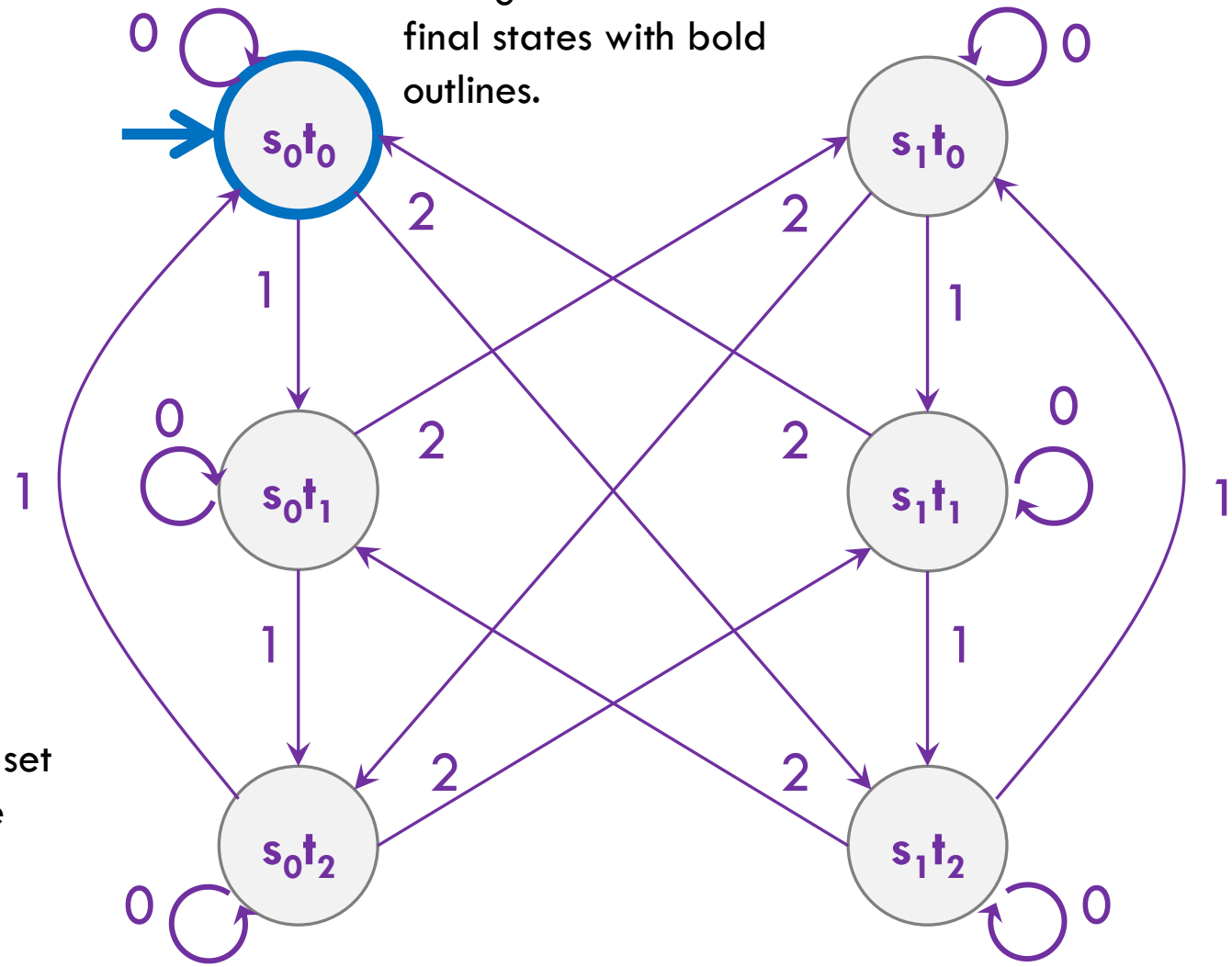
Nondeterministic Finite Automata

CSE 311 Autumn 2025
Lecture 25

Strings over {0,1,2} w/ even number of 2's **and** sum%3=0

Changed notation – final states with bold outlines.



Called the "cross product" construction (because you have a set of states equal to $Q_1 \times Q_2$ where first two DFAs had states Q_1, Q_2.
A very common trick to combine DFAs.

More formally (the "cross product construction")

The original DFAs

States: Q_1, Q_2

Start state: $q_0, {q_0}'$

Transition function: $\delta_1(q, a), \delta_2(q, a)$

Outputs state transitioned to on input a from q

Final States: F_1, F_2

The constructed DFA

States: $Q_1 \times Q_2$

Start state: (q_0, q_0')

Transition function: $\delta((q, q'), a) = (\delta_1(q, a), \delta_2(q', a)$.

Outputs ordered pair of states to transition to.

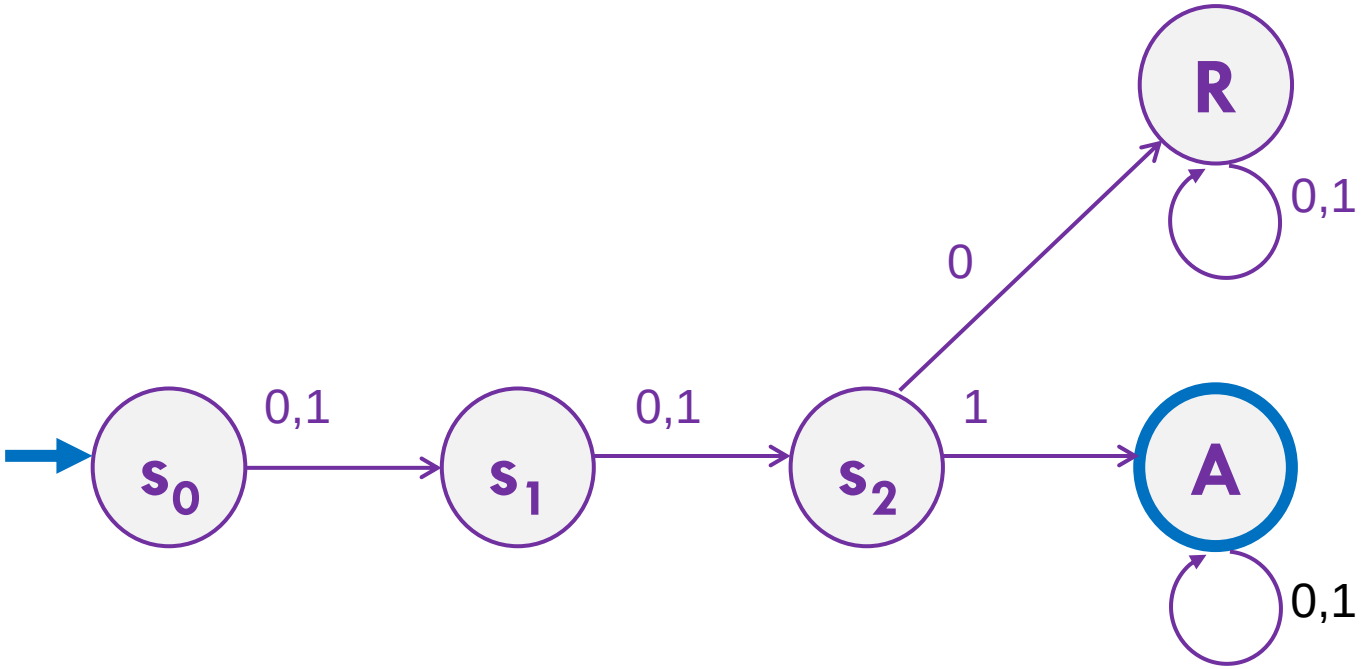
Final States: Varies. E.g.

For strings accepted by both machines, $F_1 \times F_2$

For strings accepted by at least one machine, $F_1 \times Q_2 \cup Q_1 \times F_2$

The set of binary strings with a 1 in the 3rd position from the start (1)

The set of binary strings with a 1 in the 3rd position from the start (2)



The set of binary strings with a 1 in the 3rd position from the end

What do we need to remember?

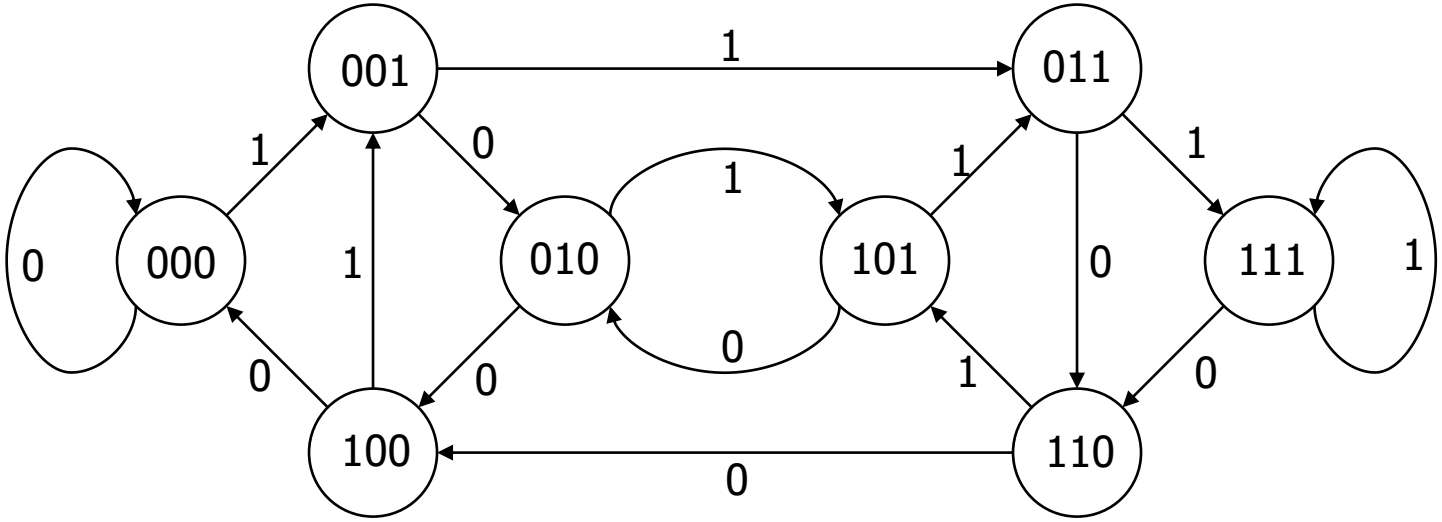
We can't know what string was third from the end until we have read the last character.

So we'll need to keep track of "the character that was 3 ago" in case this was the end of the string.

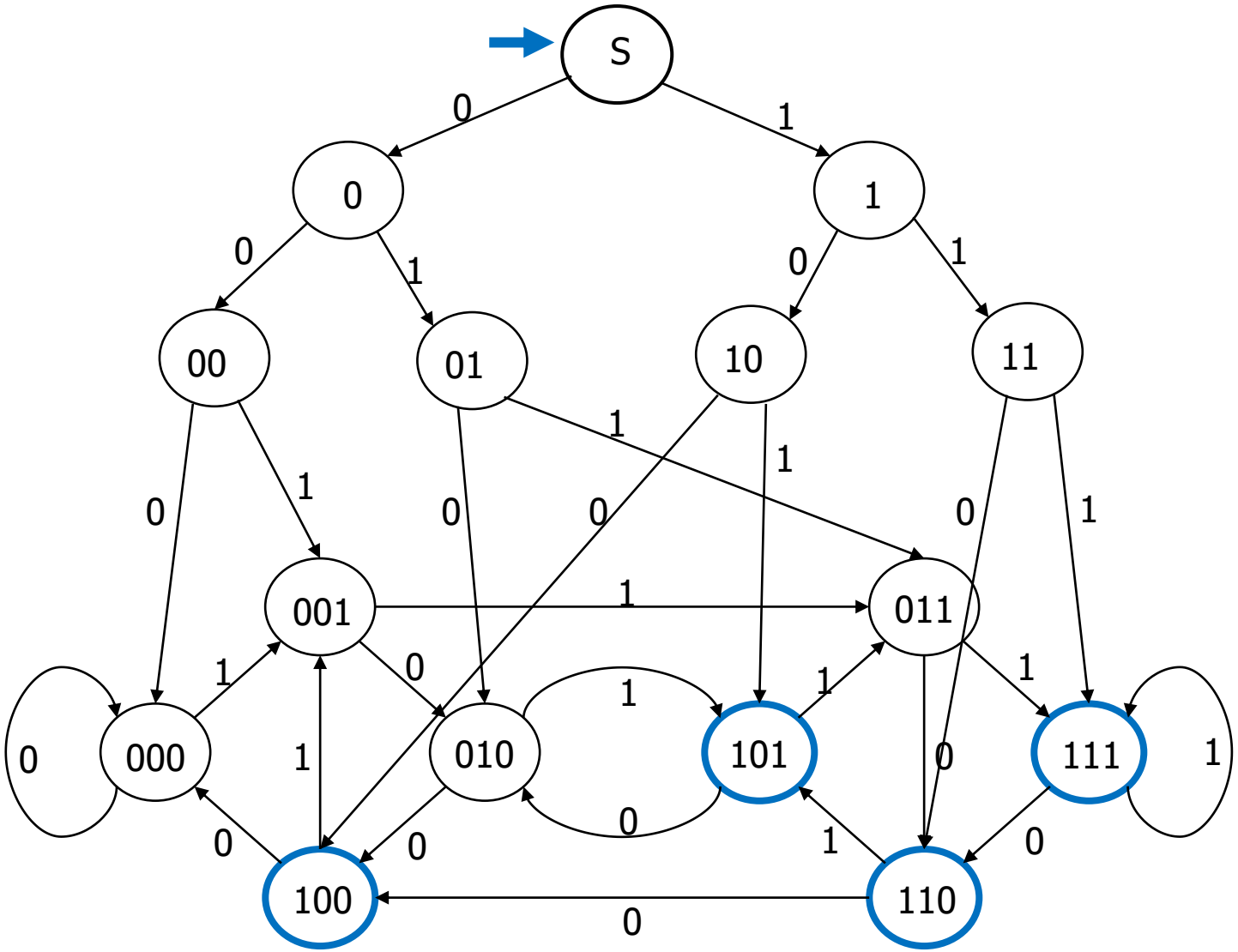
But if it's not…we'll need the character 2 ago, to update what the character 3 ago becomes. Same with the last character.

3 bit shift register

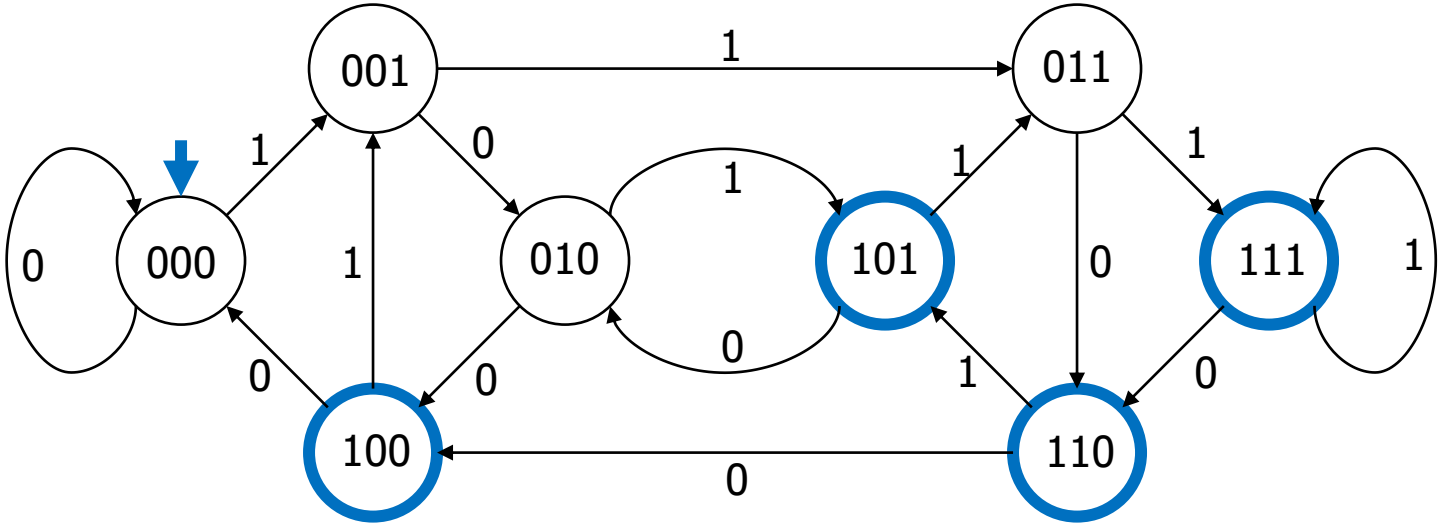
“Remember the last three bits”



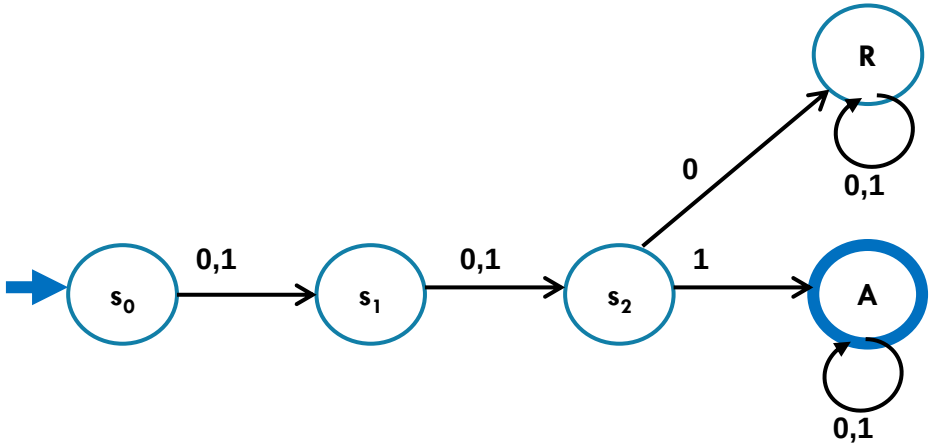
The set of binary strings with a 1 in the 3rd position from the end (solution 1)

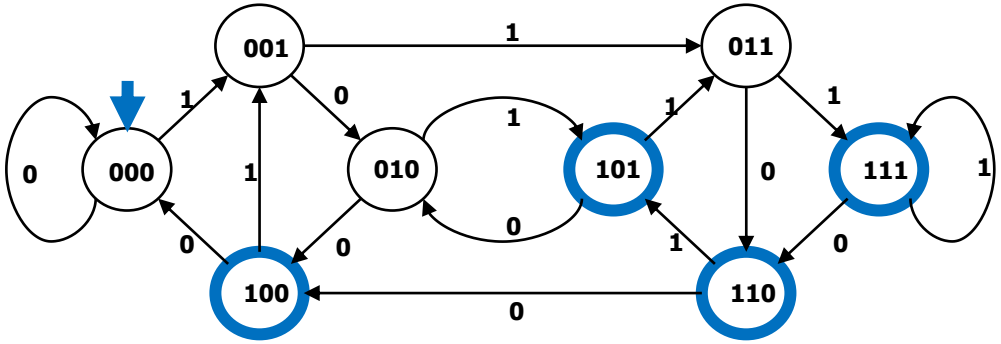


The set of binary strings with a 1 in the 3rd position from the end (solution 2)



The beginning versus the end





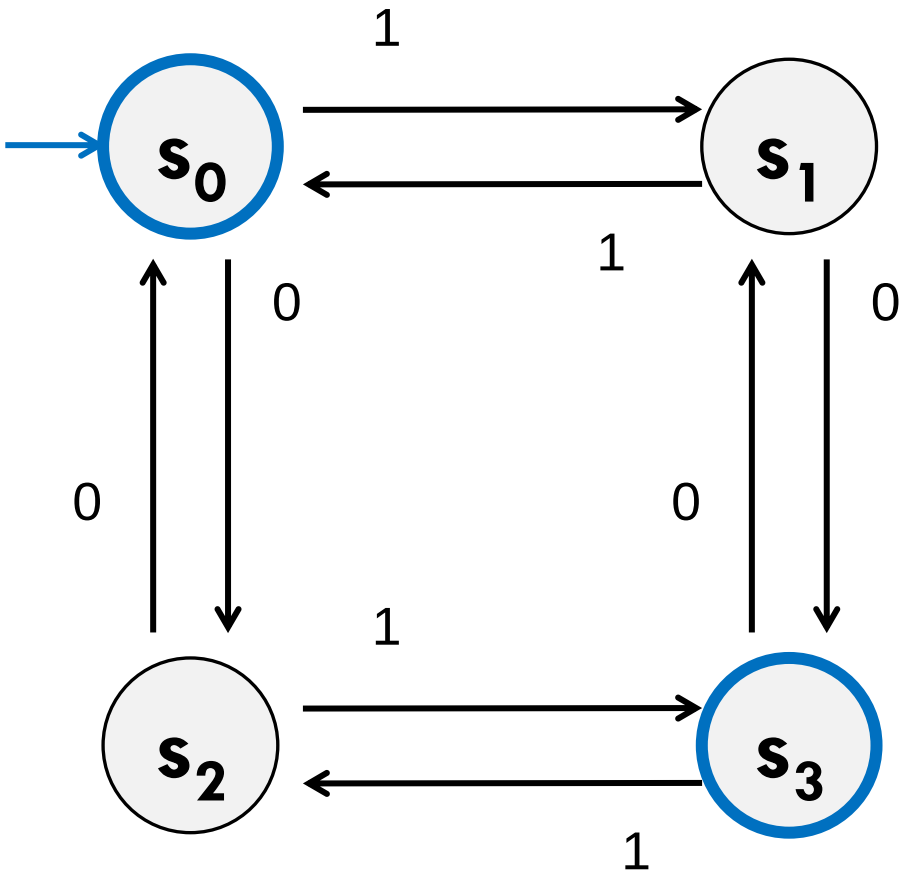
From the beginning was “easier” than “from the end”

At least in the sense that we needed fewer states.

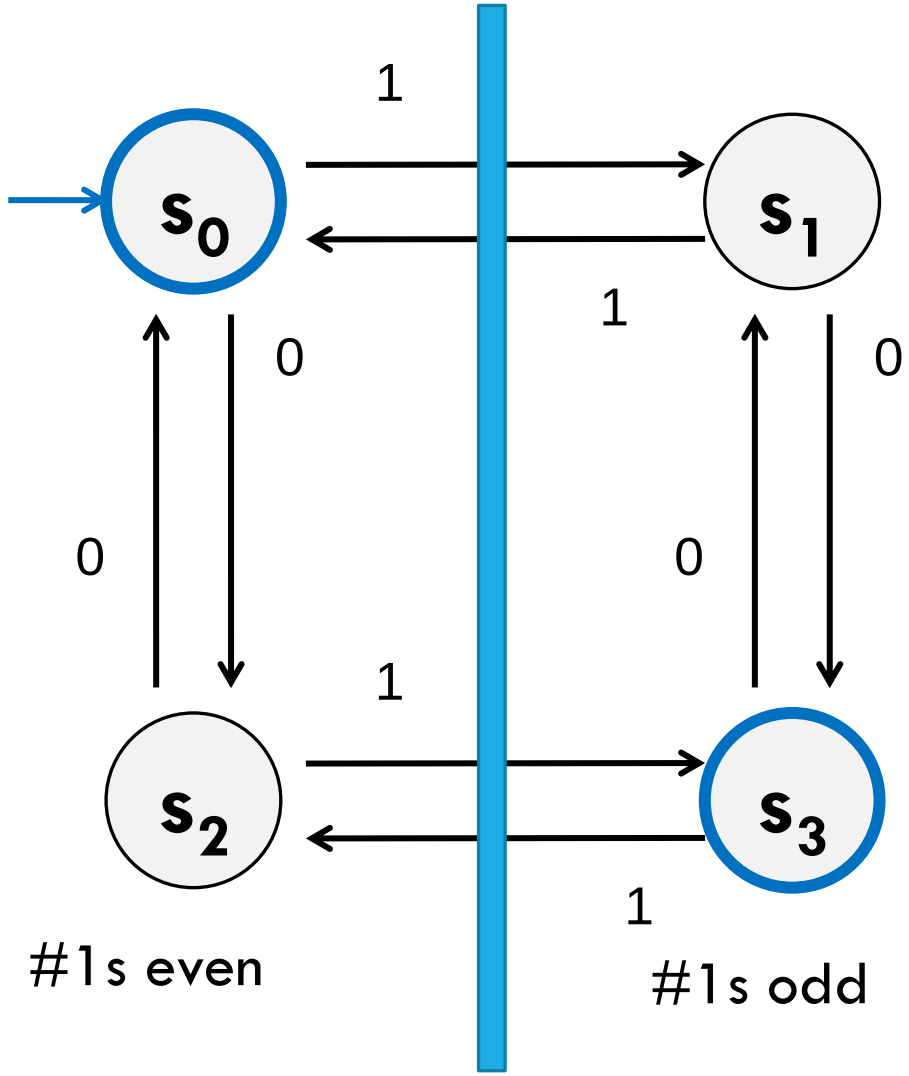
That might be surprising since a java program wouldn’t be much different for those two.

Not being able to access the full input at once limits your abilities somewhat and makes some jobs harder than others.

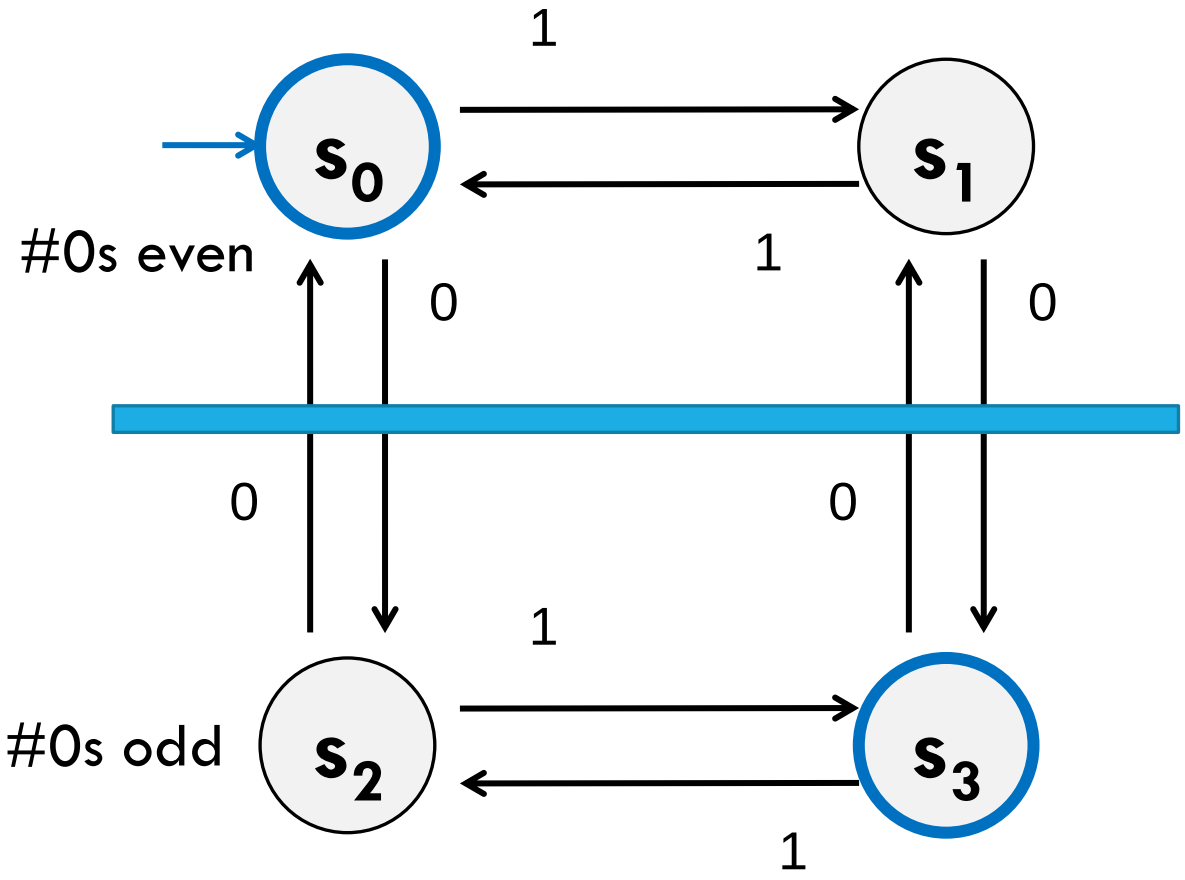
What language does this machine recognize? (1)



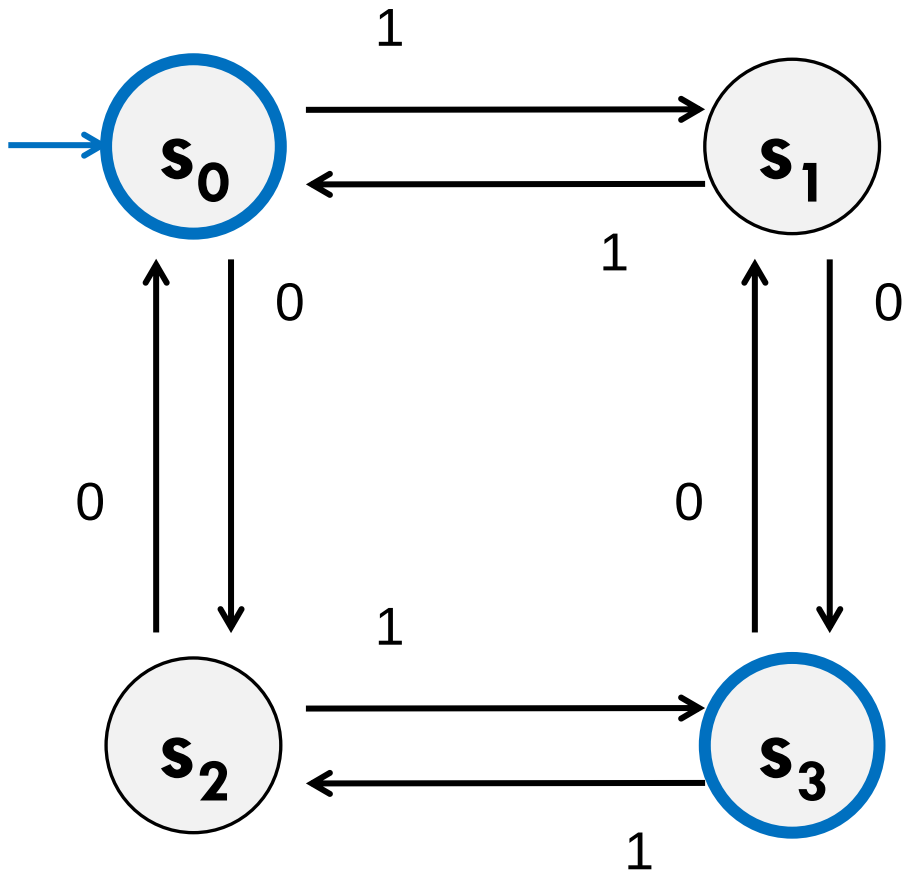
What language does this machine recognize? (2)



What language does this machine recognize? (3)



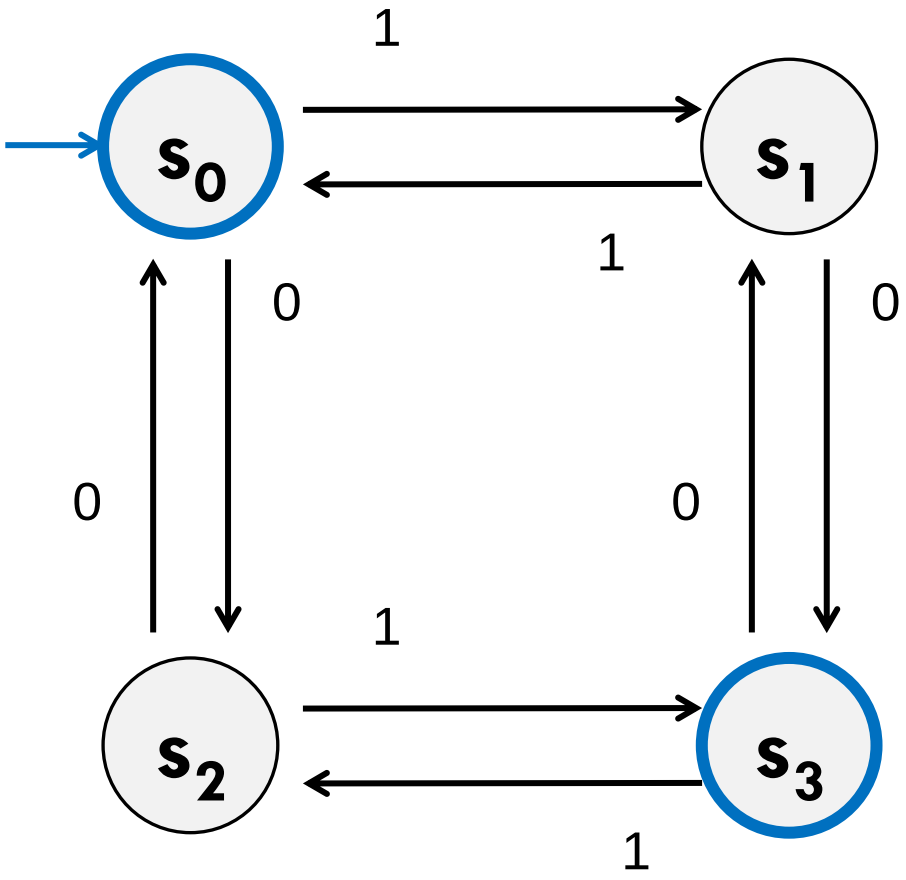
What language does this machine recognize? (4)



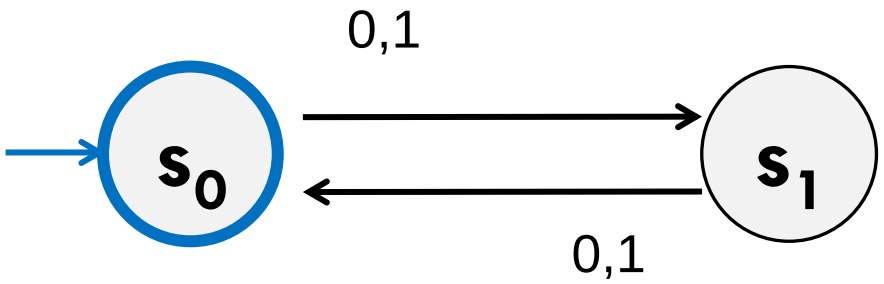
#0s is congruent to #1s (mod 2)

Wait…there's an easier way to describe that….

What language does this machine recognize? (5)



That's all binary strings of even length.



Takeaways

The first DFA might not be the simplest.

Try to think of other descriptions – you might realize you can keep track of fewer things than you thought.

Boy…it'd be nice if we could know that we have the smallest possible DFA for a given language…

DFA Minimization

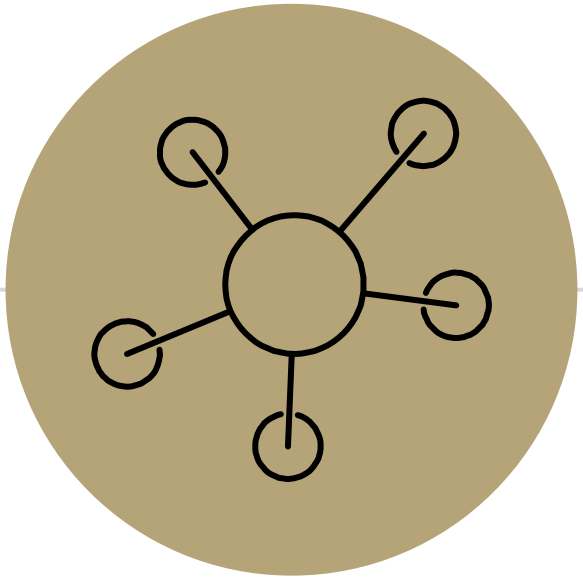
We can know!

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.

Some quarters this covered in detail. But…we ran out of time.
Optional slides – won't be required in HW or final but you might find it useful/interesting for your own learning.



Optional Content: Machines with output

What are FSMs used for? (1)

"Classic" hardware applications:

Anything where you only need to remember a very small amount of information, and have very simple update rules.

Vending machines

Elevators: need to know whether you're going up or down, where people want to go, where people are waiting, and whether you're going up or down. Simple rules to transition.

These days…general hardware is cheap, less likely to use custom hardware. BUT the programmer was probably still thinking about FSMs when writing the code.

What are FSMs used for? (2)

Theoretically – still lots of applications.

`grep` uses FSMs to analyze regular expressions (more on this later).

Useful for modeling situations where you have minimal memory.

Good model for simple AI (say simple NPCs in games).

Technically all of our computers are finite state machines…

But they're not usually how we think about them…more on this next week.

Adding Output to Finite State Machines

So far we have considered finite state machines that just accept/reject strings

called “Deterministic Finite Automata” or DFAs

One can also consider finite state machines that with output

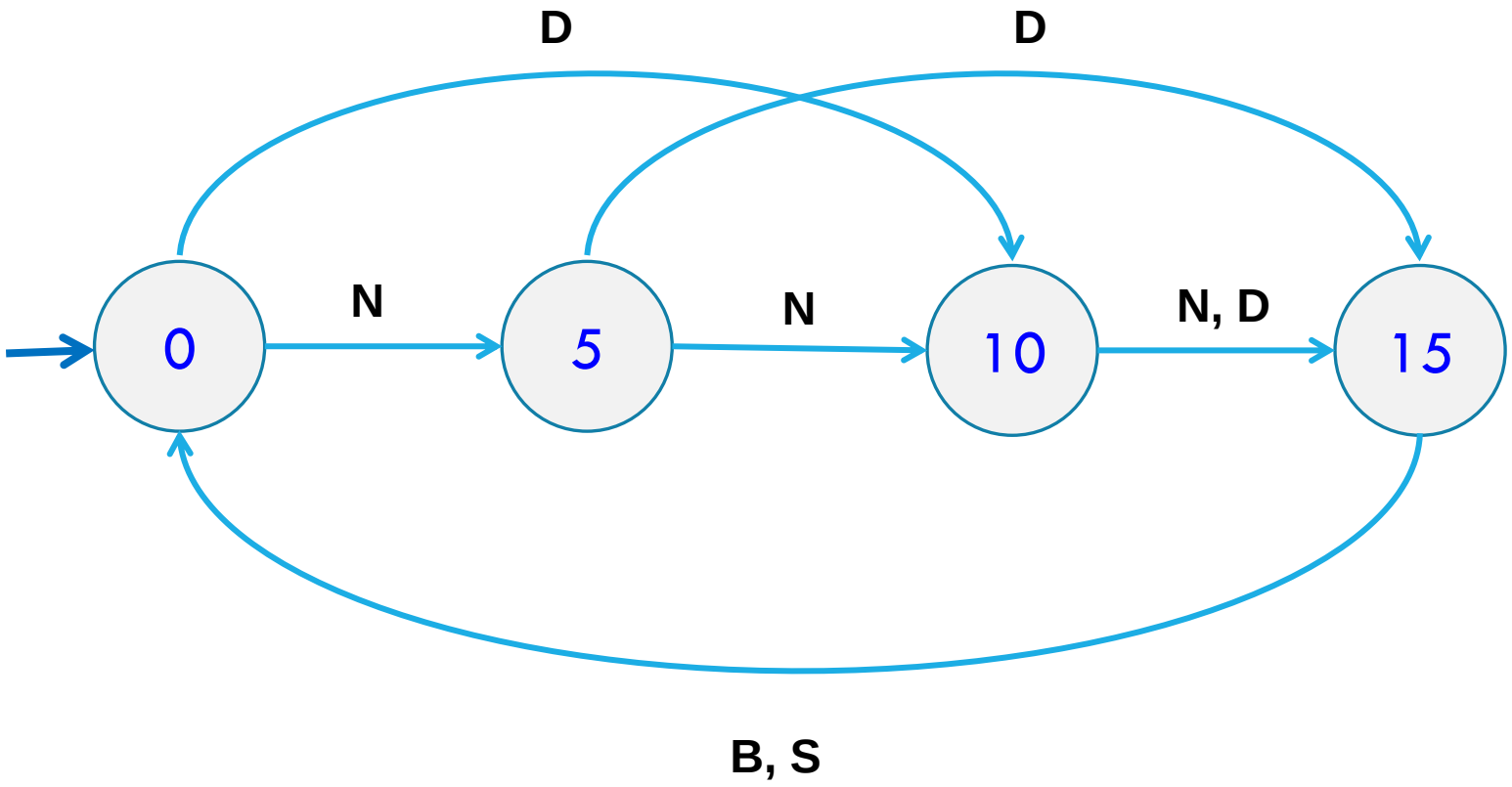
These are often used as controllers

Vending Machine

Enter 15 cents in dimes or nickels
Press S or B for a candy bar

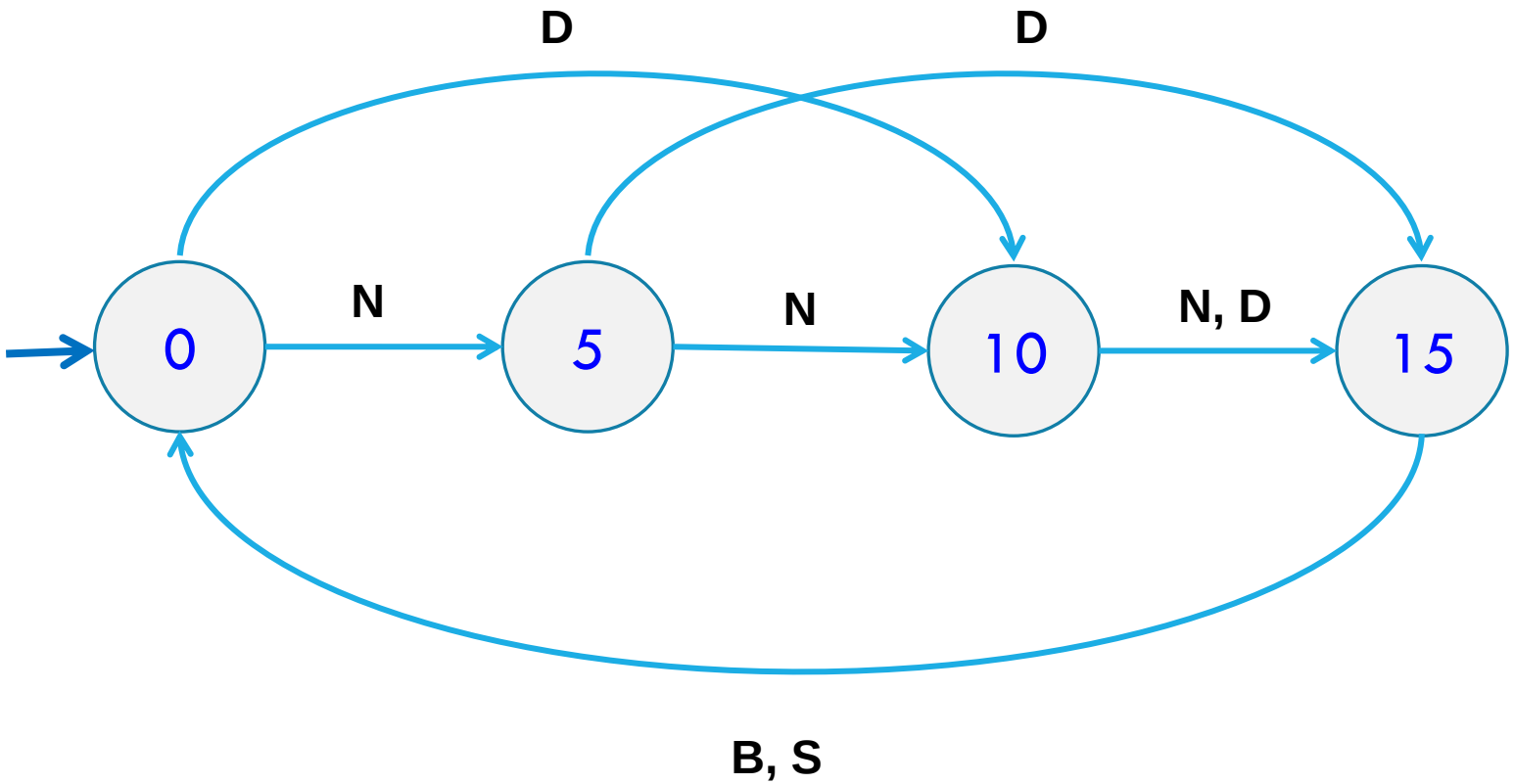
Vending Machine v0.1

Vending Machine, v0.1

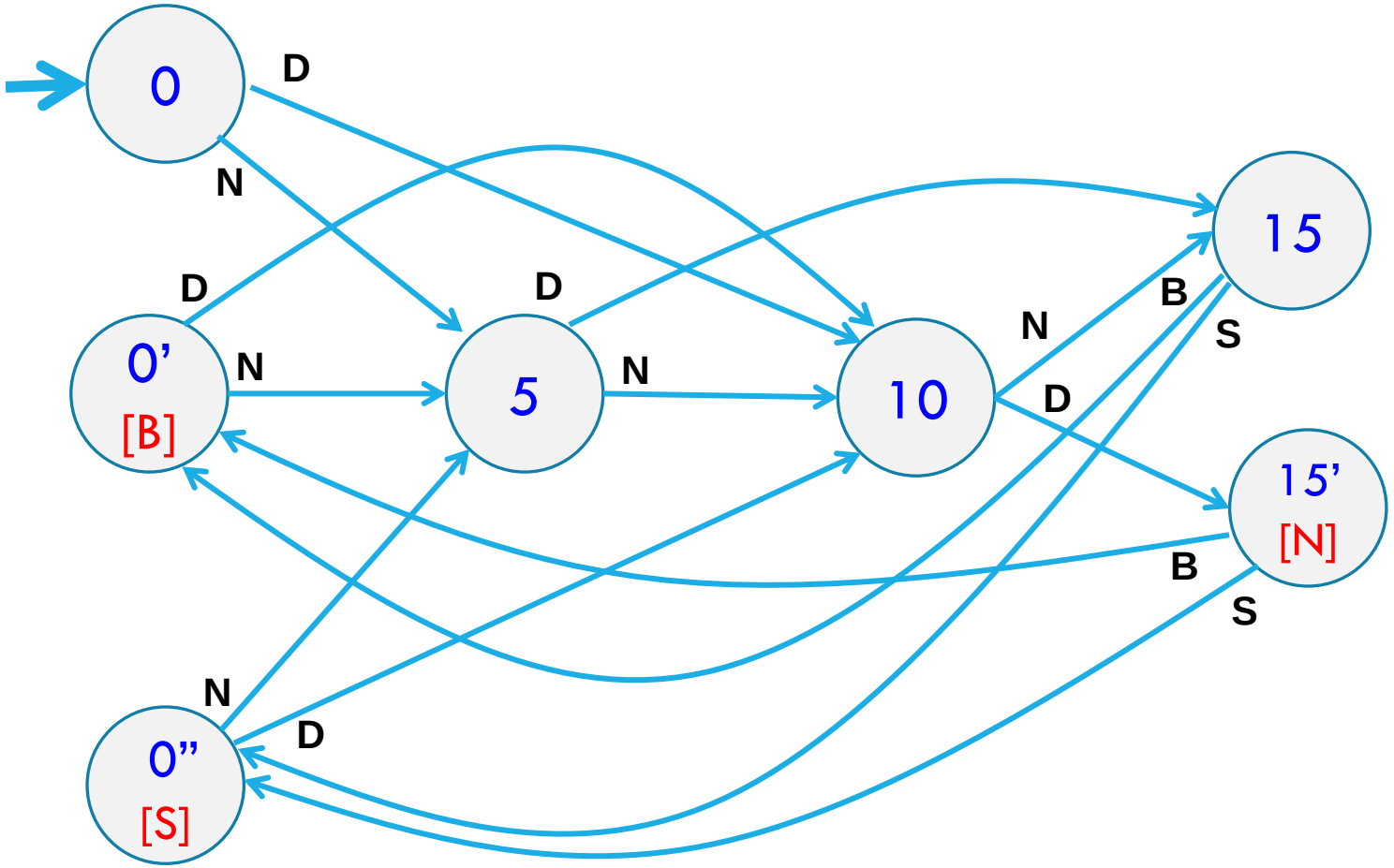


Basic transitions on **N** (nickel), **D** (dime), **B** (butterfinger), **S** (snickers)

Vending Machine v0.2

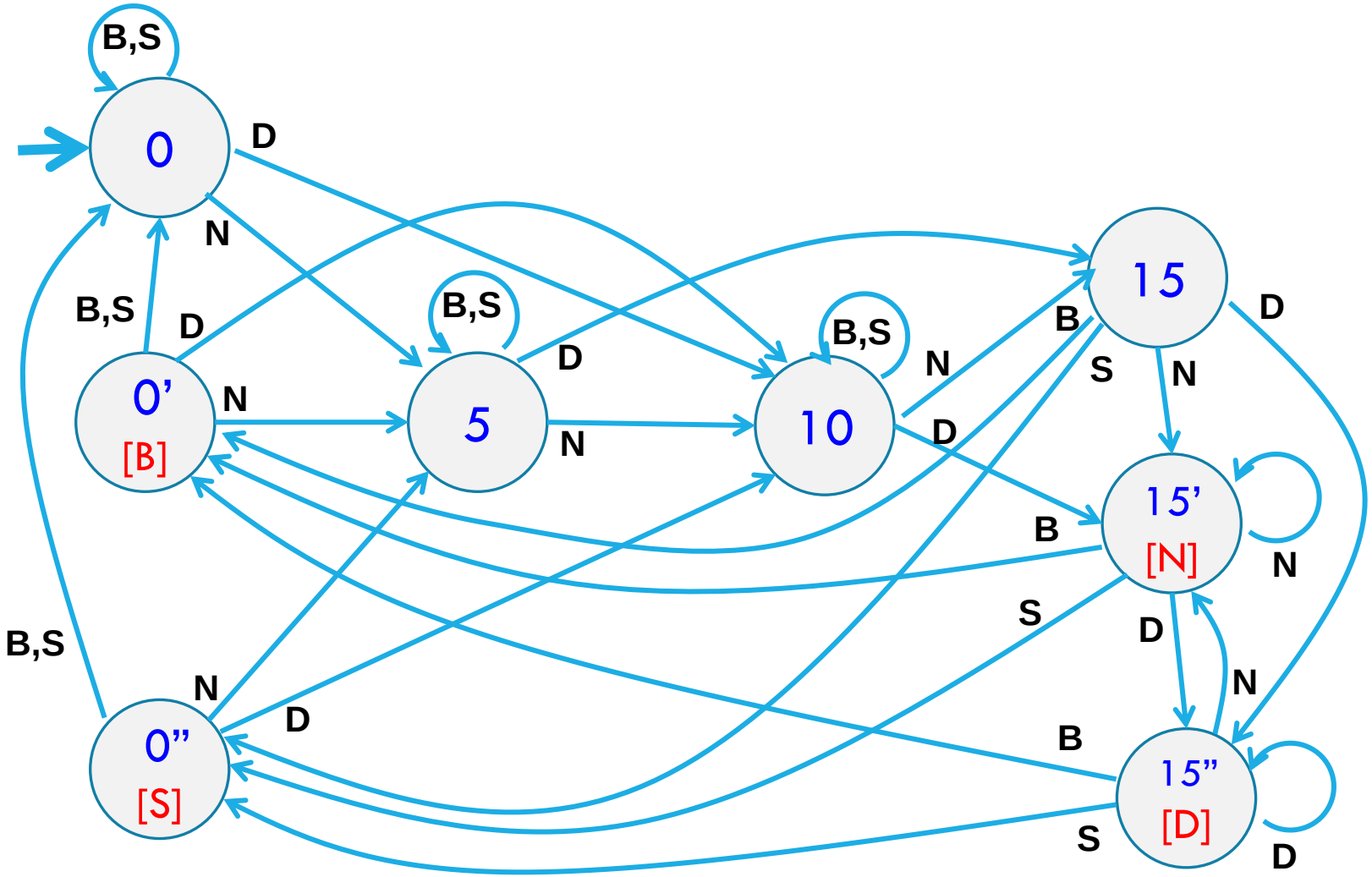


Vending Machine, v0.2

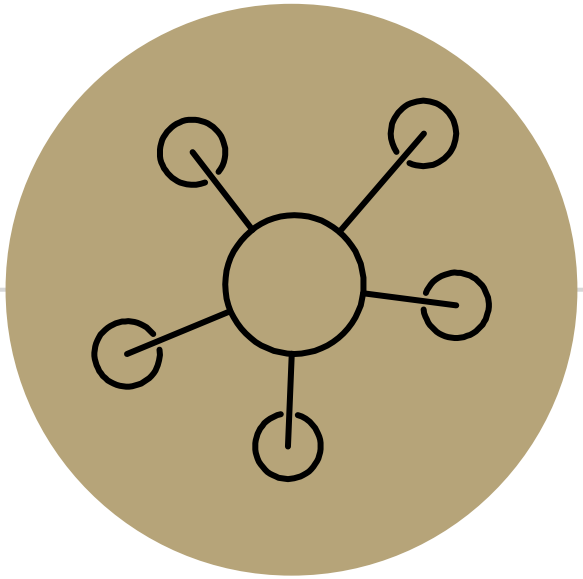


Adding output to states: N – Nickel, S – Snickers, B – Butterfinger

Vending Machine, v1.0



Adding additional “unexpected” transitions to cover all symbols for each state



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 "ε" – 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.

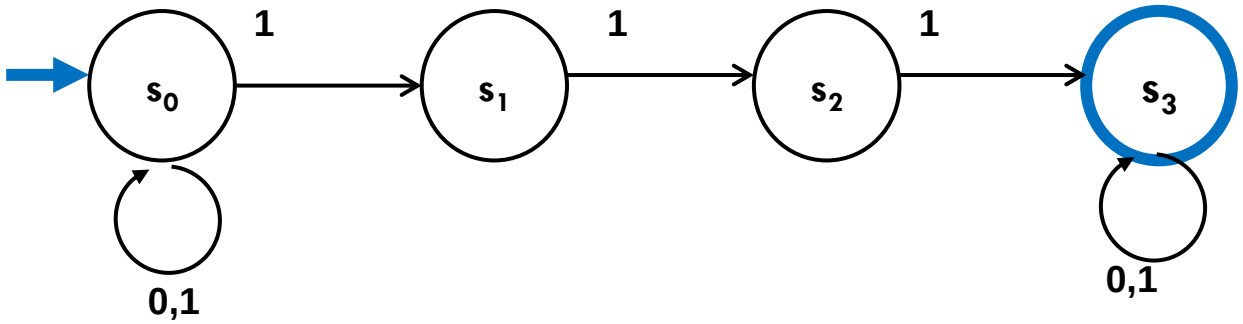
Nondeterministic Finite Automata Intro

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.

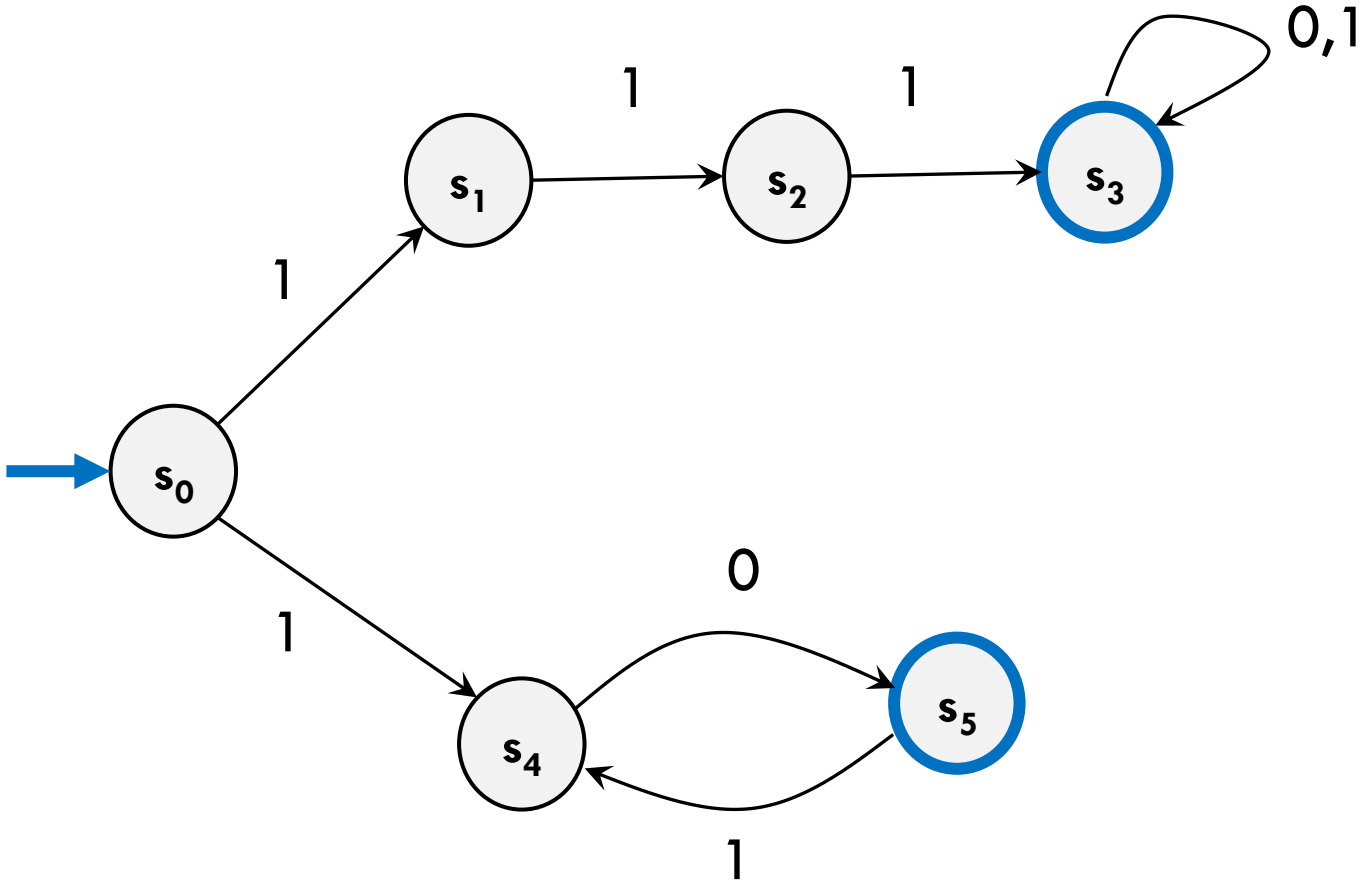
Lets us give simpler descriptions of complicated objects.

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.

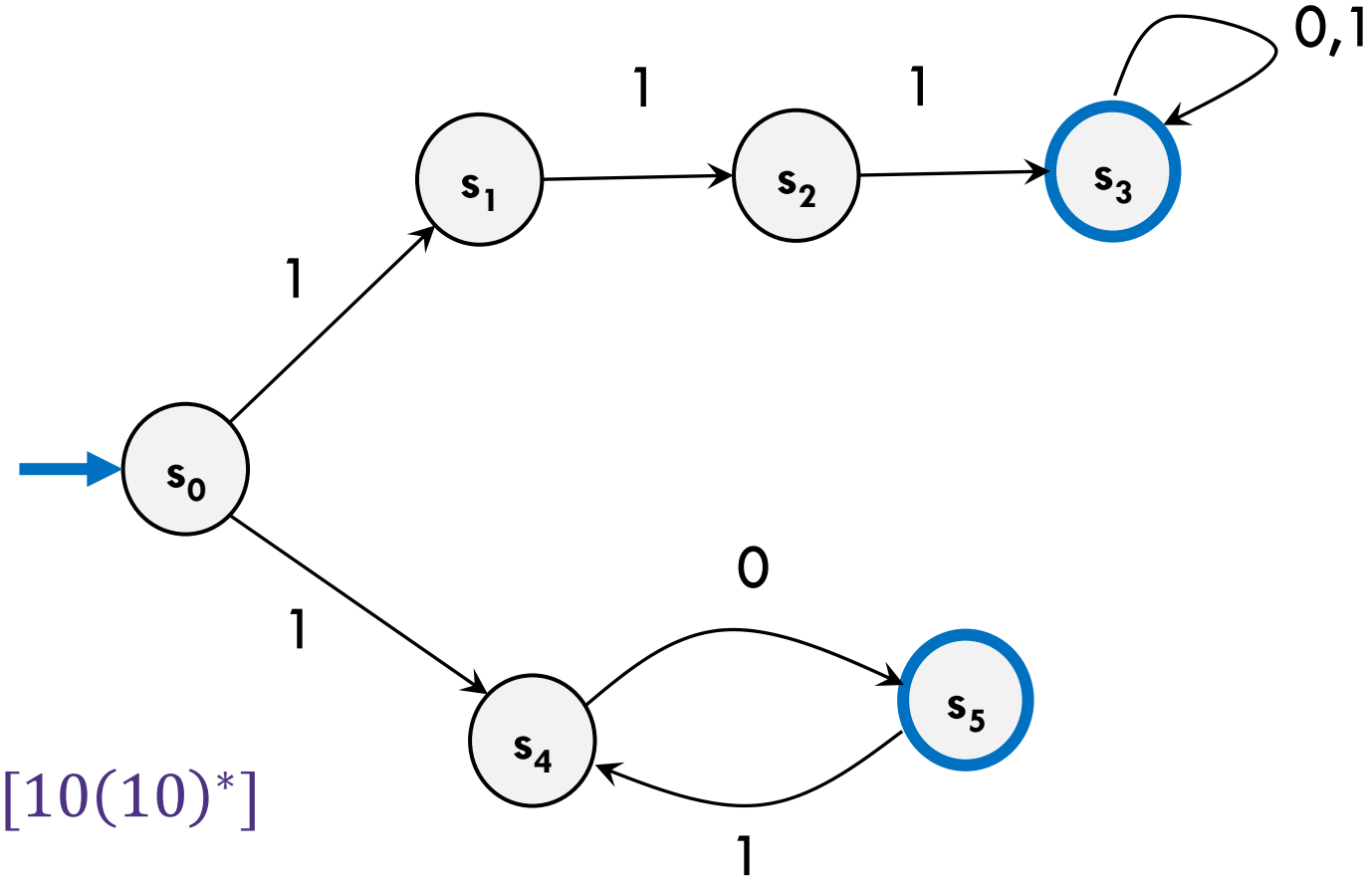
NFA practice

What is the language of this NFA?



NFA practice (solution)

What is the language of this NFA?

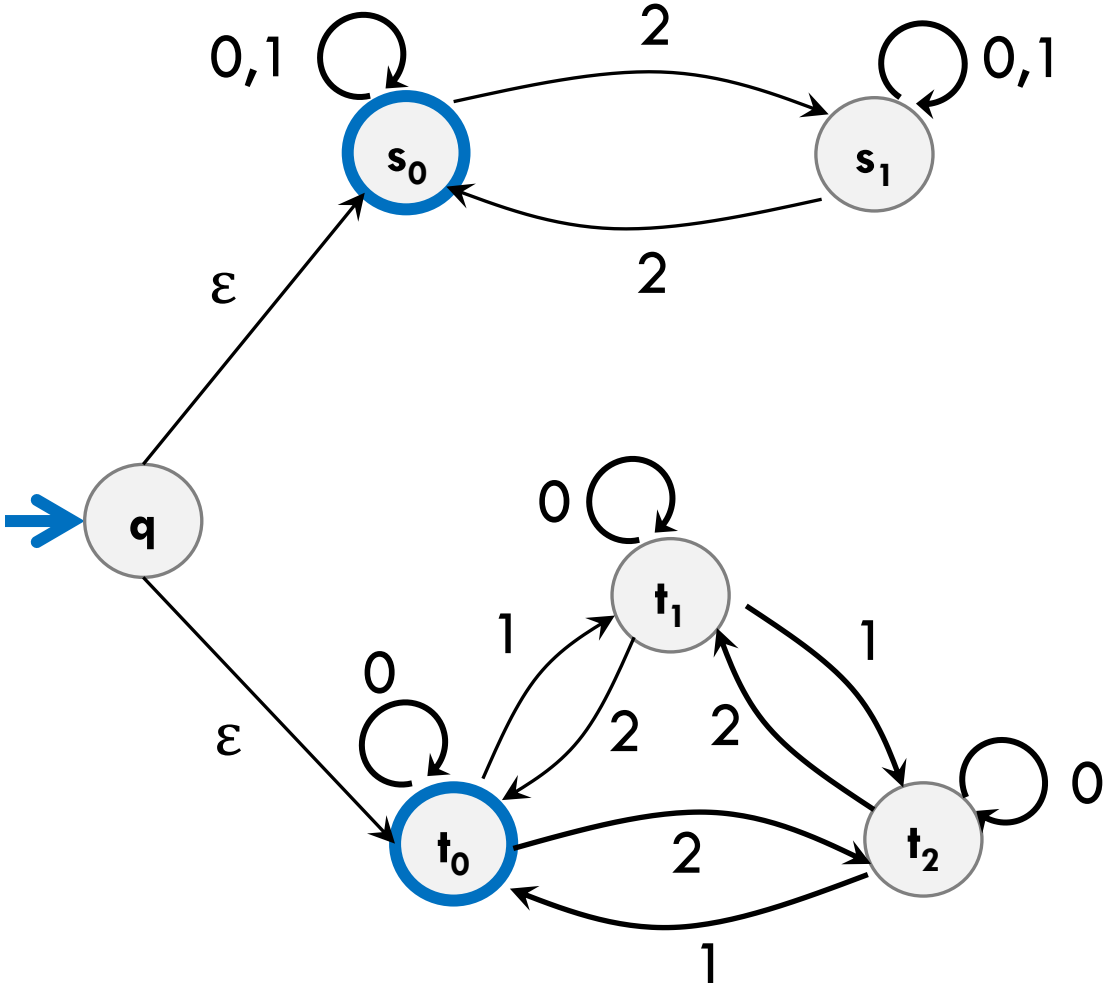


$111(0 \cup 1)^*$

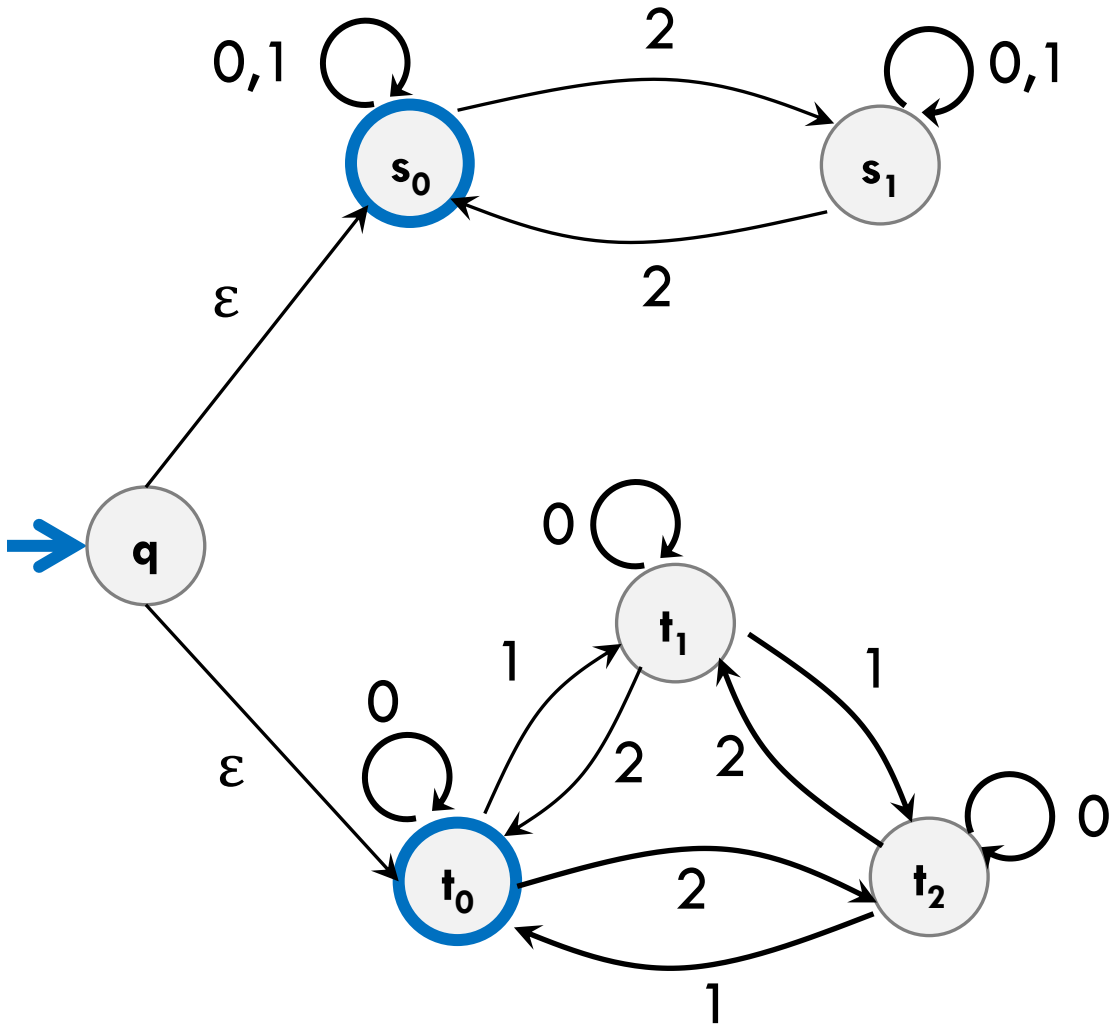
$10(10)^*$

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

What about those ε-transitions?



What about those ε-transitions? (complete)



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)

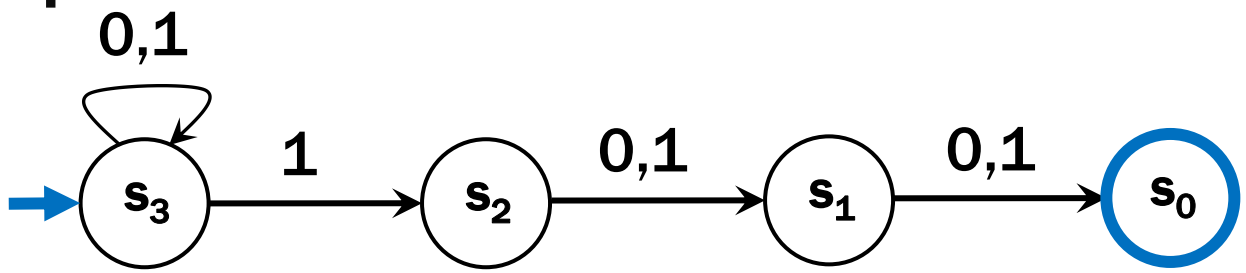
Perfect guesser view makes this easier.

Design an NFA for the language in the title.

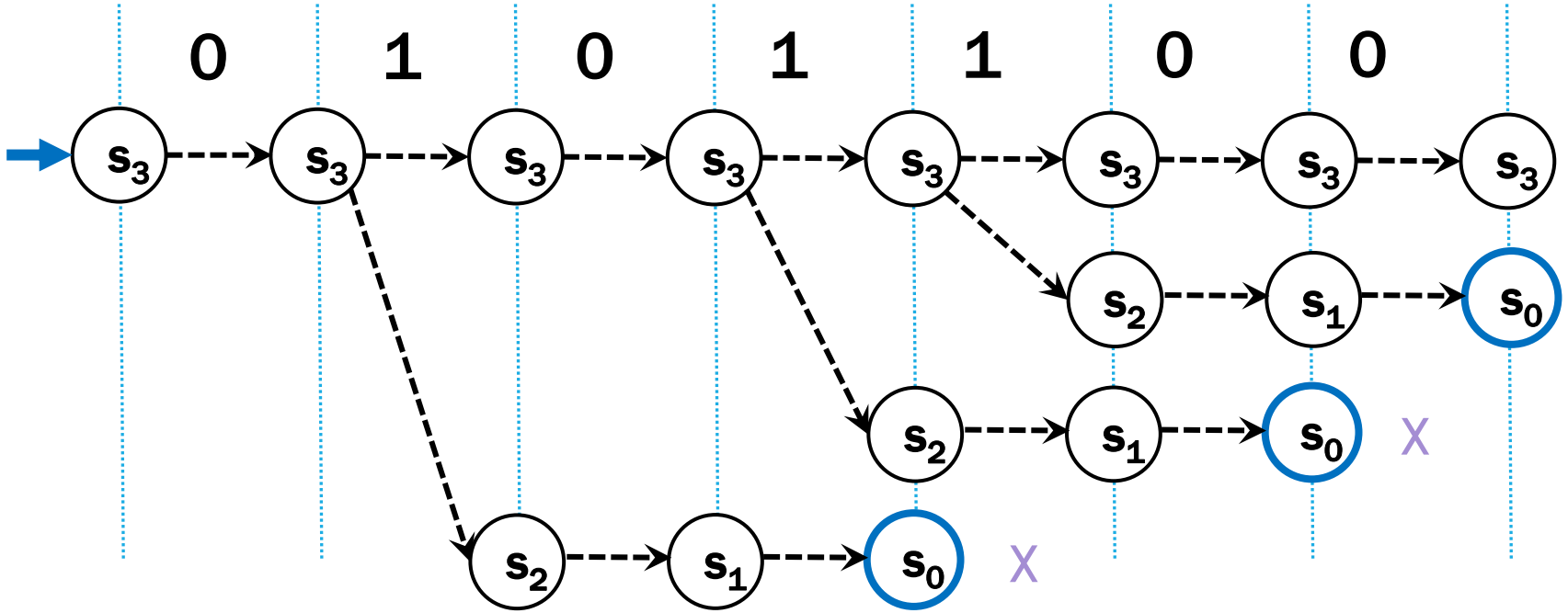
NFA that recognizes “binary strings with a 1 in the third position from the end” (solution)

That’s WAY easier than the DFA…

Parallel Exploration view of an NFA



Input string 0101100



Regularity Idea

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?

pollev.com/robbie

Regularity (definition)

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.

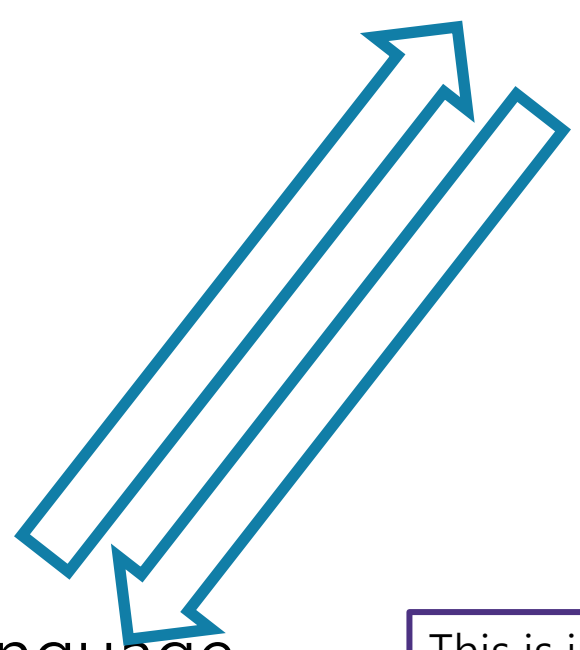
A set of strings that is recognized by a DFA (equivalently, recognized by an NFA; equivalently, the language of a regular expression) is called a **regular language**.

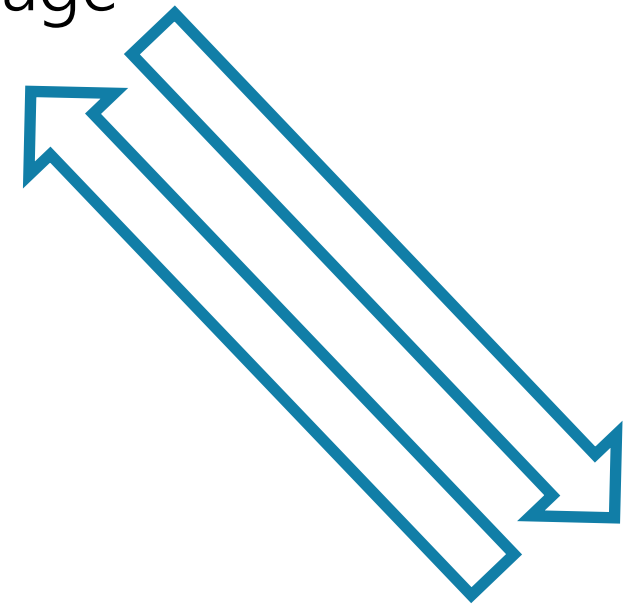
So to show a language is “regular” you just need to show one of these and prove it works. There are some “irregular” languages (that don’t have a corresponding NFA/DFA/regex).

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] (1)

L is the language of an NFA.





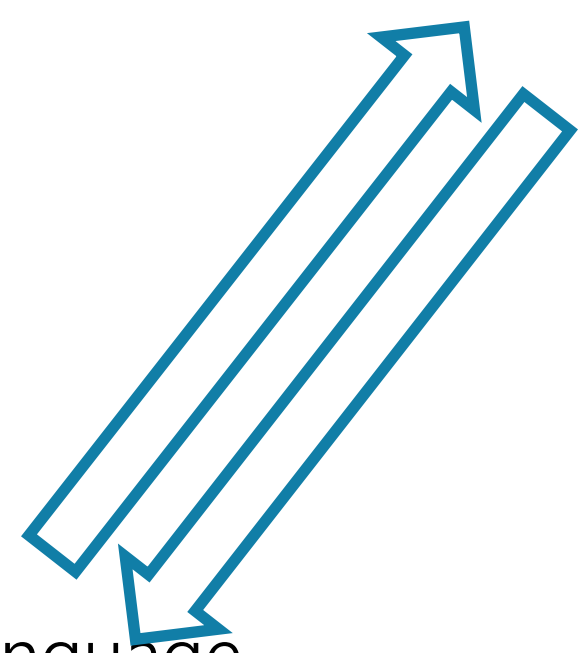
L is the language of a regular expression.

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.

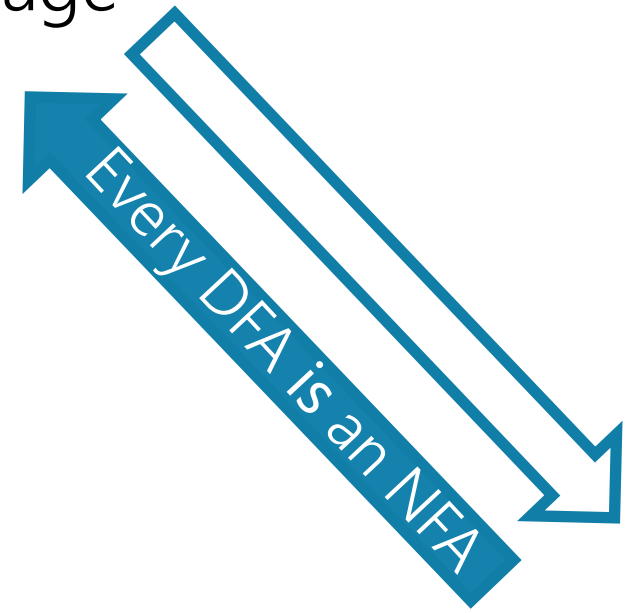
L is the language of a DFA.

Proof [sketch] (2: every DFA is an NFA)

L is the language of an NFA.



L is the language of a regular expression.

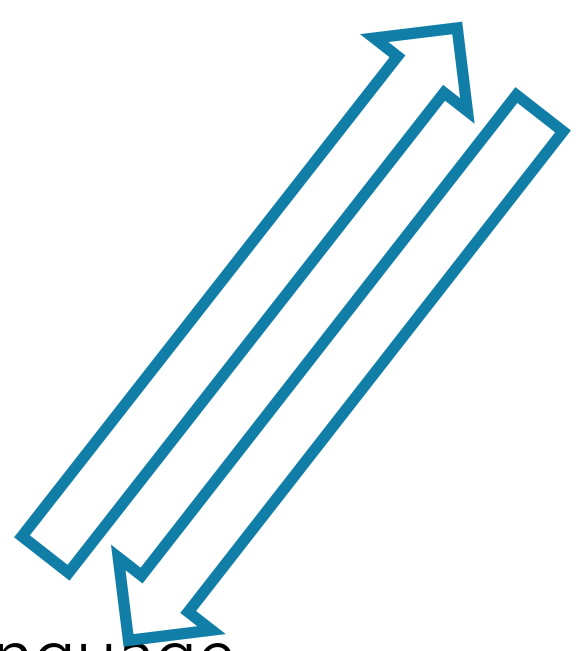


L is the language of a DFA.

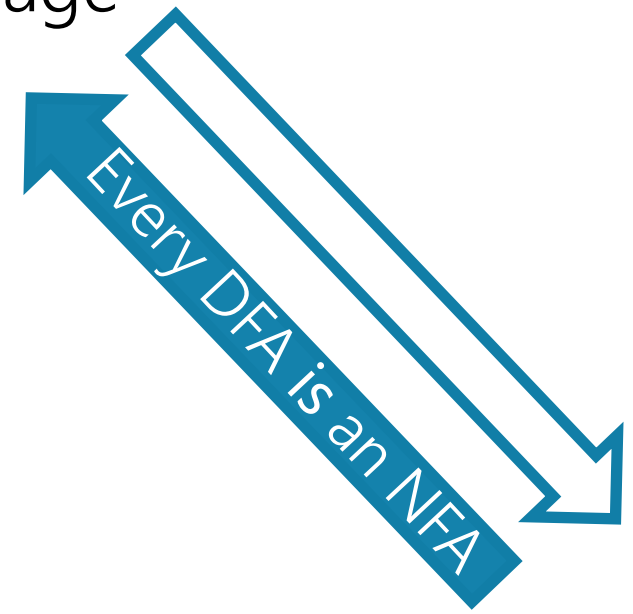
Suppose L is the language of some DFA M. M also satisfies the requirements for an NFA, so L is also the language of an NFA.

Proof [sketch] (3)

L is the language of an NFA.



L is the language of a regular expression.



L is the language of a DFA.

Can we convert an NFA to a DFA?

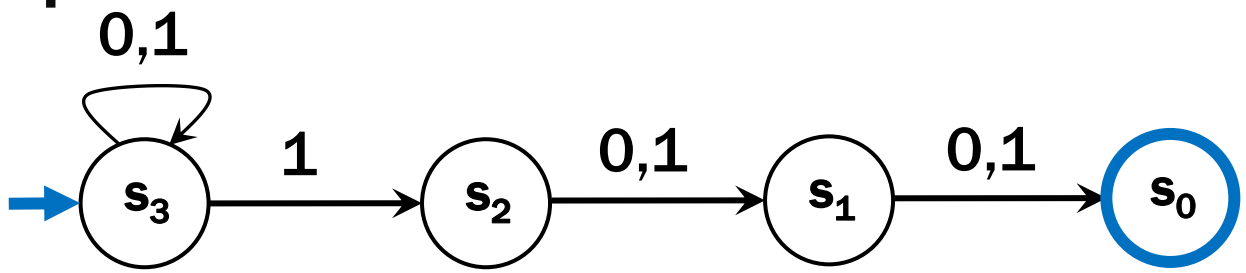
NFAs are magic though! DFAs can't guess…

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

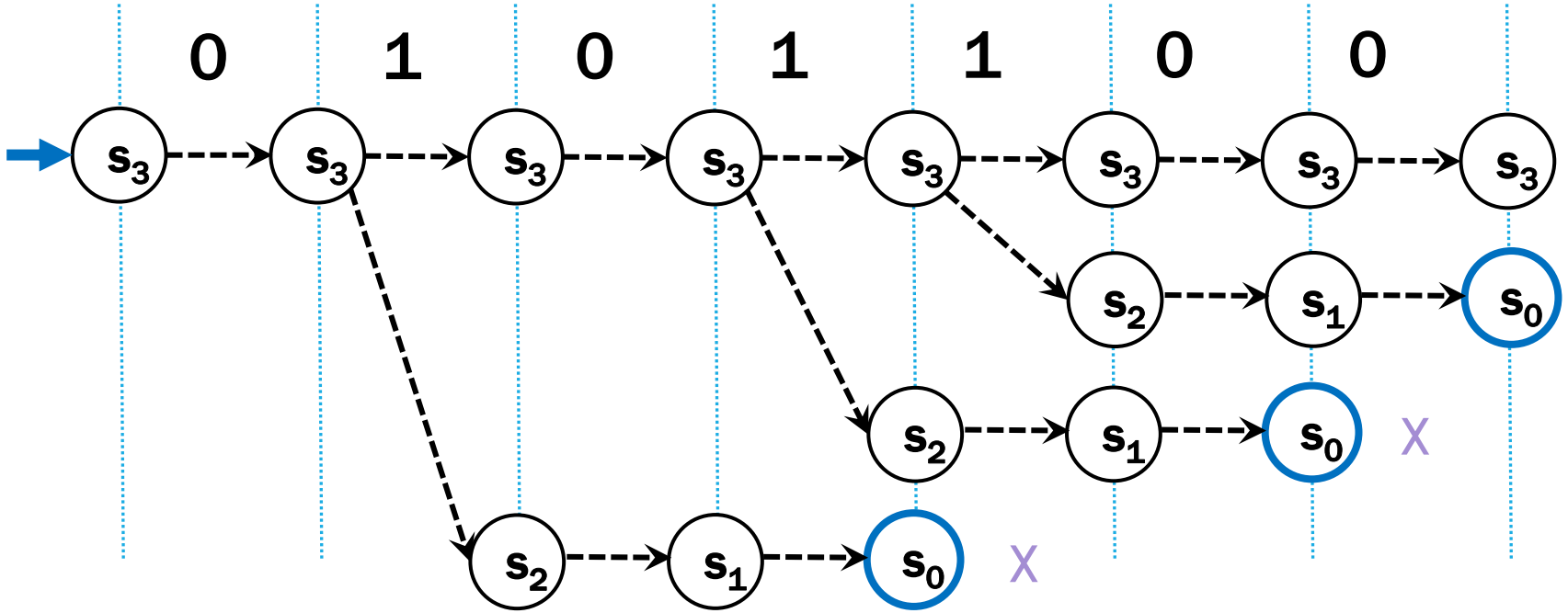
At any step, the set of all possible states we could be in is fixed!

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

Parallel Exploration view of an NFA (again)



Input string 0101100



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.

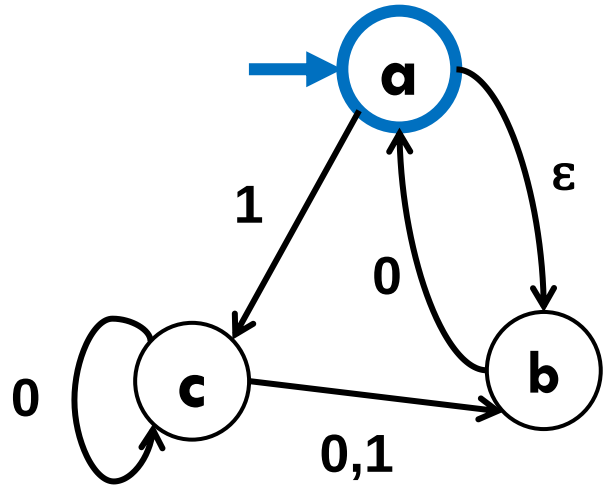
Let D be a DFA with set of states $\mathcal{P}(S)$.

How do we update?

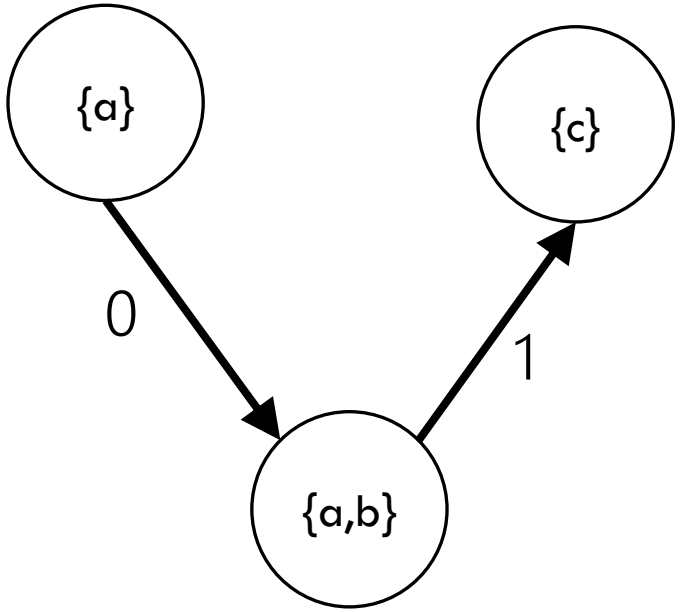
If I'm in a set of states X, if the next character to be read is a

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

An example (starting point)



N



D

Finishing the DFA

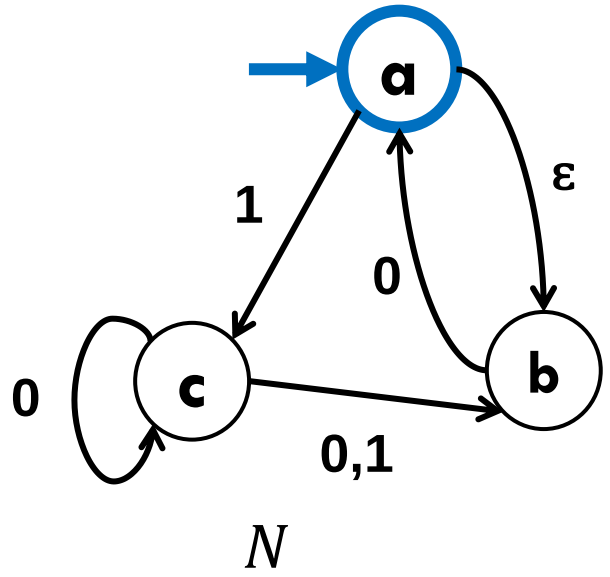
What about start and accept states?

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

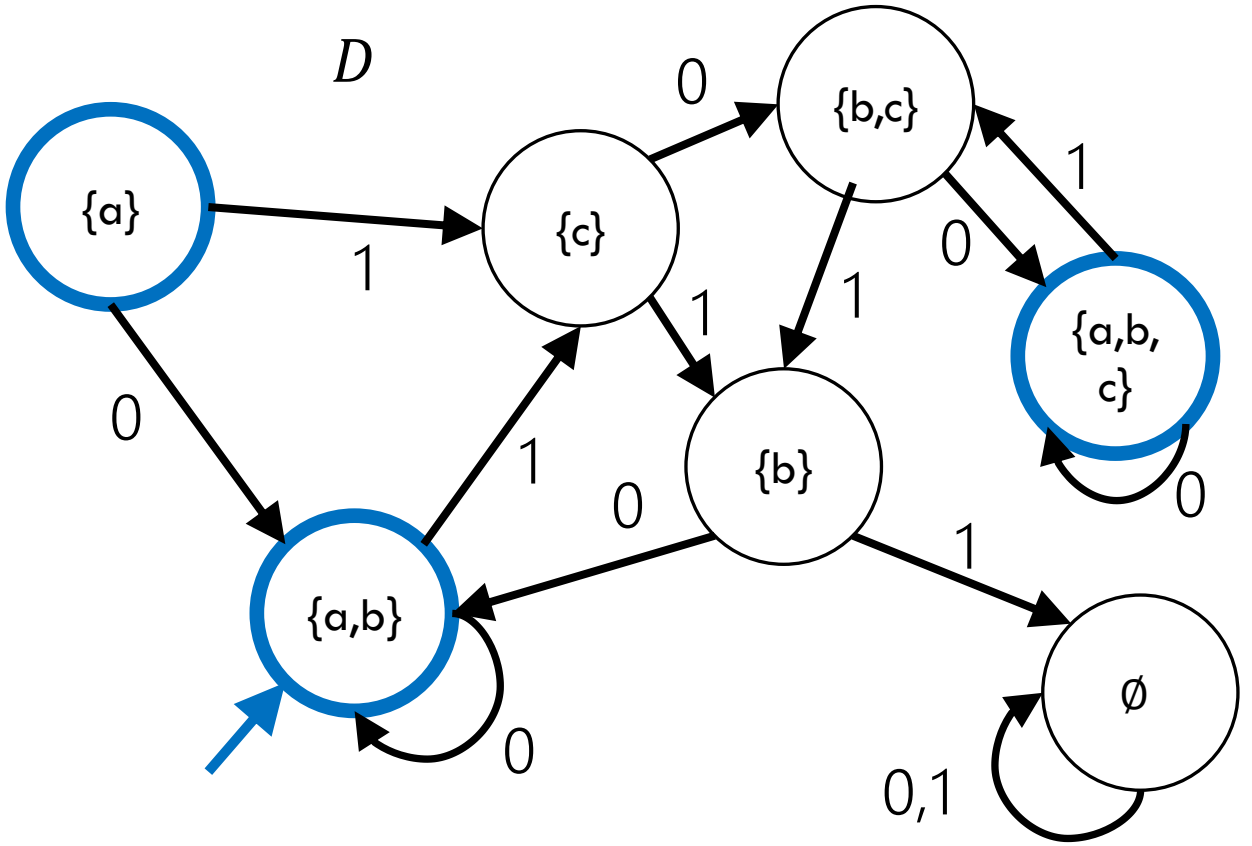
i.e. the states the NFA could be in before reading a character of the input.

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



N



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 ε-transitions)

Final States: F

The constructed DFA

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

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

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

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

Proof [sketch] (4: powerset construction)

L is the language of an NFA.

L is the language of a regular expression.

L is the language of a DFA.