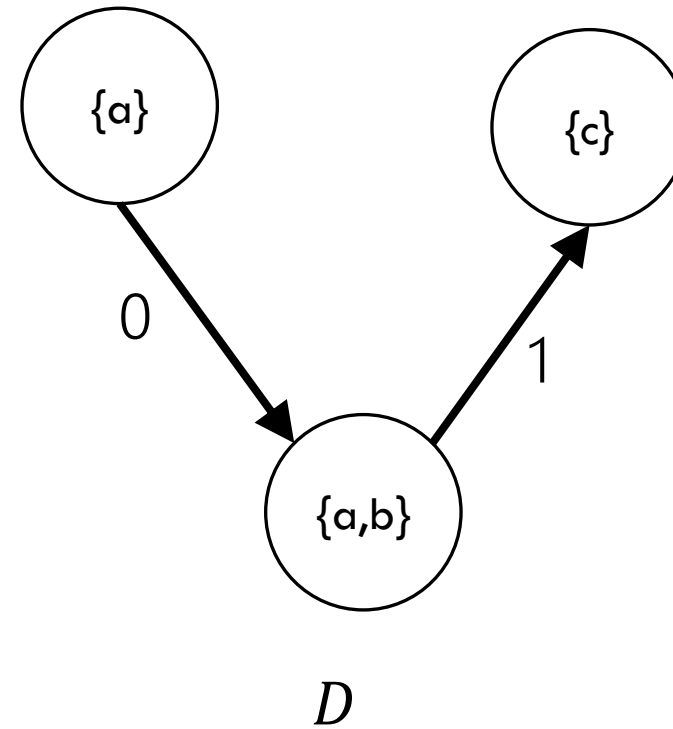
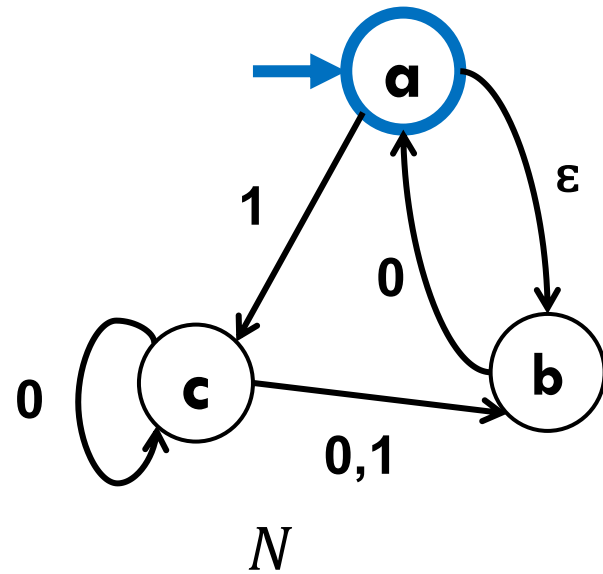
Let  $D$  be a DFA with set of states  $\mathcal{P}(S)$ .

How do we update?

If I'm in a set of states  $X$ , if the next character to be read is  $a$

Transition to  $\{y: \exists x \in X \text{ such that } y \text{ is reachable from } x \text{ in } N \text{ using exactly one } a \text{ transition and any number of } \varepsilon\text{-transitions}\}$ .

# An example (starting point)



# Finishing the DFA

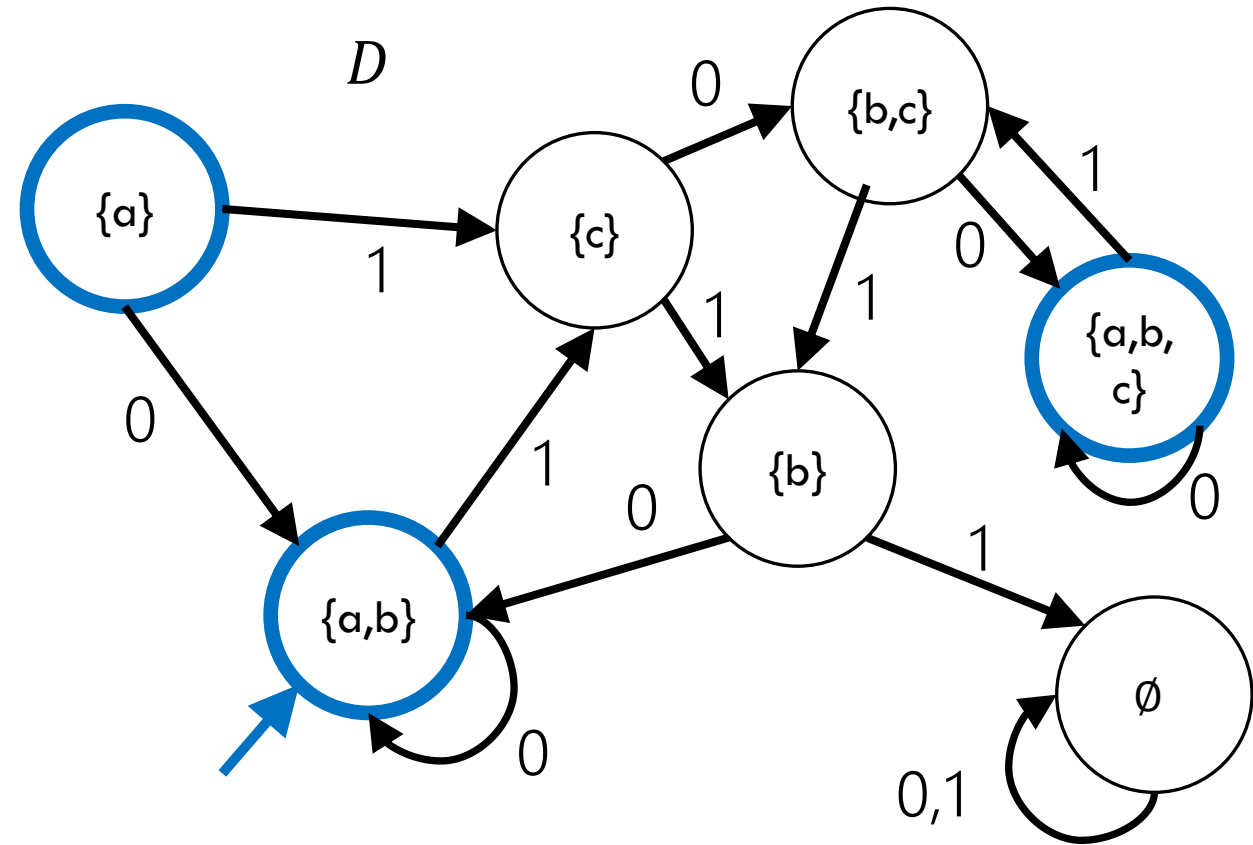
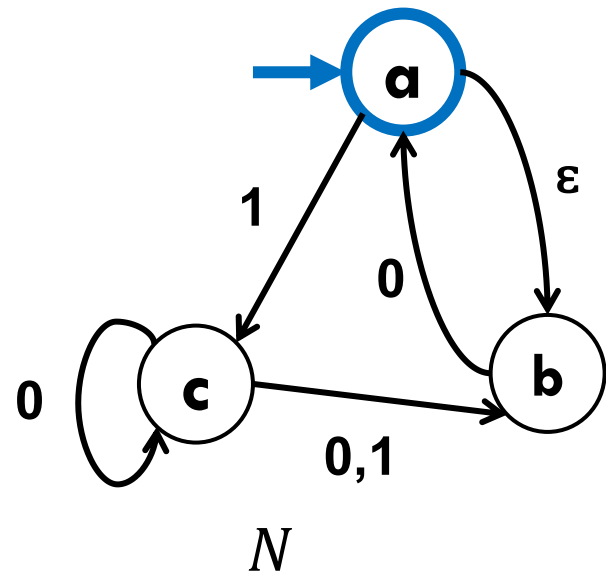
What about start and accept states?

The start state of  $D$  is  $\{x: x \text{ is the start state of } N \text{ or } x \text{ is reachable from the start state of } N \text{ with only } \varepsilon\text{-transitions}\}$

i.e. the states the NFA could be in before reading a character of the input.

Final states?  $X$  is a final state if there is an  $x \in X$  such that  $x$  is a final state of  $N$ . (If at least one version of the computation is in a final state, then the NFA will accept)

# An example



# Proof Sketch

Define  $P(n)$ : "on all strings of length  $n$ , the set of states the NFA could be in processing  $n$  corresponds to the state the DFA is in"

Show  $P(n)$  for all  $n$  by induction.

The choices of start and final states ensure  $x$  is accepted by the NFA if and only if it is accepted by the DFA.

# More formally (the “powerset construction”)

The original NFA

States:  $Q$

Start state:  $q_0$

Transition function:  $\delta(q, a)$

Outputs set of all states reachable from  $q$  using one  $a$  transition (and any number of  $\varepsilon$ -transitions)

Final States:  $F$

The constructed DFA

States:  $\mathcal{P}(Q)$

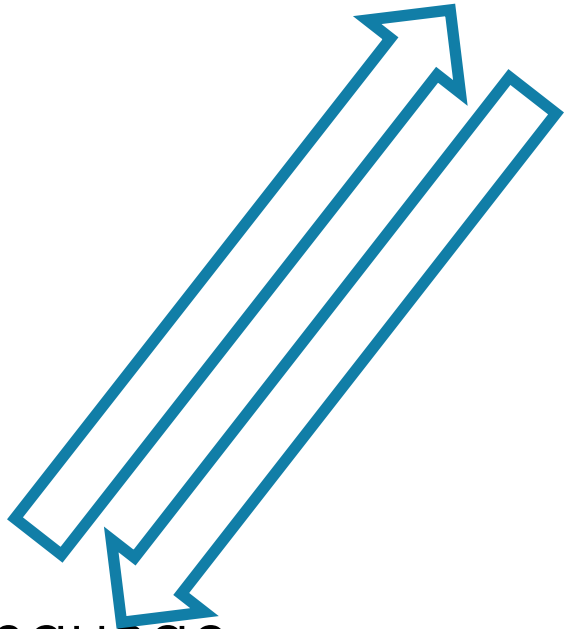
Start state:  $\{q' : q' \text{ reachable from } q_0 \text{ with only } \varepsilon\text{-transitions}\}$

Transition function:  $\delta_D(S, a) = \bigcup_{q \in S} \delta(q, a)$ .

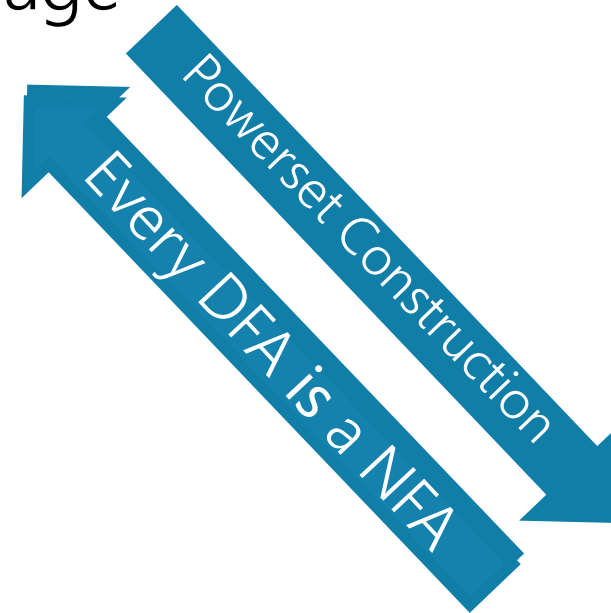
Final States:  $\{S : S \cap F \neq \emptyset\}$

# Proof [sketch]

$L$  is the language of a regular expression.



$L$  is the language of an NFA.



$L$  is the language of a DFA.

Every regular expression has a corresponding NFA.

Proof by...

Structural induction!

Regular expressions are recursively defined, so we can prove something about every regular expression via induction.

What was that definition again...

# Regular Expressions

## Basis:

$\varepsilon$  is a regular expression. The empty string itself matches the pattern (and nothing else does).

$\emptyset$  is a regular expression. No strings match this pattern.

$a$  is a regular expression, for any  $a \in \Sigma$  (i.e. any character). The character itself matching this pattern.

## Recursive

If  $A, B$  are regular expressions then  $(A \cup B)$  is a regular expression  
matched by any string that matches  $A$  or that matches  $B$  [or both]).

If  $A, B$  are regular expressions then  $AB$  is a regular expression.  
matched by any string  $x$  such that  $x = yz$ ,  $y$  matches  $A$  and  $z$  matches  $B$ .

If  $A$  is a regular expression, then  $A^*$  is a regular expression.  
matched by any string that can be divided into 0 or more strings that match  $A$ .

Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

Base Cases:

$\emptyset$

$\varepsilon$

$a$  ( $a \in \Sigma$ )

Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

Base Cases:

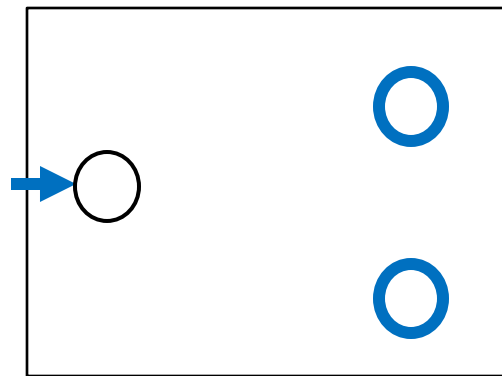


Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

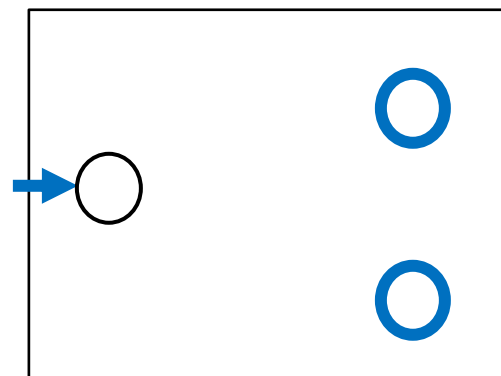
Let  $R$  be a regex not covered by the base cases. By the exclusion rule,  $R = A \cup B$  or  $AB$  or  $A^*$  from some regexes  $A, B$

Inductive Hypothesis: Suppose  $P(A)$  and  $P(B)$ .

Inductive Step: **Case 1:  $A \cup B$**



$N_A$



$N_B$

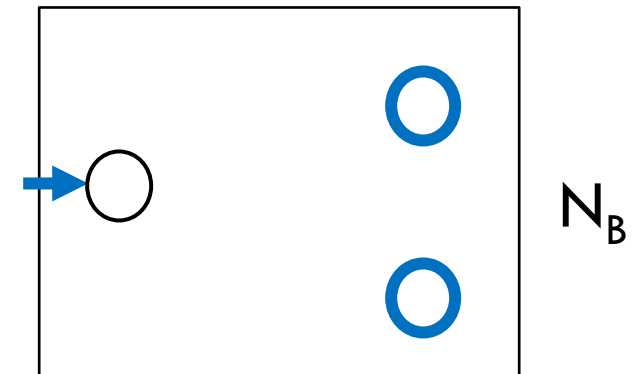
Only a sketch for this proof – so we'll just doodle stuff. Let  $N_A$  recognize  $A$ 's language, and  $N_B$  recognize  $B$ 's language.

Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

Let  $R$  be a regex not covered by the base cases. By the exclusion rule,  $R = A \cup B$  or  $AB$  or  $A^*$  from some regexes  $A, B$

Inductive Hypothesis: Suppose  $P(A)$  and  $P(B)$ .

Inductive Step: **Case 1:  $A \cup B$**



Want a machine that accepts exactly strings matched by  $A$  or  $B$ .

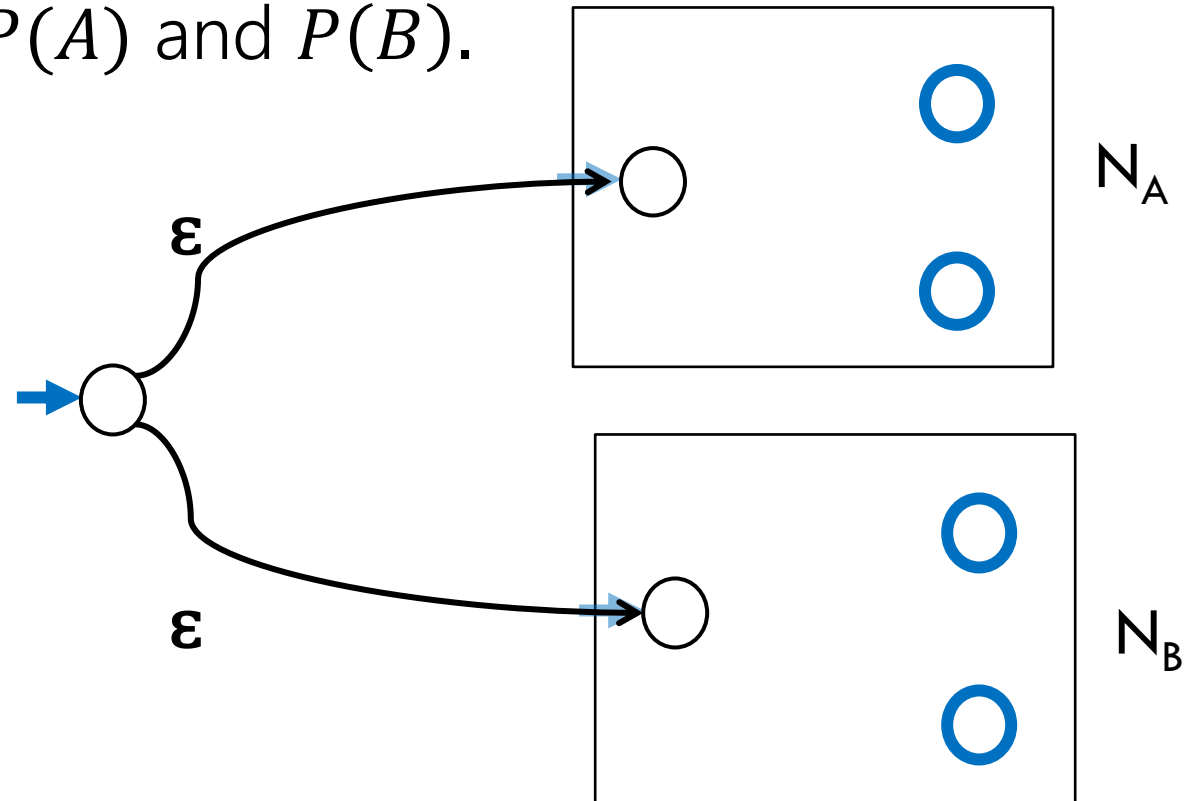
Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

Let  $R$  be a regex not covered by the base cases. By the exclusion rule,  $R = A \cup B$  or  $AB$  or  $A^*$  from some regexes  $A, B$   
Inductive Hypothesis: Suppose  $P(A)$  and  $P(B)$ .

Inductive Step: **Case 1:  $A \cup B$**

Match  $A \cup B$ ? Then you match one of the two regexes. New machine transitions into start state of appropriate old machine. Will be accepted.  
Accepted by the machine? First step **has** to be an  $\epsilon$ -transition into one of the machines, so would have been accepted by the smaller machine, so must have matched  $A$  or  $B$ .

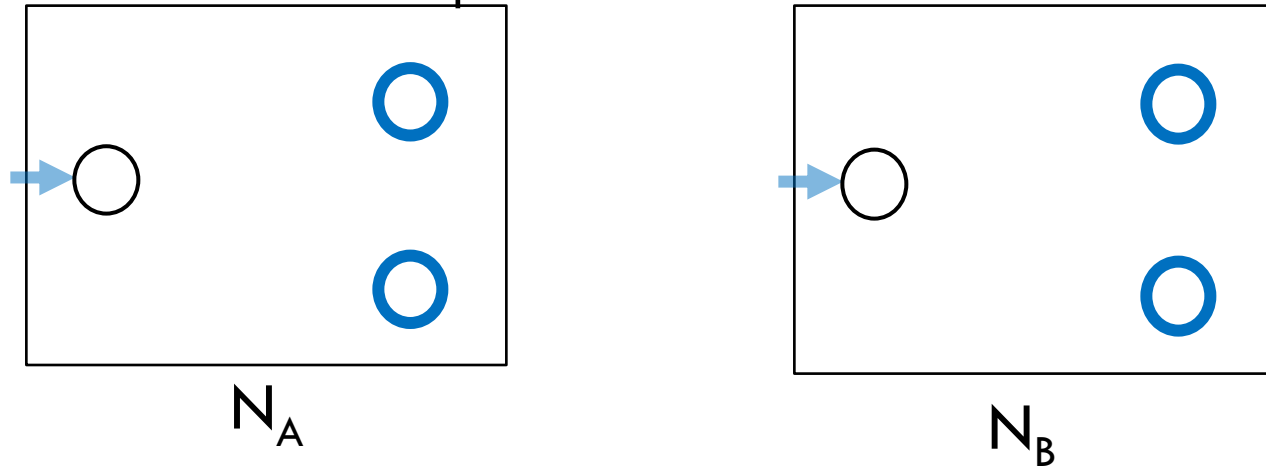
Want a machine that accepts exactly strings matched by  $A$  or  $B$ .



Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

Let  $R$  be a regex not covered by the base cases. By the exclusion rule,  $R = A \cup B$  or  $AB$  or  $A^*$  from some regexes  $A, B$   
Inductive Hypothesis: Suppose  $P(A)$  and  $P(B)$ .

Inductive Step: **Case 2:  $AB$**

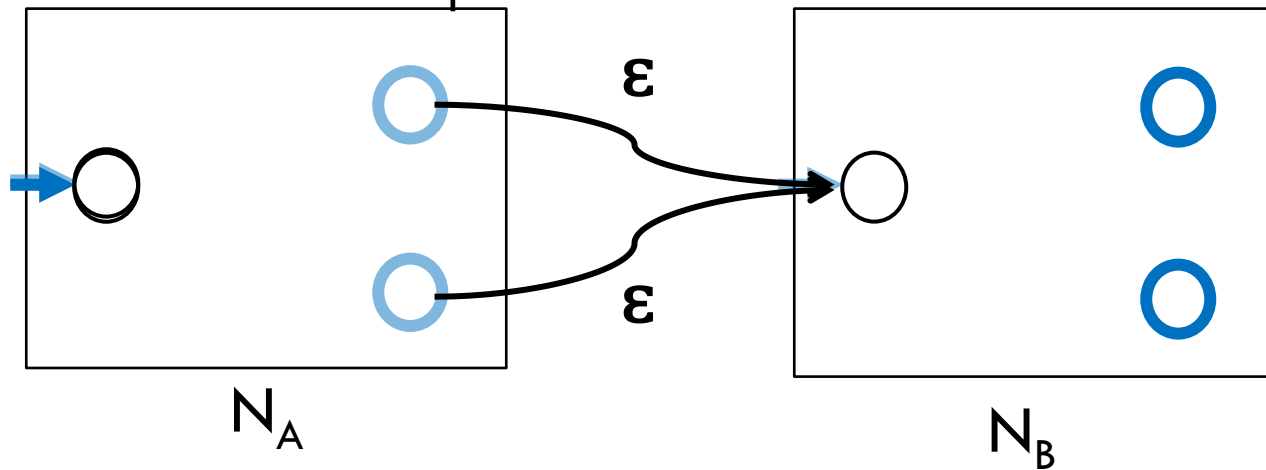


Want a machine that accepts exactly strings matched by  $AB$ .

Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

Let  $R$  be a regex not covered by the base cases. By the exclusion rule,  $R = A \cup B$  or  $AB$  or  $A^*$  from some regexes  $A, B$   
Inductive Hypothesis: Suppose  $P(A)$  and  $P(B)$ .

Inductive Step: **Case 2:  $AB$**



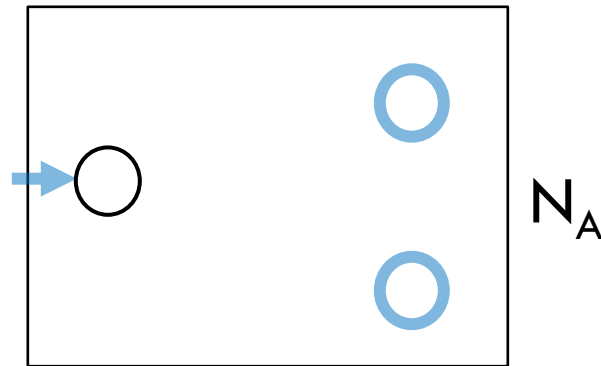
String  $x$  that matches  $AB$  can divide into  $yz$  where  $y$  matches  $A$ ,  $z$  matches  $B$ .  
NFA can run as  $N_A$  would on  $y$  take  $\epsilon$ -transition, then run as  $N_B$  would on  $z$  so accepted by  $N$   
String  $x$  that is accepted?  
 $N$  must run in  $N_A$  take  $\epsilon$ -transition, then run in  $N_B$  until acceptance. Substring read in  $N_A$  must match  $A$ . Substring read in  $N_B$  must match  $B$  (by IH) so string matches  $AB$ .

Want a machine that accepts exactly strings matched by  $AB$ .

Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

Let  $R$  be a regex not covered by the base cases. By the exclusion rule,  $R = A \cup B$  or  $AB$  or  $A^*$  from some regexes  $A, B$   
Inductive Hypothesis: Suppose  $P(A)$  and  $P(B)$ .

Inductive Step: **Case 3:  $A^*$**

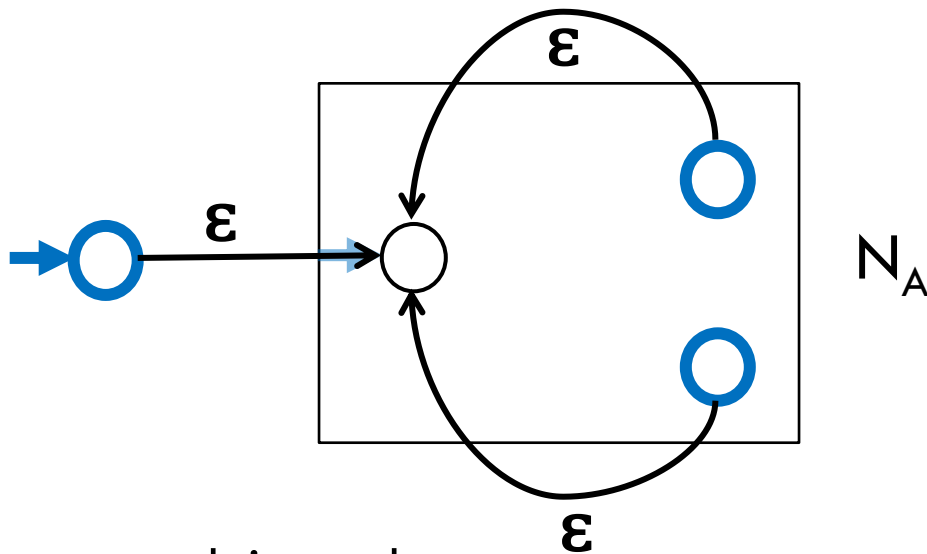


Want a machine that accepts exactly strings matched by  $A^*$ .

Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

Let  $R$  be a regex not covered by the base cases. By the exclusion rule,  $R = A \cup B$  or  $AB$  or  $A^*$  from some regexes  $A, B$   
Inductive Hypothesis: Suppose  $P(A)$  and  $P(B)$ .

Inductive Step: **Case 3:  $A^*$**



Want a machine that accepts exactly strings matched by  $A^*$ .

If  $x$  matches  $A^*$ , then by def of  $*$   $x = \epsilon$  or  $x = x_1 \dots x_k$  with each  $x_i$  matching  $A$ . If  $x = \epsilon$ , machine accepts by not transitioning. Otherwise run accepting computation in  $N_A$  for each  $x_i$  return to start until  $x_k$  then end in accept state (all possible by IH)

If accepted by  $N$ ,

Either  $\epsilon$  or go from start state of  $N_A$  to final state and  $\epsilon$ -transition back to start some number of times. So we can break string into parts accepted by  $N_A$  by IH we can break string into substrings all matched by  $A$ , i.e. we match  $A^*$ .

Let  $P(A)$  be "There is an NFA whose language is the same as the language for  $A$ ."

By principle of structural induction,  $P(A)$  holds for all regular expressions  $A$ .

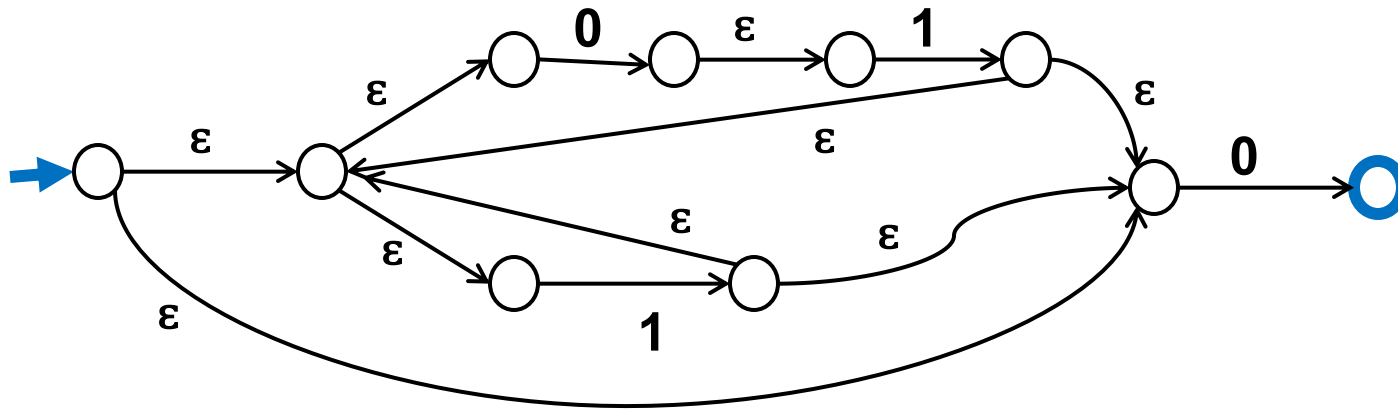
Thus every regular expression has an equivalent NFA.



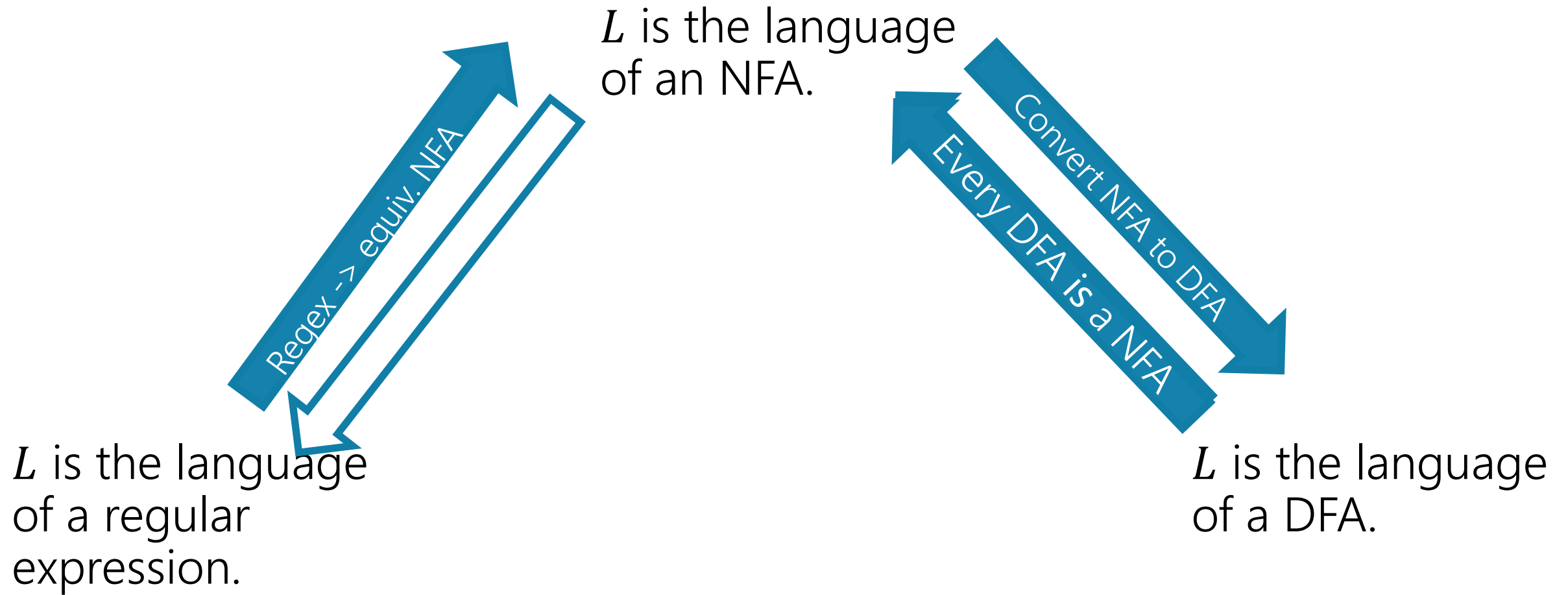
Every regex has an equivalent NFA

# An example

$(01 \cup 1)^*0$



# Proof [sketch]



# Takeaways

Nondeterminism wasn't magic. It was just efficiency.

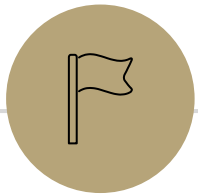
The construction we had would turn a  $k$  state NFA into a  $2^k$  state DFA.

For some languages there might be a smaller DFA. But for some it really is (essentially) that big.

“string has a 1 in the  $k^{\text{th}}$  character from the end” is an example.

The P vs. NP question asks whether nondeterminism is similar for running time on our computers (it doesn't let you do anything new, but it lets you do it MUCH more efficiently).

Next time: Showing a language is not regular!



---

## Enrichment Content

---

(optional) sketch that for every NFA there is an equivalent regular expression.

# Every NFA has an equivalent regular expression

Not responsible for this, but if you're curious:

# Generalized NFAs

Like NFAs but allow

Parallel edges

Regular Expressions as edge labels

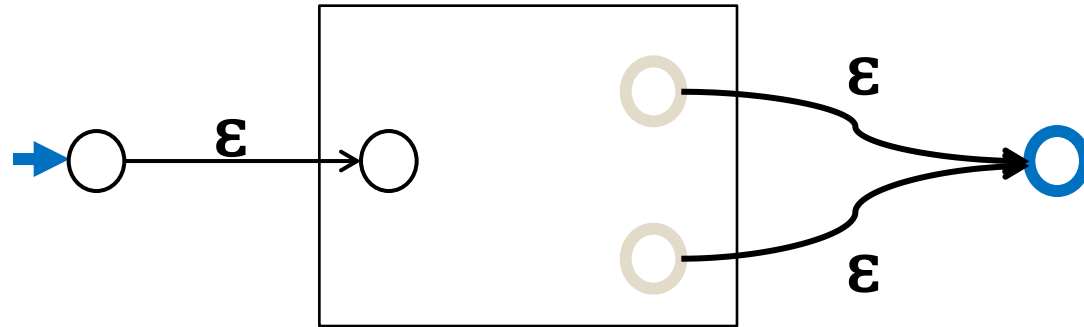
-NFAs already have edges labeled  $\epsilon$  or  $a$

An edge labeled by  $A$  can be followed by reading a string of input chars that is in the language represented by  $A$

Defn: A string  $x$  is accepted iff there is a *path* from start to final state *labeled by a regular expression* whose language contains  $x$

# Starting from an NFA

- Add new start state and final state



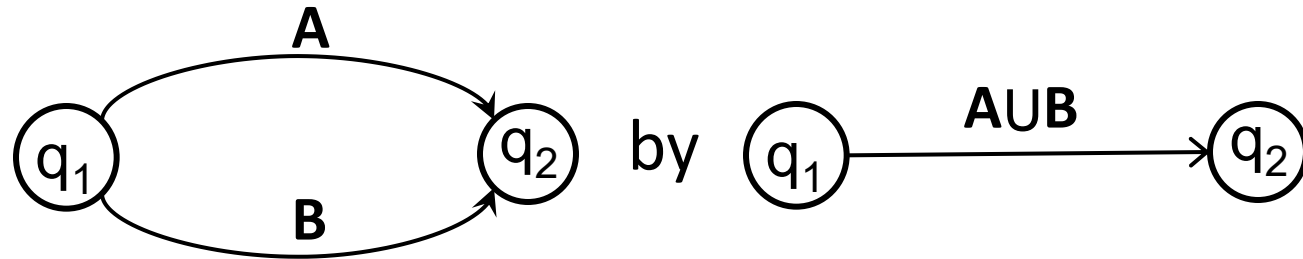
Then eliminate original states one by one, keeping the same language, until it looks like:



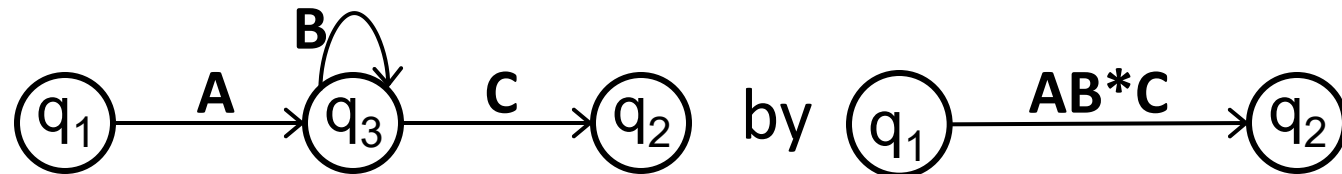
Final regular expression will be **A**

# Only two simplification rules

**Rule 1:** For any two states  $q_1$  and  $q_2$  with parallel edges (possibly  $q_1=q_2$ ), replace



**Rule 2:** Eliminate non-start/final state  $q_3$  by replacing all

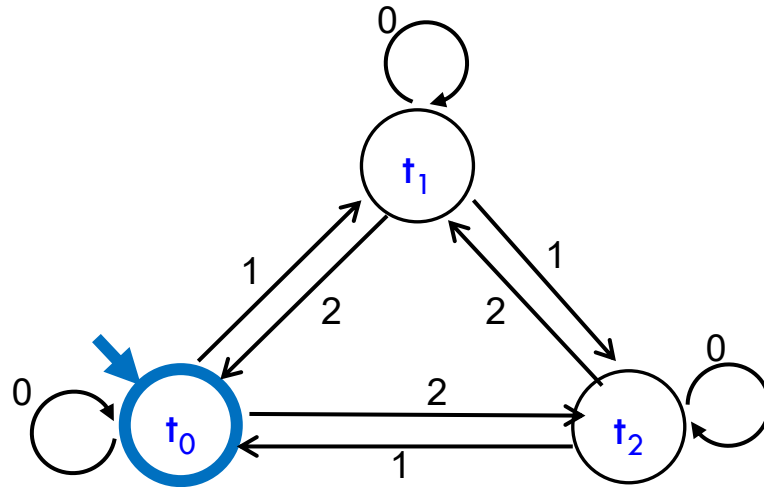


for *every* pair of states  $q_1$ ,  $q_2$  (even if  $q_1=q_2$ )

# Converting an NFA to a regular expression

Consider the DFA for the mod 3 sum

Accept strings from  $\{0,1,2\}^*$  where the digits mod 3 sum of the digits is 0



# Splicing out a state $t_1$

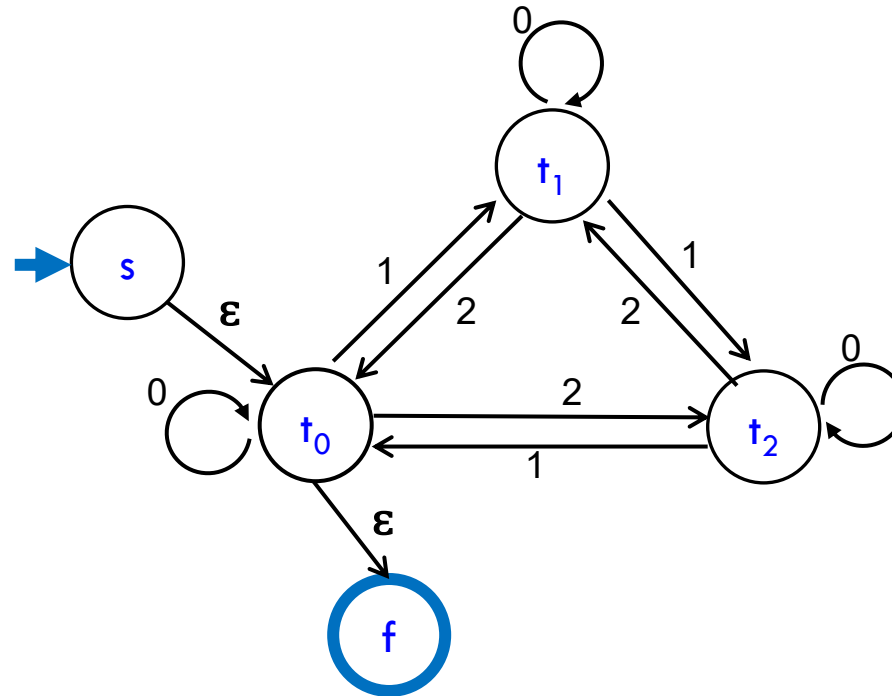
Regular expressions to add to edges

$t_0 \rightarrow t_1 \rightarrow t_0$  :  $10^*2$

$t_0 \rightarrow t_1 \rightarrow t_2$  :  $10^*1$

$t_2 \rightarrow t_1 \rightarrow t_0$  :  $20^*2$

$t_2 \rightarrow t_1 \rightarrow t_2$  :  $20^*1$



# Splicing out a state $t_1$

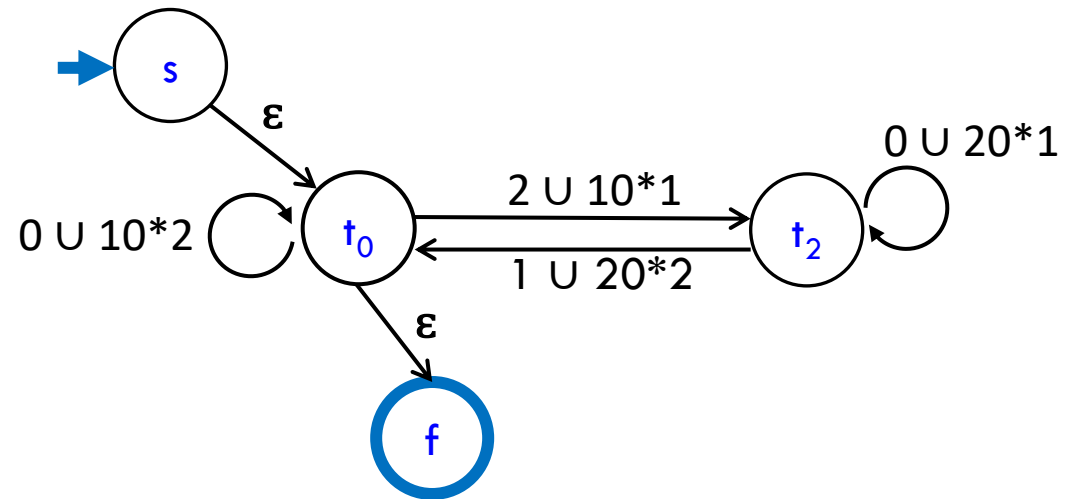
Regular expressions to add to edges

$t_0 \rightarrow t_1 \rightarrow t_0$  :  $10^*2$

$t_0 \rightarrow t_1 \rightarrow t_2$  :  $10^*1$

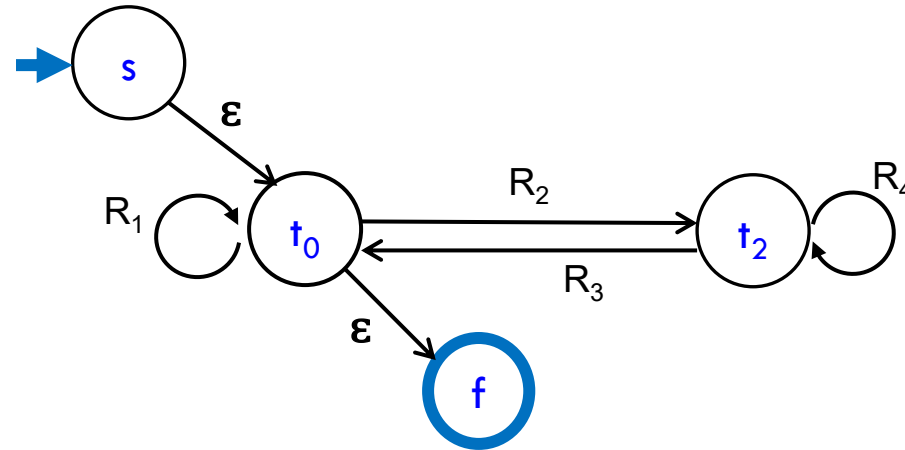
$t_2 \rightarrow t_1 \rightarrow t_0$  :  $20^*2$

$t_2 \rightarrow t_1 \rightarrow t_2$  :  $20^*1$

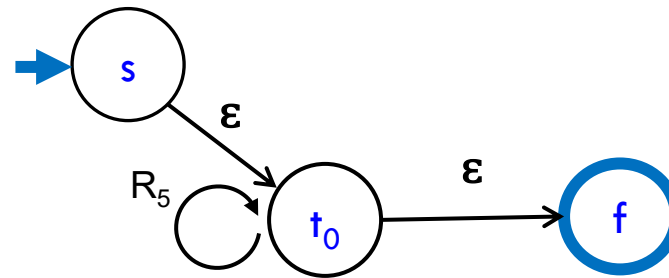


# Splicing out state $t_2$ (and then $t_0$ )

$R_1: 0 \cup 10^*2$   
 $R_2: 2 \cup 10^*1$   
 $R_3: 1 \cup 20^*2$   
 $R_4: 0 \cup 20^*1$



$R_5: R_1 \cup R_2R_4^*R_3$



Final regular expression:  $R_5^* =$

$(0 \cup 10^*2 \cup (2 \cup 10^*1)(0 \cup 20^*1)^*(1 \cup 20^*2))^*$

# Proof [sketch]



# Todo

## Tonight:

- CC 20 is out and due Wednesday at noon
- Get started on HW7 if you haven't already