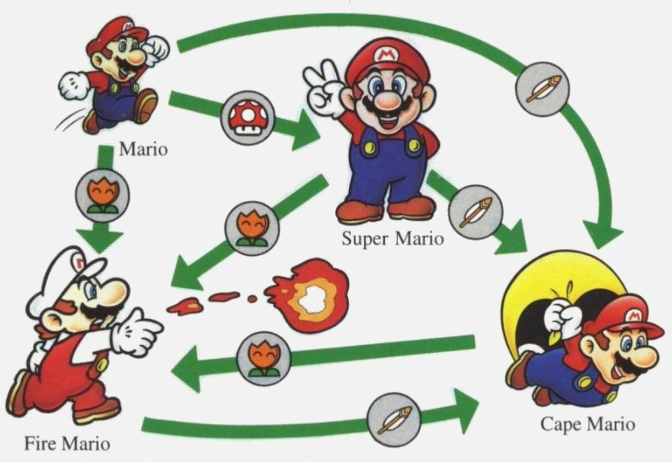
# CS420

# (Deterministic) Finite Automata

Wednesday, January 26, 2022

UMass Boston Computer Science



# Announcements

- HW 0 due Sunday 1/30 11:59pm EST

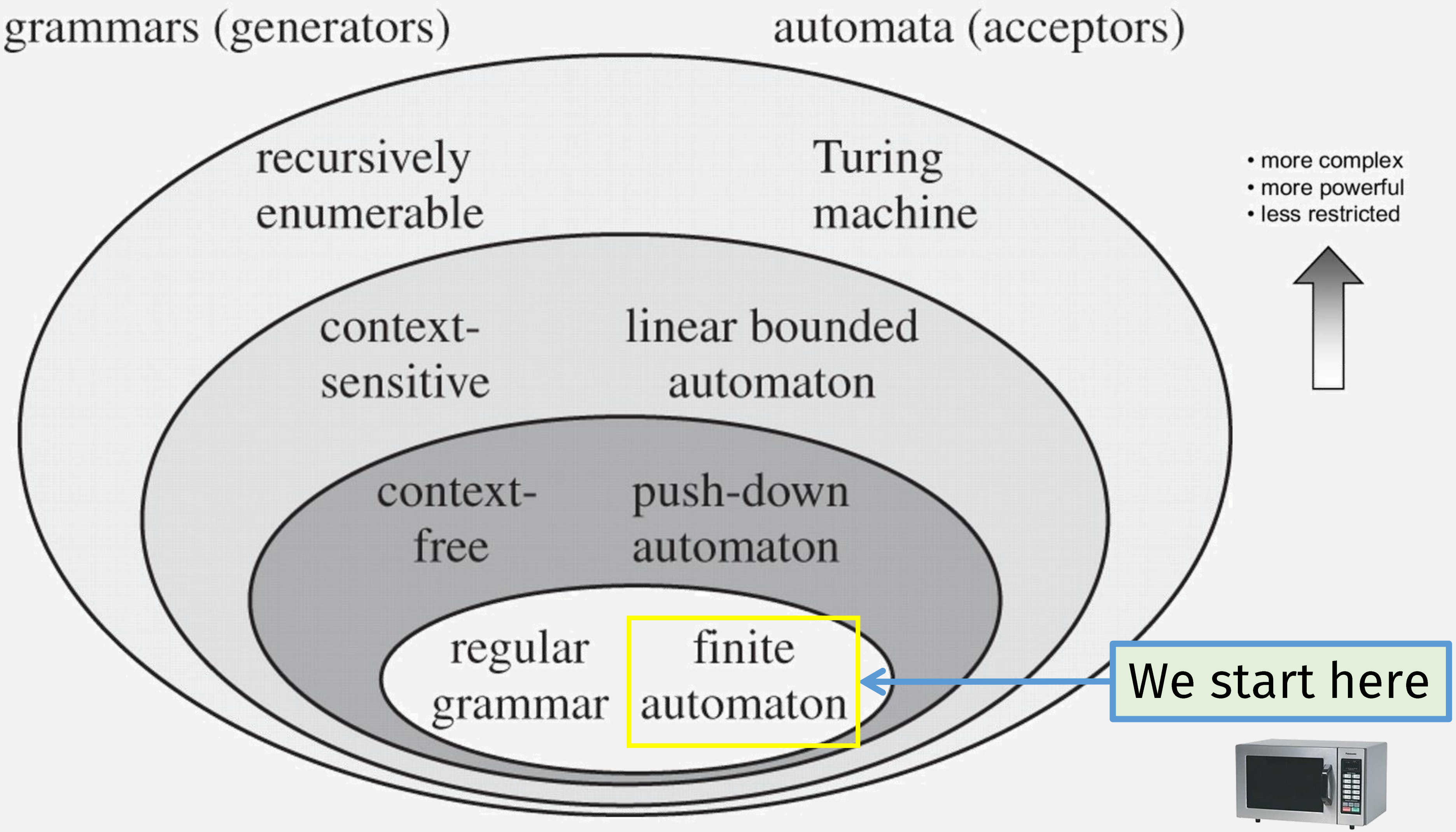
- Office Hours (via zoom)
  - Hannah: Mon 2-3:30pm
  - Me: Tue/Fri 4-5:30pm

# *Last time*: The Theory of Computation …

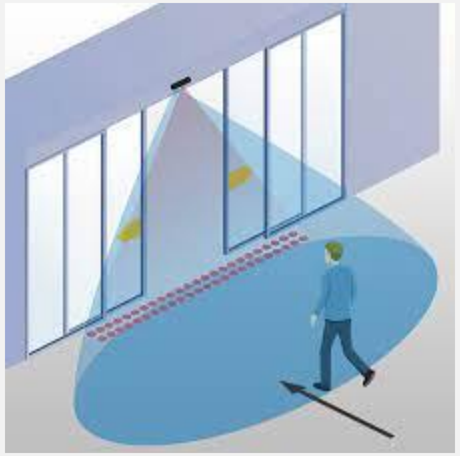
- Creates and compares mathematical models of computers

- In order to:
  - Make predictions about computer programs
  - Explore the limits of computation

# *Last time*: Levels of Computational *Power*



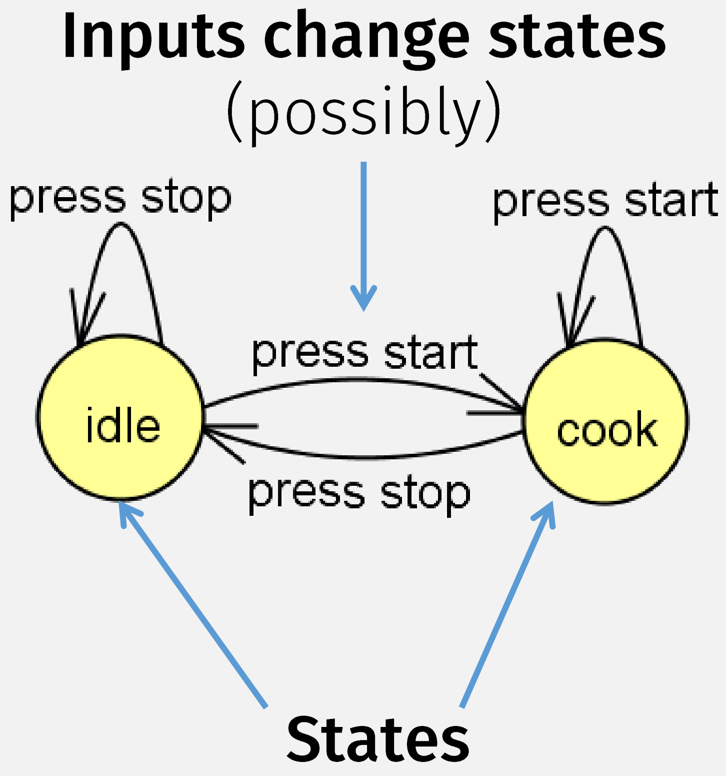
# Finite Automata: A computational model for …



# Finite Automata

- A **finite automata** or **finite state machine (FSM)** ...
- ... is a computer with a <u>finite</u> number of states

# A Microwave Finite Automata



# Finite Automata: Not Just for Microwaves

## State pattern

From Wikipedia, the free encyclopedia

The **state pattern** is a behavioral software design pattern that allows an object to alter its behavior when its internal state changes. This pattern is close to the concept of finite-state machines. The state pattern can be interpreted as a strategy pattern, which is able to switch a strategy through invocations of methods defined in the pattern's interface.

**Finite Automata:**
a common
programming pattern

<u>Note</u>: **Computers can simulate computers** (more on this later)

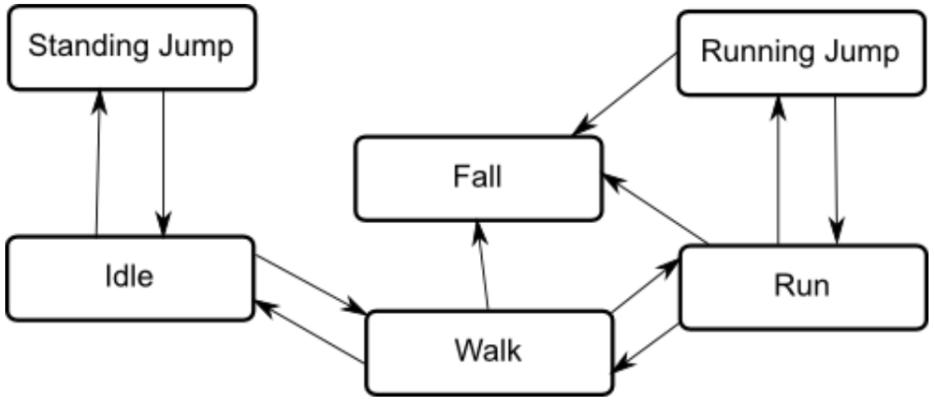
# Video Games Love Finite Automata

**Unity Documentation** Manual

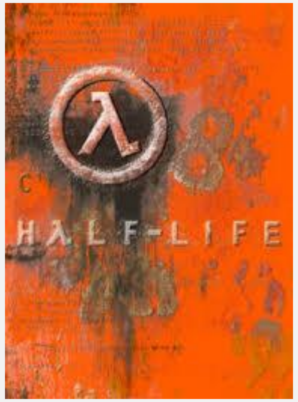
Unity User Manual 2020.3 (LTS) / Animation / Animator Controllers / Animation State Machines / State Machine Basics

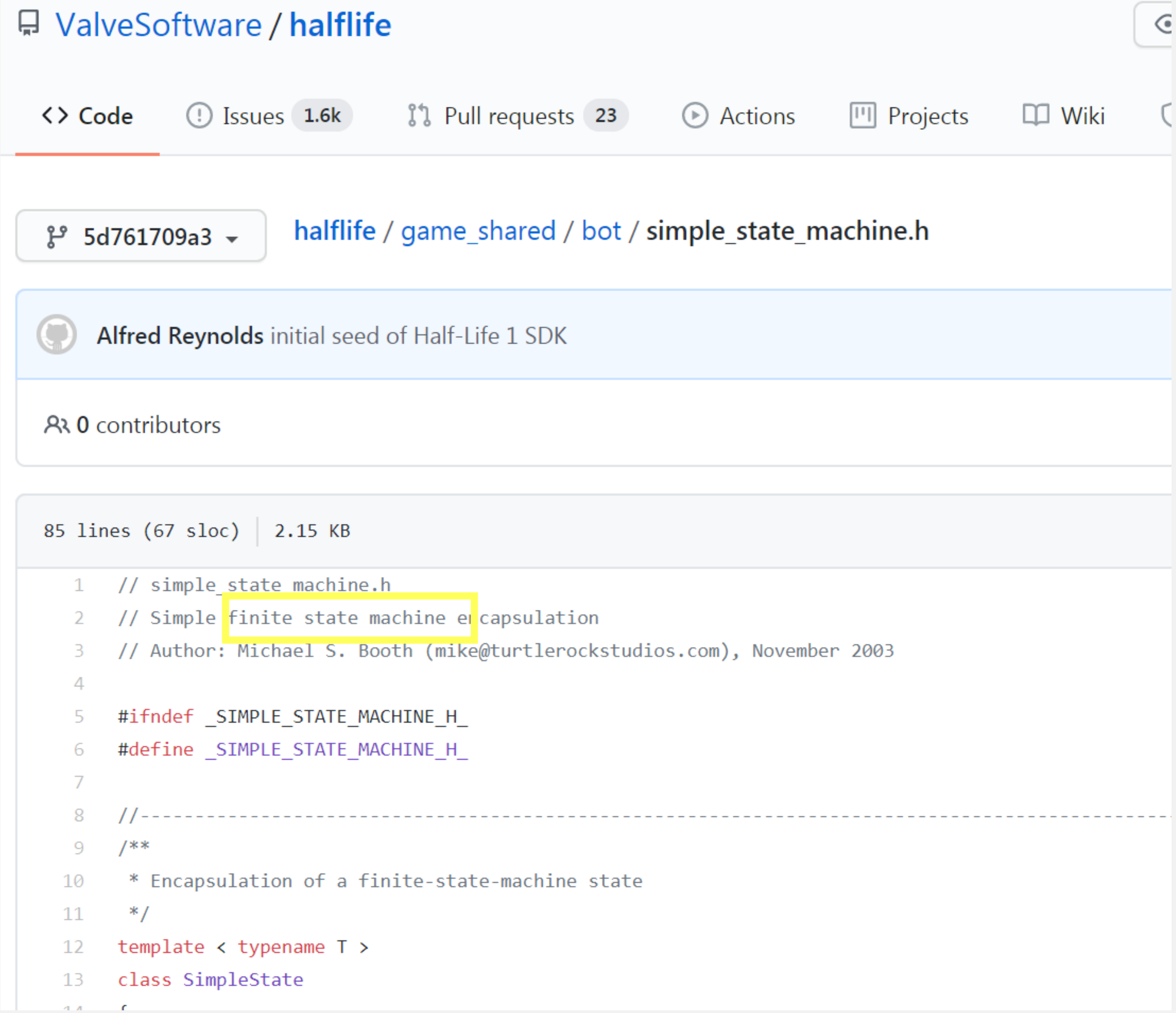
The basic idea is that a character is engaged in some particular kind of action at any given time. The actions available will depend on the type of gameplay but typical actions include things like idling, walking, running, jumping, etc. These actions are referred to as **states**, in the sense that the character is in a "state" where it is walking, idling or whatever. In general, the character will have restrictions on the next state it can go to rather than being able to switch immediately from any state to any other. For example, a running jump can only be taken when the character is already running and not when it is at a standstill, so it should never switch straight from the idle state to the running jump state. The options for the next state that a character can enter from its current state are referred to as state transitions. Taken together, the set of states, the set of transitions and the variable to remember the current state form a **state machine**.

The states and transitions of a state machine can be represented using a graph diagram, where the nodes represent the states and the arcs (arrows between nodes) represent the transitions. You can think of the current state as being a marker or highlight that is placed on one of the nodes and can then only jump to another node along one of the arrows.



# Finite Automata in Video Games

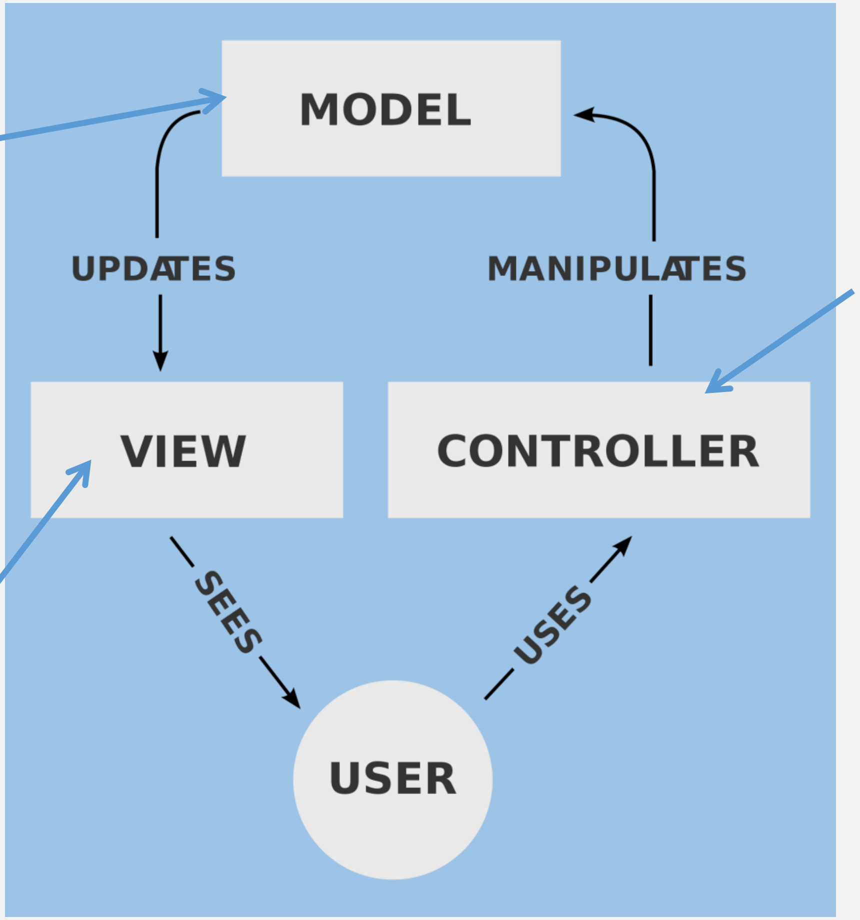


# Model-view-controller (MVC) is a FSM

**States**

**The View draws states**



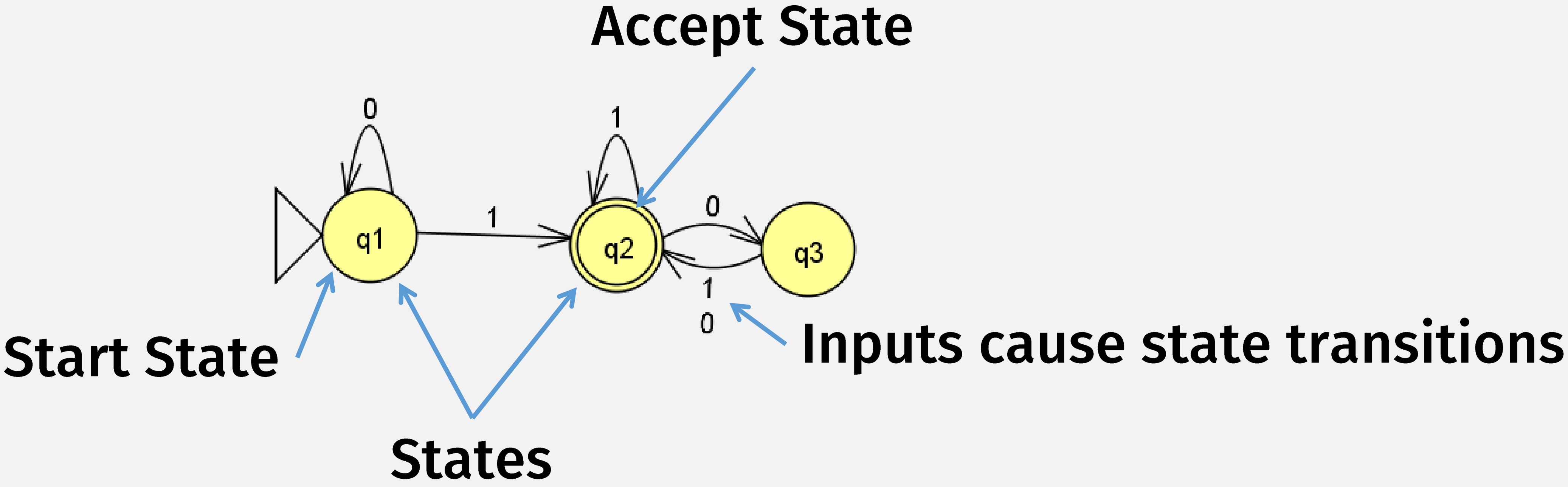

**Inputs change states**

# A Finite Automata is a Computer

- A very limited computer with finite memory
  - Actually, only 1 cell of memory!
  - States = the possible things that can be written to the memory

- Finite Automata has different representations:
  - Code
  - ➢State diagrams

# Finite Automata state diagram



# A Finite Automata is a Computer

- A very limited computer with finite memory
  - Actually, only 1 cell of memory!
  - States = the possible things that can be written to the memory

- Finite Automata has different representations:
  - Code
  - State diagrams
  - ➢ Formal mathematical model

# Finite Automata: The Formal Definition

5 components

**DEFINITION**

deterministic

A ***finite automaton*** is a 5-tuple $\boxed{(Q, \Sigma, \delta, q_0, F)}$, where

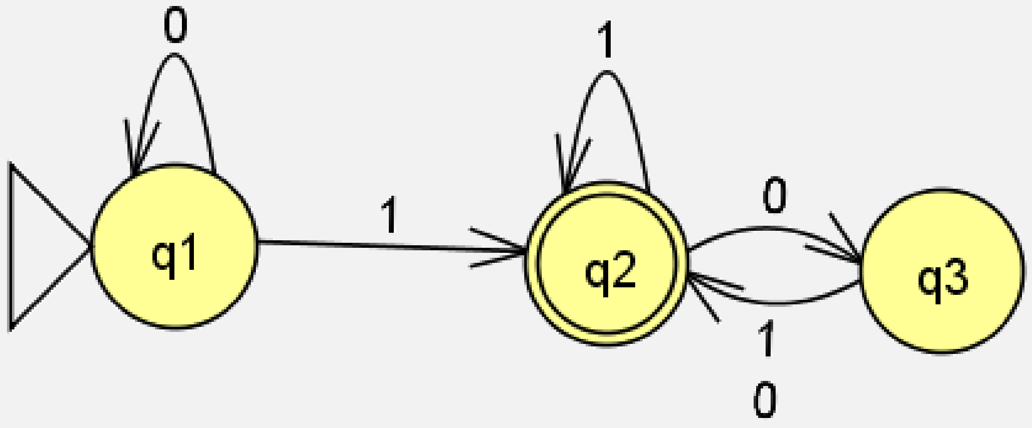
1. $Q$ is a finite set called the ***states***,
2. $\Sigma$ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Also called a **Deterministic Finite Automata (DFA)**

**DEFINITION**

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q$ is a finite set called the ***states***,
2. $\Sigma$ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

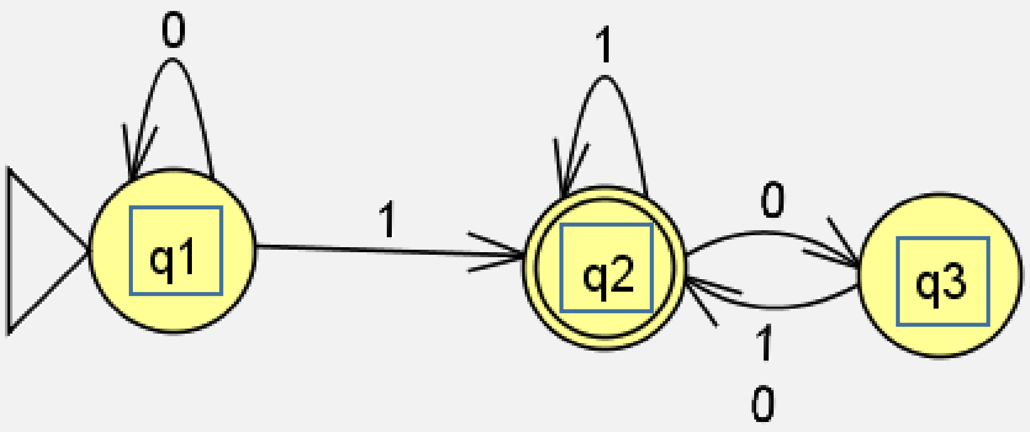



Example: as state diagram

**DEFINITION**

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q$ is a finite set called the ***states***,
2. $\Sigma$ is a finite set called the ***alphabet***,
3. $\delta \colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.




Example: as state diagram

Example: as formal description

$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. $\delta$ is described as

| | 0 | 1 |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_2$ |
| $q_3$ | $q_2$ | $q_2$, |

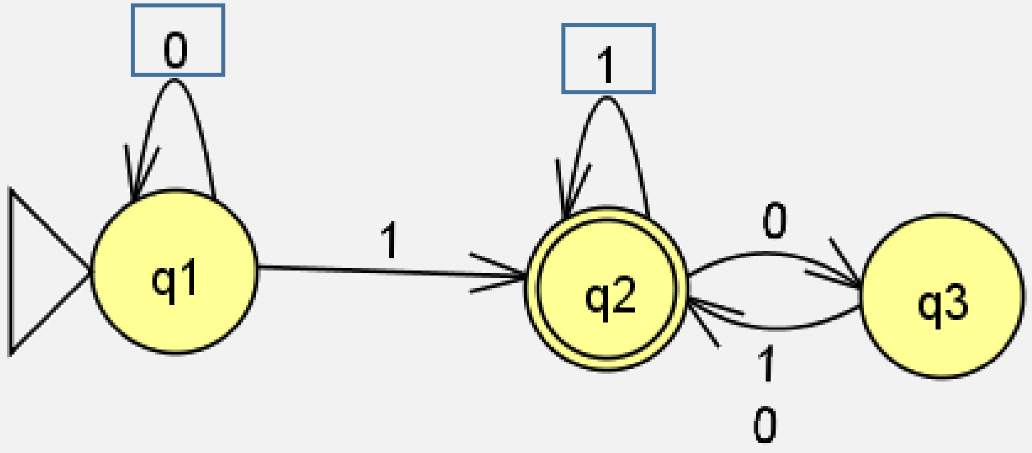
4. $q_1$ is the start state, and
5. $F = \{q_2\}$.

braces = set notation (no duplicates)

**DEFINITION**

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q$ is a finite set called the ***states***,
2. $\Sigma$ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.




Example: as state diagram

Example: as formal description

$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$, ← Possible inputs
3. $\delta$ is described as

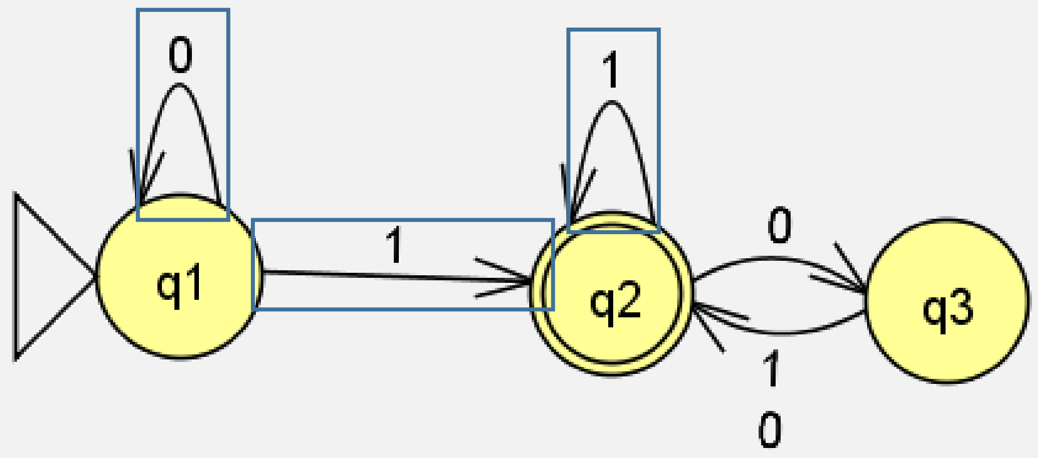
| | 0 | 1 |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_2$ |
| $q_3$ | $q_2$ | $q_2$, |

4. $q_1$ is the start state, and
5. $F = \{q_2\}$.

**DEFINITION**

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q$ is a finite set called the ***states***,
2. $\Sigma$ is a finite set called the ***alphabet***,
3. $\delta \colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.




Example: as state diagram

Example: as formal description

$$M_1 = (Q, \Sigma, \delta, q_1, F), \text{ where}$$

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. $\delta$ is described as

"And this is next input symbol"

"If in this state"

"Then go to this state"

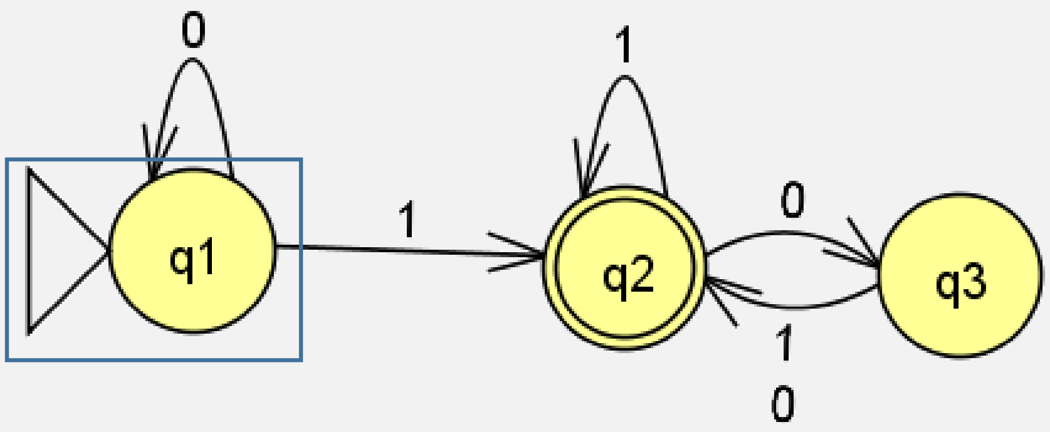
| | 0 | 1 |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_2$ |
| $q_3$ | $q_2$ | $q_2$, |

4. $q_1$ is the start state, and
5. $F = \{q_2\}$.

**DEFINITION**

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q$ is a finite set called the ***states***,
2. $\Sigma$ is a finite set called the ***alphabet***,
3. $\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.




Example: as <u>state diagram</u>

Example: as <u>formal description</u>

$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. $\delta$ is described as

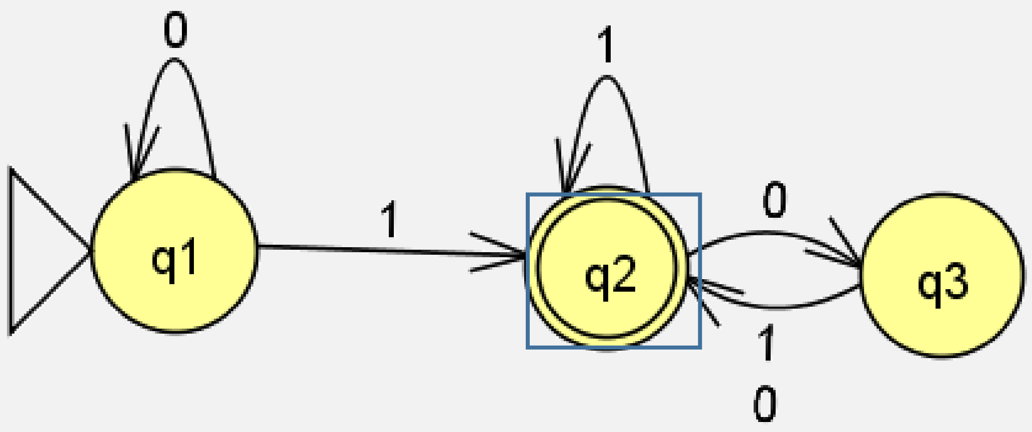
| | 0 | 1 |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_2$ |
| $q_3$ | $q_2$ | $q_2$, |

4. $q_1$ is the start state, and
5. $F = \{q_2\}$.

**DEFINITION**

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. $Q$ is a finite set called the ***states***,
2. $\Sigma$ is a finite set called the ***alphabet***,
3. $\delta \colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.




Example: as state diagram

Example: as formal description

$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. $\delta$ is described as

| | 0 | 1 |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_2$ |
| $q_3$ | $q_2$ | $q_2$, |

4. $q_1$ is the start state, and
5. $F = \{q_2\}$.

# Precise Terminology is Important

- A **finite automata** or **finite state machine (FSM)** is a …
  … computer with a finite number of states

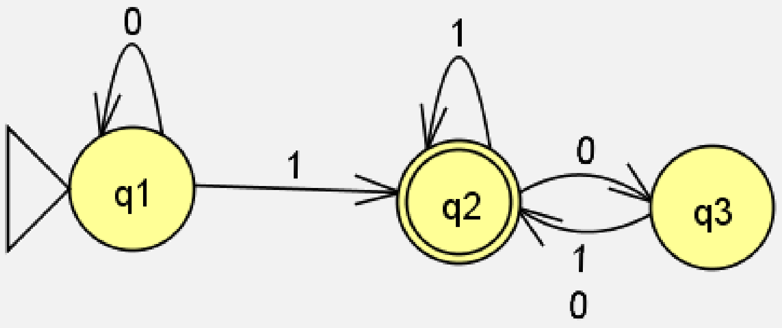
- There are many FSM variations. We've learned one so far:
  - the **Deterministic Finite Automata (DFA)**
  - (So currently, the term DFA and FSM refer to the same definition)

- Eventually, we'll learn other FSM variations,
  - e.g., **Nondeterministic Finite Automata (NFA)**

- Then, you will need to be more careful with terminology

- We will show that: all FSMs are related; they are equivalent in "power"

# Computation on an FSM (JFLAP demo)

- **FSM:** 


- **Program:** “`1101`”

# FSM Computation Model

| *Informally* | *Formally (i.e., mathematically)* |
| --- | --- |

- Computer = a finite automata
- Program = input string of chars, e.g. "`1101`"

To run a program:

- Start in "start state"
- Read 1 char at a time, changing states according to the transition table
- Result =
  - "Accept" if last state is "Accept" state
  - "Reject" otherwise

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$
- $r_0 = q_0$
- $\delta(r_i, w_{i+1}) = r_{i+1}, \text{ for } i = 0, \ldots, n-1$

**Let's come up with new notation to represent this part**

- $M$ ***accepts*** $w$ if

  sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists ...

  with $r_n \in F$

**Still a little verbose**

$\delta : Q \times \Sigma \longrightarrow Q$ is the ***transition function***

# An Extended Transition Function

Define the extended transition function: $\hat{\delta} : Q \times \Sigma^* \to Q$

* = "0 or more"

- *Inputs*:
  - Some beginning state $q \in Q$ (not necessarily the start state)
  - Input string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- *Output*:
  - Some ending state (not necessarily an accept state)

(Defined recursively)

Empty string

- Base case: $\hat{\delta}(q, \varepsilon) = q$

First char

Remaining chars

- Recursive case: $\hat{\delta}(q, w) = \hat{\delta}(\delta(q, w_1), w_2 \cdots w_n)$

Recursive call

Single transition step

# FSM Computation Model

| *Informally* | *Formally (i.e., mathematically)* |
| --- | --- |
| • Computer = a finite automata | • $M = (Q, \Sigma, \delta, q_0, F)$ |
| • Program = input string of chars | • $w = w_1 w_2 \cdots w_n$ |
| To run a program: | |
| • Start in "start state" | • $r_0 = q_0$ |
| • Read 1 char at a time, changing states according to transition table | • $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$ |
| • Result = <br>• "Accept" if last state is "Accept" state <br>• "Reject" otherwise | • $M$ ***accepts*** $w$ if $\hat{\delta}(q_0, w) \in F$ <br>sequence of states $r_0, r_1, \ldots, r_n$ in $Q$ exists ... with $r_n \in F$ |

**Still a little verbose**

# Languages

- A **language** is a set of strings
- A **string** is a finite sequence of symbols from an alphabet
- An **alphabet** is a non-empty finite set of symbols

$$\Sigma_1 = \{0,1\}$$

$$\Sigma_2 = \{a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, z\}$$

# Computers and Languages

- Every computer is associated with a **language**
- The **language of a machine** is the set of all strings that it accepts
- E.g., An FSM $M$ ***accepts*** $w$ if $\hat{\delta}(q_0, w) \in F$
- Language of $M = L(M) = \{w \mid M \text{ accepts } w\}$

"the set of all …"

"such that …"

# Language Terminology

- $M$ ***accepts*** $w$

string

- $M$ ***recognizes language*** $A$

  if $A = \{w \mid M \text{ accepts } w\}$

Set of strings

# Computation and Classes of Languages

- Every computer is associated with a **language**
- The **language of a machine** is the set of all strings that it accepts
- A **computation model** is represented by a **set of machines**
- E.g., all possible FSMs represent a computation model
- Or: a **computation model** is represented by a **set of languages**

# Regular Languages

A language is called a ***regular language*** if some finite automaton recognizes it.

A **language** is a set of strings.

$M$ ***recognizes language*** $A$ if $A = \{w \mid M \text{ accepts } w\}$

# A language, regular or not?

- If given: Finite Automata $M$
  - We know: the language recognized by $M$ is a regular language
- If given: some Language $A$
  - Is $A$ is a regular language?
    - Not necessarily!
  - How do we determine, i.e., *prove*, that $A$ is a regular language?

A language is called a ***regular language*** if some finite automaton recognizes it.

# Kinds of Mathematical Proof

- Proof by construction
  - Construct the mathematical object in question

Example:

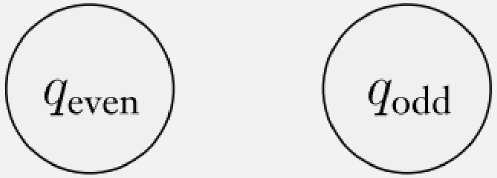
- To prove that a language is regular …
- … <u>construct</u> a finite automata recognizing the language
- (Because that's what definition of a regular language says)

# Designing Finite Automata: Tips

- Input may only be read once, one char at a time
- Must decide accept/reject after that
- States = the machine's **memory**!
  - # states must be decided in advance
  - So think about what information must be remembered.
- Every state/symbol pair must have a transition (for DFAs)

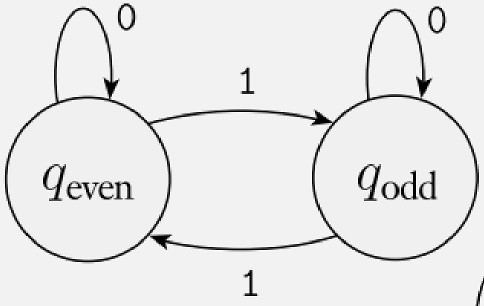
# Design a DFA: accept strs with odd # **1**s

- <u>States</u>:
  - 2 states:
    - seen even **1**s so far
    - seen odds **1**s so far

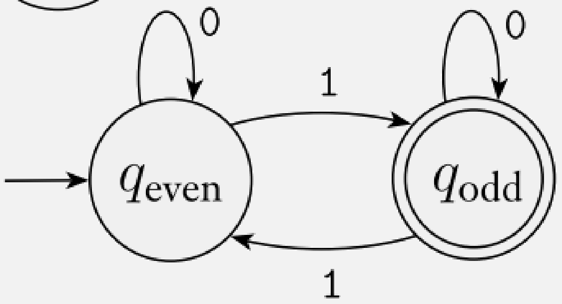



- <u>Alphabet</u>: `0` and `1`

- <u>Transitions</u>:




- <u>Start</u> / <u>Accept</u> states:




So finite automata are used to <u>recognize simple string patterns</u>?

**Yes!**

Do you know of anything else used to <u>recognize simple string patterns</u>?

# Combining Automata

# Combining DFAs?

## Password Requirements

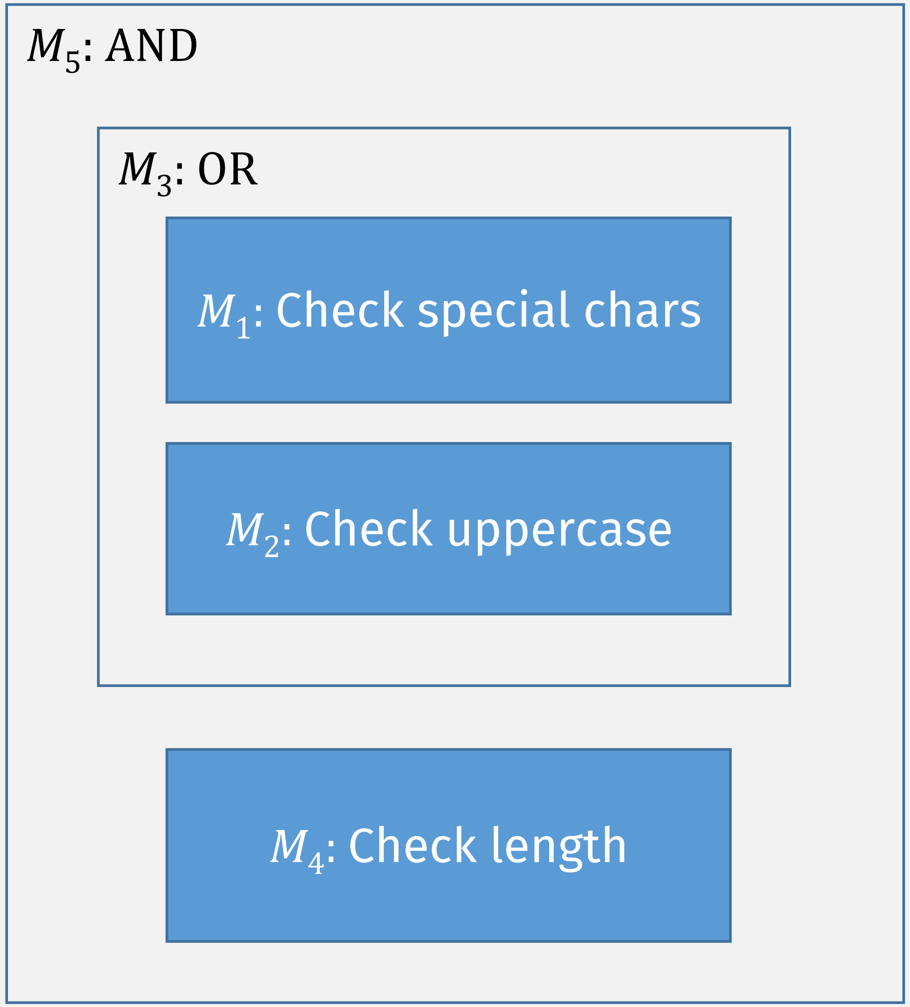
DFA

- Passwords must have a minimum length of ten (10) characters - but more is better!
- Passwords **must include at least 3** different types of characters:
  - upper-case letters (A-Z) ← DFA
  - DFA → lower-case letters (a-z)
  - symbols or special characters (%, &, *, $, etc.) ← DFA
  - numbers (0-9) ← DFA
- Passwords cannot contain all or part of your email address ← DFA
- Passwords cannot be re-used ← DFA

s://www.umb.edu/it/password

To match <u>all</u> requirements,
can we combine smaller DFAs?

# Password checker




Want to be able to easily combine finite automata machines

To combine more than once, operations must be **closed**!

# “Closed” Operations

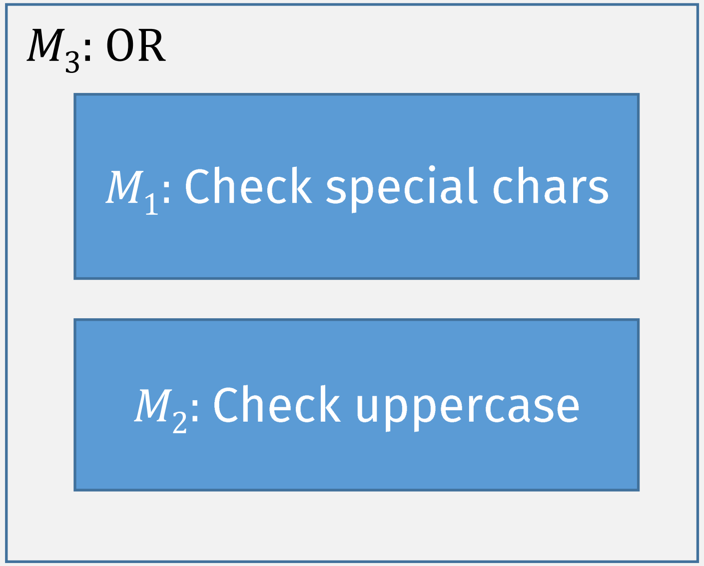
A set is **<u>closed</u>** under an operation if:
the result of applying the operation to members of the set is still in the set

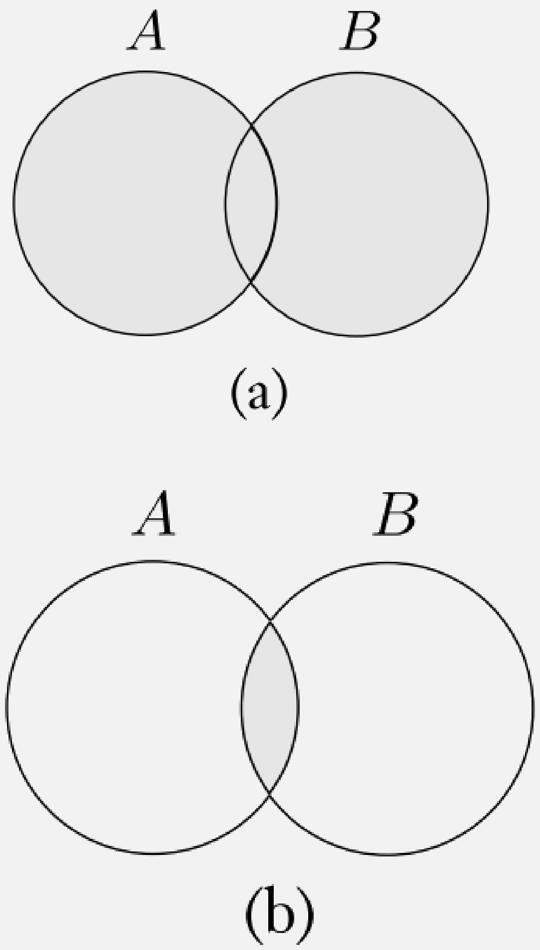
- Natural numbers = $\{0, 1, 2, \ldots\}$
  - <u>Closed</u> under addition:
    - if $x$ and $y$ are Natural number,
    - then $z = x + y$ is a Natural number
  - Closed under multiplication?
    - **yes**
  - Closed under subtraction?
    - **no**
- Integers = $\{\ldots, -2, -1, 0, 1, 2, \ldots\}$
  - <u>Closed</u> under addition and multiplication
  - Closed under subtraction?
    - **yes**
  - Closed under division?
    - **no**
- Rational numbers = $\{x \mid x = y/z, y$ and $z$ are Integers$\}$
  - Closed under division?
    - **No?**
    - **Yes if** $z$ **!=0**

# Why Care About Closed Ops on Reg Langs?

- Closed operations preserves “regularness”
- I.e., it preserves the same computation model!
- So result of combining machines can be combined again

# Password checker: “Or” = “Union”



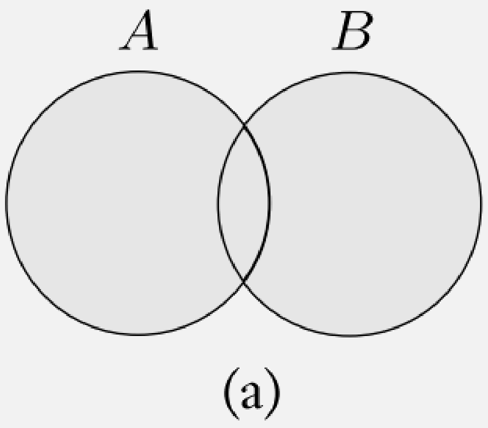

???

# Password checker: “Or” = “Union”

$M_3$: OR

$M_1$: Check special chars

$M_2$: Check uppercase



**Union**: $A \cup B = \{x \mid x \in A \text{ or } x \in B\}$

# Union of Languages

Let the alphabet $\Sigma$ be the standard 26 letters $\{\mathtt{a}, \mathtt{b}, \ldots, \mathtt{z}\}$.

If $A = \{\mathtt{good}, \mathtt{bad}\}$ and $B = \{\mathtt{boy}, \mathtt{girl}\}$, then

$$A \cup B = \{\mathtt{good}, \mathtt{bad}, \mathtt{boy}, \mathtt{girl}\}$$

# A Closed Operation: Union

**THEOREM**

The class of regular languages is closed under the union operation.

In other words, if $A_1$ and $A_2$ are regular languages, so is $A_1 \cup A_2$.

- How do we prove that a language is regular?
  - Create a FSM recognizing it!
- So to prove this theorem …
  create a machine that combines the machines of $A_1$ and $A_2$.

# Kinds of Mathematical Proof

- Proof by construction
  - Construct the mathematical object in question

- E.g., To prove that language $A_1 \cup A_2$ is regular … construct a finite state machine recognizing it!

# Union Closed?

**THEOREM**

The class of regular languages is closed under the union operation.

In other words, if $A_1$ and $A_2$ are regular languages, so is $A_1 \cup A_2$.

*Proof*

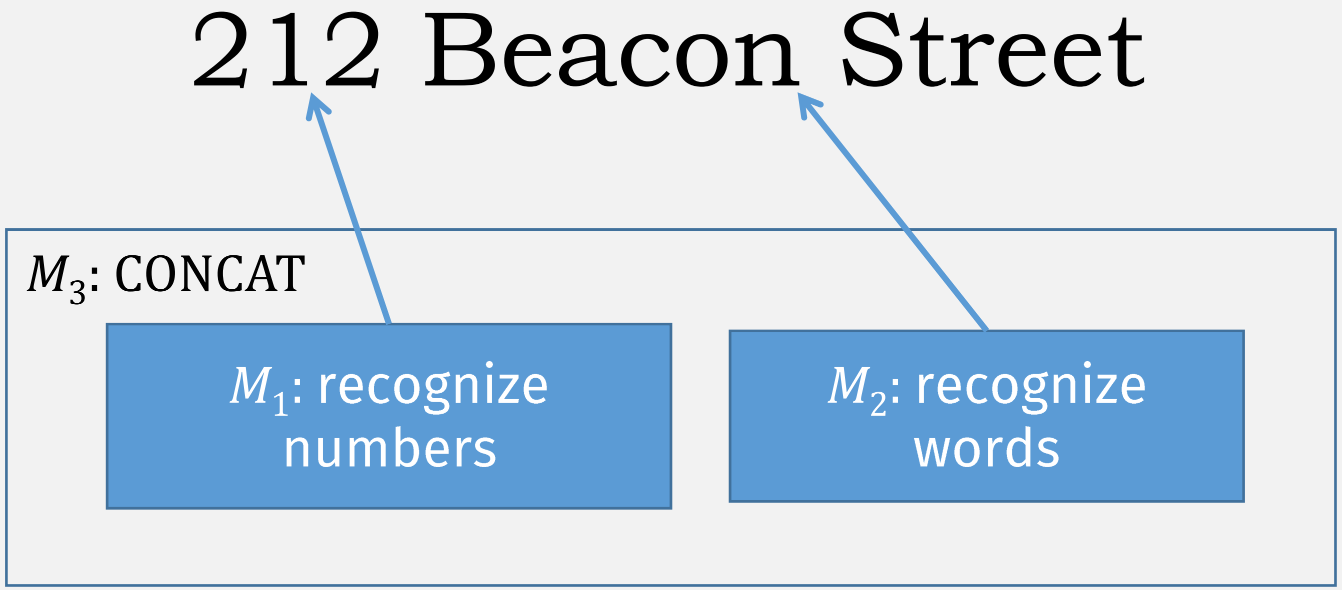
- Given: $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, recognize $A_1$,
  $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$, recognize $A_2$,

*M* runs its input on both $M_1$ and $M_2$ in parallel; accept if either accepts

- Construct: a new machine $M = (Q, \Sigma, \delta, q_0, F)$ using $M_1$ and $M_2$
- states of ***M***: $Q = \{(r_1, r_2) \mid r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
  This set is the ***Cartesian product*** of sets $Q_1$ and $Q_2$
- ***M*** transition fn: $\delta\big((r_1, r_2), a\big) = \big(\delta_1(r_1, a), \delta_2(r_2, a)\big)$ — a step in $M_1$, a step in $M_2$
- ***M*** start state: $(q_1, q_2)$
- ***M*** accept states: $F = \{(r_1, r_2) \mid r_1 \in F_1 \text{ or } r_2 \in F_2\}$ — Accept if either $M_1$ or $M_2$ accept

# Another operation: Concatenation

Example: **Recognizing street addresses**



**Concatenation**: $A \circ B = \{xy | \, x \in A \text{ and } y \in B\}$

# Concatenation of Languages

Let the alphabet $\Sigma$ be the standard 26 letters $\{\texttt{a}, \texttt{b}, \ldots, \texttt{z}\}$.

If $A = \{\texttt{good}, \texttt{bad}\}$ and $B = \{\texttt{boy}, \texttt{girl}\}$, then

$$A \circ B = \{\texttt{goodboy}, \texttt{goodgirl}, \texttt{badboy}, \texttt{badgirl}\}$$

# Is Concatenation Closed?

**THEOREM**

The class of regular languages is closed under the concatenation operation.

In other words, if $A_1$ and $A_2$ are regular languages then so is $A_1 \circ A_2$.

- Construct a <u>new</u> machine $M$? (like union)
  - From DFA $M_1$ (which recognizes $A_1$),
  - and DFA $M_2$ (which recognizes $A_2$)

# Is Concatenation Closed?

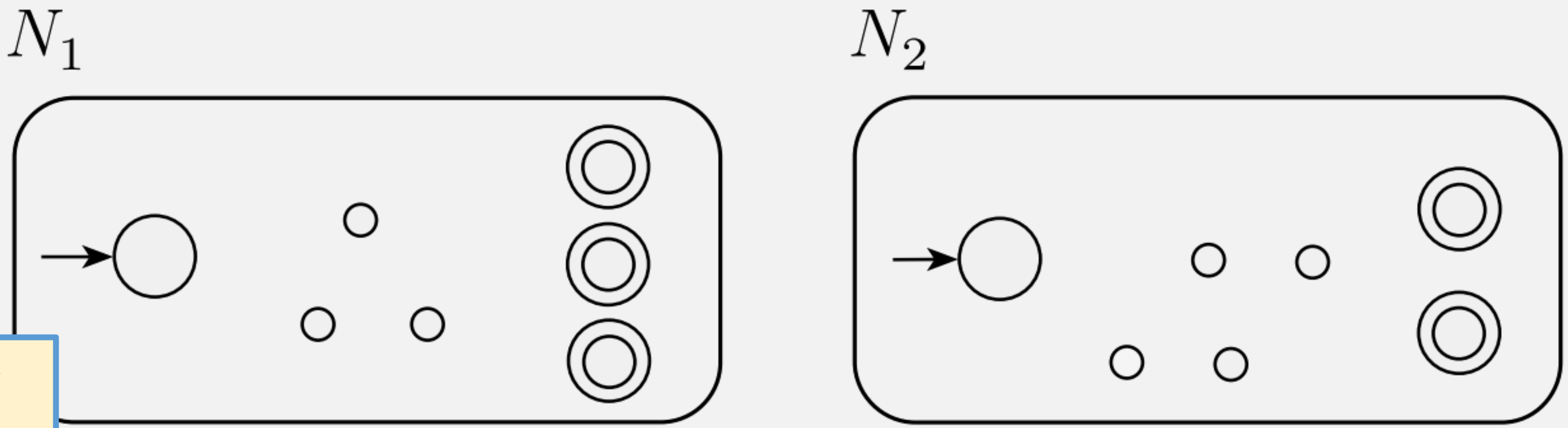
**THEOREM**

The class of regular languages is closed under the concatenation operation.

In other words, if $A_1$ and $A_2$ are regular languages then so is $A_1 \circ A_2$.

- Can't directly combine $A_1$ and $A_2$ because:
  - Need switch from $A_1$ to $A_2$
  - But don't know when! (can only read input once)
- Need a new kind of machine!
- So is concatenation not closed for reg langs???

## Concatentation



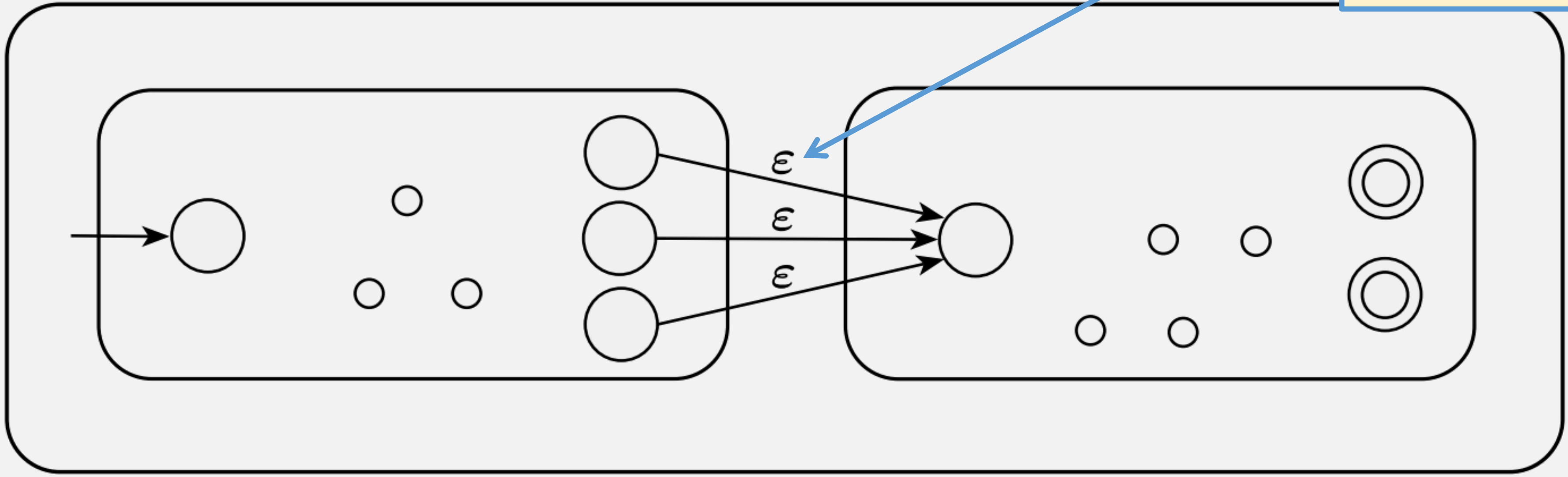

$N$ is a new kind of machine, an **NFA!**
(next time)

Let $N_1$ recognize $A_1$, and $N_2$ recognize $A_2$.

Want: Construction of $N$ to recognize $A_1 \circ A_2$

ε = empty string = no input

So $N$ can:
- stay in current state **and**
- move to next state



# Check-in Quiz 1/26

On gradescope