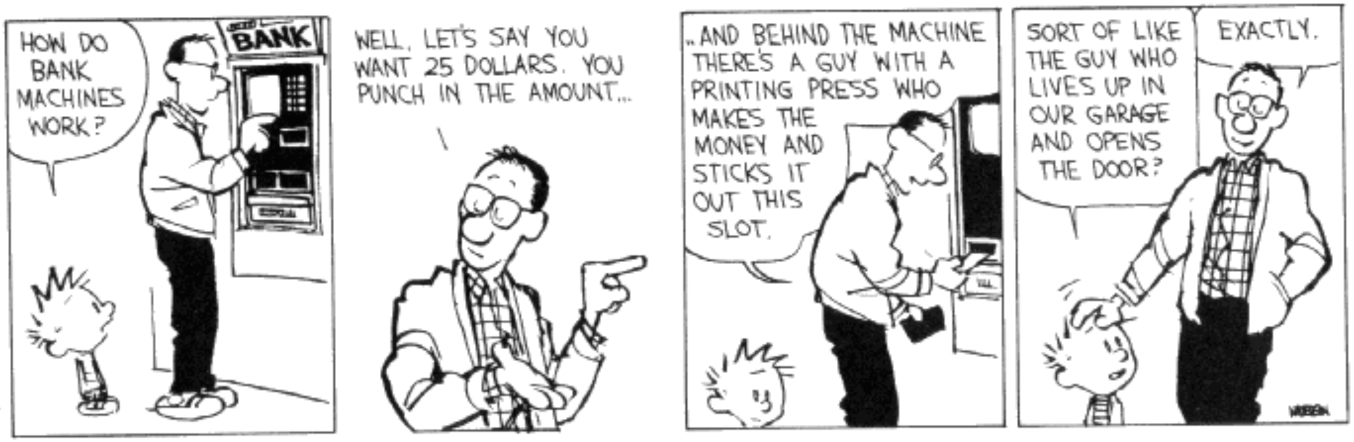
CSE 311: Foundations of Computing

Topic 10: Finite State Machines



Last time: Languages — REs and CFGs

Saw two new ways of defining languages

- Regular Expressions $(0 \cup 1)^*\,0110\,(0 \cup 1)^*$
  - easy to understand (declarative)
- Context-free Grammars $S \rightarrow SS \mid 0S1 \mid 1S0 \mid \varepsilon$
  - more expressive
  - (≈ recursively-defined sets)

We will connect these to machines shortly.

But first, we need some new math terminology….

Alternative Set Notation

We defined Cartesian Product as

$$A \times B ::= \{x : \exists a \in A, \exists b \in B \ (x = (a, b)) \}$$

Alternative notation for this is

$$A \times B ::= \{(a, b) : a \in A, b \in B\}$$

"The set of all (a, b) such that a ∈ A and b ∈ B"

Relations

Let A and B be sets,
A **binary relation from** A **to** B is a subset of A × B

Let A be a set,
A **binary relation on** A is a subset of A × A

Relations You Already Know

$\geq$ on $\mathbb{N}$

That is: $\{(x,y) : x \geq y \text{ and } x, y \in \mathbb{N}\}$

$<$ on $\mathbb{R}$

That is: $\{(x,y) : x < y \text{ and } x, y \in \mathbb{R}\}$

$=$ on Σ^*

That is: $\{(x,y) : x = y \text{ and } x, y \in \Sigma^*\}$

$\subseteq$ on $\mathcal{P}(U)$ for universe U

That is: $\{(A,B) : A \subseteq B \text{ and } A, B \in \mathcal{P}(U)\}$

More Relation Examples

R_1 = {(a, 1), (a, 2), (b, 1), (b, 3), (c, 3)}

R_2 = {(x, y) : $x \equiv_5 y$ }

R_3 = {(c_1, c_2) : c_1 is a prerequisite of c_2 }

R_4 = {(s, c) : student s has taken course c }

Properties of Relations

Let R be a relation on A.

R is **reflexive** iff $(a,a) \in R$ for every $a \in A$

R is **symmetric** iff $(a,b) \in R$ implies $(b,a) \in R$

R is **antisymmetric** iff $(a,b) \in R$ and $a \neq b$ implies $(b,a) \notin R$

R is **transitive** iff $(a,b) \in R$ and $(b,c) \in R$ implies $(a,c) \in R$

Which relations have which properties?

$\geq$ on $\mathbb{N}$:

$<$ on $\mathbb{R}$:

$=$ on $\sum^*$:

$\subseteq$ on $\mathcal{P}$(U):

R_2 = $\{(x, y) : x \equiv_5 y\}$:

R_3 = $\{(c_1, c_2) : c_1$ is a prerequisite of $c_2\}$:

R is **reflexive** iff $(a,a) \in R$ for every $a \in A$
R is **symmetric** iff $(a,b) \in R$ implies $(b, a) \in R$
R is **antisymmetric** iff $(a,b) \in R$ and $a \neq b$ implies $(b,a) \notin R$
R is **transitive** iff $(a,b) \in R$ and $(b, c) \in R$ implies $(a, c) \in R$

Which relations have which properties?

$\geq$ on $\mathbb{N}$: Reflexive, Antisymmetric, Transitive

$<$ on $\mathbb{R}$: Antisymmetric, Transitive

$=$ on Σ^* : Reflexive, Symmetric, Antisymmetric, Transitive

$\subseteq$ on $\mathcal{P}(U)$: Reflexive, Antisymmetric, Transitive

$R_2 = \{(x, y) : x \equiv_5 y\}$: Reflexive, Symmetric, Transitive

$R_3 = \{(c_1, c_2) : c_1$ is a prerequisite of $c_2\}$: Antisymmetric

R is **reflexive** iff $(a,a) \in R$ for every $a \in A$
R is **symmetric** iff $(a,b) \in R$ implies $(b, a) \in R$
R is **antisymmetric** iff $(a,b) \in R$ and $a \neq b$ implies $(b,a) \notin R$
R is **transitive** iff $(a,b) \in R$ and $(b, c) \in R$ implies $(a, c) \in R$

Combining Relations

Let R be a relation from A to B.
Let S be a relation from B to C.

The composition of R and S, $R \circ S$ is the relation from A to C defined by:

$$R \circ S = \{(a, c) : \exists\ b \text{ such that } (a, b) \in R \text{ and } (b, c) \in S\}$$

Intuitively, a pair is in the composition if there is a “connection” from the first to the second.

Examples

(a,b) ∈ Parent iff b is a parent of a

(a,b) ∈ Sister iff b is a sister of a

When is (x,y) ∈ Parent ∘ Sister?

When is (x,y) ∈ Sister ∘ Parent?

R ∘ S = {(a, c) : ∃ b such that (a,b)∈ R and (b,c)∈ S}

Examples

Using only the relations Parent, Child, Father, Son, Brother, Sibling, Husband

and *composition*, express the following:

Uncle: b is an uncle of a

Cousin: b is a cousin of a

Powers of a Relation

$$\begin{aligned} R^2 &::= R \circ R \\ &= \{(a, c) : \exists b \text{ such that } (a, b) \in R \text{ and } (b, c) \in R\} \end{aligned}$$

$R^0 \quad ::= \{(a, a) : a \in A\}$ "the equality relation on A"

$R^{n+1} ::= R^n \circ R$ for $n \geq 0$

e.g., $R^1 = R^0 \circ R = R$

$R^2 = R^1 \circ R = R \circ R$

Non-constructive Definitions

Recursively defined sets and functions describe these objects by explaining how to construct / compute them

But sets can also be defined non-constructively:

$$\boxed{S = \{x : P(x)\}}$$

How can we define functions non-constructively?

- (useful for writing a function specification)

Functions

A function $f : A \to B$ (A as input and B as output) is a special type of relation.

A **function** f **from** A **to** B is a relation from A to B such that: for every $a \in A$, there is *exactly one* $b \in B$ with $(a, b) \in f$

I.e., for every input $a \in A$, there is one output $b \in B$.
We denote this b by $f(a)$.

(When attempting to define a function this way, we sometimes say the function is "well defined" if the *exactly one* part holds)

Functions

A function $f : A \to B$ (A as input and B as output) is a special type of relation.

A **function** f **from** A **to** B is a relation from A to B such that: for every $a \in A$, there is *exactly one* $b \in B$ with $(a, b) \in f$

Ex: {((a, b), d) : d is the largest integer dividing a and b}

- **gcd :** $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$
- **defined without knowing how to compute it**

Matrix Representation

Relation $\boldsymbol{R}$ on $\boldsymbol{A} = \{a_1, \ldots, a_p\}$

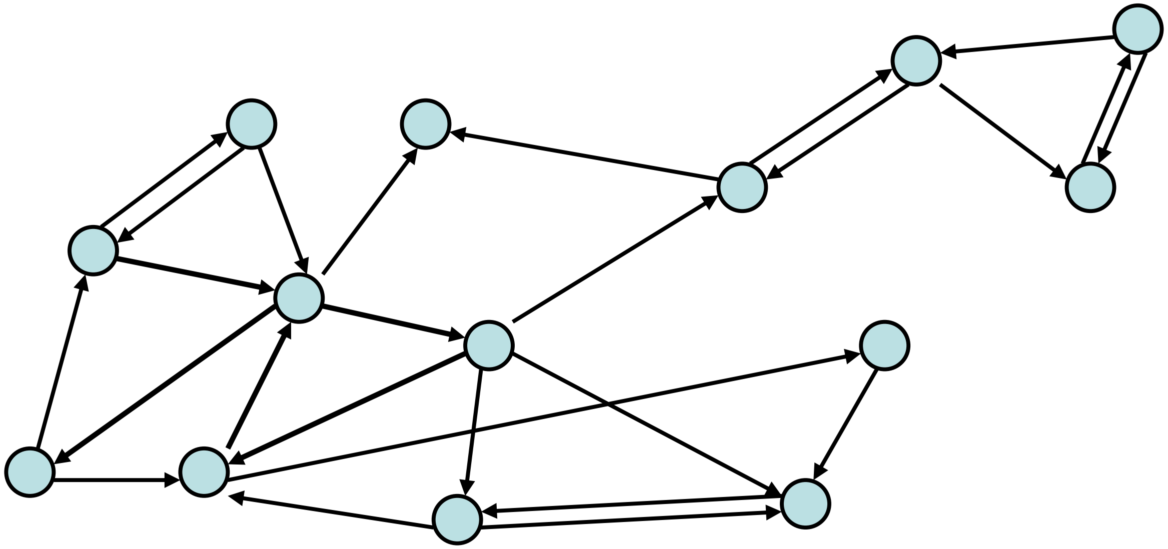
$$\boldsymbol{m_{ij}} = \begin{cases} 1 & \text{if } (a_i, a_j) \in \boldsymbol{R} \\ 0 & \text{if } (a_i, a_j) \notin \boldsymbol{R} \end{cases}$$

{ (1, 1), (1, 2), (1, 4), (2, 1), (2, 3), (3, 2), (3, 3), (4, 2), (4, 3) }

	1	**2**	**3**	**4**
1	1	1	0	1
2	1	0	1	0
3	0	1	1	0
4	0	1	1	0

Directed Graphs

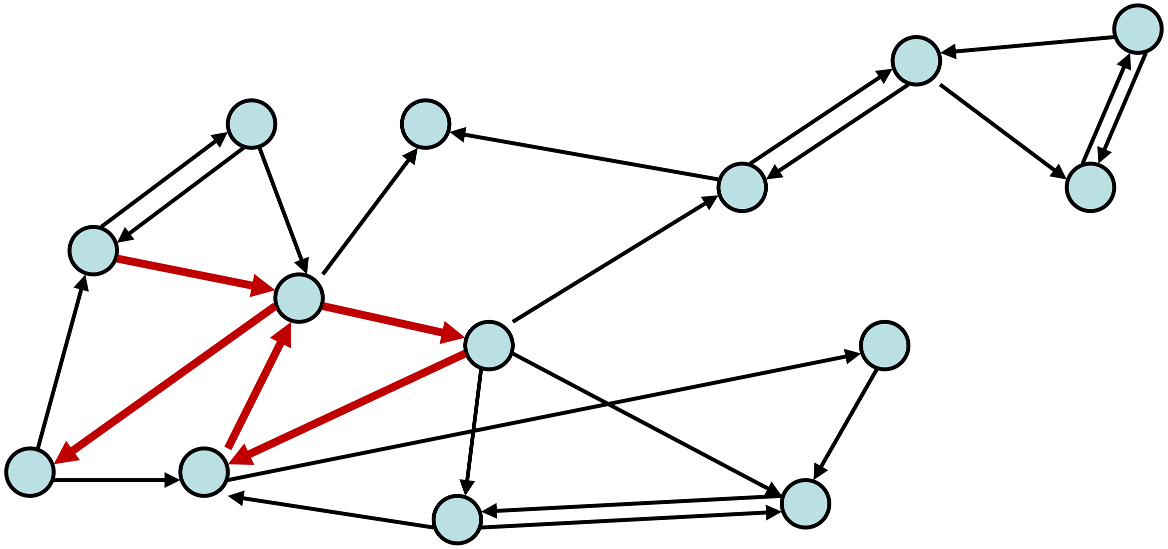
G = (V, E) V – vertices
E – edges (relation on vertices)



Directed Graphs

G = (V, E)   V – vertices
E – edges   (relation on vertices)

Path: $v_0, v_1, \ldots, v_k$ with each (v_i, v_{i+1}) in E

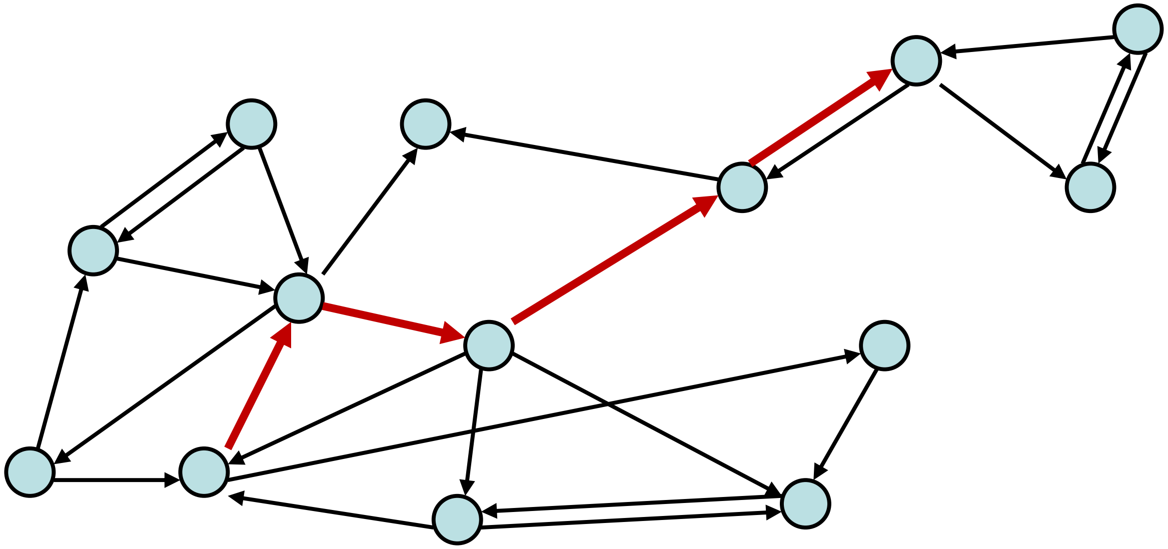


Directed Graphs

G = (V, E) V – vertices
E – edges (relation on vertices)

Path: $v_0, v_1, \ldots, v_k$ with each (v_i, v_{i+1}) in E

Simple Path: none of $\mathbf{v_0}, \ldots, \mathbf{v_k}$ repeated
Cycle: $\mathbf{v_0} = \mathbf{v_k}$
Simple Cycle: $\mathbf{v_0} = \mathbf{v_k}$, none of $\mathbf{v_1}, \ldots, \mathbf{v_k}$ repeated

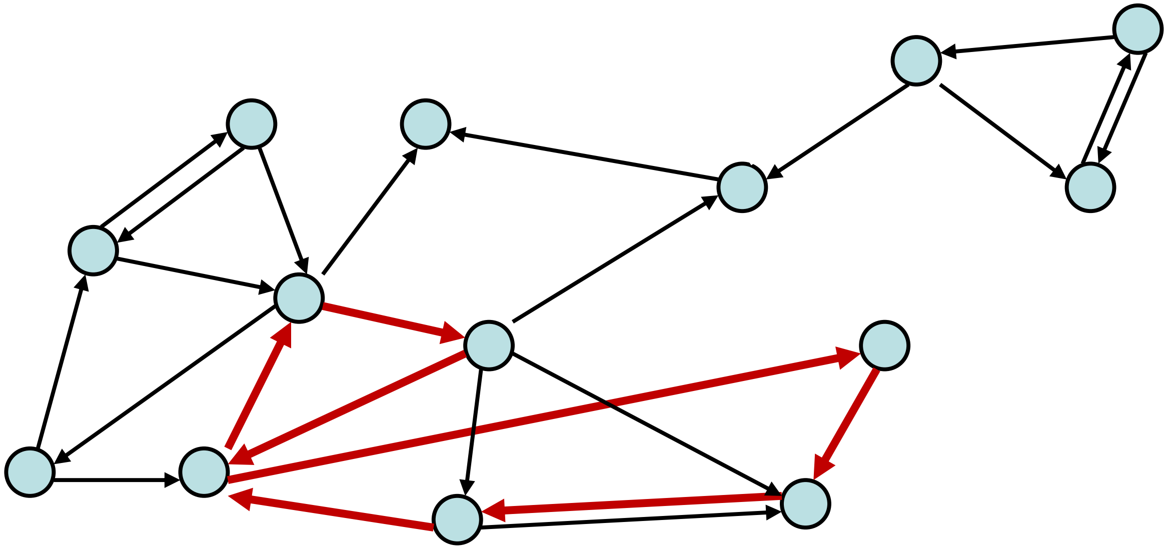


Directed Graphs

G = (V, E) V – vertices
E – edges (relation on vertices)

Path: $v_0, v_1, \ldots, v_k$ with each (v_i, v_{i+1}) in E

Simple Path: none of $\mathbf{v_0}, \ldots, \mathbf{v_k}$ repeated
Cycle: $\mathbf{v_0} = \mathbf{v_k}$
Simple Cycle: $\mathbf{v_0} = \mathbf{v_k}$, none of $\mathbf{v_1}, \ldots, \mathbf{v_k}$ repeated

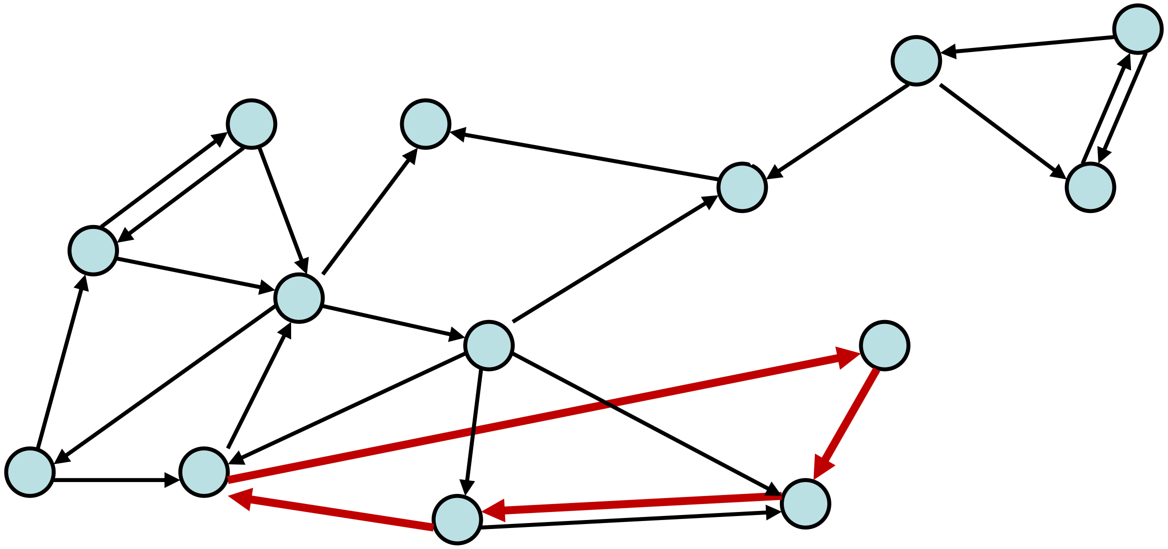


Directed Graphs

G = (V, E) V – vertices
E – edges (relation on vertices)

Path: $v_0, v_1, \ldots, v_k$ with each (v_i, v_{i+1}) in E

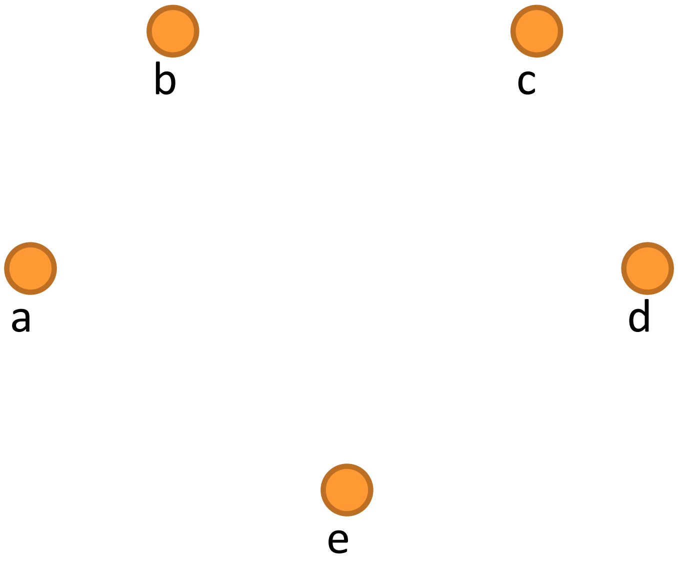
Simple Path: none of $\mathbf{v_0}, \ldots, \mathbf{v_k}$ repeated
Cycle: $\mathbf{v_0} = \mathbf{v_k}$
Simple Cycle: $\mathbf{v_0} = \mathbf{v_k}$, none of $\mathbf{v_1}, \ldots, \mathbf{v_k}$ repeated



Representation of Relations

Directed Graph Representation (Digraph)

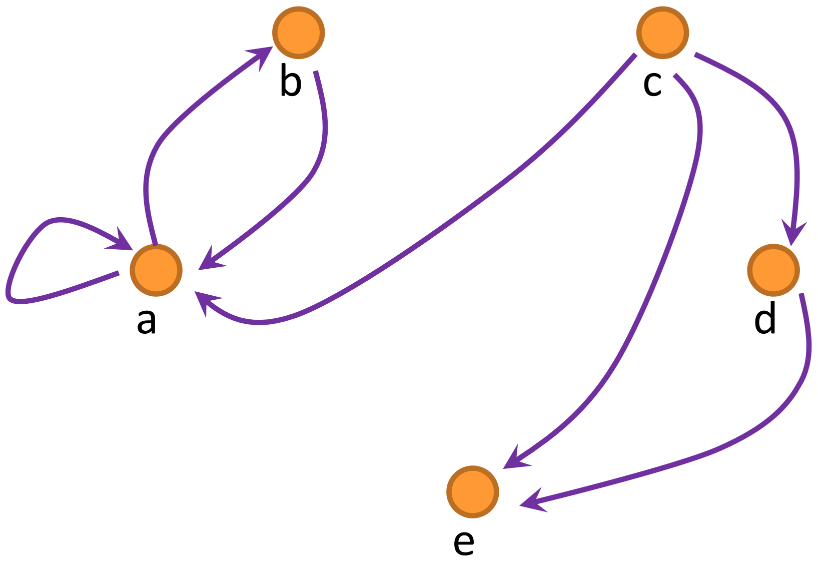
{(a, b), (a, a), (b, a), (c, a), (c, d), (c, e) (d, e) }



Representation of Relations

Directed Graph Representation (Digraph)

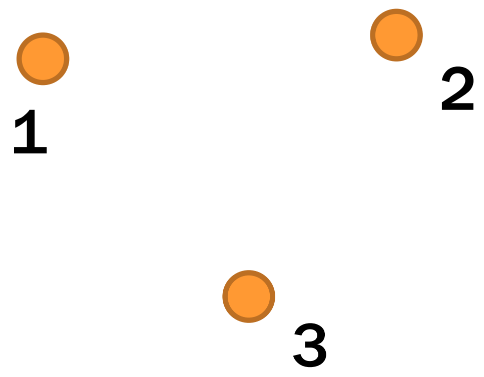
{(a, b), (a, a), (b, a), (c, a), (c, d), (c, e) (d, e) }

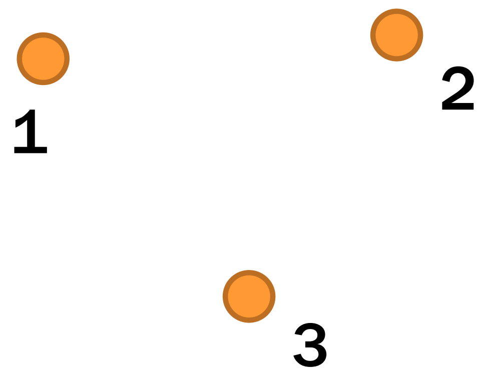


Relational Composition using Digraphs

If $S = \{(2,2),(2,3),(3,1)\}$ and $R = \{(1,2),(2,1),(1,3)\}$

Compute $R \circ S$

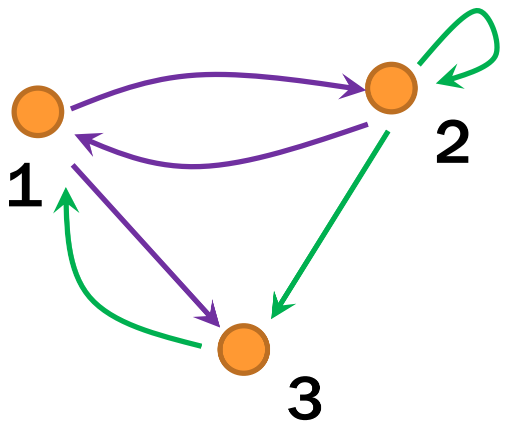


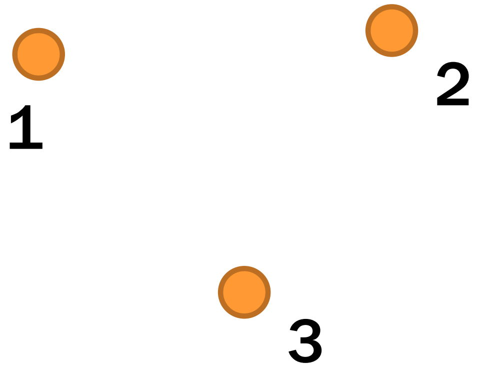


Relational Composition using Digraphs

If $S = \{(2,2),(2,3),(3,1)\}$ and $R = \{(1,2),(2,1),(1,3)\}$

Compute $R \circ S$

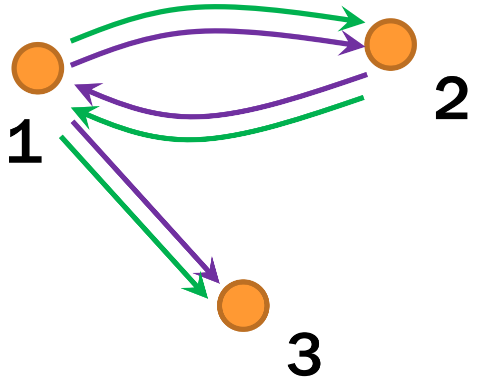


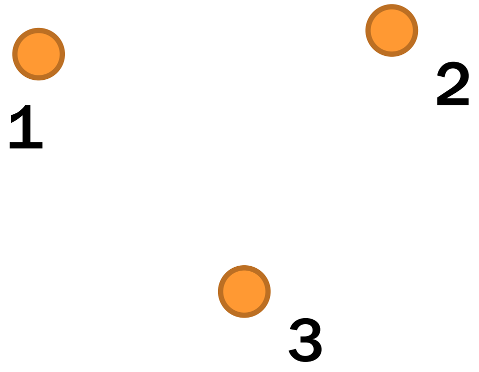


Relational Composition using Digraphs

If $\boldsymbol{R = \{(1,2),(2,1),(1,3)\}}$ and $\boldsymbol{R = \{(1,2),(2,1),(1,3)\}}$

Compute $\boldsymbol{R \circ R}$





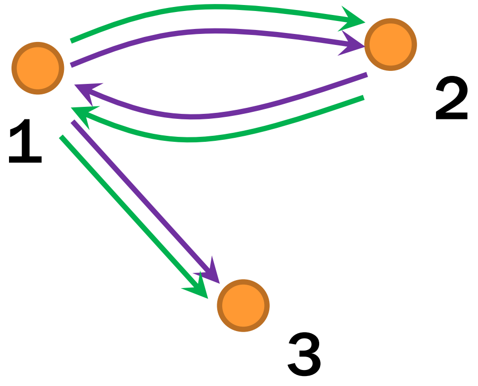
$(a,c) \in R \circ R = R^2$ iff $\exists b\ ((a,b) \in R \land (b,c) \in R)$

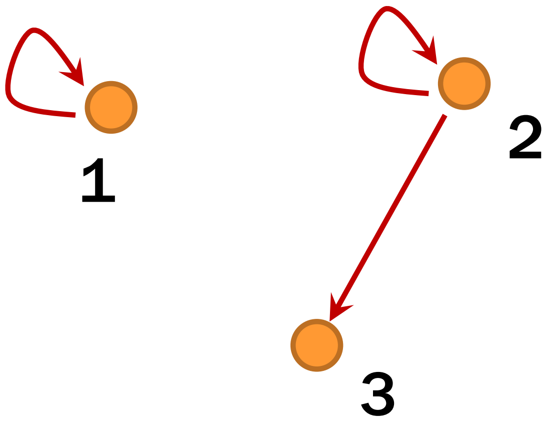
iff $\exists b$ such that a, b, c is a path

Relational Composition using Digraphs

If $\boldsymbol{R} = \{(\mathbf{1},\mathbf{2}),(\mathbf{2},\mathbf{1}),(\mathbf{1},\mathbf{3})\}$ and $\boldsymbol{R} = \{(\mathbf{1},\mathbf{2}),(\mathbf{2},\mathbf{1}),(\mathbf{1},\mathbf{3})\}$

Compute $\boldsymbol{R} \circ \boldsymbol{R}$





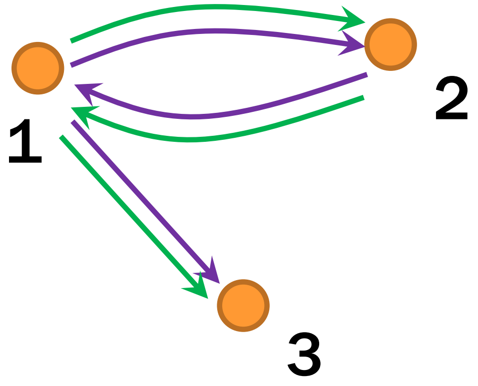
$(a,c) \in R \circ R = R^2$ iff $\exists b\ ((a,b) \in R \land (b,c) \in R)$

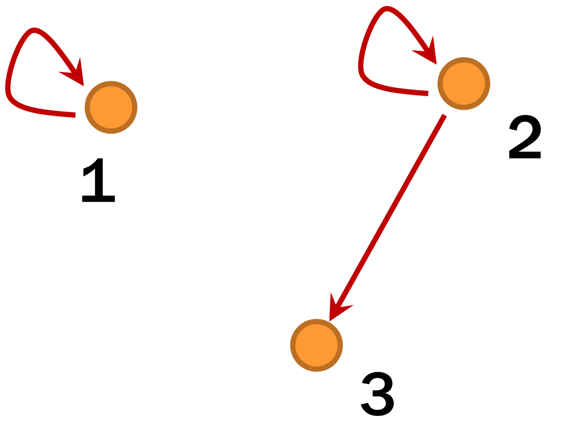
iff $\exists b$ such that a, b, c is a path

Relational Composition using Digraphs

If $R = \{(1,2),(2,1),(1,3)\}$ and $R = \{(1,2),(2,1),(1,3)\}$

Compute $R \circ R$





Special case: $R \circ R$ is paths of length 2.

- R is paths of length 1
- R^0 is paths of length 0 (can't go anywhere)
- $R^3 = R^2 \circ R$ etc, so is R^n paths of length n

Paths in Relations and Graphs

Def: The **length** of a path in a graph is the number of edges in it (counting repetitions if edge used > once).

Let $\boldsymbol{R}$ be a relation on a set $\boldsymbol{A}$. There is a path of length $\boldsymbol{n}$ from **a** to **b** if and only if (**a,b**) $\in \boldsymbol{R}^{\boldsymbol{n}}$

Connectivity In Graphs

Def: Two vertices in a graph are **connected** iff there is a path between them.

Let $\boldsymbol{R}$ be a relation on a set $\boldsymbol{A}$. The **connectivity** relation $\boldsymbol{R}^*$ consists of the pairs (a, b) such that there is a path from a to b in $\boldsymbol{R}$.

$$R^* = \bigcup_{k=0}^{\infty} R^k$$

Note: The text uses the wrong definition of this quantity. What the text defines (ignoring k=0) is usually called R⁺

How Properties of Relations show up in Graphs

Let R be a relation on A.

R is **reflexive** iff $(a,a) \in R$ for every $a \in A$

R is **symmetric** iff $(a,b) \in R$ implies $(b, a) \in R$

R is **antisymmetric** iff $(a,b) \in R$ and $a \neq b$ implies $(b,a) \notin R$

R is **transitive** iff $(a,b) \in R$ and $(b, c) \in R$ implies $(a, c) \in R$

How Properties of Relations show up in Graphs

Let R be a relation on A.

R is **reflexive** iff (a,a) ∈ R for every a ∈ A

at every node

R is **symmetric** iff (a,b) ∈ R implies (b, a)∈ R

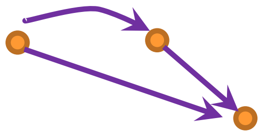
or

R is **antisymmetric** iff (a,b) ∈ R and a ≠ b implies (b,a) ∉ R

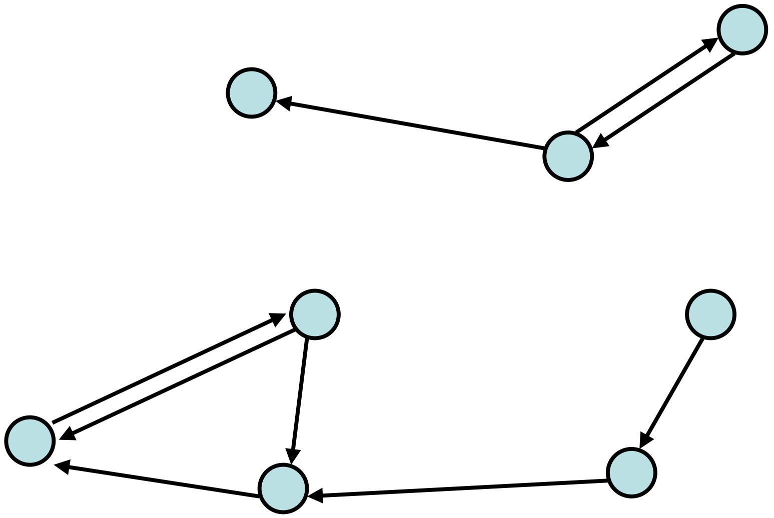
or

or

R is **transitive** iff (a,b)∈ R and (b, c)∈ R implies (a, c) ∈ R

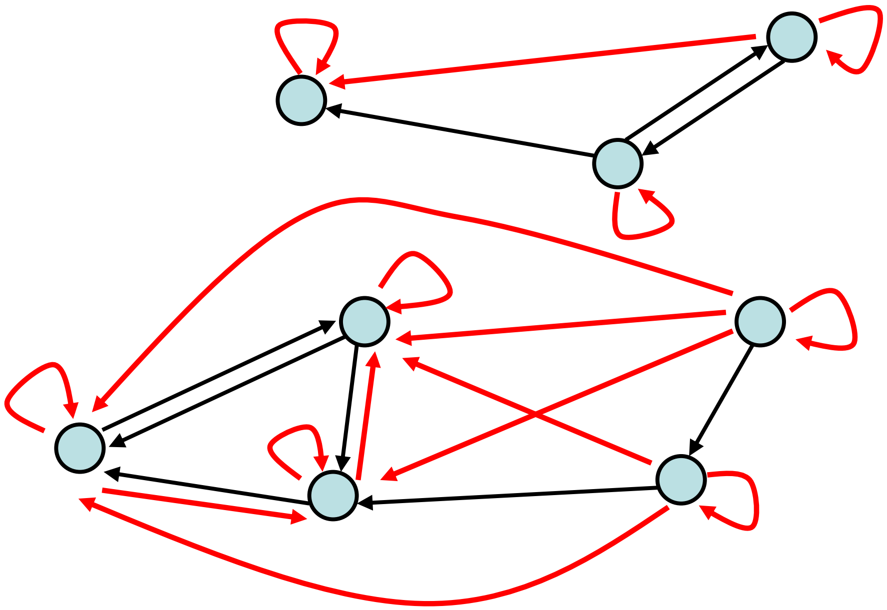


Transitive-Reflexive Closure



Add the **minimum possible** number of edges to make the relation transitive and reflexive.

Transitive-Reflexive Closure



Relation with the **minimum possible** number of **extra edges** to make the relation both transitive and reflexive.

The **transitive-reflexive closure** of a relation $\boldsymbol{R}$ is the connectivity relation $\boldsymbol{R}^*$

n-ary Relations

Let $\boldsymbol{A_1}, \boldsymbol{A_2}, \ldots, \boldsymbol{An}$ be sets. An **$\boldsymbol{n}$-ary** relation on these sets is a subset of $\boldsymbol{A_1} \times \boldsymbol{A_2} \times \cdots \times \boldsymbol{A_n}$.

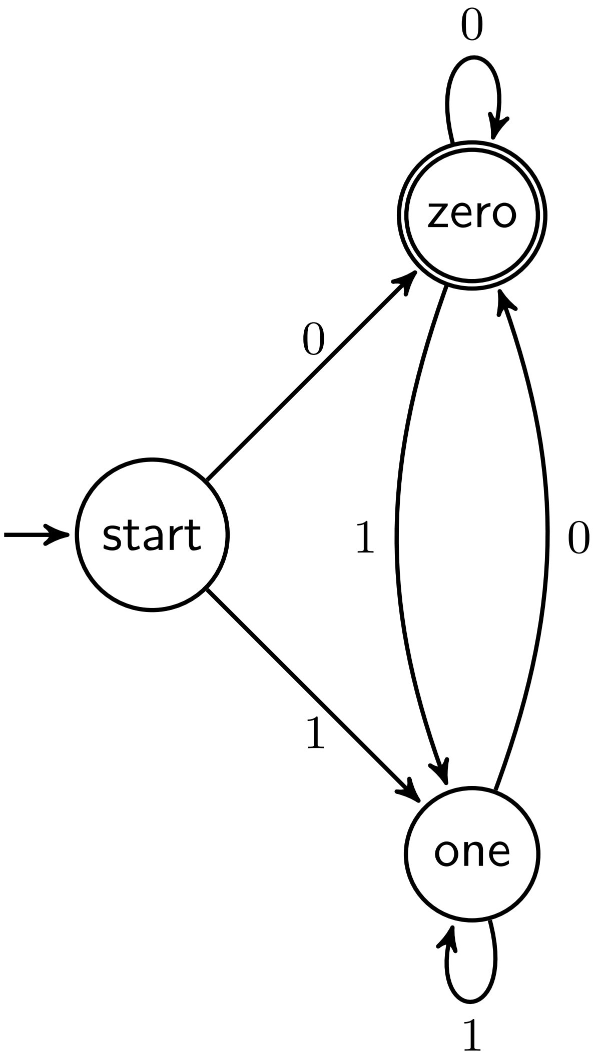
Relational Databases

STUDENT

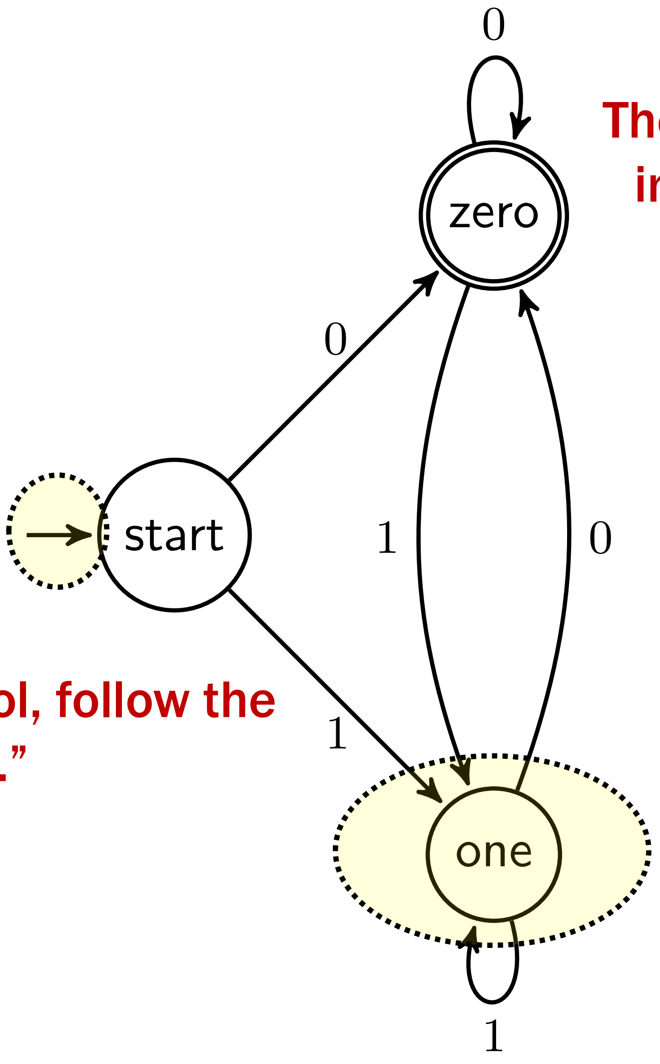
Student_Name	ID_Number	Office	GPA
Knuth	328012098	022	4.00
Von Neuman	481080220	555	3.78
Russell	238082388	022	3.85
Einstein	238001920	022	2.11
Newton	1727017	333	3.61
Karp	348882811	022	3.98
Bernoulli	2921938	022	3.21

Back to Languages

Selecting strings using labeled graphs as “machines”



Finite State Machines



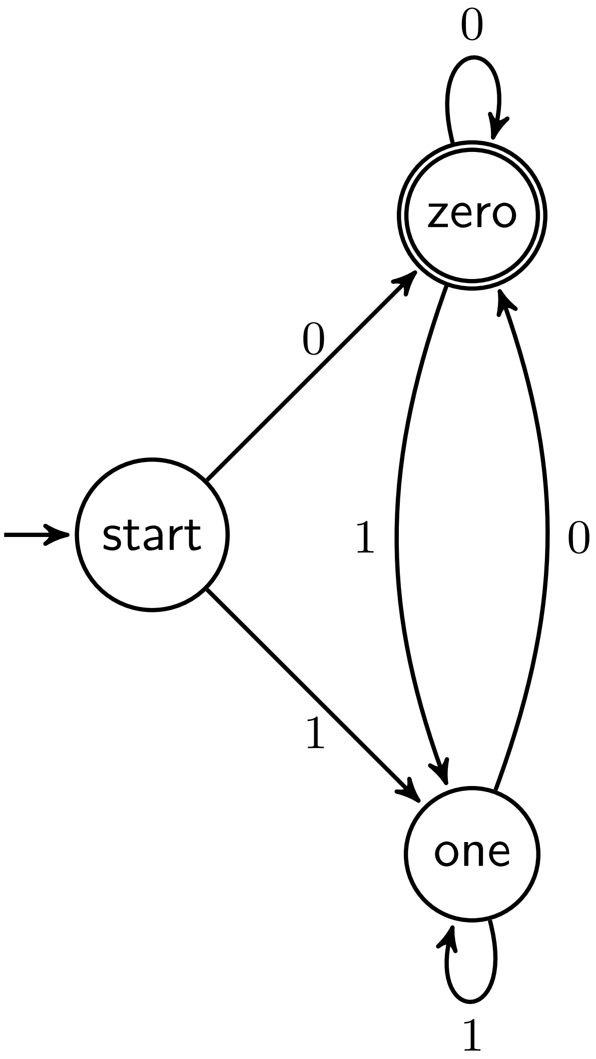
The “double circle” means “the input is good if it ends here”

“Start here”

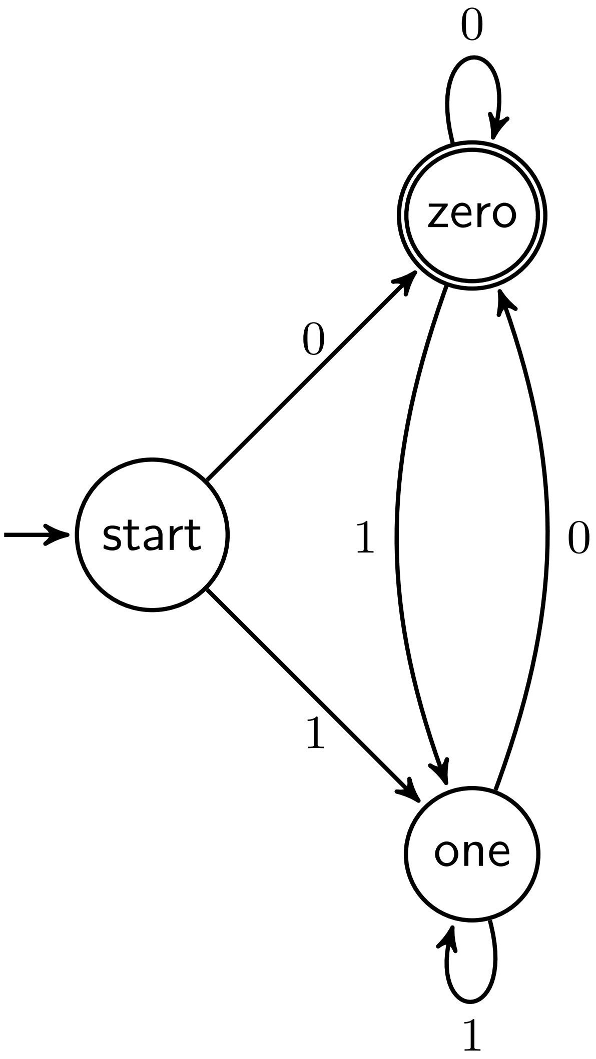
“If I get this symbol, follow the arrow…”

The circles are called “states”
We’re only in a single state at any point in time…

Which strings does this machine say are OK?



Which strings does this machine say are OK?

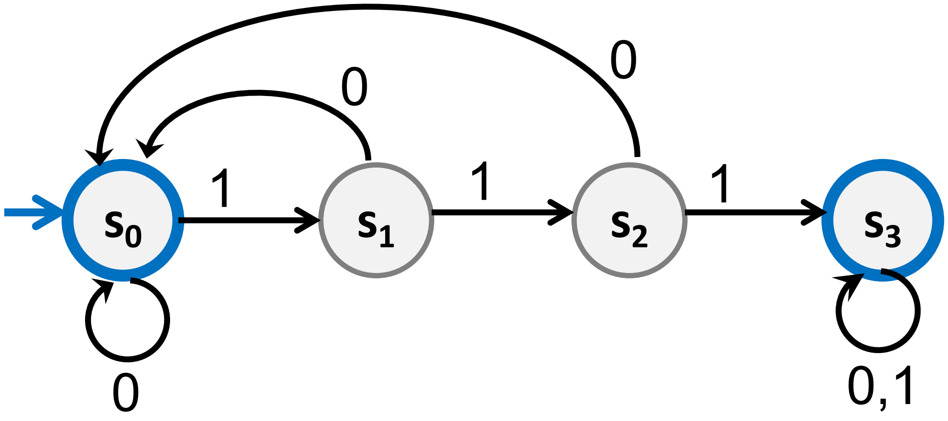


The set of all binary strings that end in 0

Finite State Machines

- States
- Transitions on input symbols
- Start state and final states
- The "language recognized" by the machine is the set of strings that reach a final state from the start

Old State	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_3
s_3	s_3	s_3

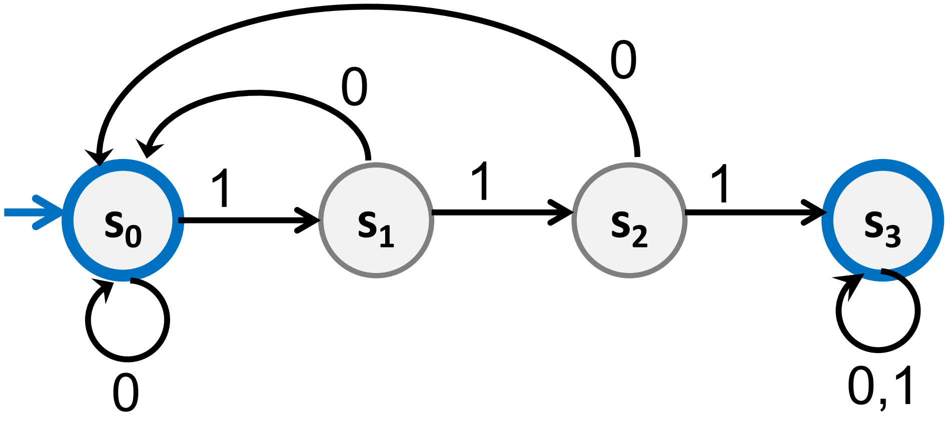


Finite State Machines

- Each machine designed for strings over some fixed alphabet Σ.

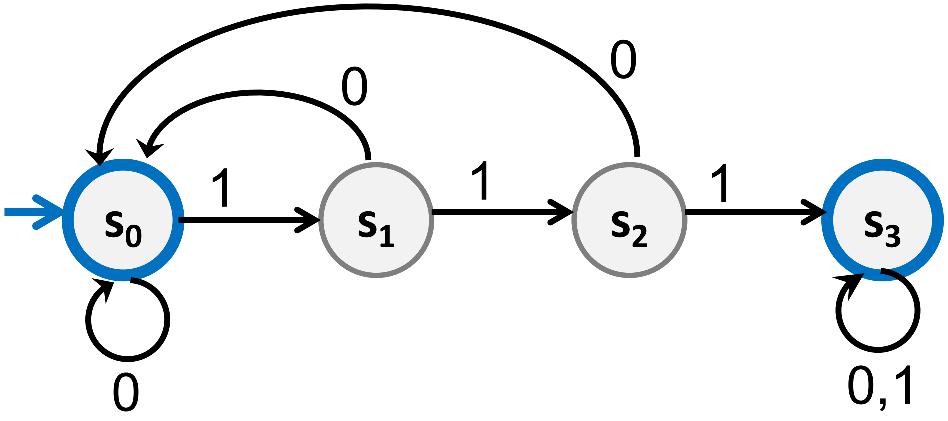
- Must have a transition defined from each state for ***every*** symbol in Σ.

Old State	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_3
s_3	s_3	s_3



What language does this machine recognize?

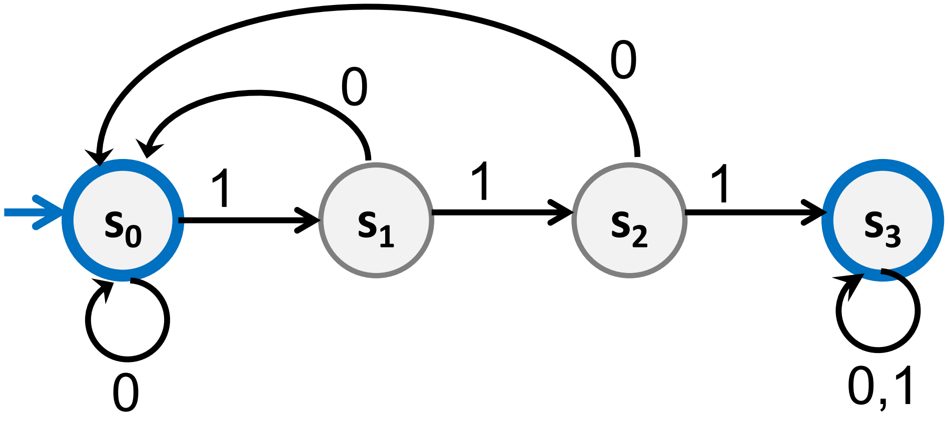
Old State	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_3
s_3	s_3	s_3



What language does this machine recognize?

The set of all binary strings that contain 111 or don't end in 1

Old State	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_3
s_3	s_3	s_3



Applications of FSMs (a.k.a. Finite Automata)

- Implementation of regular expression matching in programs like `grep`
- Control structures for sequential logic in digital circuits
- Algorithms for communication and cache-coherence protocols
  - Each agent runs its own FSM
- Design specifications for reactive systems
  - Components are communicating FSMs

Applications of FSMs (a.k.a. Finite Automata)

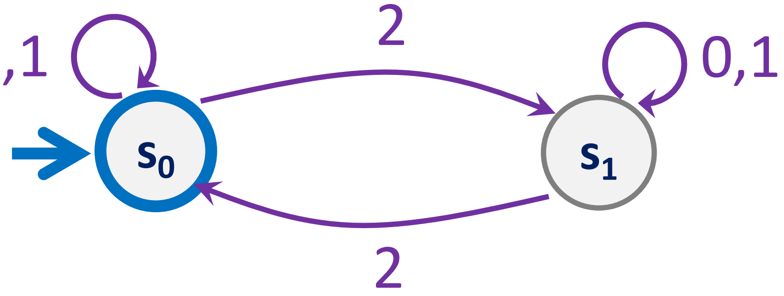
- Formal verification of systems
  - Is an unsafe state reachable?
- Computer games
  - FSMs implement non-player characters
- Minimization algorithms for FSMs can be extended to more general models used in
  - Text prediction
  - Speech recognition

Strings over {0, 1, 2}

M_1: Strings with an even number of 2's

Strings over {0, 1, 2}

M_1: Strings with an even number of 2's

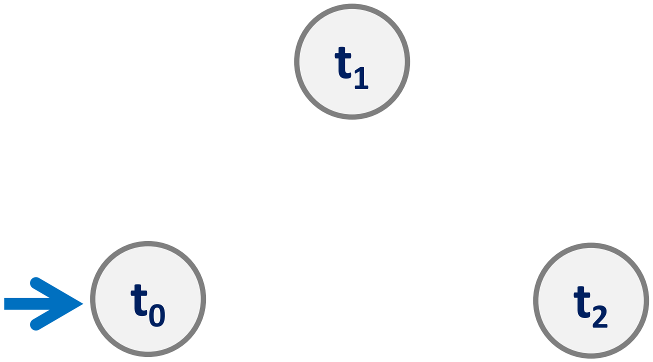


FSM as abstraction of Java code

```
boolean sumCongruentToZero(String str) {
   int sum = 0;
   for (int i = 0; i < str.length(); i++) {
      if (str.charAt(i) == '2’)
         sum = (sum + 2) % 3;
      if (str.charAt(i) == '1’)
         sum = (sum + 1) % 3;
      if (str.charAt(i) == ‘0’)
         sum = (sum + 0) % 3;
   }
   return sum == 0;
}
```

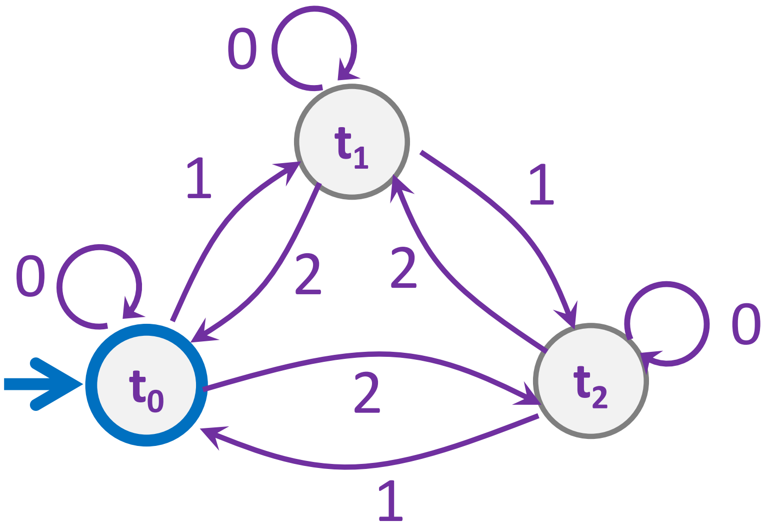
Strings over {0, 1, 2}

M_2: Strings where the sum of digits mod 3 is 0



Strings over {0, 1, 2}

M_2: Strings where the sum of digits mod 3 is 0



FSM as abstraction of Java code

```
boolean sumCongruentToZero(String str) {
    int sum = 0;
    for (int i = 0; i < str.length(); i++) {
        if (str.charAt(i) == '2’)
            sum = (sum + 2) % 3;
        if (str.charAt(i) == '1’)
            sum = (sum + 1) % 3;
        if (str.charAt(i) == ‘0’)
            sum = (sum + 0) % 3;
    }
    return sum == 0
}
```

FSMs can model Java code with
a **finite** number of **fixed-size** variables
that makes **one pass** through input

FSM to Java code

```
int[][] TRANSITION = {...};

boolean sumCongruentToZero(String str) {
    int state = 0;
    for (int i = 0; i < str.length(); i++) {
        int d = str.charAt(i) - ‘0’;
        state = TRANSITION[state][d];
    }
    return state == 0;
}
```

State Machine Design Recipe

Given a language, how do you design a state machine for it?

Need enough states to:

- Decide whether to accept or reject at the end
- Update the state on each new character

State Machine Design Recipe

M_2: Strings where the sum of digits mod 3 is 0

State Machine Design Recipe

M_2: Strings where the sum of digits mod 3 is 0

Can we get away with two states?

- One for 0 mod 3 and one for everything else

State Machine Design Recipe

M_2: Strings where the sum of digits mod 3 is 0

Can we get away with two states?

- One for 0 mod 3 and one for everything else

This would be enough to decide at the end!

But can’t update the state on each new character

State Machine Design Recipe

M_2: Strings where the sum of digits mod 3 is 0

Can we get away with two states?

- One for 0 mod 3 and one for everything else

This would be enough to decide at the end!

But can't update the state on each new character:

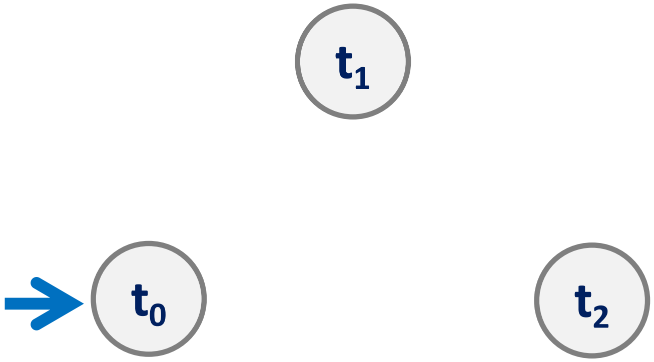
- If you're in the "not 0 mod 3" state, and the next character is 1, which state should you go to?

State Machine Design Recipe

M_2: Strings where the sum of digits mod 3 is 0

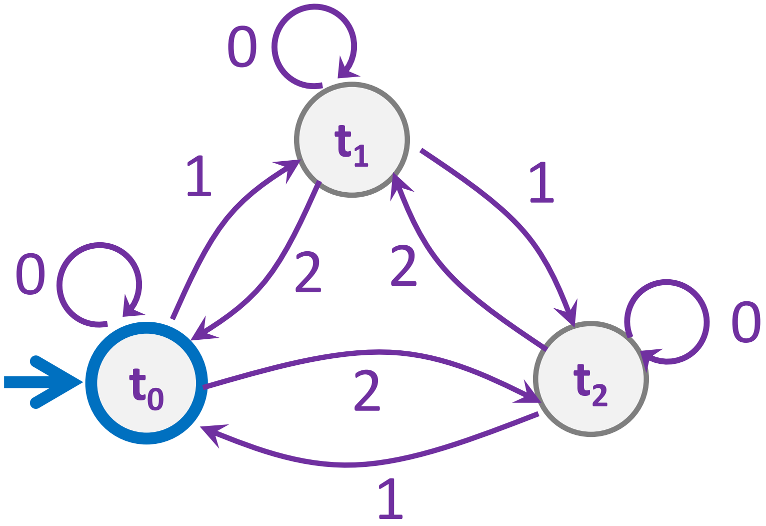
So, we need three states.

What information should we track?



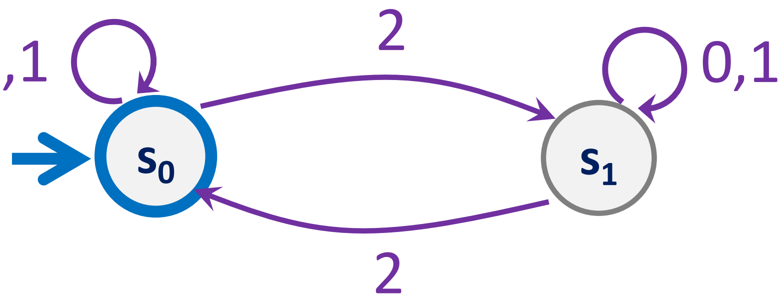
Strings over {0, 1, 2}

M_2: Strings where the sum of digits mod 3 is 0

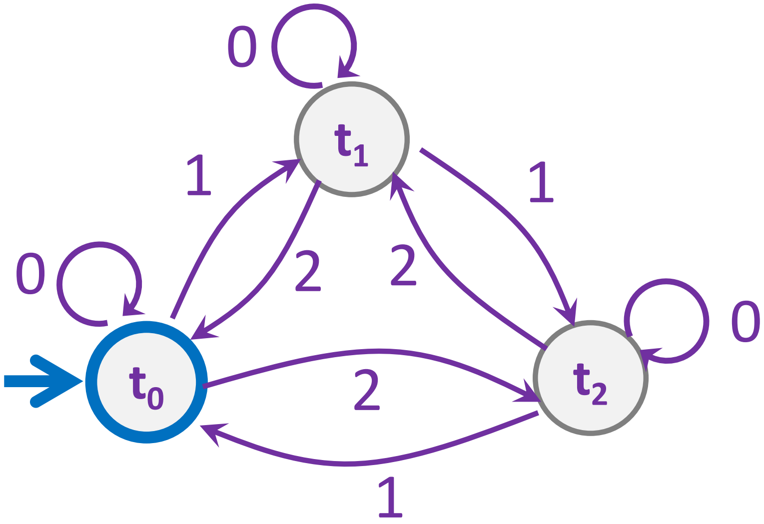


Strings over {0, 1, 2}

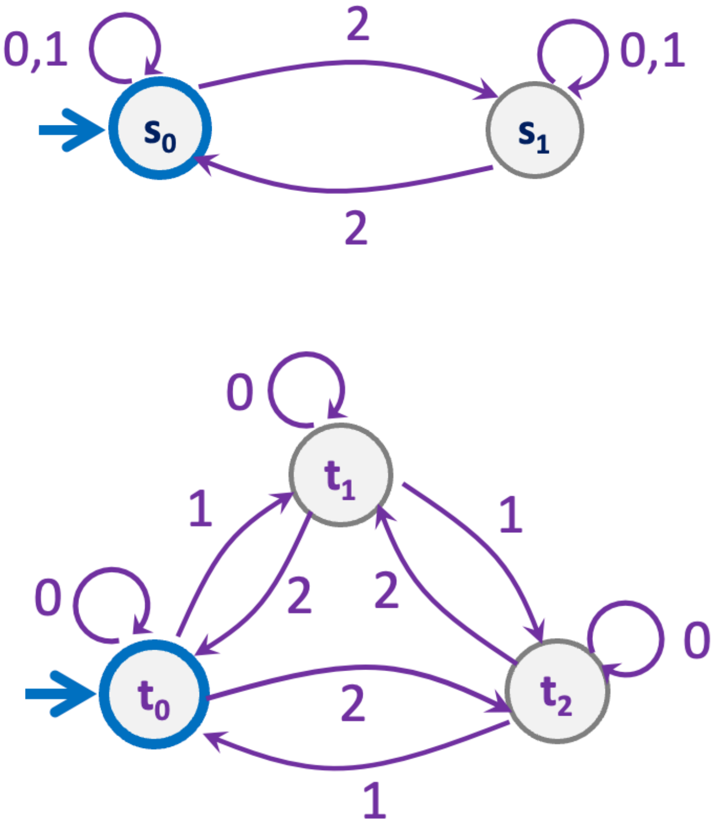
M_1: Strings with an even number of 2's

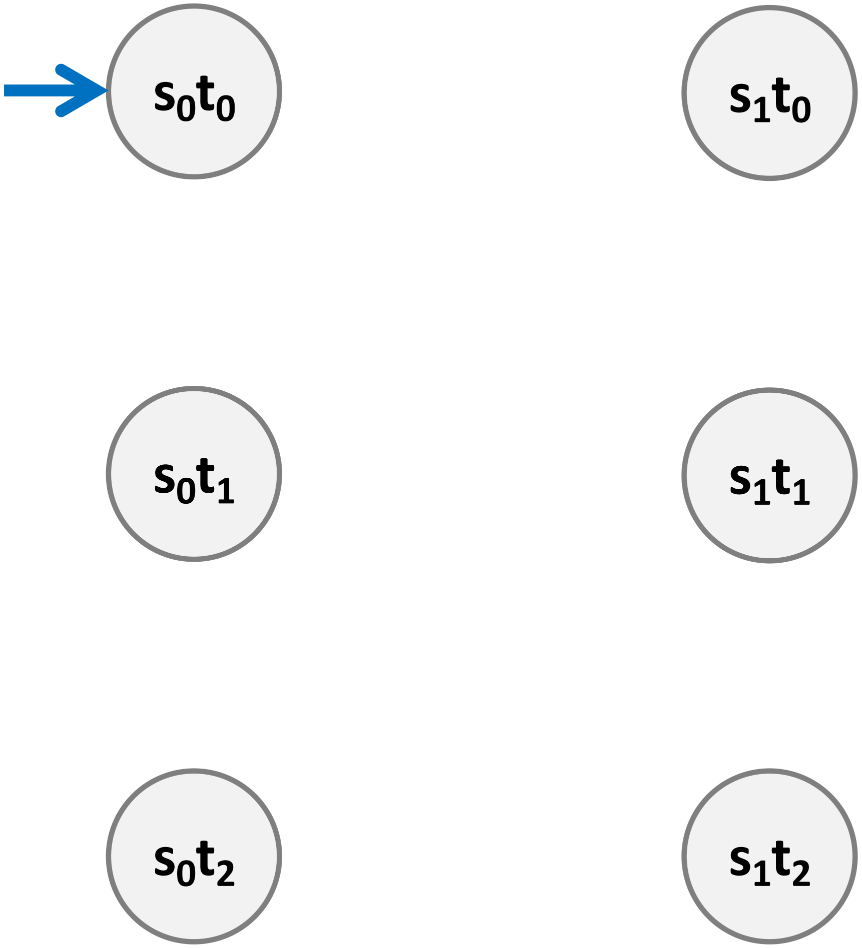


M_2: Strings where the sum of digits mod 3 is 0

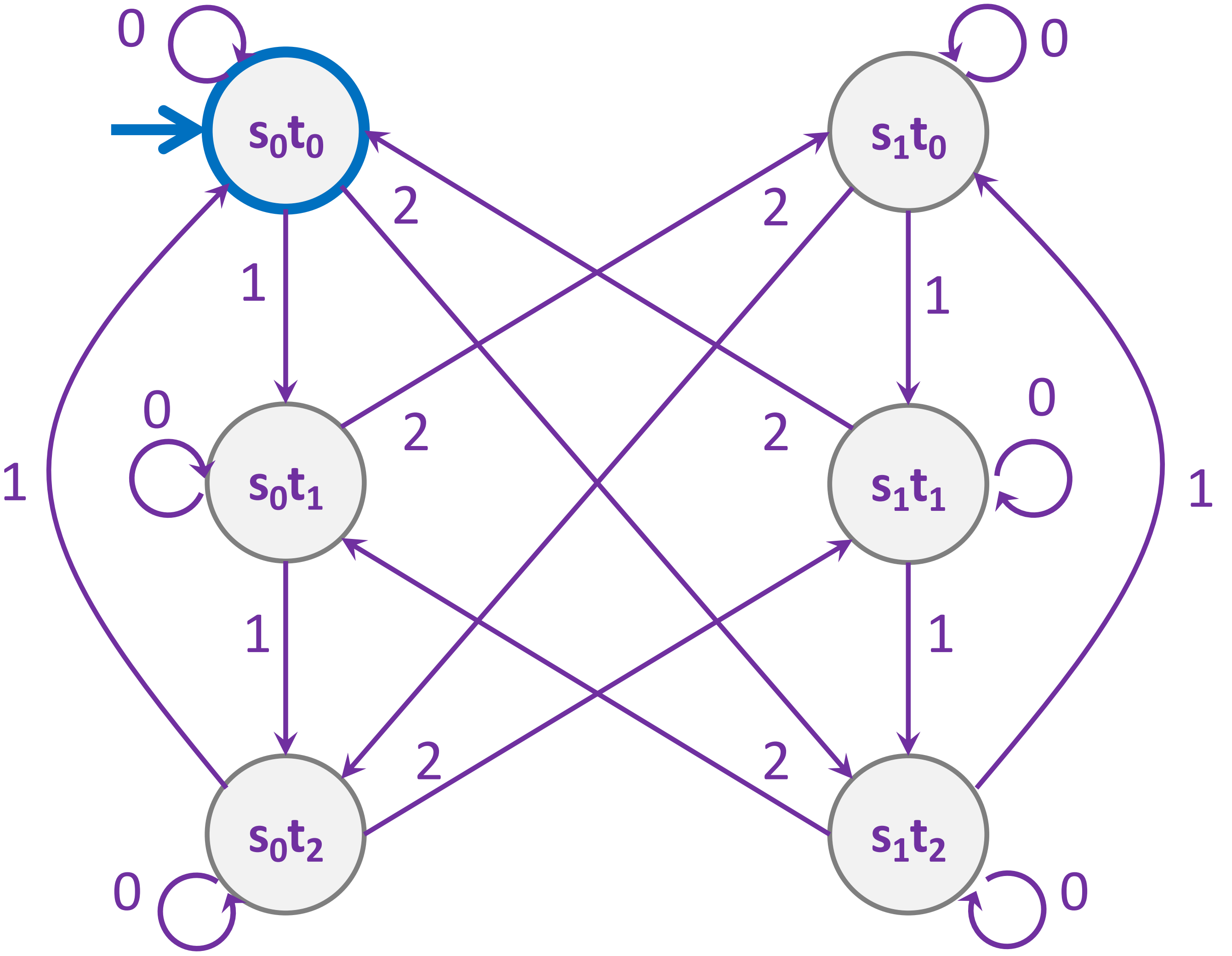


Strings over {0,1,2} w/ even number of 2's AND mod 3 sum 0

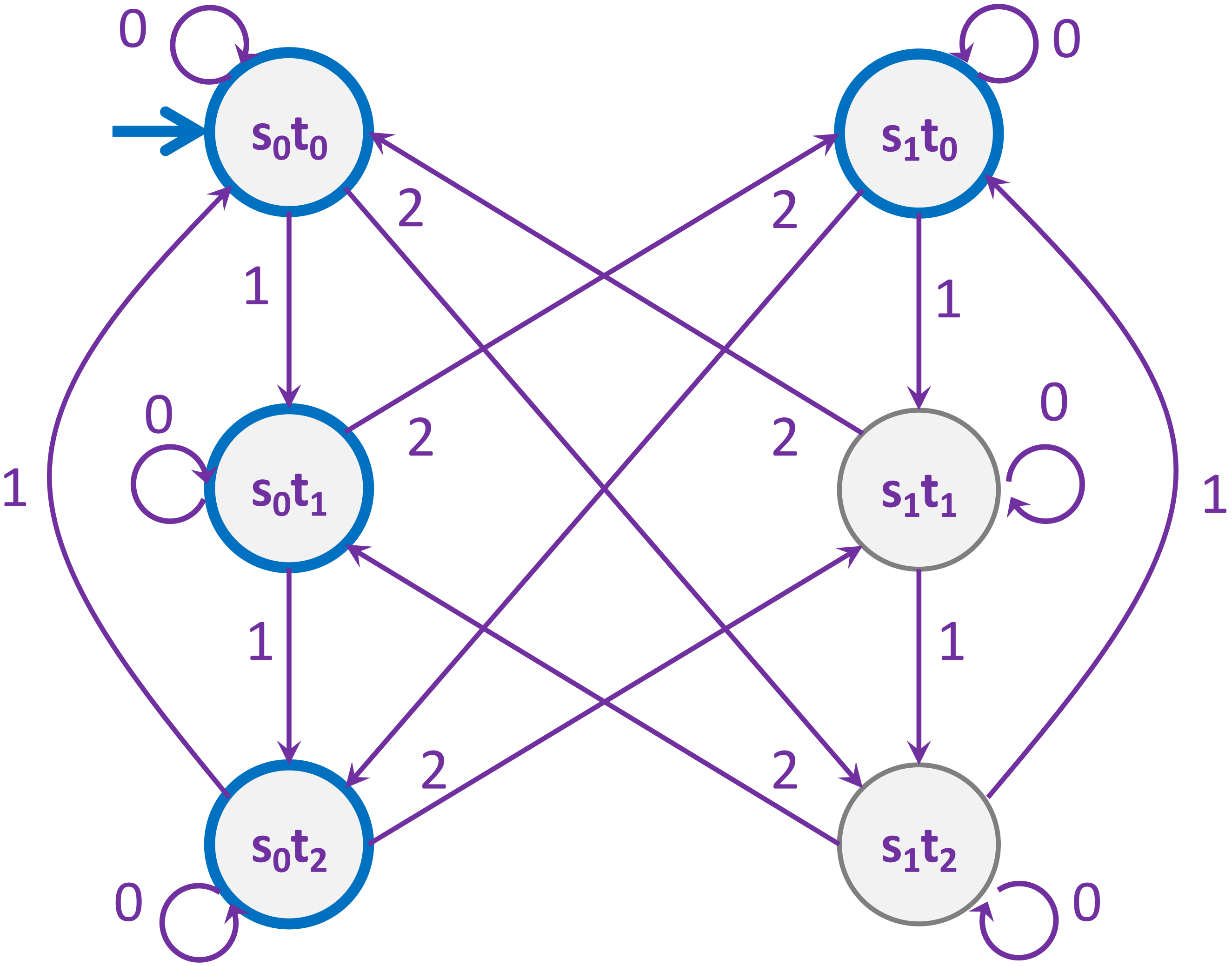




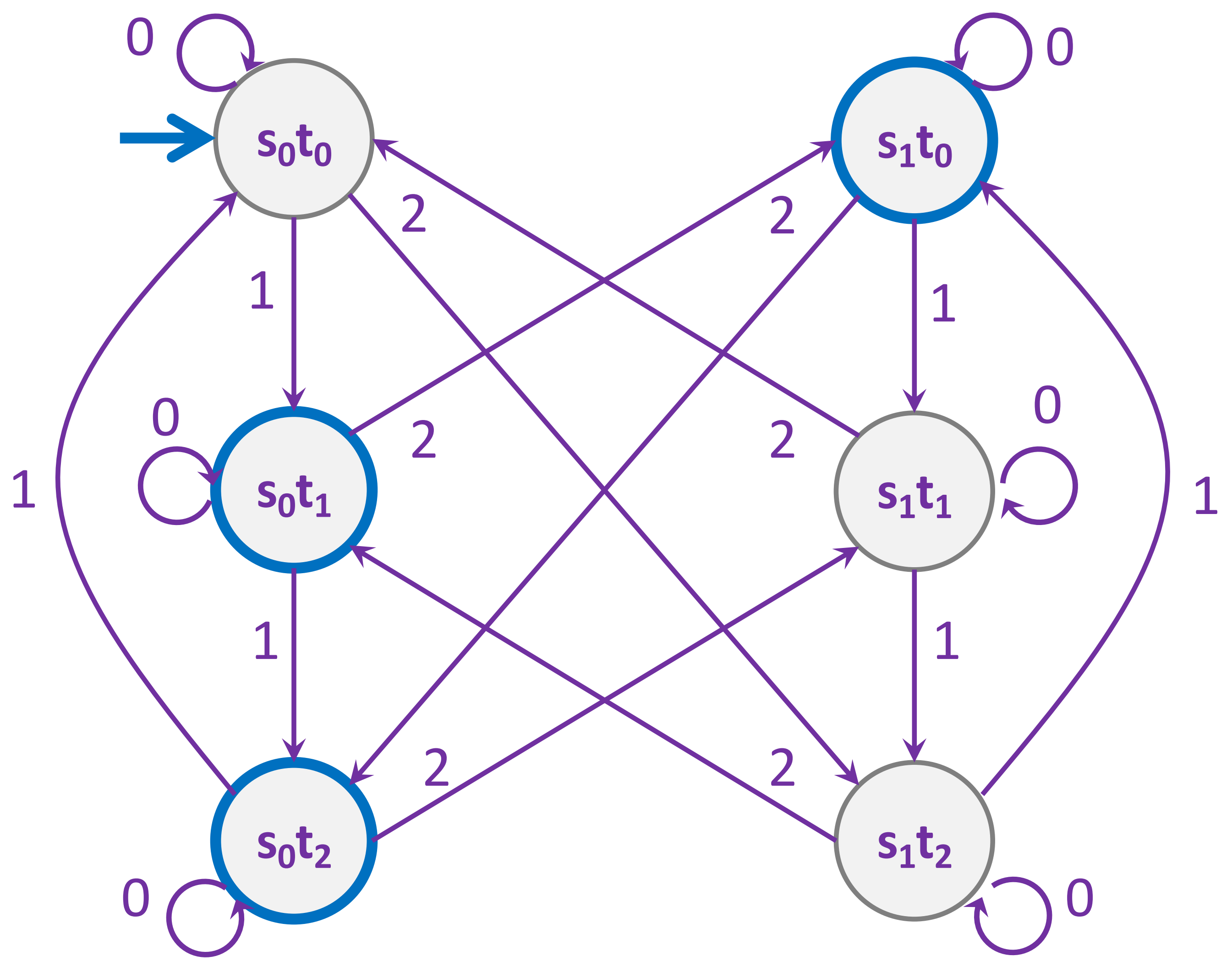
Strings over {0,1,2} w/ even number of 2's AND mod 3 sum 0



Strings over {0,1,2} w/ even number of 2's OR mod 3 sum 0

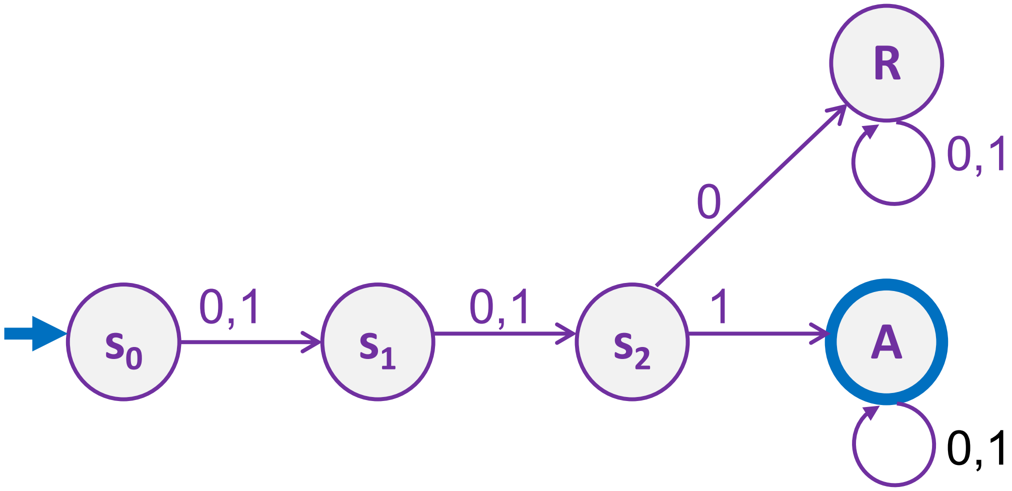


Strings over {0,1,2} w/ even number of 2's XOR mod 3 sum 0



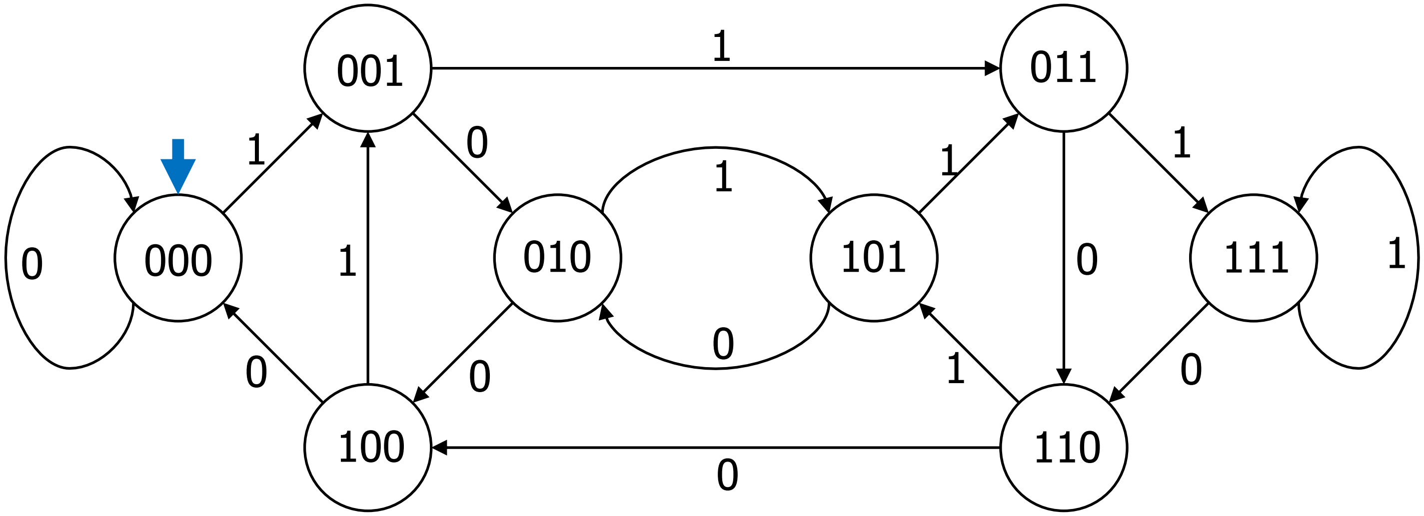
The set of binary strings with a 1 in the 3rd position from the start

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

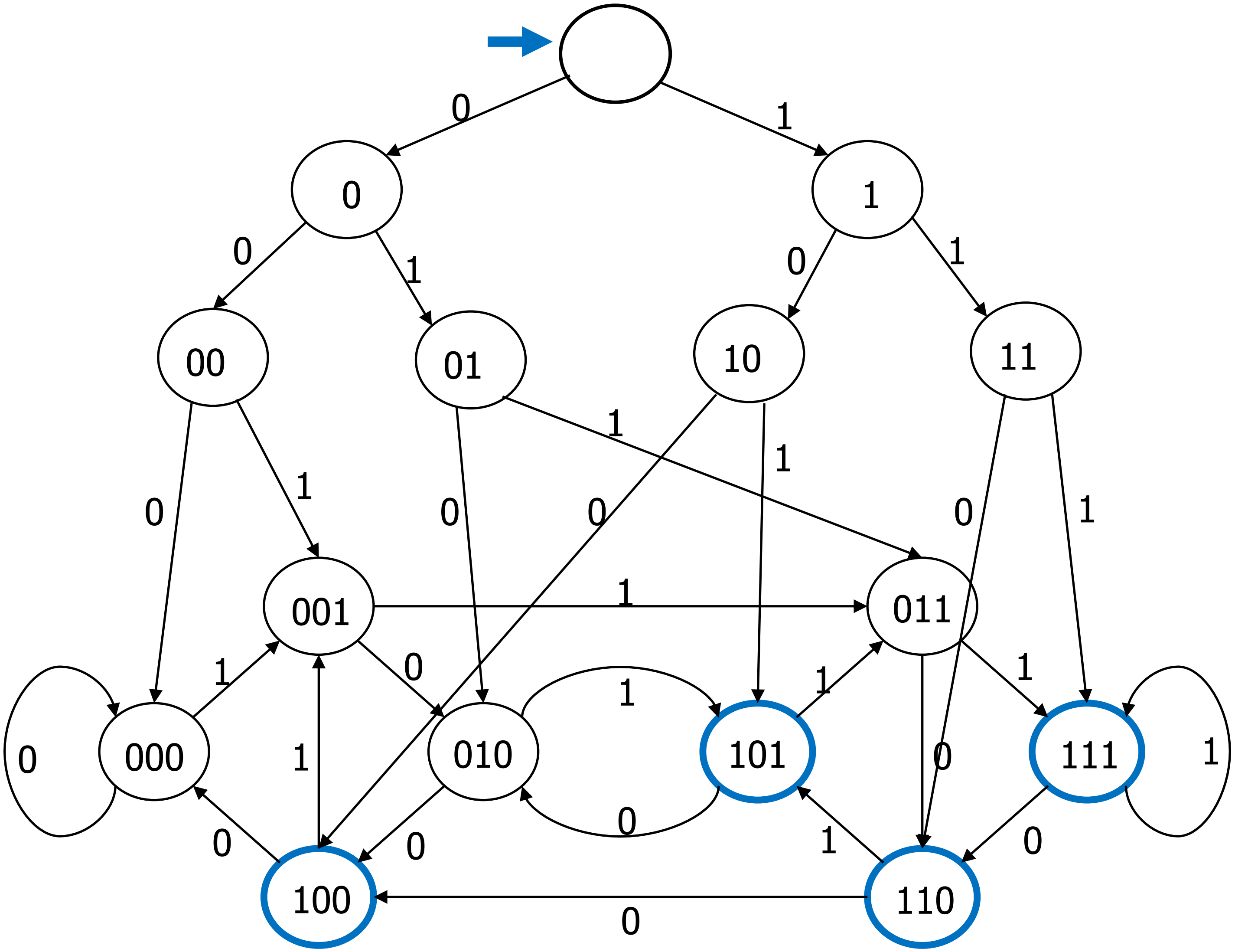


The set of binary strings with a 1 in the 3rd position from the end

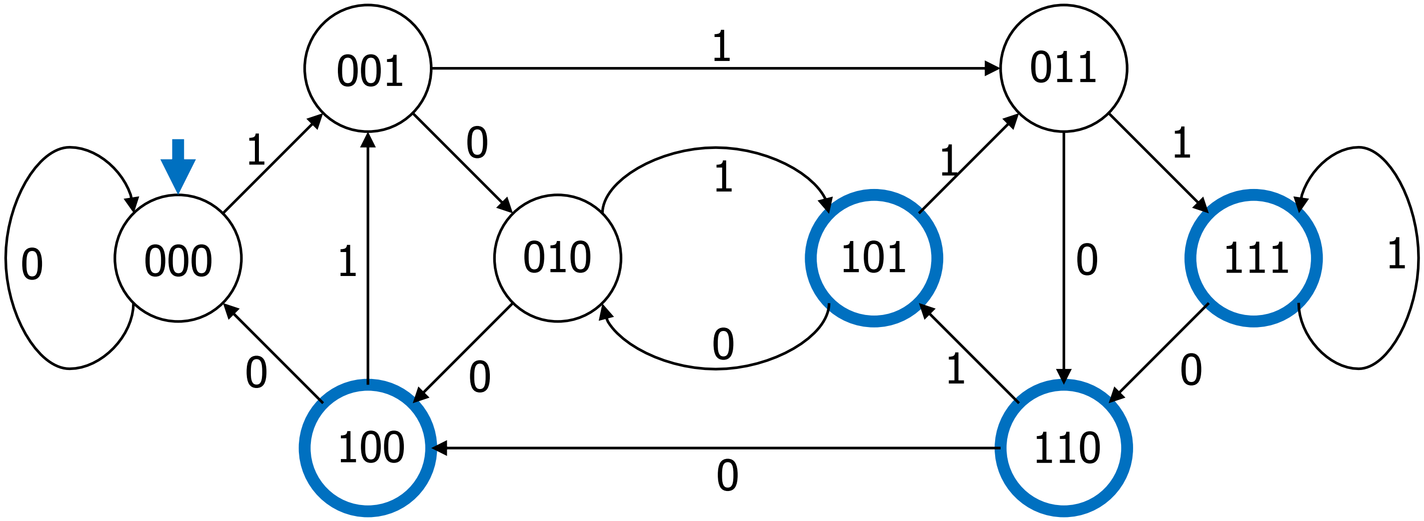
3 bit shift register “Remember the last three bits”



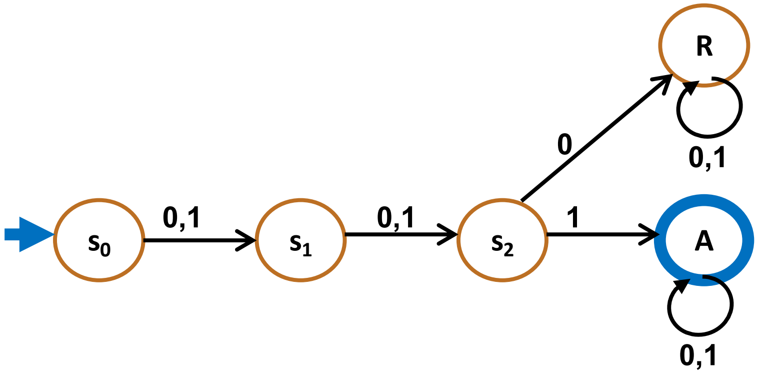
The set of binary strings with a 1 in the 3rd position from the end

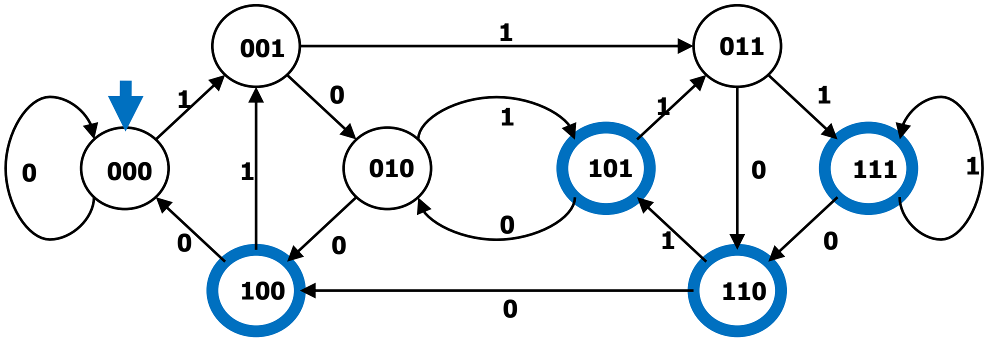


The set of binary strings with a 1 in the 3rd position from the end



The beginning versus the end





Adding Output to Finite State Machines

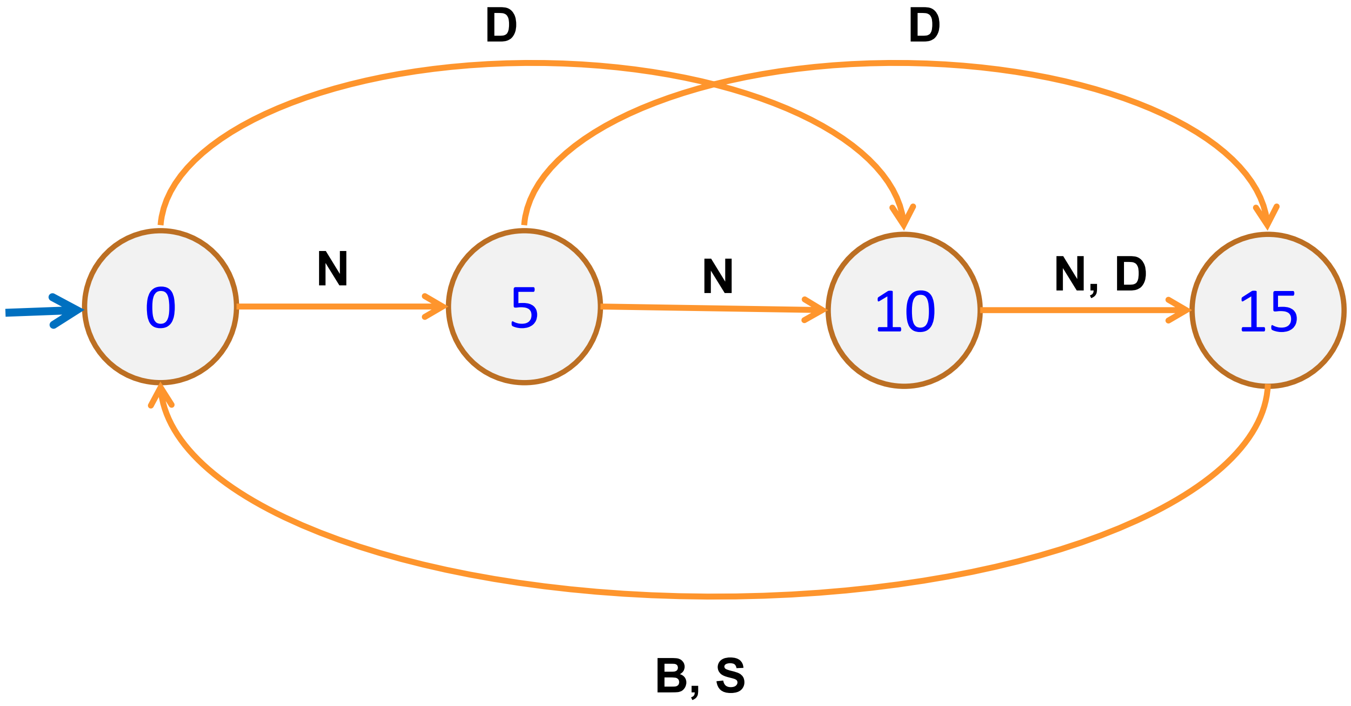
- So far we have considered finite state machines that just accept/reject strings
  - called “Deterministic Finite Automata” or DFAs

- Now we consider finite state machines *with output*
  - These are the kinds used as controllers

Vending Machine

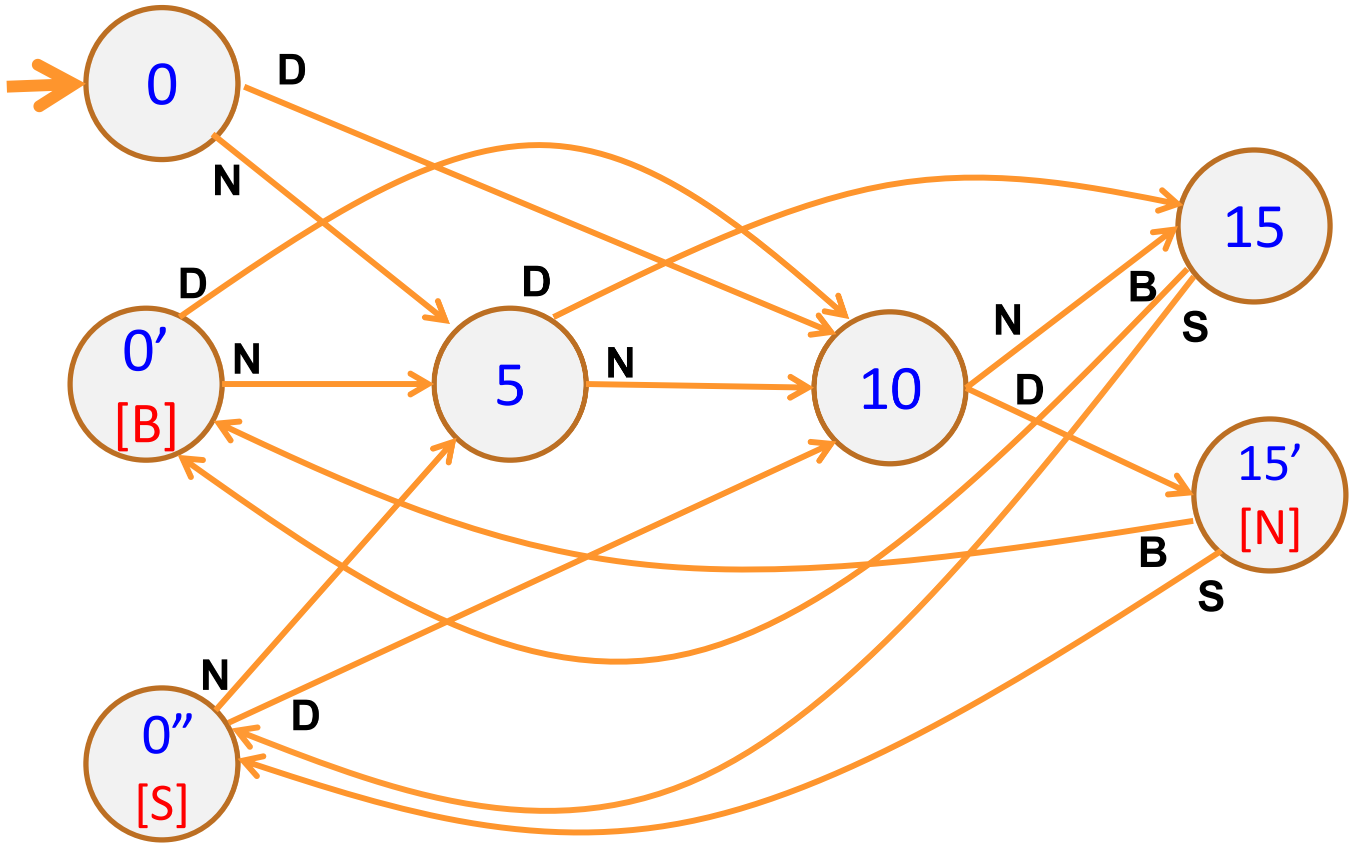
Enter 15 cents in dimes or nickels
Press S or B for a candy bar

Vending Machine, v0.1



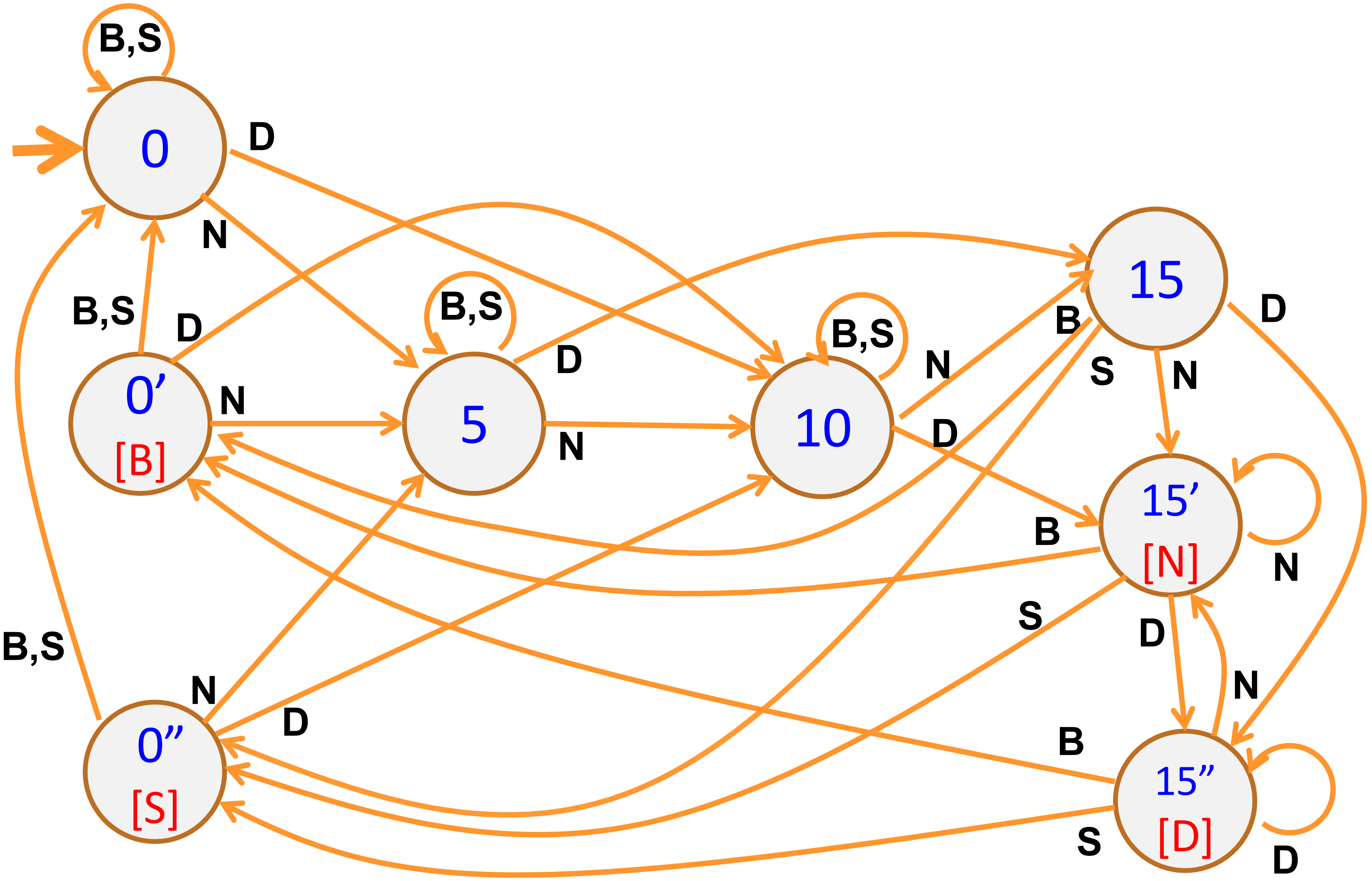
Basic transitions on **N** (nickel), **D** (dime), **B** (butterfinger), **S** (snickers)

Vending Machine, v0.2



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

Vending Machine, v1.0

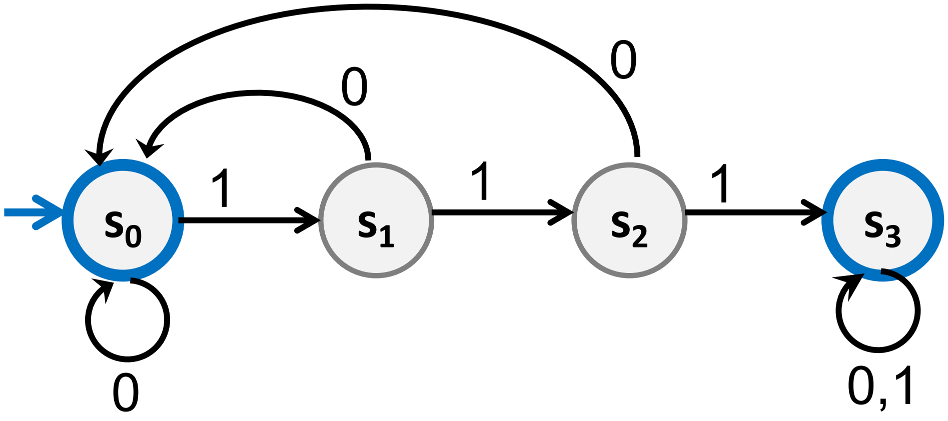


Adding additional “unexpected” transitions to cover all symbols for each state

Recall: Finite State Machines

- States
- Transitions on input symbols
- Start state and final states
- The "language recognized" by the machine is the set of strings that reach a final state from the start

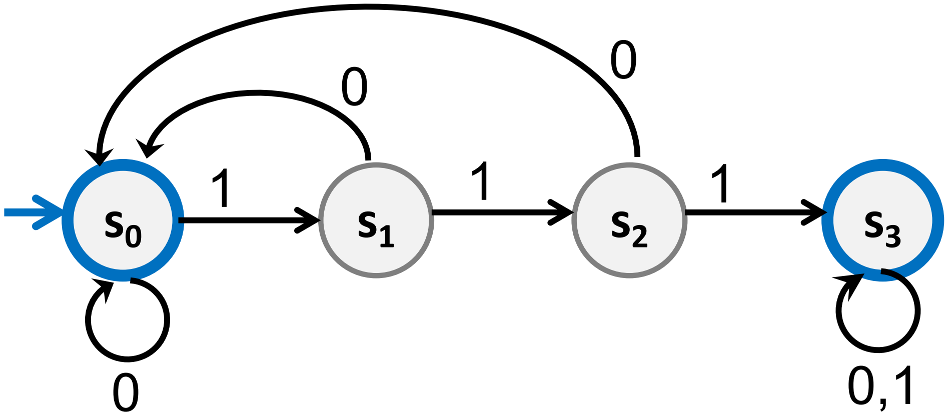
Old State	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_3
s_3	s_3	s_3



Recall: Finite State Machines

- Each machine designed for strings over some fixed alphabet Σ.
- Must have a transition defined from each state for ***every*** symbol in Σ.

Old State	0	1
s_0	s_0	s_1
s_1	s_0	s_2
s_2	s_0	s_3
s_3	s_3	s_3



State Minimization

- Many FSMs (DFAs) for the same problem
- Take a given FSM and try to reduce its state set by combining states
    - Algorithm will always produce the unique minimal equivalent machine (up to renaming of states) but we won't prove this

State Minimization Algorithm

- Put states into groups
- Try to find groups that can be collapsed into one state
  - states can keep track of information that isn't necessary to determine whether to accept or reject
- Group states together until we can *prove* that collapsing them can change the accept/reject result
  - find a specific string x such that:

    starting from state A, following edges according to x ends in accept

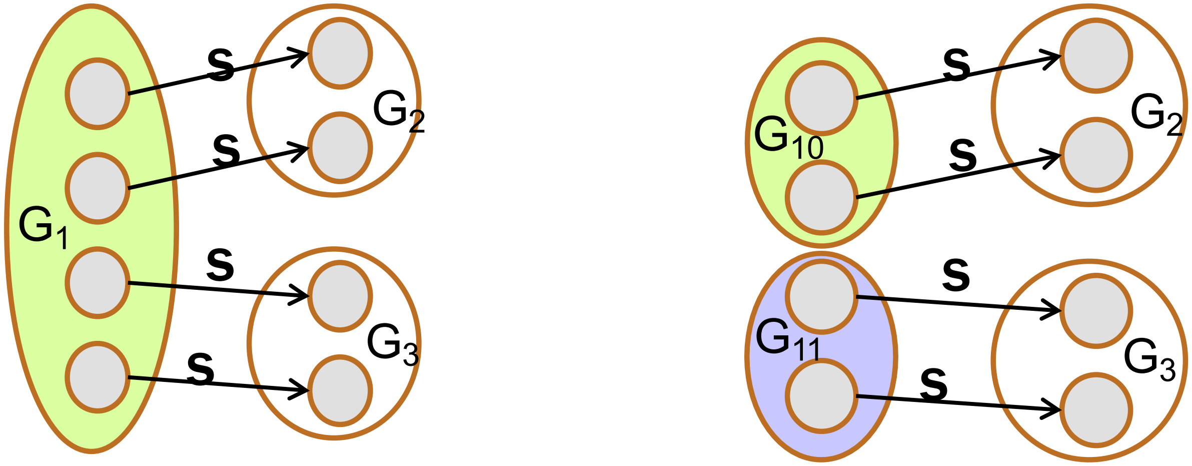
    starting from state B, following edges according to x ends in reject
  - (algorithm below could be modified to show these strings)

State Minimization Algorithm

1. Put states into groups based on their outputs (whether they accept or reject)

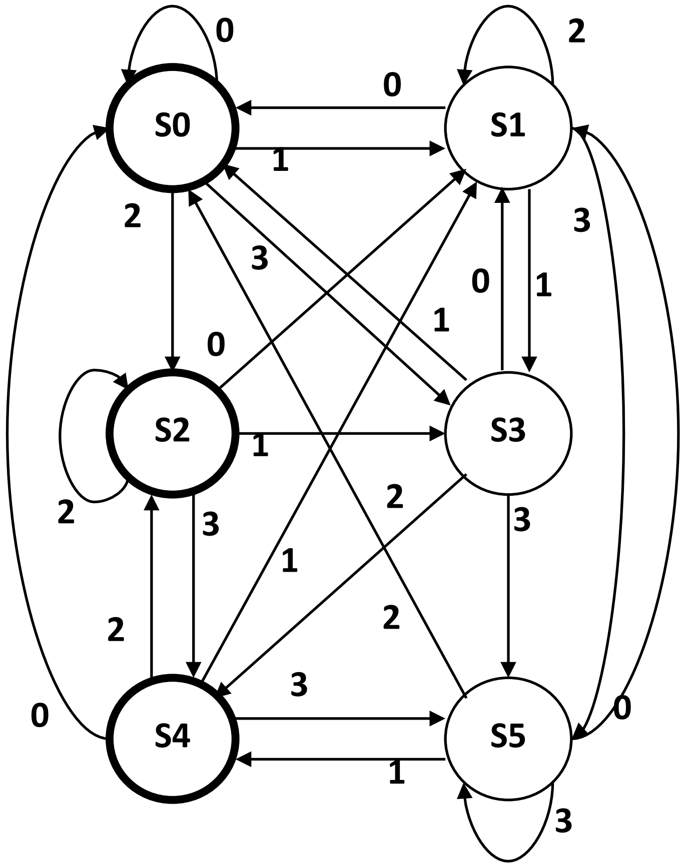
State Minimization Algorithm

1. Put states into groups based on their outputs (whether they accept or reject)
2. Repeat the following until no change happens
   a. If there is a symbol s so that not all states in a group G agree on which group s leads to, split G into smaller groups based on which group the states go to on s



3. Finally, convert groups to states

State Minimization Example

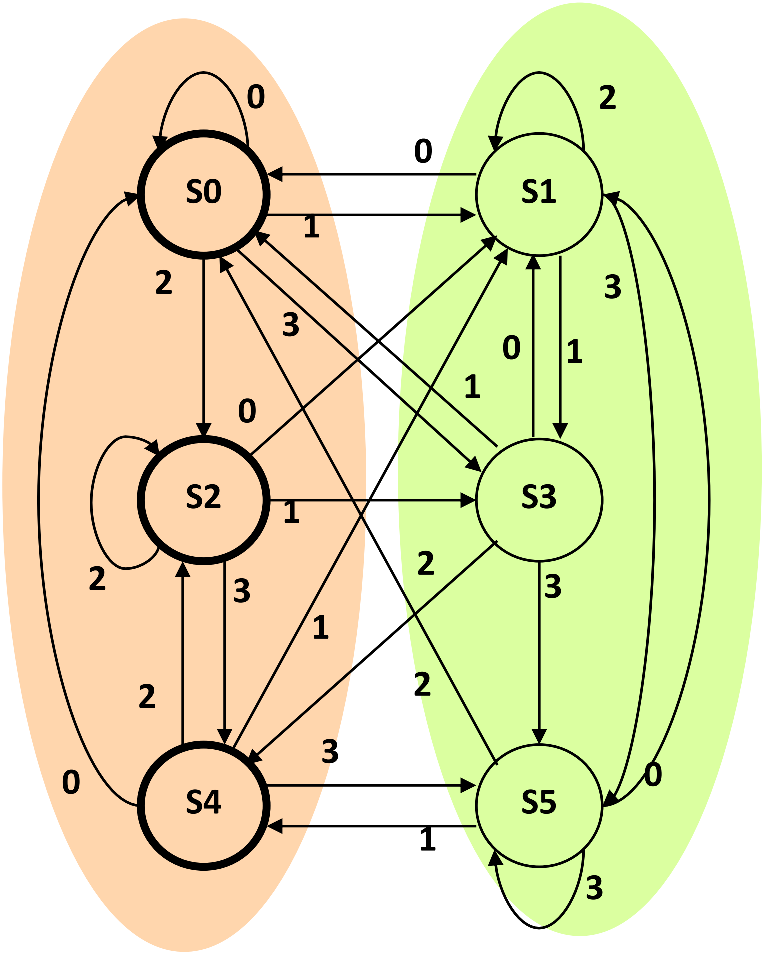


present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S5	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

state
transition table

Put states into groups based on their outputs (or whether they accept or reject)

State Minimization Example

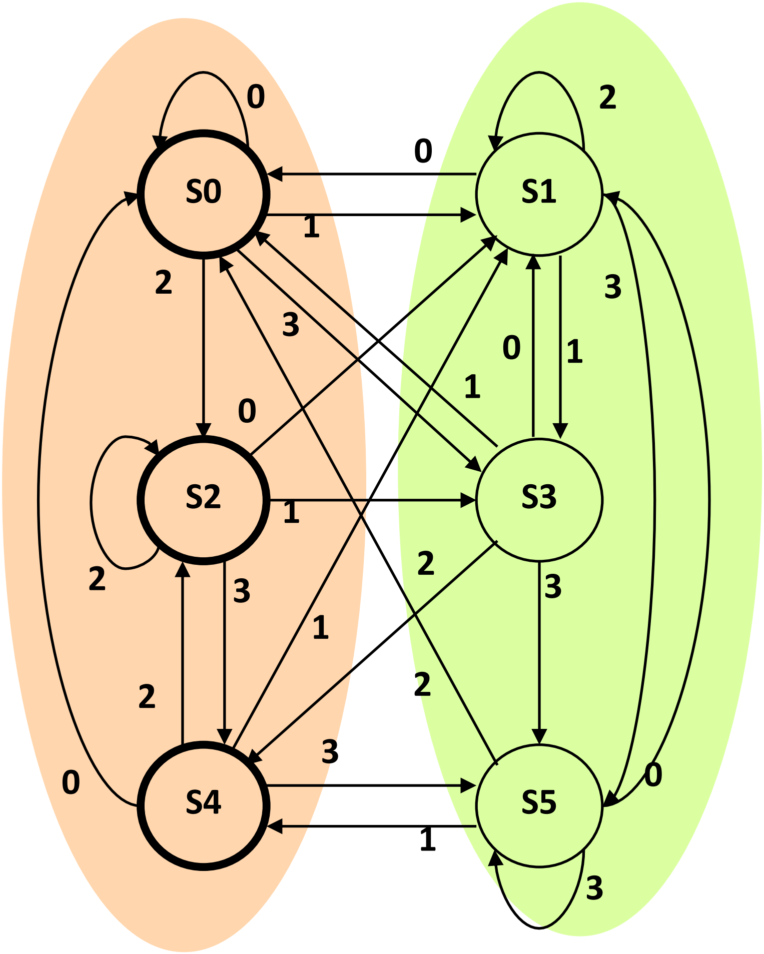


present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S5	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

state transition table

Put states into groups based on their outputs (or whether they accept or reject)

State Minimization Example



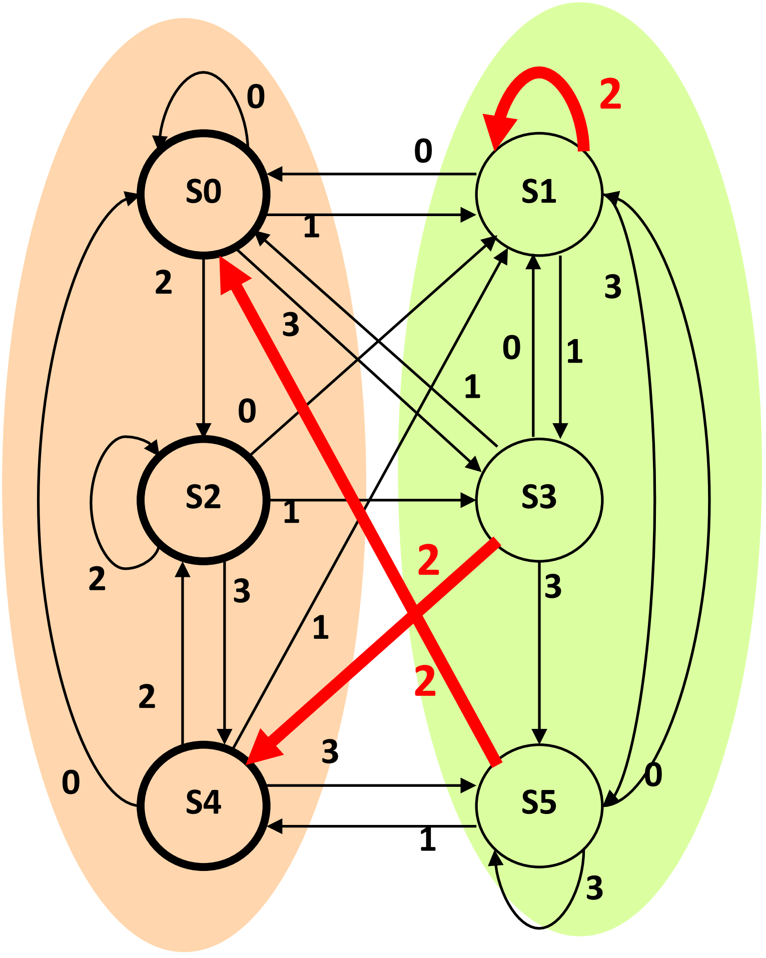
present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S5	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

state
transition table

Put states into groups based on their outputs (or whether they accept or reject)

If there is a symbol **s** so that not all states in a group **G** agree on which group **s** leads to, split **G** based on which group the states go to on **s**

State Minimization Example



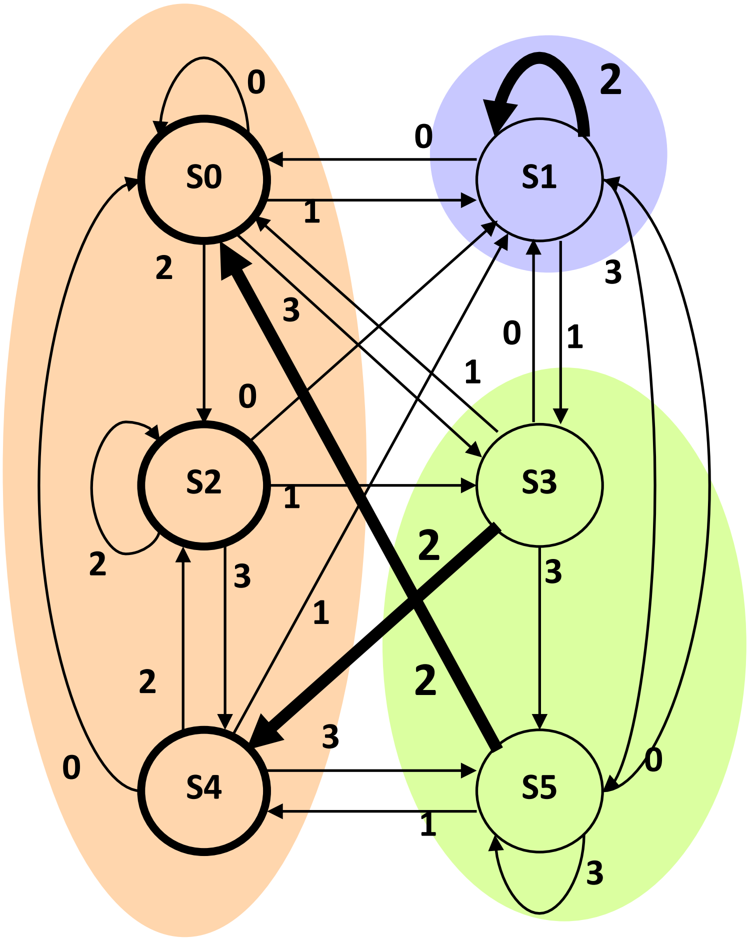
present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S5	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

state
transition table

Put states into groups based on their outputs (or whether they accept or reject)

If there is a symbol **s** so that not all states in a group **G** agree on which group **s** leads to, split **G** based on which group the states go to on **s**

State Minimization Example



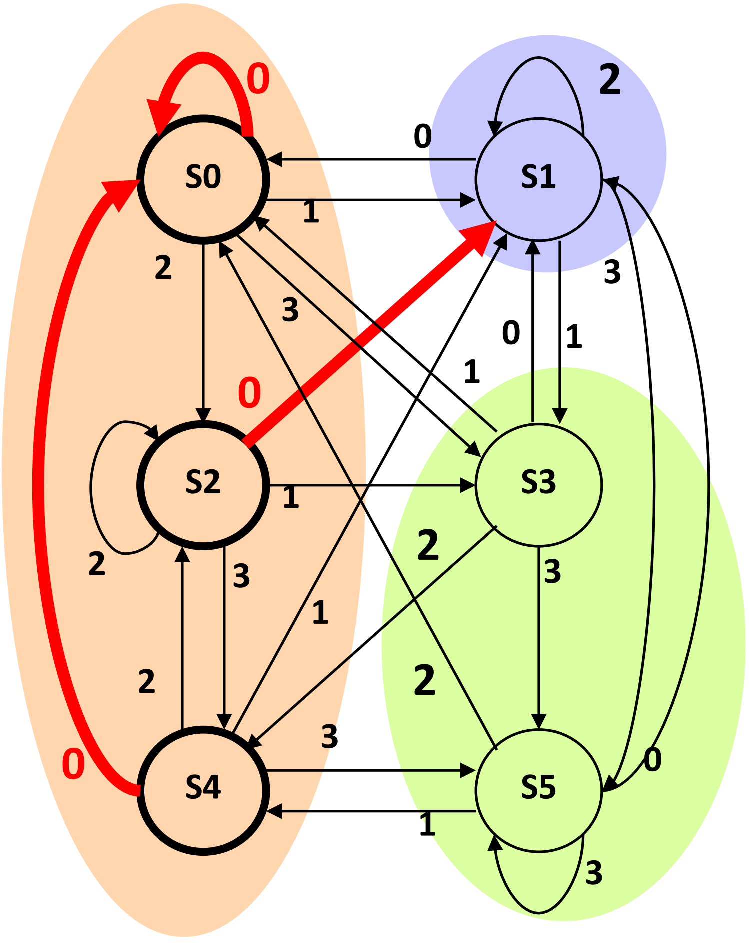
present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S5	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

state
transition table

Put states into groups based on their outputs (or whether they accept or reject)

If there is a symbol **s** so that not all states in a group **G** agree on which group **s** leads to, split **G** based on which group the states go to on **s**

State Minimization Example



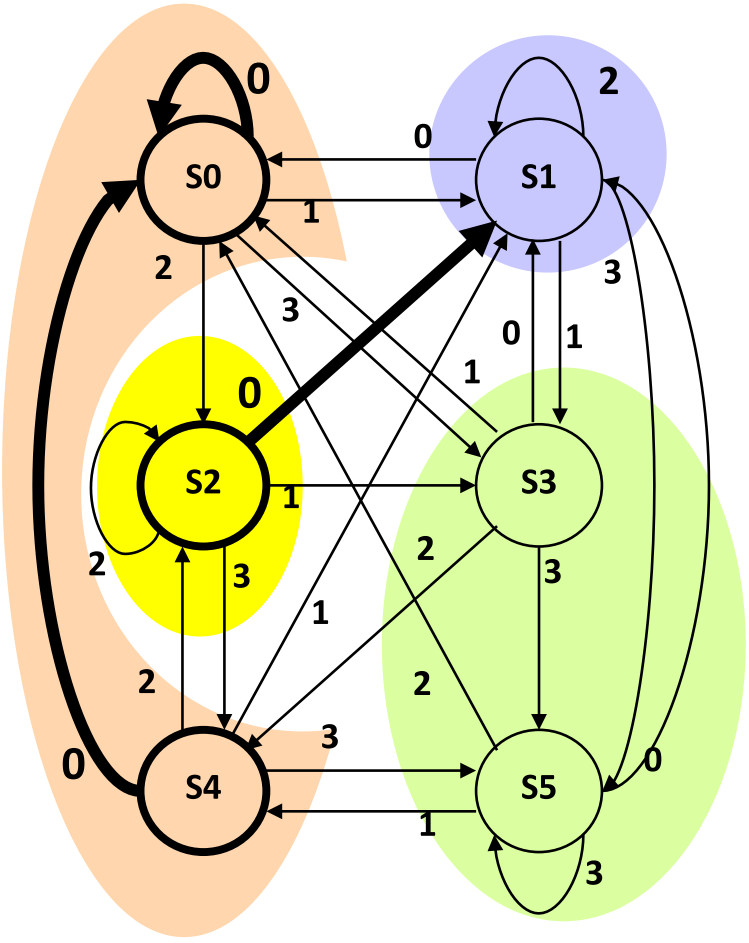
present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S5	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

state
transition table

Put states into groups based on their outputs (or whether they accept or reject)

If there is a symbol **s** so that not all states in a group **G** agree on which group **s** leads to, split **G** based on which group the states go to on **s**

State Minimization Example



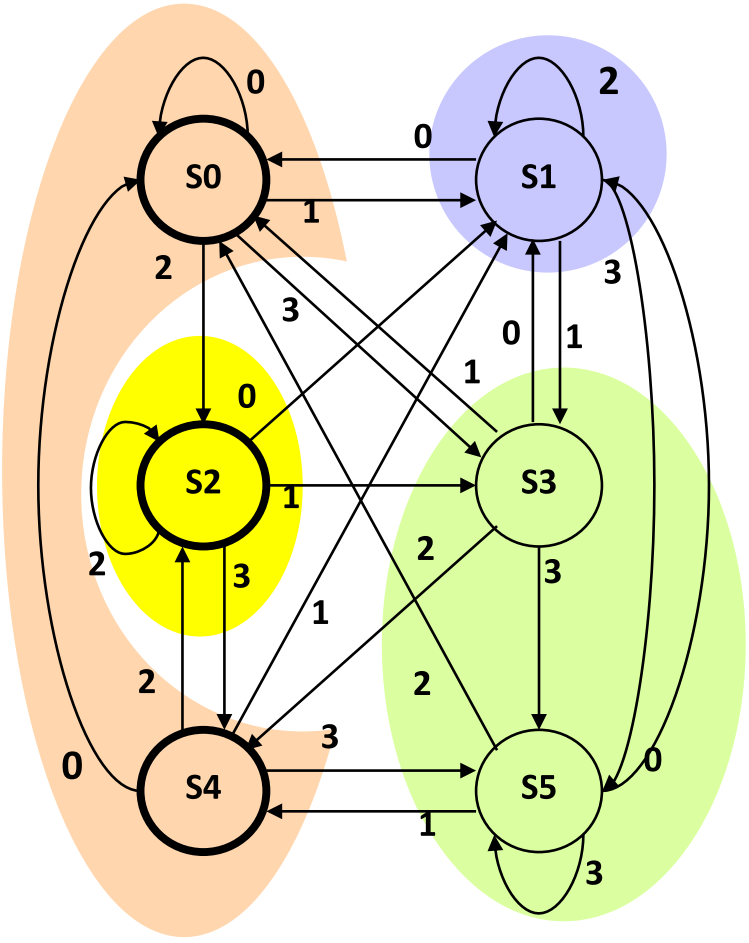
present state	next state				output
	0	1	2	3	
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S5	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

state
transition table

Put states into groups based on their outputs (or whether they accept or reject)

If there is a symbol **s** so that not all states in a group **G** agree on which group **s** leads to, split **G** based on which group the states go to on **s**

State Minimization Example



present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S5	0
S2	S1	S3	S2	S4	1
S3	S1	S0	S4	S5	0
S4	S0	S1	S2	S5	1
S5	S1	S4	S0	S5	0

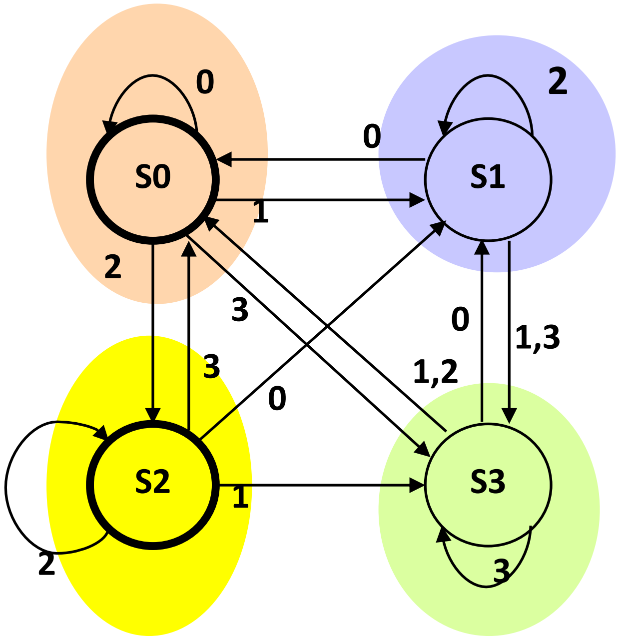
state
transition table

Finally convert groups to states:

Can combine states S0-S4 and S3-S5.

In table replace all S4 with S0 and all S5 with S3

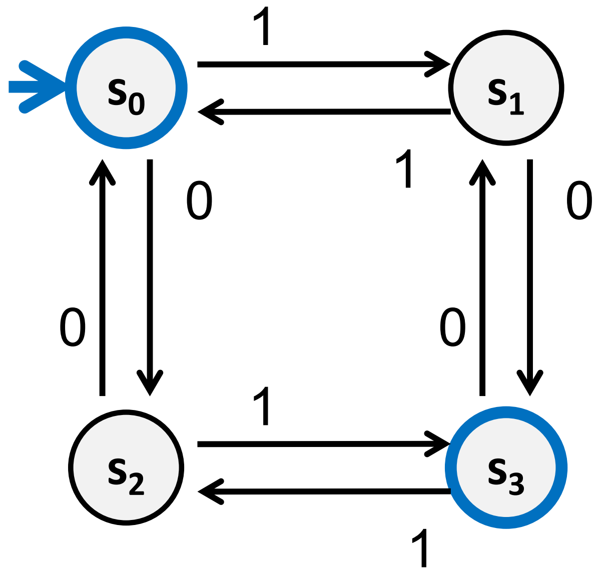
Minimized Machine



present state	next state 0	1	2	3	output
S0	S0	S1	S2	S3	1
S1	S0	S3	S1	S3	0
S2	S1	S3	S2	S0	1
S3	S1	S0	S0	S3	0

state
transition table

A Simpler Minimization Example



#0s is even

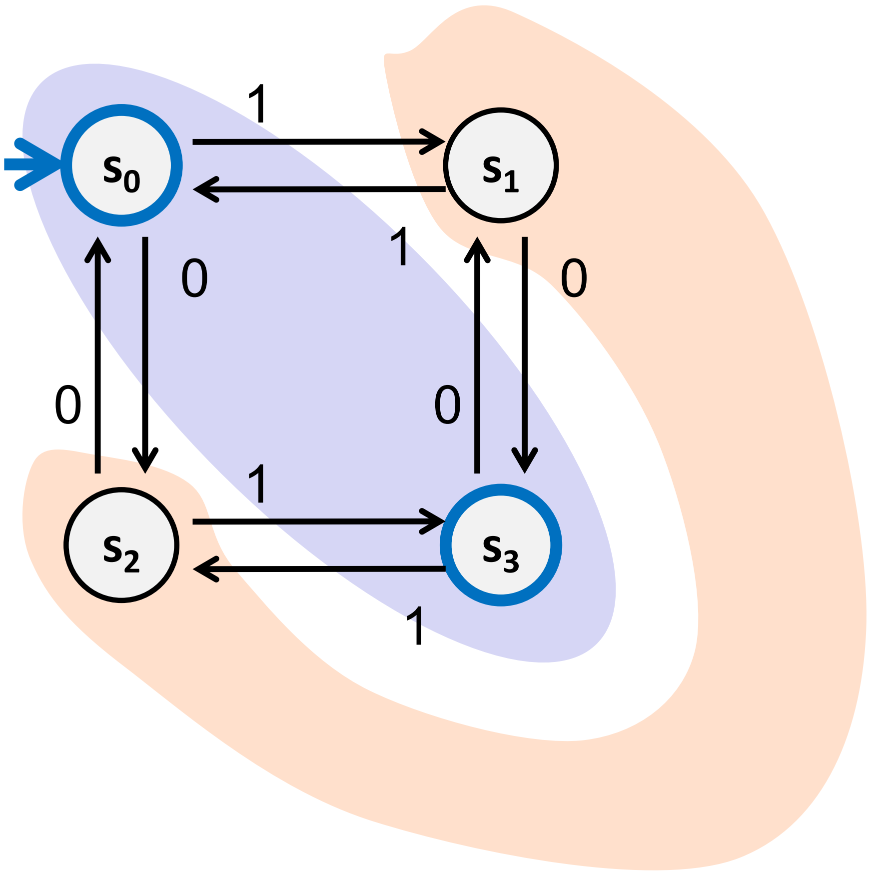
#0s is odd

#1s is even

#1s is odd

The set of all binary strings with # of 1's ≡ # of 0's (mod 2).

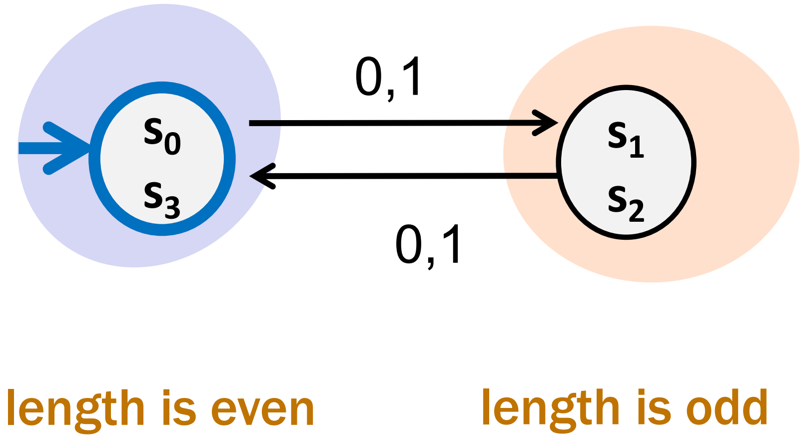
A Simpler Minimization Example



Split states into accept/reject groups

Every symbol causes the DFA to go from one group to the other so neither group needs to be split

Minimized DFA

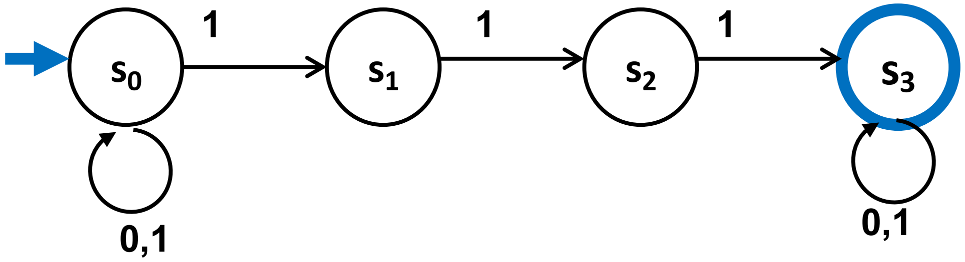


The set of all binary strings with # of 1’s $\equiv$ # of 0’s (mod 2).

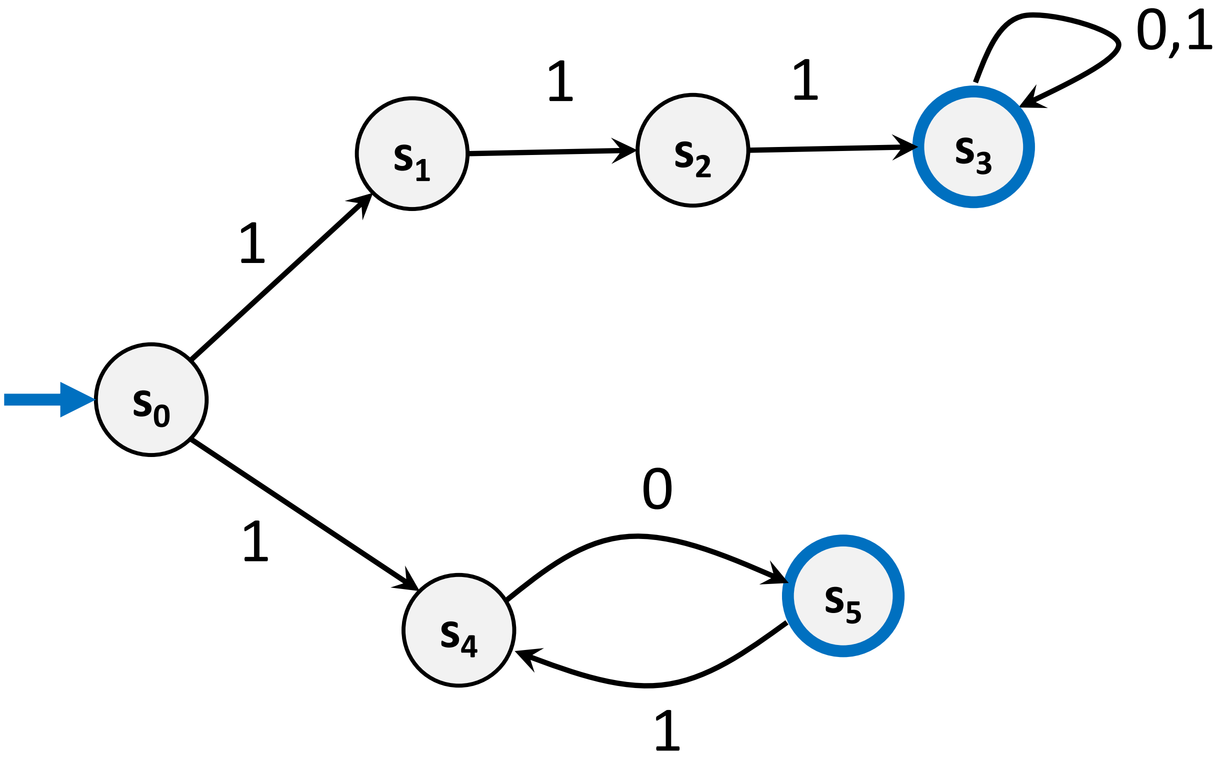
= The set of all binary strings with even length.

Nondeterministic Finite Automata (NFA)

- Graph with start state, final states, edges labeled by symbols (like DFA) but
  - Not required to have exactly 1 edge out of each state labeled by each symbol— can have 0 or >1
  - Also can have edges labeled by empty string ε
- **Definition:** x is in the language recognized by an NFA if and only if <u>some</u> valid execution of the machine gets to an accept state

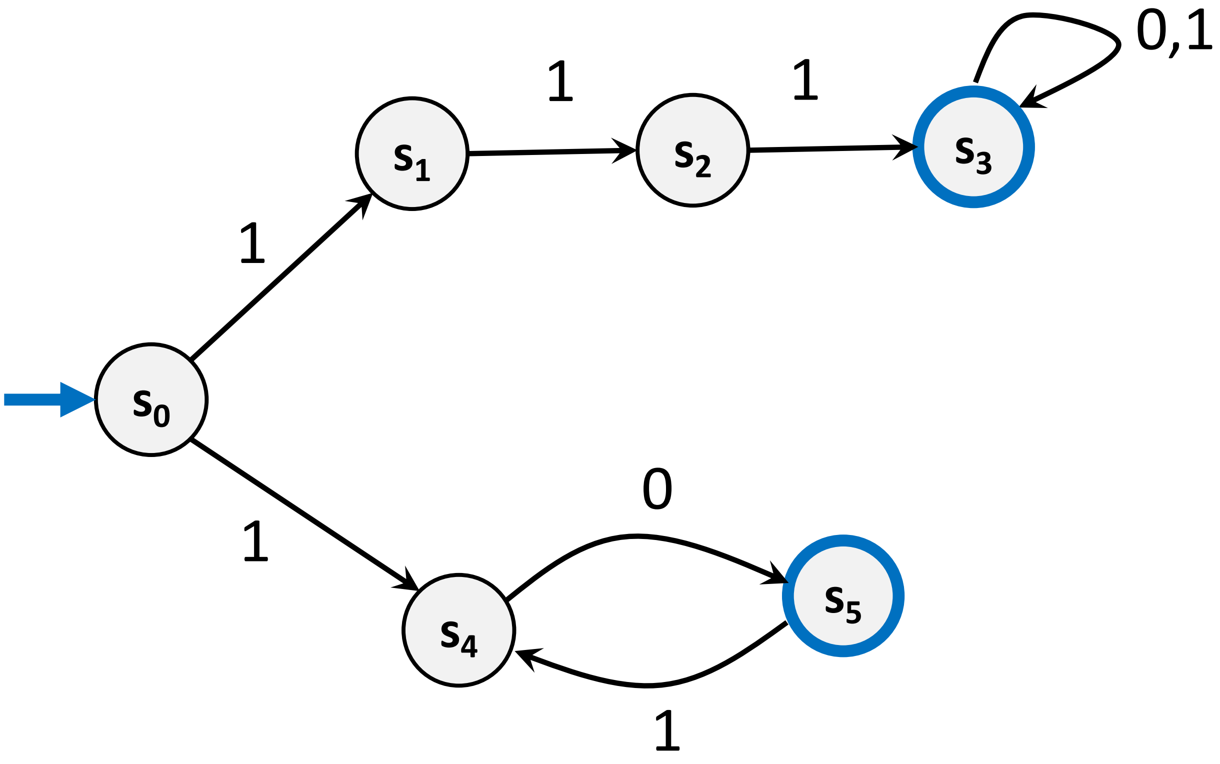


Consider This NFA



What language does this NFA accept?

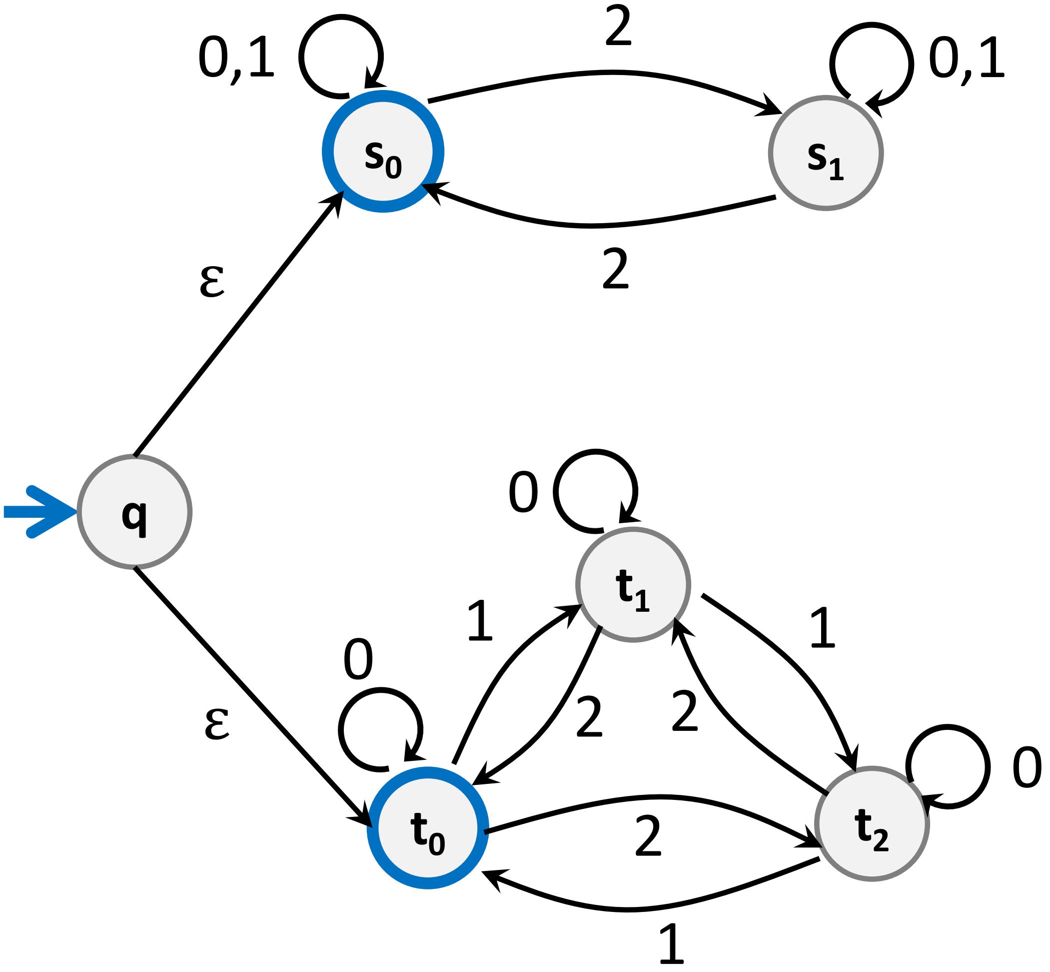
Consider This NFA



What language does this NFA accept?

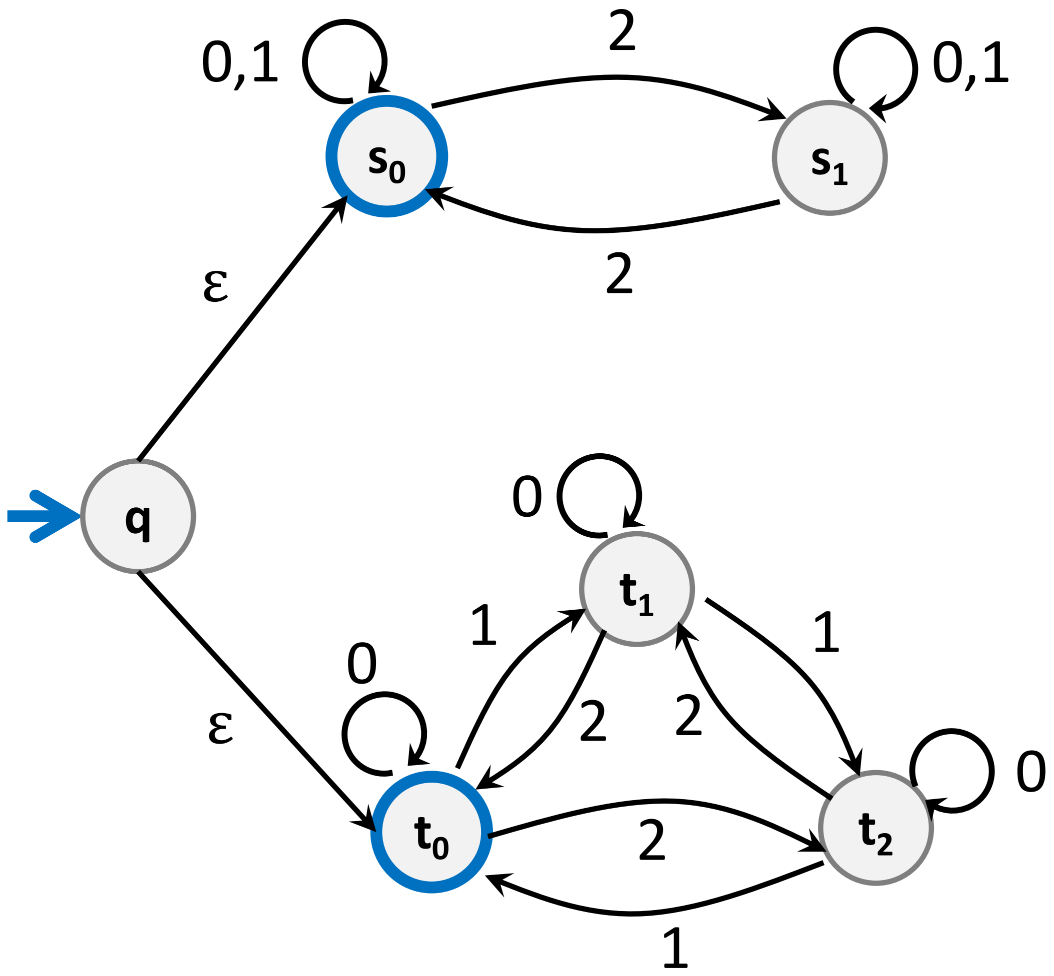
10(10)* U 111 (0 U 1)*

NFA ε-moves



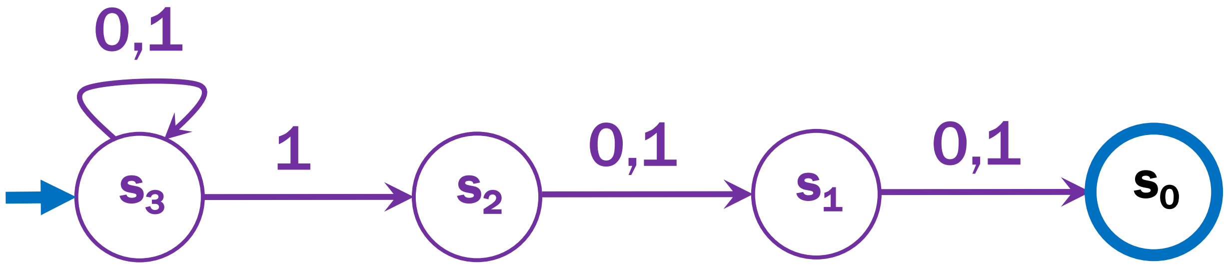
NFA ε-moves

Strings over {0,1,2} w/even # of 2’s OR sum to 0 mod 3

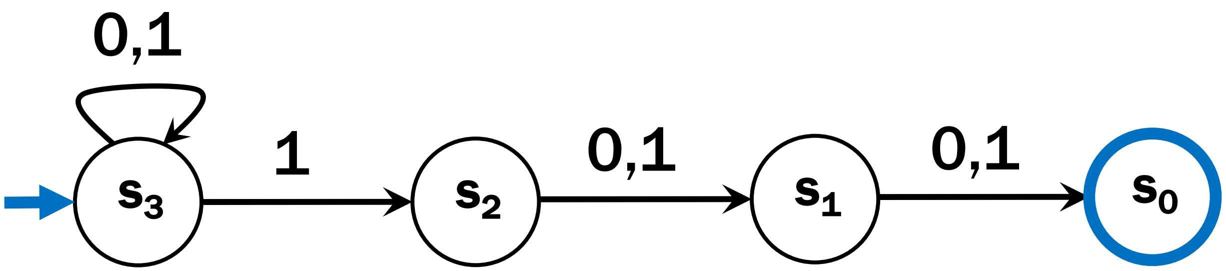


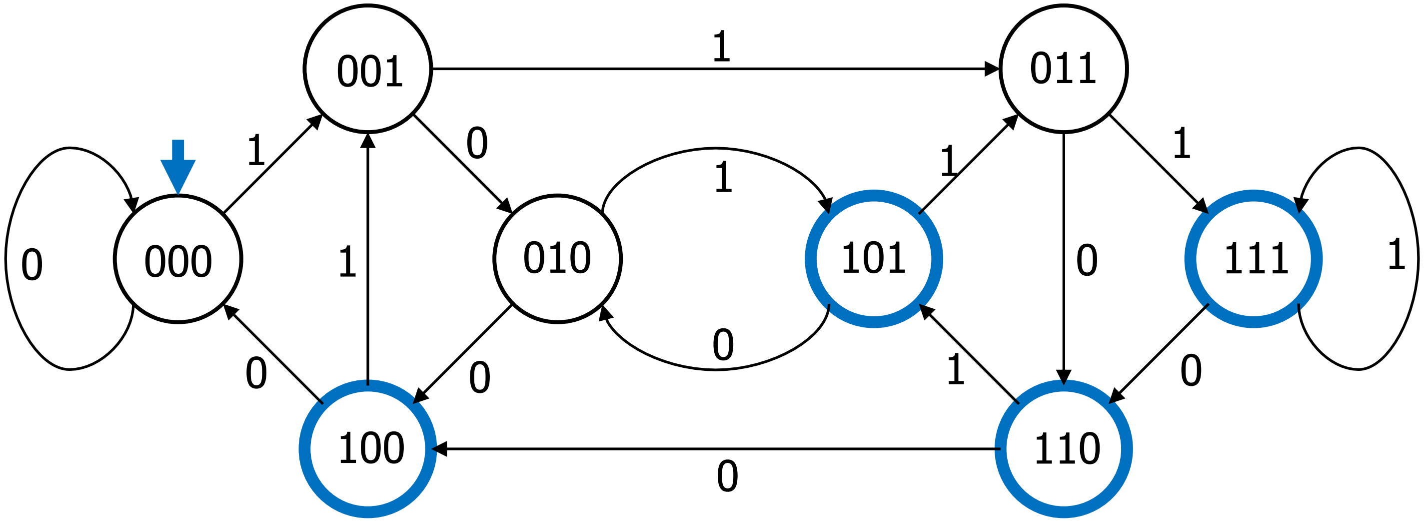
NFA for set of binary strings with a 1 in the 3rd position from the end

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



Compare with the smallest DFA





Summary of NFAs

- Generalization of DFAs
  - drop two restrictions of DFAs
  - every DFA <u>is</u> an NFA
- *Seem* to be more powerful
  - designing is easier than with DFAs
- *Seem* related to regular expressions

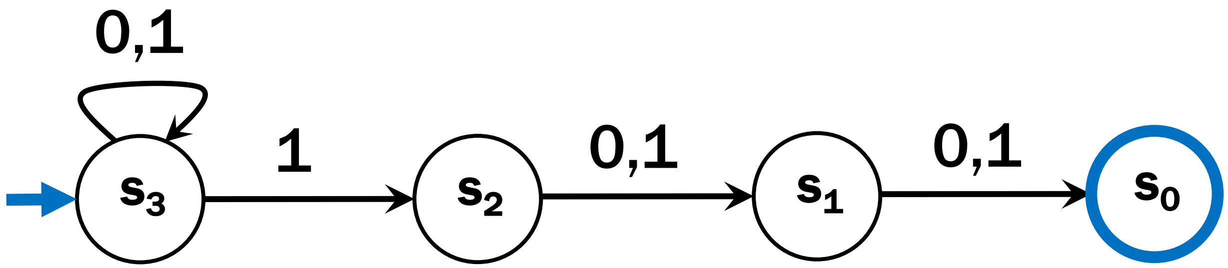
Three ways of thinking about NFAs

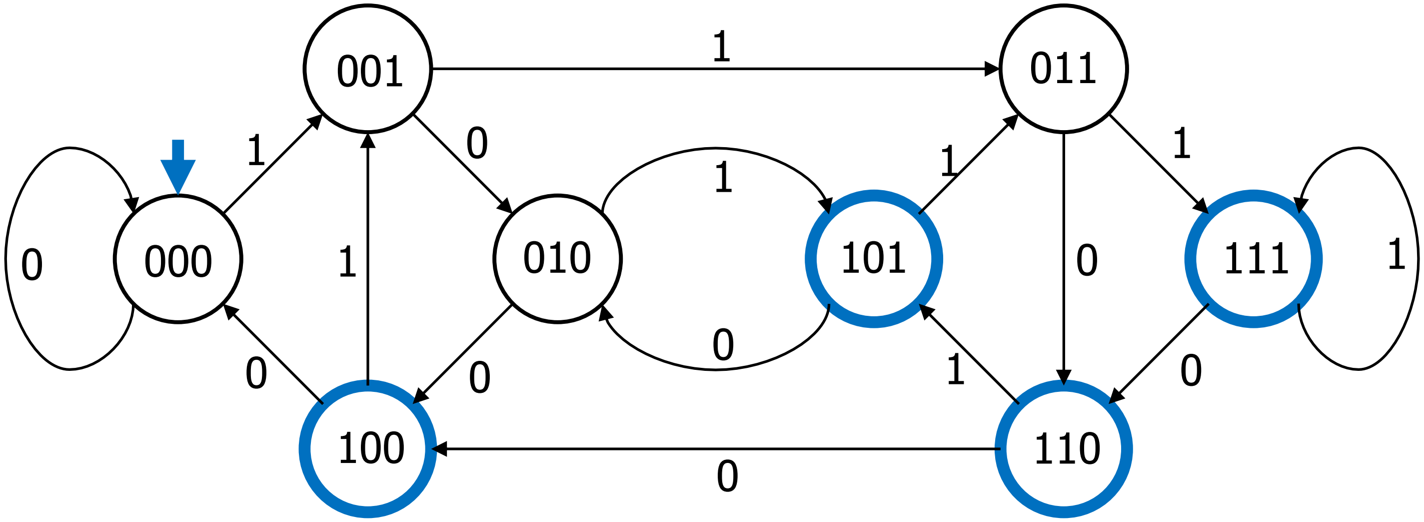
- Perfect guesser: The NFA has input x and whenever there is a choice of what to do it magically guesses a good one (if one exists)

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

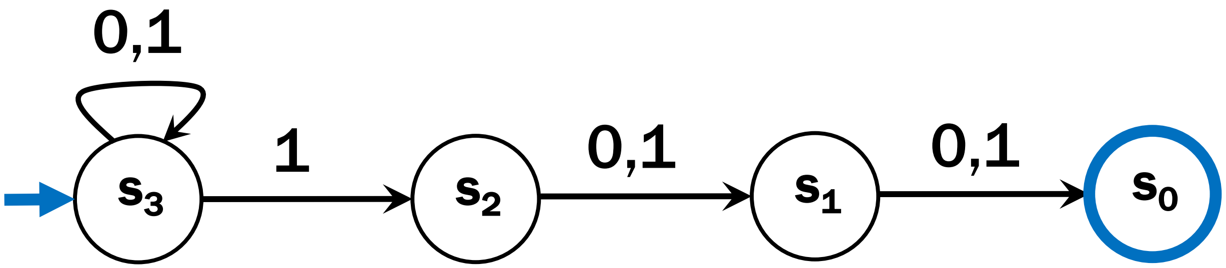
- Outside observer: Is there a path labeled by x from the start state to some accepting state?

Compare with the smallest DFA

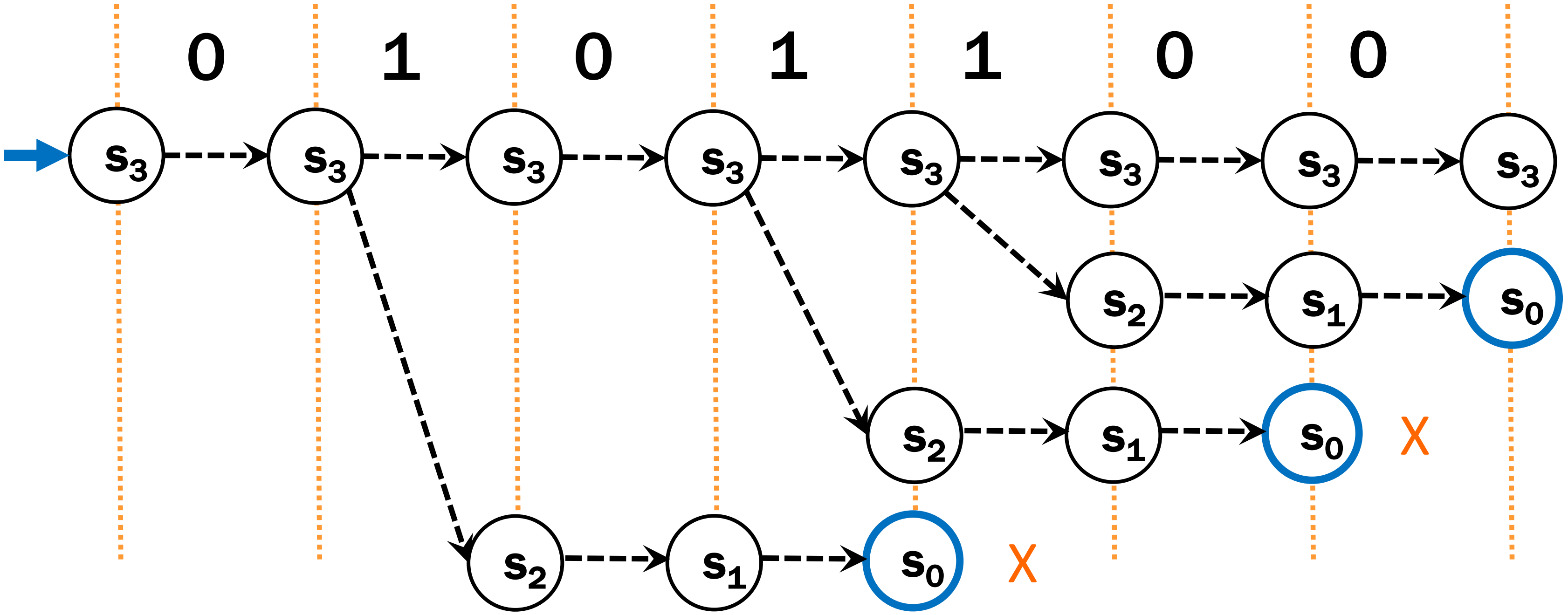




Parallel Exploration view of an NFA



Input string 0101100



Three ways of thinking about NFAs

- Perfect guesser: The NFA has input x and whenever there is a choice of what to do it magically guesses a good one (if one exists)

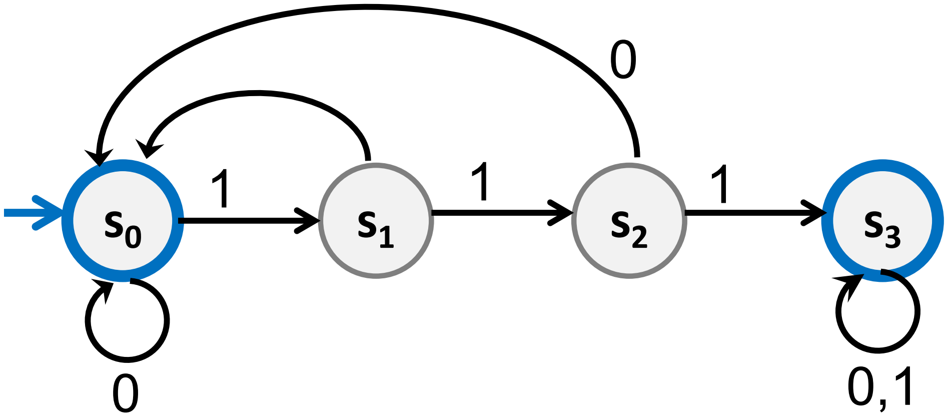
- Parallel exploration: The NFA computation runs all possible computations on x step-by-step at the same time in parallel

- Outside observer: Is there a path labeled by x from the start state to some accepting state?

Path Labels

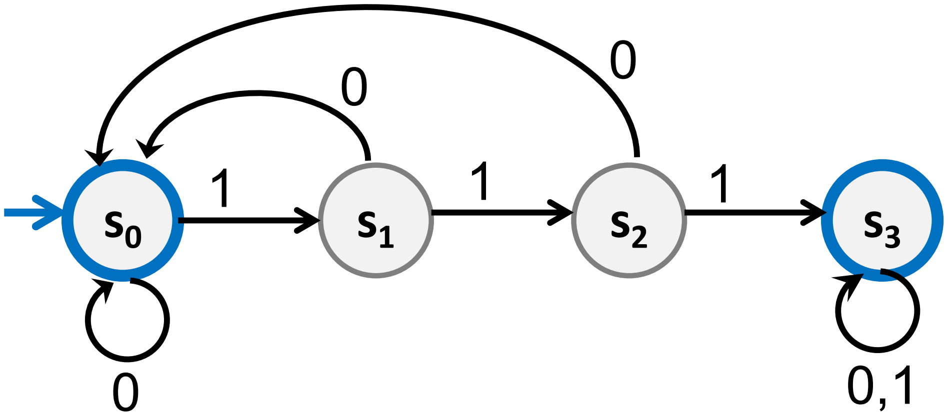
Def: The label of path $v_0, v_1, \ldots, v_n$ is the <u>concatenation</u> of the labels of the edges $(v_0, v_1), (v_1, v_2), \ldots, (v_{n-1}, v_n)$

Example: The label of path s_0, s_1, s_2, s_0, s_0 is 1100



Deterministic Finite Automata (DFA)

- **Theorem:** x is in the language recognized by an DFA if and only if x labels a path from the start state to some final state

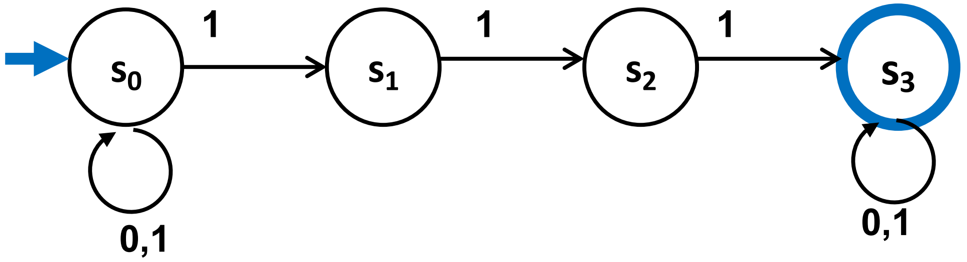


- Path $v_0, v_1, \ldots, v_n$ with $v_0 = s_0$ and label x describes a correct simulation of the DFA on input x
  - i-th step must match the i-th character of x

Nondeterministic Finite Automata (NFA)

- Graph with start state, final states, edges labeled by symbols (like DFA) but
  - Not required to have exactly 1 edge out of each state labeled by each symbol— can have 0 or >1
  - Can also have edges labeled by empty string ε

- **Theorem:** x is in the language recognized by an NFA if and only if x labels <u>some</u> **path** from the start state to an accepting state



Summary of NFAs

- Generalization of DFAs
  - drop two restrictions of DFAs
  - every DFA <u>is</u> an NFA

- *Seem* to be more powerful
  - designing is easier than with DFAs

- *Seem* related to regular expressions

The story so far...

REs $\subseteq$ **CFGs**

DFAs $\subseteq$ **NFAs**

NFAs and regular expressions

Theorem: For any set of strings (language) A described by a regular expression, there is an NFA that recognizes A.

Proof idea: Structural induction based on the recursive definition of regular expressions...

Regular Expressions over Σ

- **Basis:**
  - ε is a regular expression
  - ***a*** is a regular expression for any *a* $\in \Sigma$

- **Recursive step:**
  - If **A** and **B** are regular expressions, then so are:

    A $\cup$ B

    AB

    A*

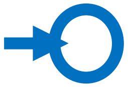
Base Case

- Case ε:

- Case ***a***:

Base Case

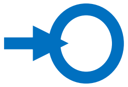
- Case ε:



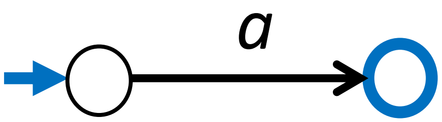
- Case ***a***:

Base Case

- Case ε:



- Case ***a***:



Regular Expressions over Σ

- **Basis:**
  - ε is a regular expression
  - ***a*** is a regular expression for any *a* ∈ Σ

- **Recursive step:**
  - If **A** and **B** are regular expressions, then so are:

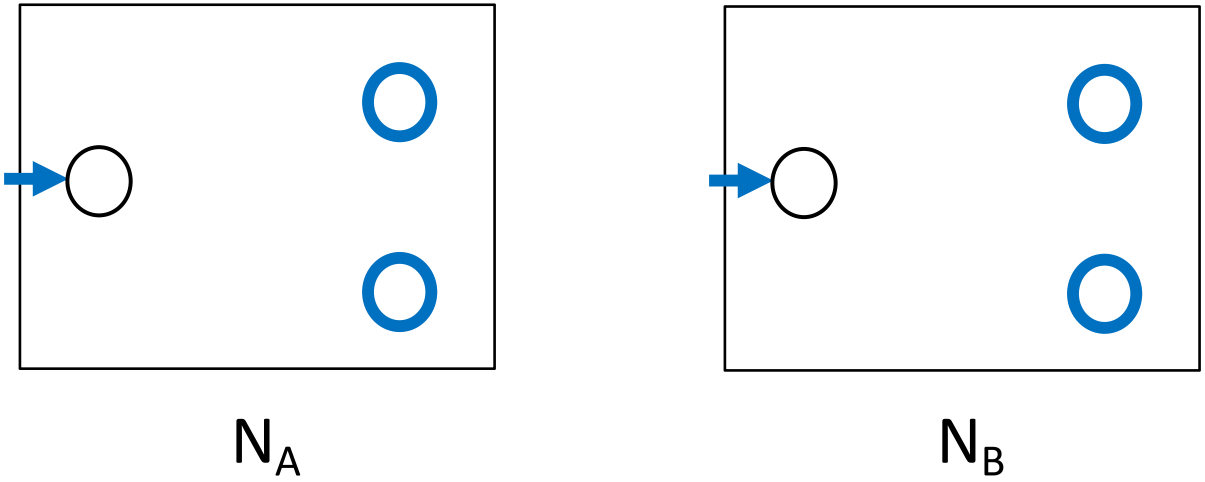
    A ∪ B

    AB

    A*

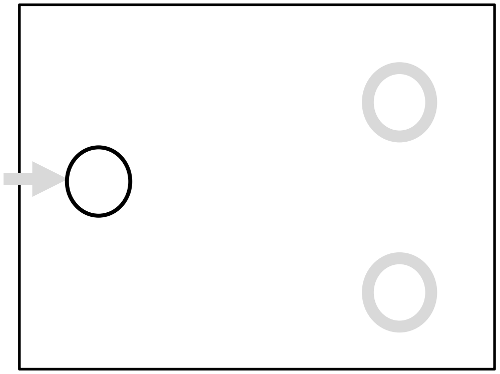
Inductive Hypothesis

- Suppose that for some regular expressions A and B there exist NFAs N_A and N_B such that N_A recognizes the language given by A and N_B recognizes the language given by B

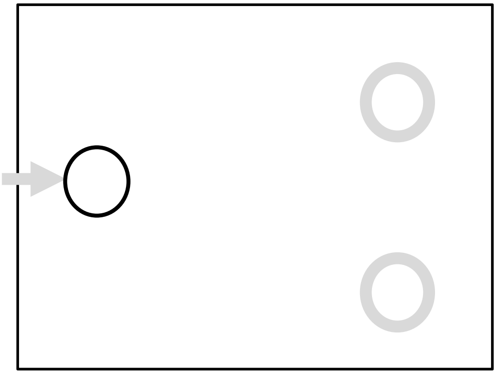


Inductive Step

Case A ∪ B:



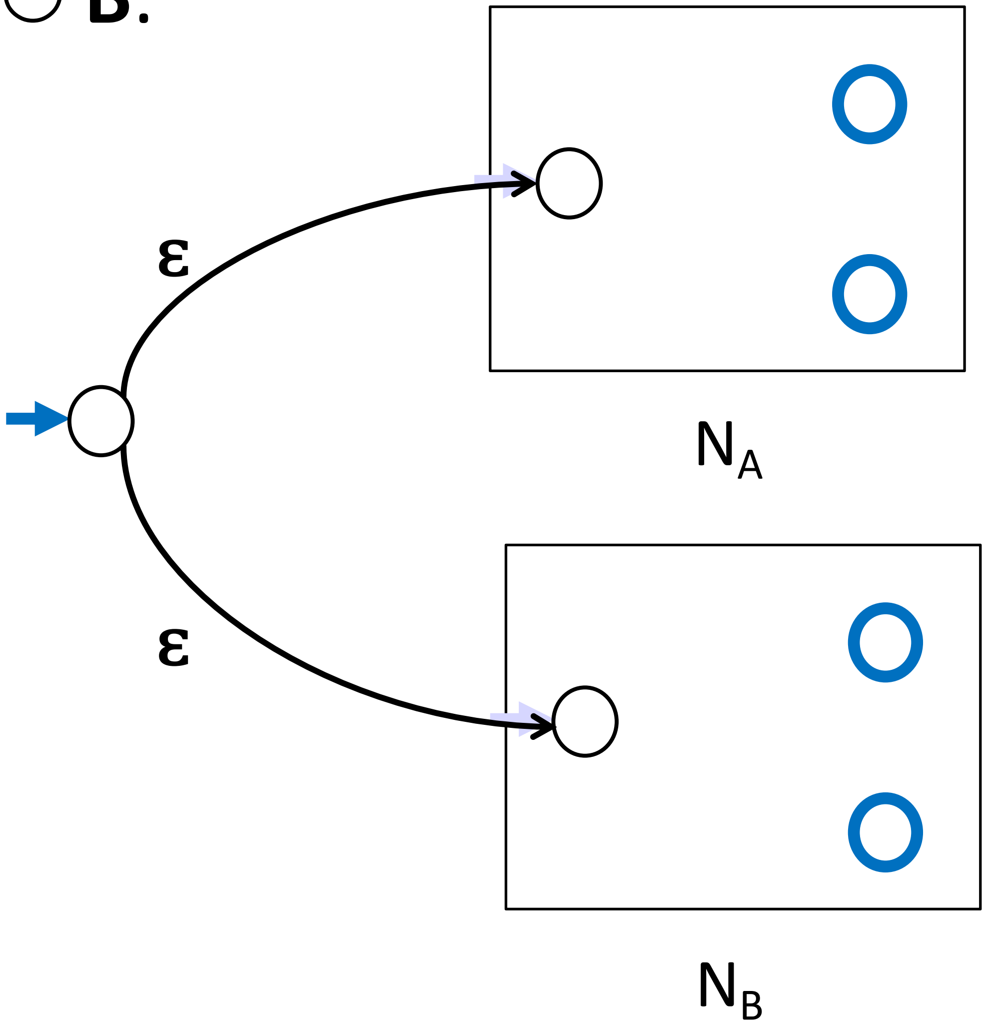
N_A



N_B

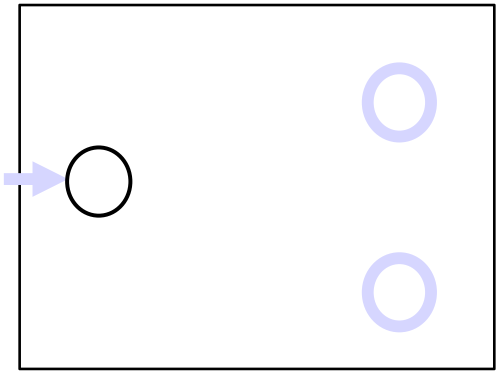
Inductive Step

Case A ∪ B:

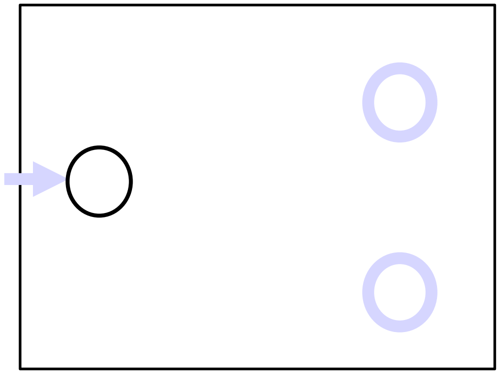


Inductive Step

Case AB:



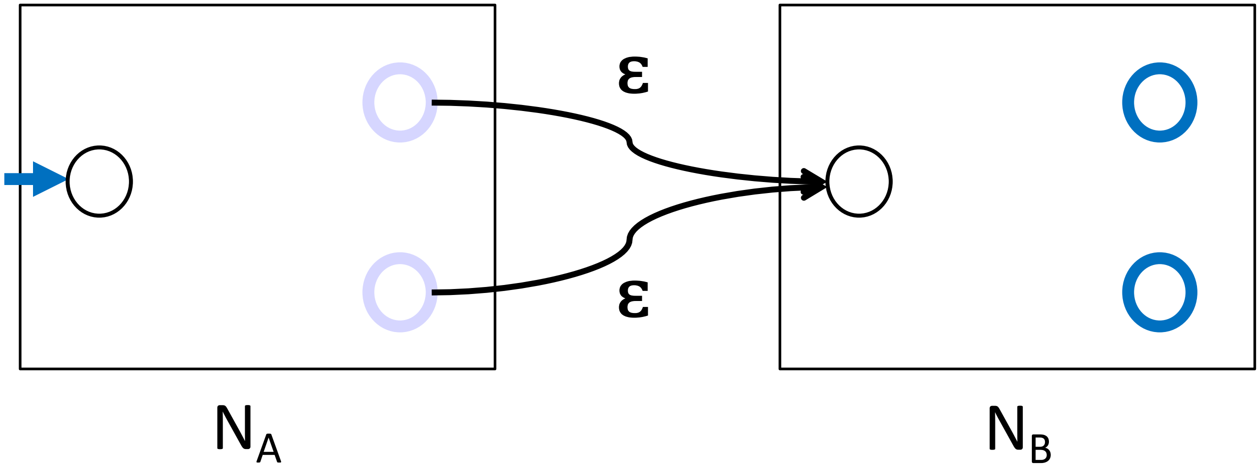
N_A



N_B

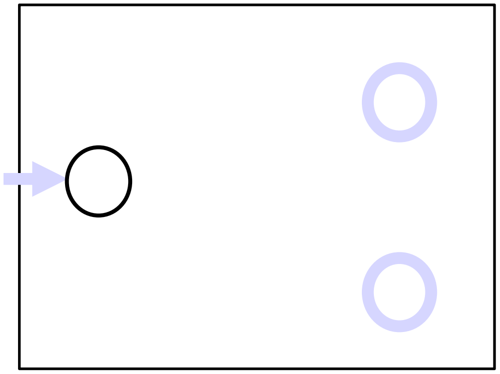
Inductive Step

Case AB:



Inductive Step

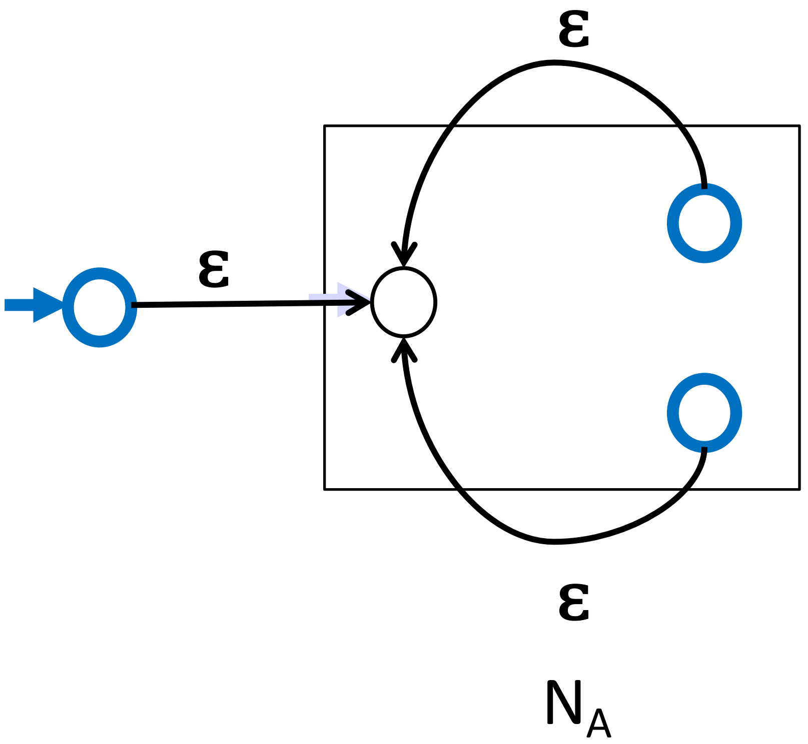
Case A*



N_A

Inductive Step

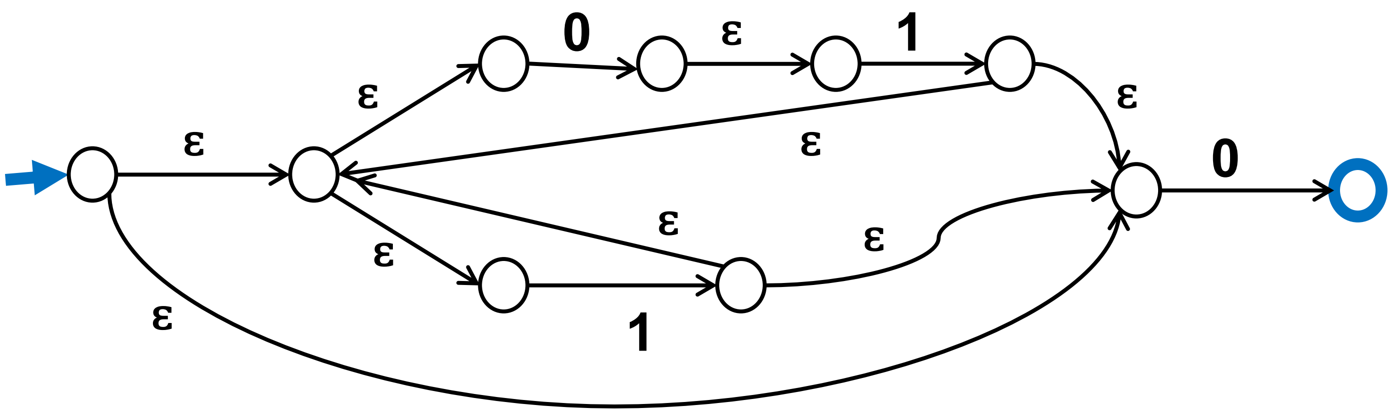
Case A*



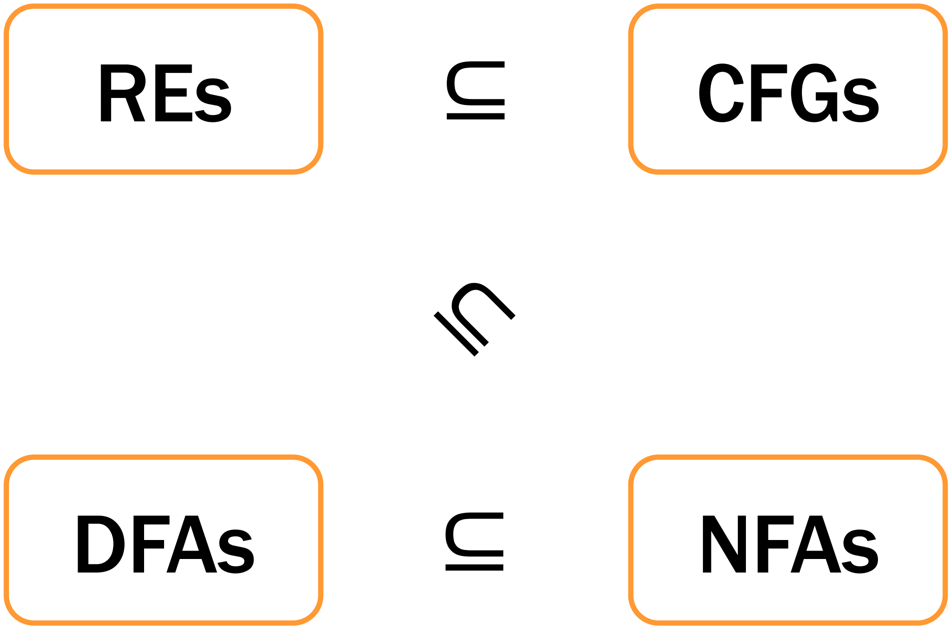
Build an NFA for $(01 \cup 1)^*0$

Solution

(01 ∪1)*0



The story so far...



REs ⊆ CFGs

DFAs ⊆ NFAs

NFAs and DFAs

Every DFA is an NFA

- DFAs have requirements that NFAs don’t have

Can NFAs recognize more languages?

NFAs and DFAs

Every DFA **is** an NFA

- DFAs have requirements that NFAs don't have

Can NFAs recognize more languages? No!

Theorem: For every NFA there is a DFA that recognizes exactly the same language

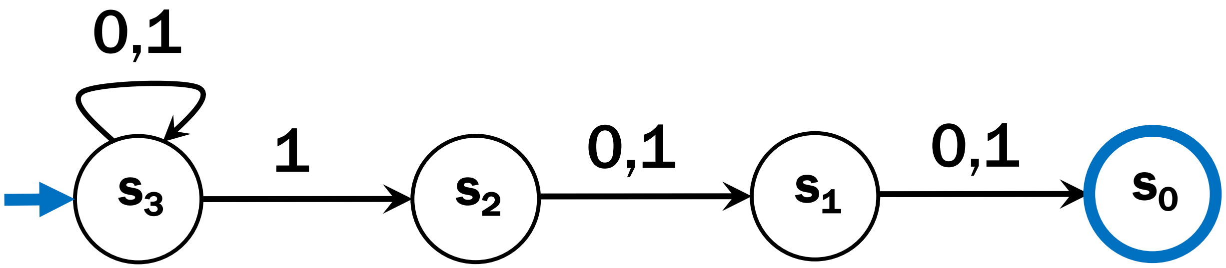
Three ways of thinking about NFAs

- Perfect guesser: The NFA has input x and whenever there is a choice of what to do it magically guesses a good one (if one exists)

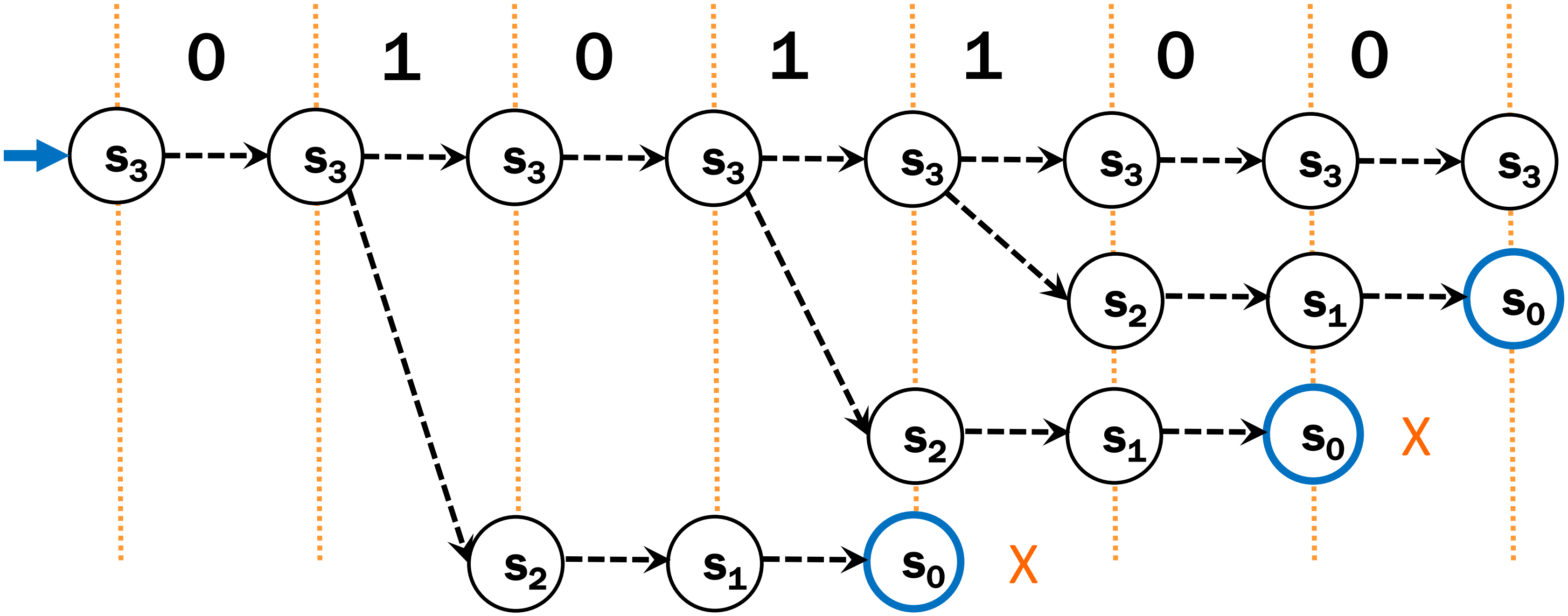
- Parallel exploration: The NFA computation runs all possible computations on x step-by-step at the same time in parallel

- Outside observer: Is there a path labeled by x from the start state to some final state?

Parallel Exploration view of an NFA



Input string 0101100



Conversion of NFAs to a DFAs

- **Construction Idea:**
  - **The DFA keeps track of ALL states reachable in the NFA along a path labeled by the input so far**

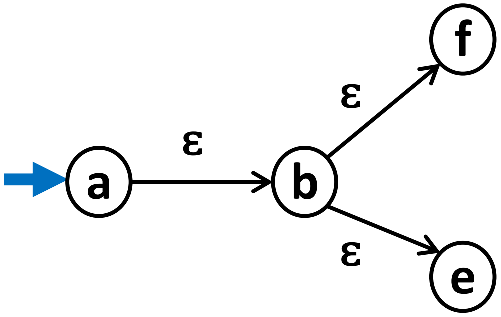
    (Note: not all *paths*; all *last states* on those paths.)

  - **There will be one state in the DFA for each *subset* of states of the NFA that can be reached by some string**

Conversion of NFAs to a DFAs

New start state for DFA

- The set of all states reachable from the start state of the NFA using only edges labeled ε



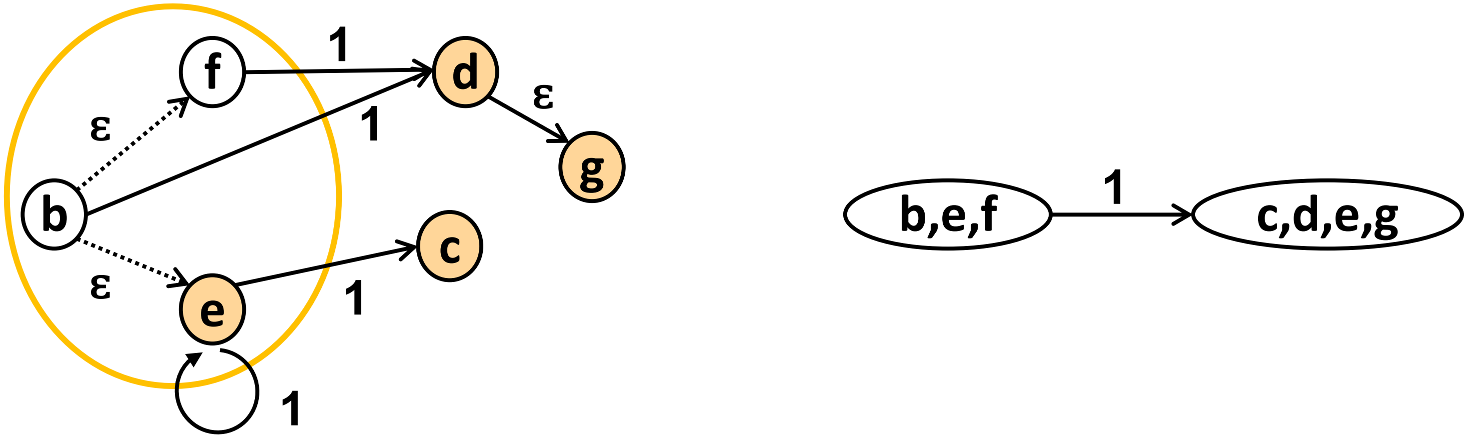
NFA

DFA

Conversion of NFAs to a DFAs

For each state of the DFA corresponding to a set S of states of the NFA and each symbol s

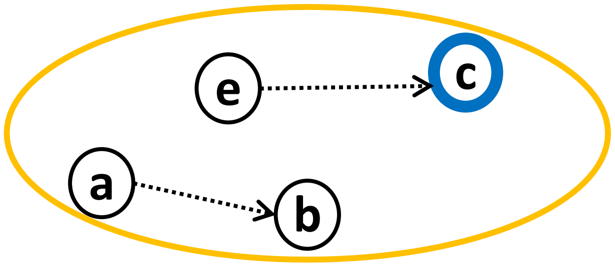
- **Add an edge labeled** s **to state corresponding to** T, **the set of states of the NFA reached by**
  - starting from some state in S, then
  - following one edge labeled by s, and then following some number of edges labeled by **ε**
- T **will be ∅ if no edges from** S **labeled** s **exist**



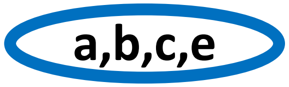
Conversion of NFAs to a DFAs

Final states for the DFA

- All states whose set contain some final state of the NFA

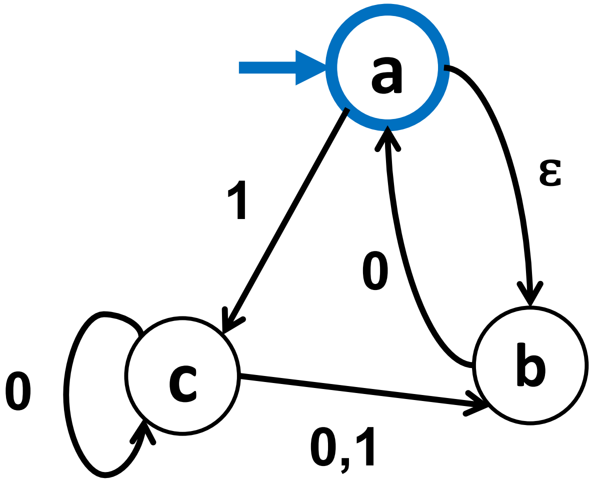


NFA

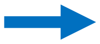


DFA

Example: NFA to DFA

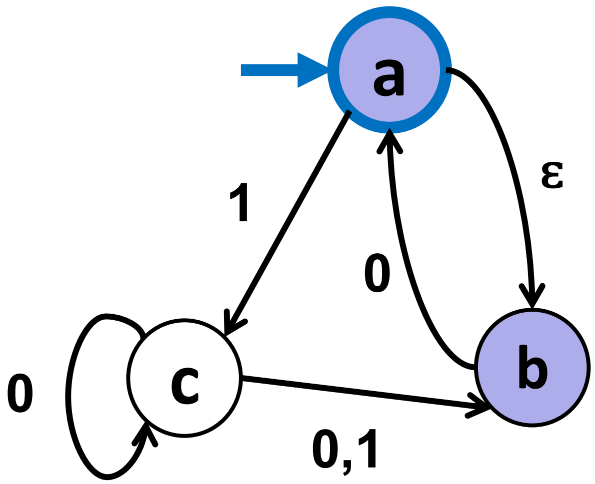


NFA

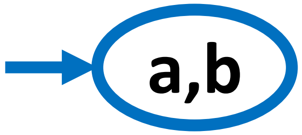


DFA

Example: NFA to DFA

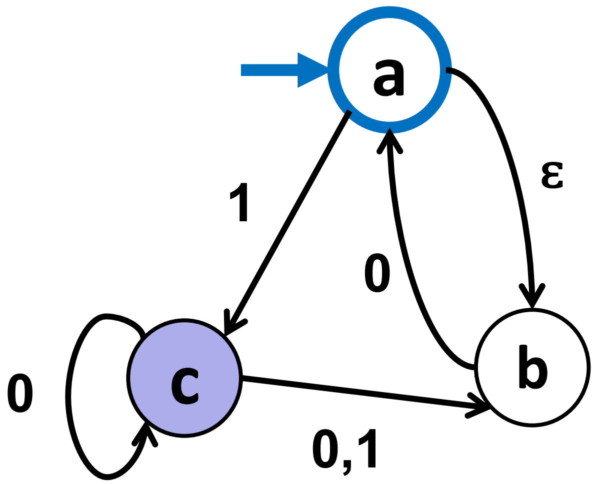


NFA

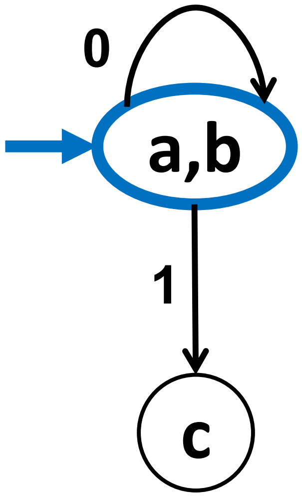


DFA

Example: NFA to DFA

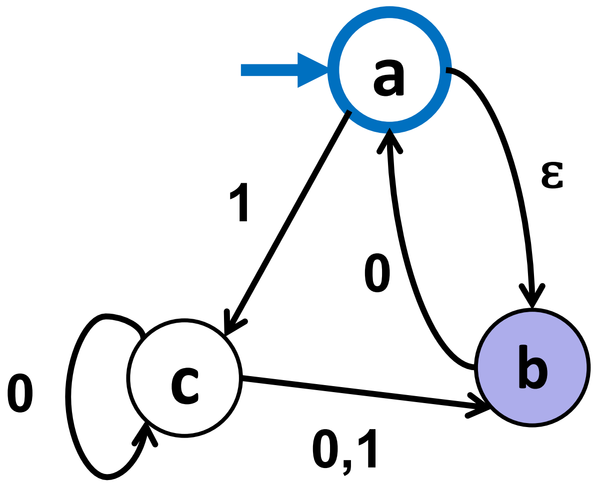


NFA

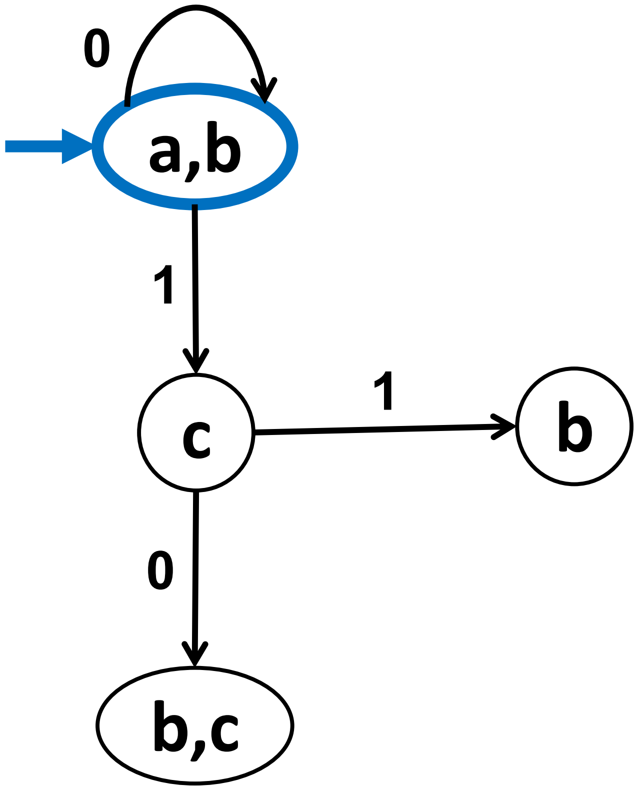


DFA

Example: NFA to DFA

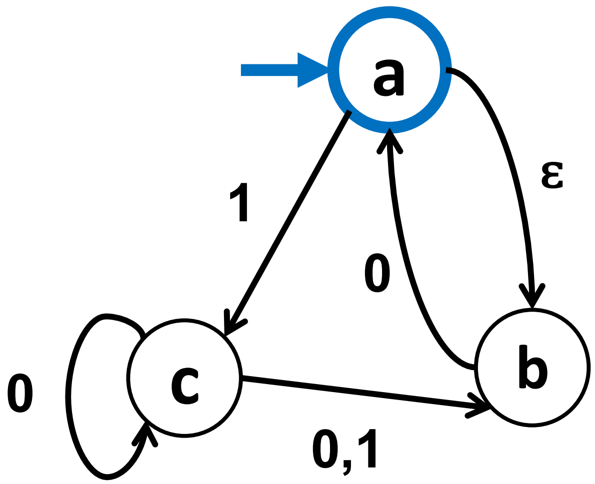


NFA

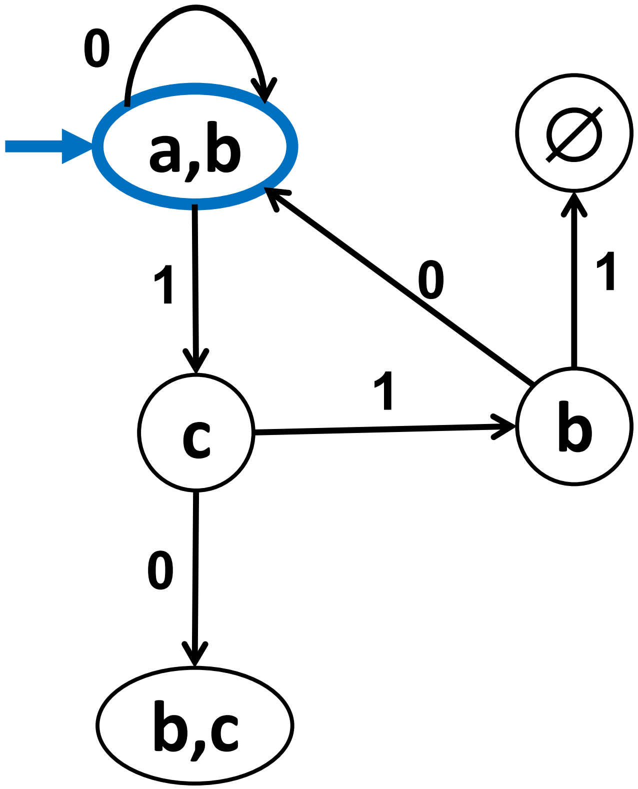


DFA

Example: NFA to DFA

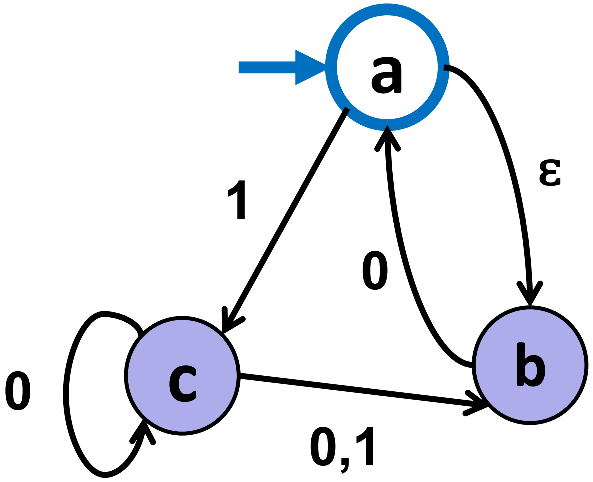


NFA

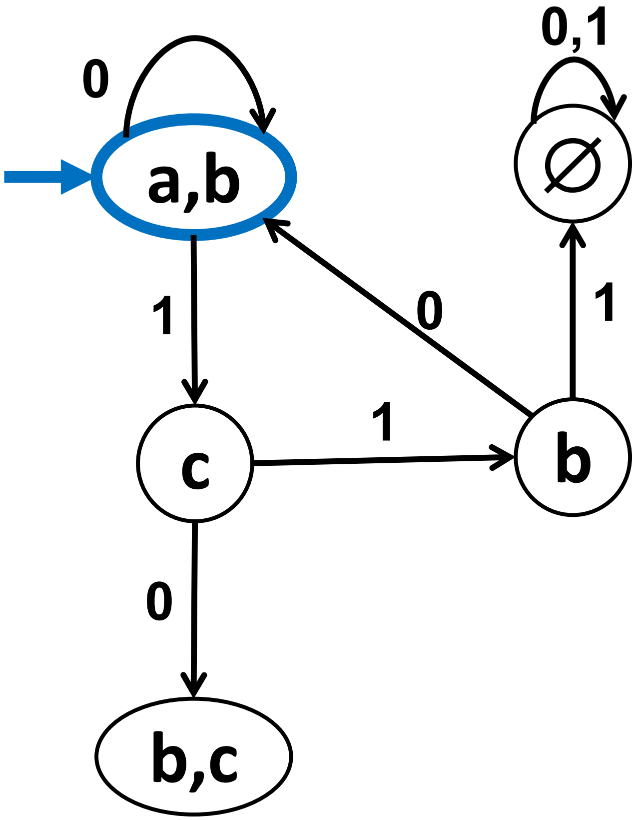


DFA

Example: NFA to DFA

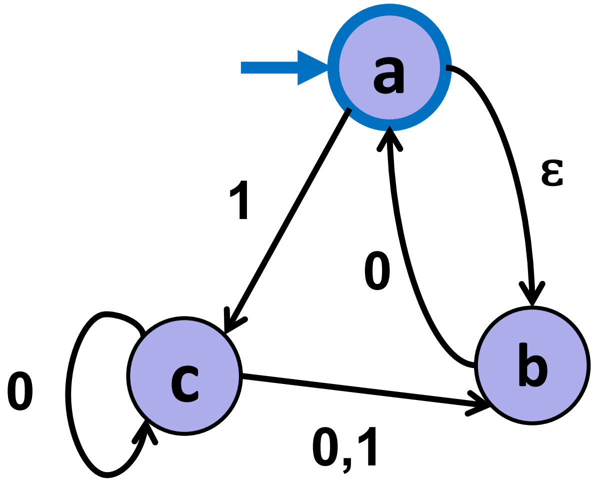


NFA

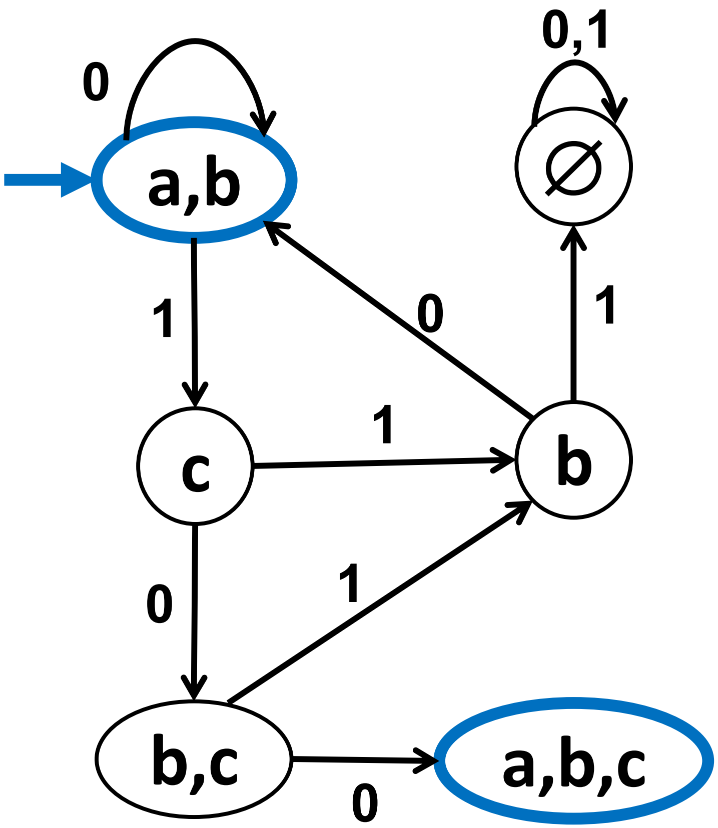


DFA

Example: NFA to DFA

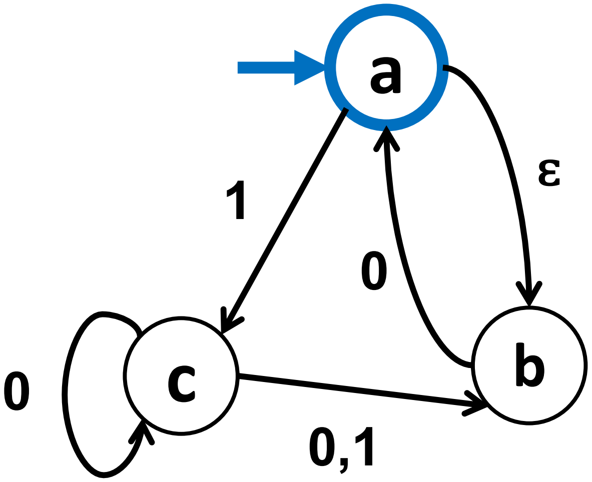


NFA

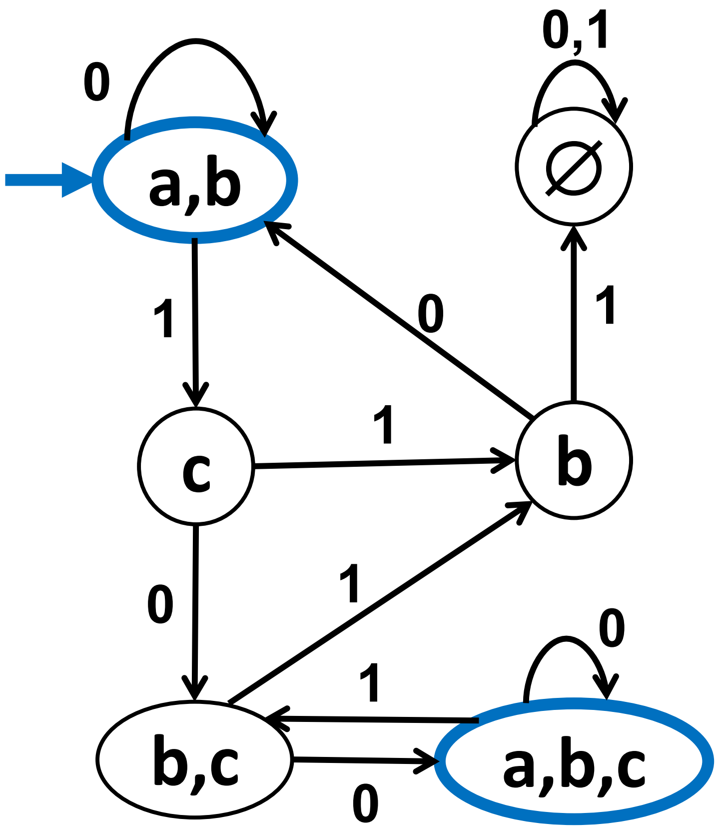


DFA

Example: NFA to DFA

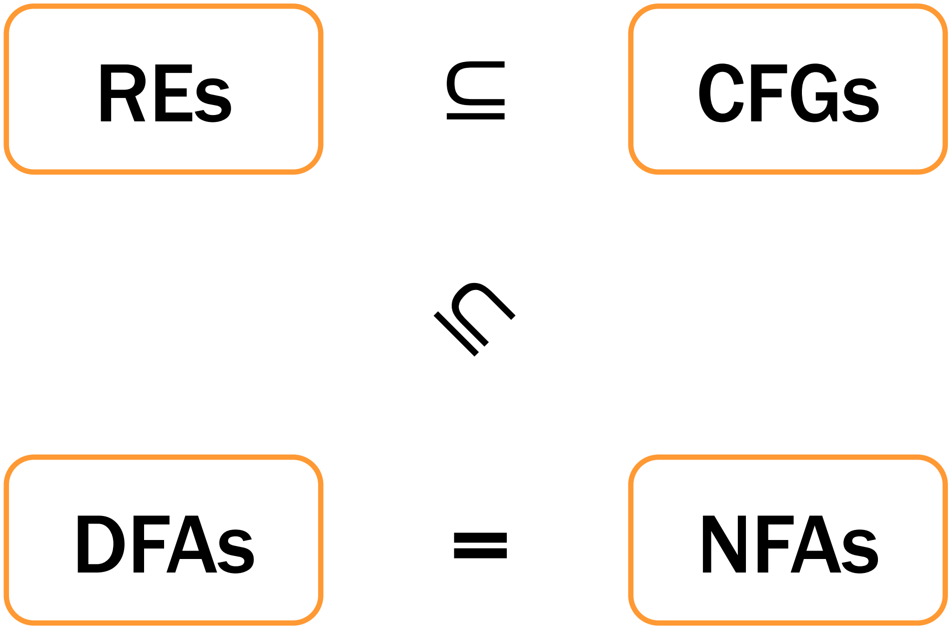


NFA



DFA

The story so far...



REs ⊆ CFGs

⊆

DFAs = NFAs

Regular expressions ⊆ NFAs ≡ DFAs

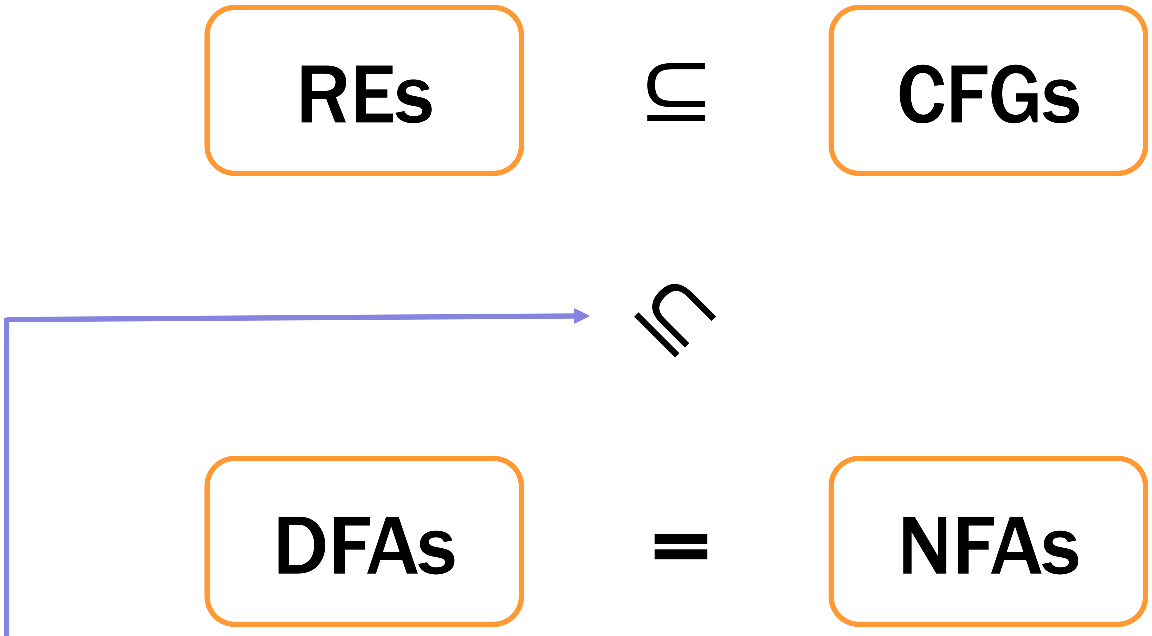
We have shown how to build an optimal DFA for every regular expression

- Build NFA
- Convert NFA to DFA using subset construction
- Minimize resulting DFA

Thus, we could now implement a RegExp library

- most RegExp libraries actually simulate the NFA
- (even better: one can combine the two approaches: apply DFA minimization lazily while simulating the NFA)

The story so far…



Is this ⊆ really “=” or “⊊”?

Regular expressions ≡ NFAs ≡ DFAs

Theorem: For any NFA, there is a regular expression that accepts the same language

Corollary: A language is recognized by a DFA (or NFA) if and only if it has a regular expression

You need to know these facts

- the construction for the Theorem is included in the slides after this, but you will not be tested on it

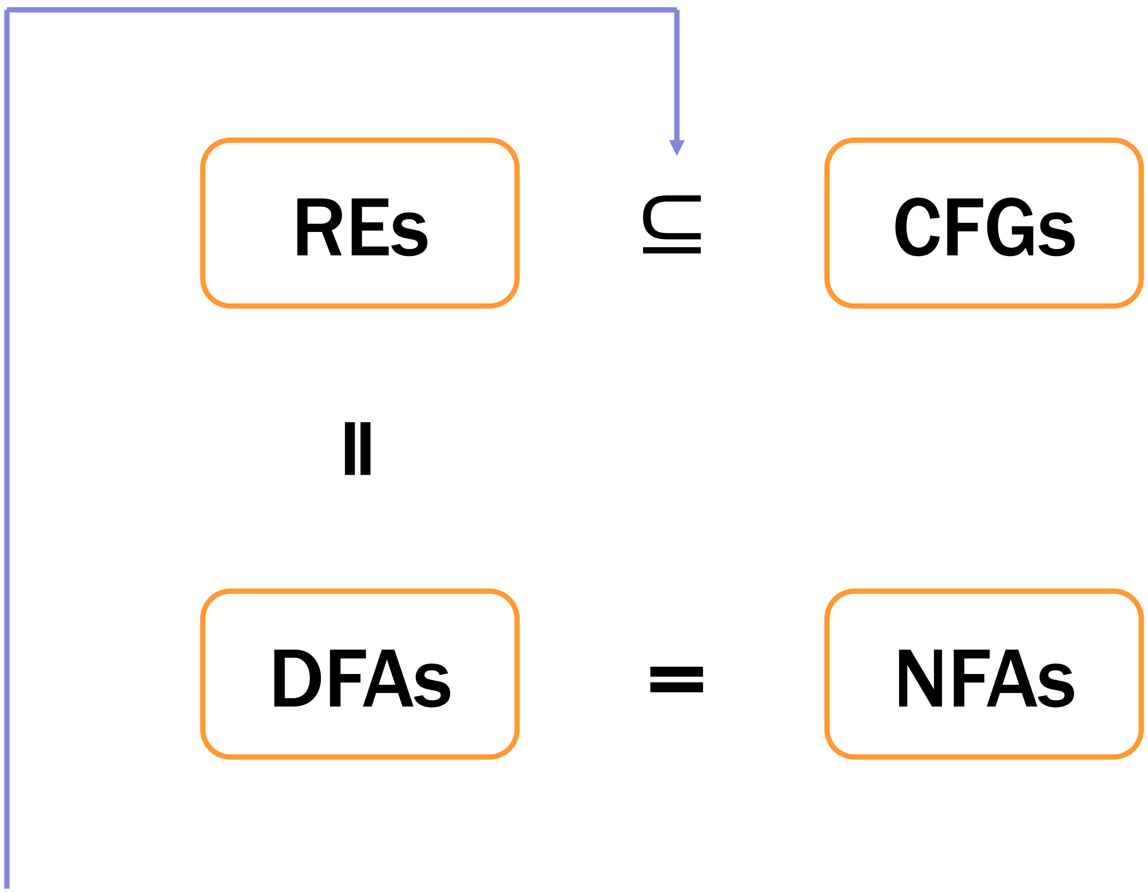
The story so far...

REs $\subseteq$ CFGs

$\parallel$

DFAs $=$ NFAs

The story so far...



REs ⊆ CFGs

‖

DFAs = NFAs

Next time: Is this ⊆ really "=" or "⊊"?

Recall: Algorithms for Regular Languages

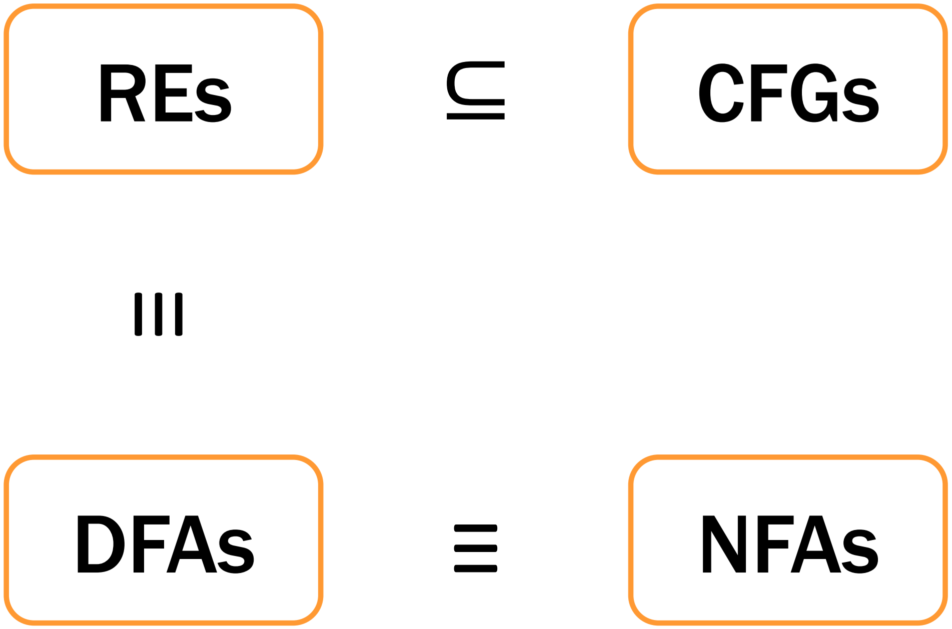
We have seen algorithms for

- RE to NFA
- NFA to DFA
- DFA/NFA to RE (not tested)
- DFA minimization

Practice three of these in HW.

(May also be on the final.)

The story so far…



REs ⊆ CFGs

≡

DFAs ≡ NFAs

Languages represented by DFA, NFAs, or regular expressions are called **Regular Languages**

Regular expressions ≡ NFAs ≡ DFAs

We have shown how to build an optimal DFA for every regular expression

- Build NFA
- Convert NFA to DFA using subset construction
- Minimize resulting DFA

Thus, we could now implement a RegExp library

- most RegExp libraries actually simulate the NFA
- (even better: one can combine the two approaches: apply DFA minimization lazily while simulating the NFA)

Example Corollary of These Results

Corollary: If $\mathbf{A}$ is the language of a regular expression, then $\overline{\mathbf{A}}$ is the language of a regular expression*.

(This is the complement with respect to the universe of all strings over the alphabet, i.e., $\overline{\mathbf{A}} = \mathbf{\Sigma}^* \setminus \mathbf{A}$.)

The story so far...

REs $\subseteq$ **CFGs**

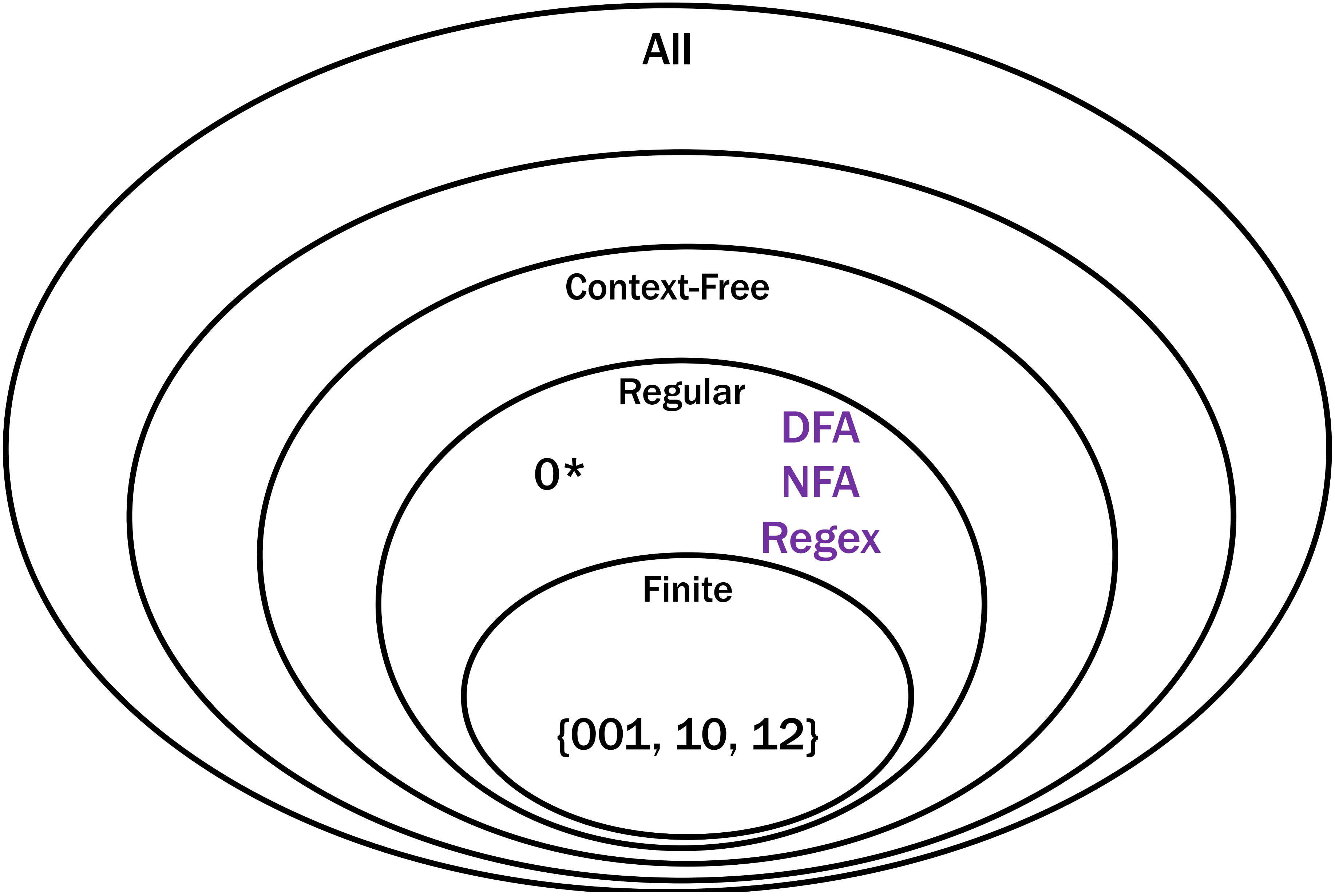
⦀

DFAs $\equiv$ **NFAs**

What languages have DFAs? CFGs?

All of them?

Languages and Representations!



All

Context-Free

Regular

DFA

NFA

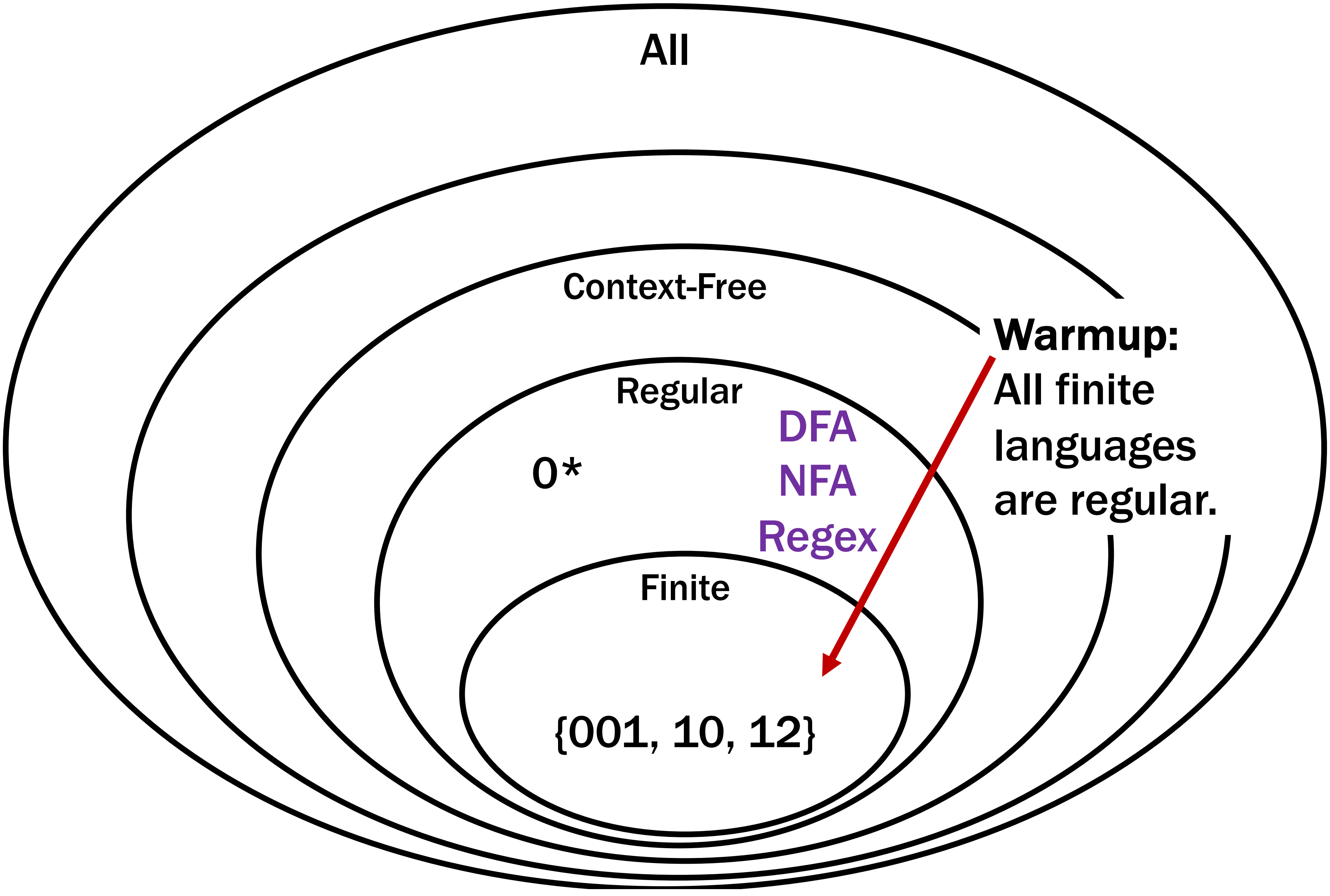
Regex

0*

Finite

{001, 10, 12}

Languages and Representations!



All

Context-Free

Regular

0*

DFA
NFA
Regex

Finite

{001, 10, 12}

Warmup:
All finite
languages
are regular.

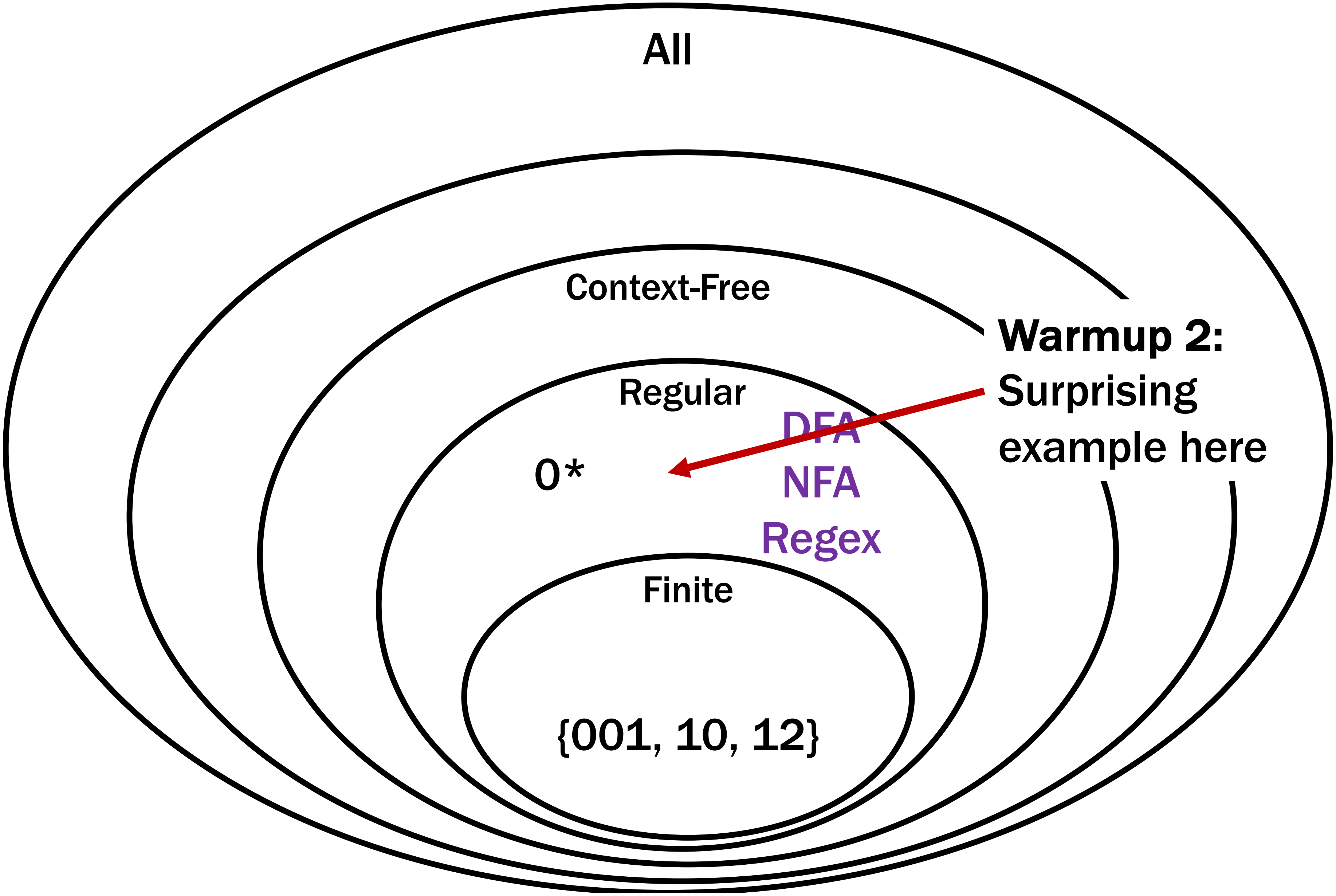
DFAs Recognize Any Finite Language

DFAs Recognize Any Finite Language

Construct a DFA for each string in the language.

Then, put them together using the union construction.

Languages and Machines!



Warmup 2: Surprising example here

An Interesting Infinite Regular Language

$L = \{x \in \{0, 1\}^*: x$ has an equal number of substrings 01 and 10$\}$.

L is infinite.

0, 00, 000, ...

L is regular. How could this be?

That seems to require comparing counts...

- easy for a CFG
- but seems hard for DFAs!

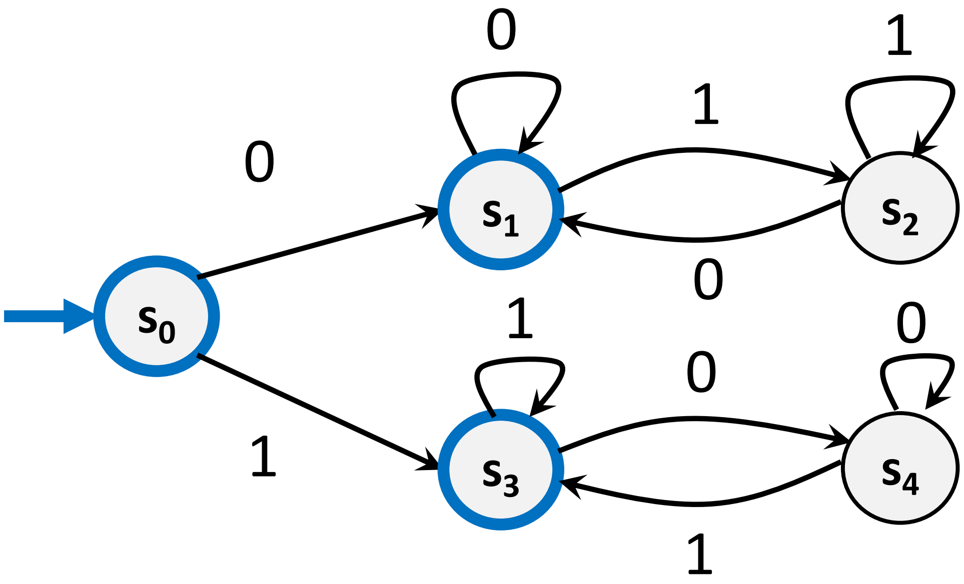
An Interesting Infinite Regular Language

L = {x∈ {0, 1}*: x has an equal number of substrings 01 and 10}.

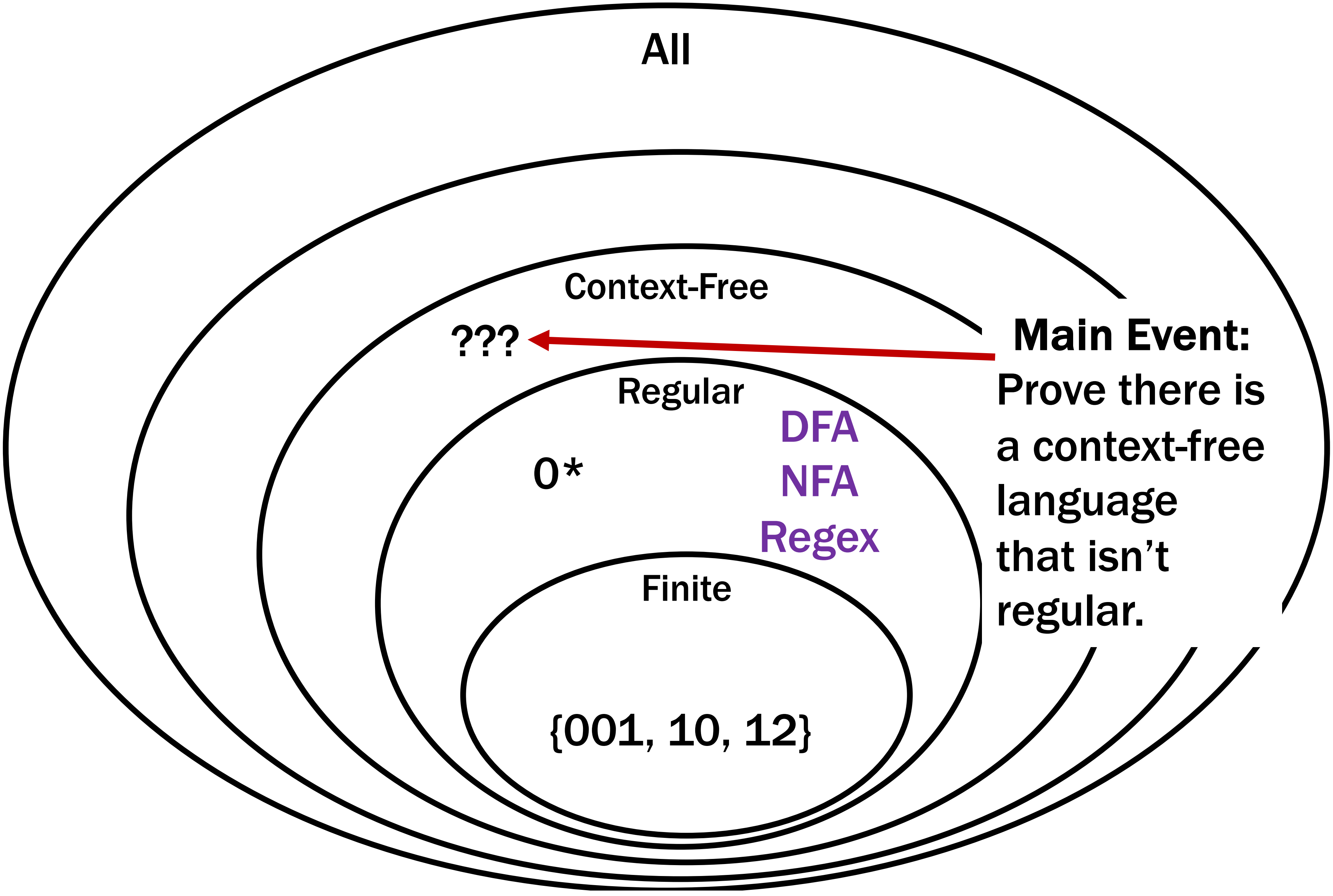
L is infinite.

0, 00, 000, …

L is regular. How could this be? It is just the set of binary strings that are empty or begin and end with the same character!



Languages and Representations!



All

Context-Free

???

Regular

DFA
NFA
Regex

0*

Finite

{001, 10, 12}

Main Event: Prove there is a context-free language that isn't regular.

The language of “Binary Palindromes” is Context-Free

$$S \rightarrow \varepsilon \mid 0 \mid 1 \mid 0S0 \mid 1S1$$

Is the language of “Binary Palindromes” Regular ?

Intuition (NOT A PROOF!):

Q: What would a DFA need to keep track of to decide?

A: It would need to keep track of the “first part” of the input in order to check the second part against it

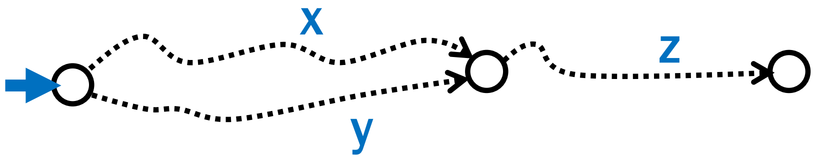
…but there are an infinite # of possible first parts and we only have finitely many states.

Proof idea: any machine that does not remember the entire first half will be wrong for some inputs

Useful Lemmas about DFAs

Lemma 1: If DFA **M** takes $\mathbf{x}, \mathbf{y} \in \mathbf{\Sigma}^*$ to the same state, then for every $\mathbf{z} \in \mathbf{\Sigma}^*$, M accepts $\mathbf{x} \bullet \mathbf{z}$ iff it accepts $\mathbf{y} \bullet \mathbf{z}$.

M can't remember that the input was $\mathbf{x}$, not $\mathbf{y}$.



$$x \bullet z = x_1\, x_2 \ldots x_n\, z_1\, z_2 \ldots z_k$$

$$y \bullet z = y_1\, y_2 \ldots y_m\, z_1\, z_2 \ldots z_k$$

Useful Lemmas about DFAs

Lemma 2: If DFA **M** has **n** states and a set **S** contains *more* than **n** strings, then **M** takes at least two strings from **S** to the same state.

M can’t take n+1 or more strings to different states because it doesn’t have n+1 different states.

So, some pair of strings must go to the same state.

B = {binary palindromes} can't be recognized by any DFA

Suppose for contradiction that some DFA, M, recognizes B.

We will show M accepts or rejects a string it shouldn't.

Consider S = {1, 01, 001, 0001, 00001, ...} = $\{0^n1 : n \geq 0\}$.

B = {binary palindromes} can't be recognized by any DFA

Suppose for contradiction that some DFA, **M**, accepts **B**.

We will show **M** accepts or rejects a string it shouldn't.

Consider $S = \{1, 01, 001, 0001, 00001, \ldots\} = \{0^n 1 : n \geq 0\}$.

Since there are finitely many states in M and infinitely many strings in S, by Lemma 2, there exist strings $0^a 1 \in S$ and $0^b 1 \in S$ with $a \neq b$ that end in the same state of M.

SUPER IMPORTANT POINT: You do not get to choose what a and b are. Remember, we've just proven they exist…we must take the ones we're given!

B = {binary palindromes} can't be recognized by any DFA

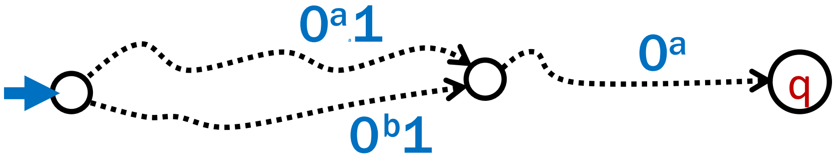
Suppose for contradiction that some DFA, M, accepts B.

We will show M accepts or rejects a string it shouldn't.

Consider S = {1, 01, 001, 0001, 00001, …} = {$0^n1 : n \geq 0$}.

Since there are finitely many states in M and infinitely many strings in S, by Lemma 2, there exist strings $0^a1 \in S$ **and** $0^b1 \in S$ **with** $a \neq b$ **that end in the same state of M.**

Now, consider appending *0^a* ***to both strings.***



B = {binary palindromes} can't be recognized by any DFA

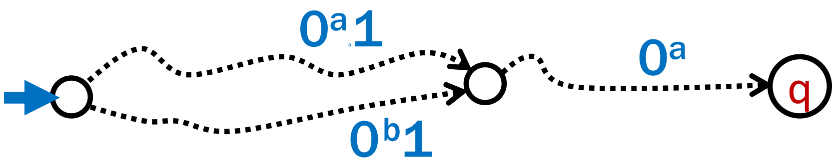
Suppose for contradiction that some DFA, M, accepts B.

We will show M accepts or rejects a string it shouldn't.

Consider $S = \{1, 01, 001, 0001, 00001, \ldots\} = \{0^n1 : n \geq 0\}$.

Since there are finitely many states in M and infinitely many strings in S, by Lemma 2, there exist strings $0^a1 \in S$ and $0^b1 \in S$ with $a \neq b$ that end in the same state of M.

Now, consider appending 0^a to both strings.



Since 0^a1 *and* 0^b1 *end in the same state,* 0^a10^a *and* 0^b10^a *also end in the same state, call it* q*. But then* ***M*** *makes a mistake:* q *needs to be an accept state since* $0^a10^a \in$ ***B****, but* ***M*** *would accept* $0^b10^a \notin$ ***B****, which is an error.*

B = {binary palindromes} can't be recognized by any DFA

Suppose for contradiction that some DFA, M, accepts B.

We will show M accepts or rejects a string it shouldn't.

Consider $S = \{1, 01, 001, 0001, 00001, \ldots\} = \{0^n1 : n \geq 0\}$.

Since there are finitely many states in M and infinitely many strings in S, by Lemma 2, there exist strings $0^a1 \in S$ and $0^b1 \in S$ with $a \neq b$ that end in the same state of M.

Now, consider appending 0^a to both strings.

Since 0^a1 and 0^b1 end in the same state, 0^a10^a and 0^b10^a also end in the same state, call it q. But then M makes a mistake: q needs to be an accept state since $0^a10^a \in B$, but M would accept $0^b10^a \notin B$, which is an error.

This proves that M does not recognize B, contradicting our assumption that it does. Thus, no DFA recognizes B.

Showing that a Language L is not regular

1. "Suppose for contradiction that some DFA M recognizes L."
2. Consider an **INFINITE** set S of prefixes (which we intend to complete later).
3. "Since S is infinite and M has finitely many states, there must be two strings s_a and s_b in S for $s_a \neq s_b$ that end up at the same state of M."
4. Consider appending the (correct) completion t to each of the two strings.
5. "Since s_a and s_b both end up at the same state of M, and we appended the same string t, both $s_a t$ and $s_b t$ end at the same state q of M. Since $s_a t \in L$ and $s_b t \notin L$, M does not recognize L."
6. "Thus, no DFA recognizes L."

Showing that a Language L is not regular

The choice of S is the creative part of the proof

You must find an infinite set S with the property that *no two* strings can be taken to the same state

- i.e., for ***every pair*** of strings S there is an "accept" completion that the two strings DO NOT SHARE

Prove A = $\{0^n1^n : n \geq 0\}$ is not regular

Suppose for contradiction that some DFA, M, recognizes A.

Let S =

Prove **A** $= \{0^n 1^n : n \geq 0\}$ is not regular

Suppose for contradiction that some DFA, **M**, recognizes **A**.

Let **S** $= \{0^n : n \geq 0\}$. Since **S** is infinite and **M** has finitely many states, there must be two strings, 0^a and 0^b for some $a \neq b$ that end in the same state in **M**.

Prove A = $\{0^n1^n : n \geq 0\}$ is not regular

Suppose for contradiction that some DFA, M, recognizes A.

Let S = $\{0^n : n \geq 0\}$. Since S is infinite and M has finitely many states, there must be two strings, 0^a and 0^b for some $a \neq b$ that end in the same state in M.

Consider appending 1^a to both strings.

Prove A = $\{0^n1^n : n \geq 0\}$ is not regular

Suppose for contradiction that some DFA, M, recognizes A.

Let $S = \{0^n : n \geq 0\}$. Since S is infinite and M has finitely many states, there must be two strings, 0^a and 0^b for some $a \neq b$ that end in the same state in M.

Consider appending 1^a to both strings.

Note that $0^a1^a \in A$, but $0^b1^a \notin A$ since $a \neq b$. But they both end up in the same state of M, call it q. Since $0^a1^a \in A$, state q must be an accept state but then M would incorrectly accept $0^b1^a \notin A$ so M does not recognize A.

Thus, no DFA recognizes A.

Prove P = {balanced parentheses} **is not regular**

Suppose for contradiction that some DFA, M, accepts P.

Let S =

Prove P = {balanced parentheses} **is not regular**

Suppose for contradiction that some DFA, M, recognizes P.

Let **S =** { $(^{n} : n \geq 0$}**. Since S is infinite and M has finitely many states, there must be two strings,** $($^a **and** $($^b **for some** $a \neq b$ **that end in the same state in M.**

Prove P = {balanced parentheses} **is not regular**

Suppose for contradiction that some DFA, M, recognizes P.

Let S = $\{ (^n : n \geq 0\}$. Since S is infinite and M has finitely many states, there must be two strings, $(^a$ and $(^b$ for some $a \neq b$ that end in the same state in M.

Consider appending $)^a$ to both strings.

Prove P = {balanced parentheses} **is not regular**

Suppose for contradiction that some DFA, M, recognizes P.

Let $S = \{ (^n : n \geq 0\}$. Since S is infinite and M has finitely many states, there must be two strings, $(^a$ and $(^b$ for some $a \neq b$ that end in the same state in M.

Consider appending $)^a$ to both strings.

Note that $(^a)^a \in P$, but $(^b)^a \notin P$ since $a \neq b$. But they both end up in the same state of M, call it **q**. Since $(^a)^a \in P$, state **q** must be an accept state but then M would incorrectly accept $(^b)^a \notin P$ so M does not recognize P.

Thus, no DFA recognizes P.

Showing that a Language L is not regular

1. “Suppose for contradiction that some DFA M recognizes L.”
2. Consider an **INFINITE** set S of prefixes (which we intend to complete later). It is imperative that for ***every pair*** of strings in our set there is an “accept” completion that the two strings DO NOT SHARE.
3. “Since S is infinite and M has finitely many states, there must be two strings s_a and s_b in S for $s_a \neq s_b$ that end up at the same state of M.”
4. Consider appending the (correct) completion t to each of the two strings.
5. “Since s_a and s_b both end up at the same state of M, and we appended the same string t, both $s_a t$ and $s_b t$ end at the same state q of M. Since $s_a t \in L$ and $s_b t \notin L$, M does not recognize L.”
6. “Thus, no DFA recognizes L.”

Fact: This method is optimal

- Suppose that for a language L, the set S is a *largest* set of prefixes with the property that, for every pair $s_a \neq s_b \in S$, there is some string t such that one of $s_a t$, $s_b t$ is in L but the other isn't.
- If S is infinite, then L is not regular
- If S is finite, then the minimal DFA for L has precisely $|S|$ states, one reached by each member of S.

Fact: This method is optimal

- Suppose that for a language L, the set S is a *largest* set of prefixes with the property that, for every pair $s_a \neq s_b \in S$, there is some string t such that one of $s_a t$, $s_b t$ is in L but the other isn't.
- If S is infinite, then L is not regular
- If S is finite, then the minimal DFA for L has precisely $|S|$ states, one reached by each member of S.

Corollary: Our minimization algorithm was correct.

- we separated *exactly* those states for which some t would make one accept and another not accept

Important Notes

- It is not necessary for our strings **xz** with **x** ∈ **L** to allow any string in the language
  - we only need to find a small "core" set of strings that must be distinguished by the machine

- It is **not true** that, if **L** is irregular and **L ⊆ U**, then **U** is irregular!
  - we always have **L ⊆ Σ*** and **Σ*** is regular!
  - our argument needs different answers: $xz \in L \not\leftrightarrow yz \in L$

    for **Σ***, both strings are always in the language

Do not claim in your proof that, because L ⊆ U, U is also irregular