CS 420 / CS 620

(Deterministic) Finite Automata

Wednesday, September 10, 2025

UMass Boston Computer Science

Announcements

HW 0

- ~~Due: 9/10 noon EDT~~

HW 1

- Out: 9/8
- Due: 9/15 noon EDT

Lecture videos (from 420-02)

- Posted to Canvas
- Presented as-is, no guarantees
  - Not accepting questions
- For emergency / supplemental use only
- Attendance still taken in lectures

HW Hints and Reminders

- Problems must be: <u>assigned to the correct pages</u>

- Proof format must be: a **Statements** and **Justifications** table

- Machine formal descriptions must have:
  - **a name** (if required), e.g., M = …
  - **a tuple with 5 components,**
    - e.g., $M = (Q, \Sigma, \delta, q_0, F)$ (where each variable is subsequently defined)
    - **inline is ok** (make sure it's readable)
  - **components of the correct type**
    - E.g., **set** or **sequence** or **???**

How to ask for HW help

(there's **no such thing** as a **stupid question**, but …)

… there **is such thing** as a **less useful question** (gets less useful answers)

- **"Is this correct?"**
- **"I don't get it"**
- **"Give me a hint?"**
- **"Do I need to do the thing DFA thing?"**

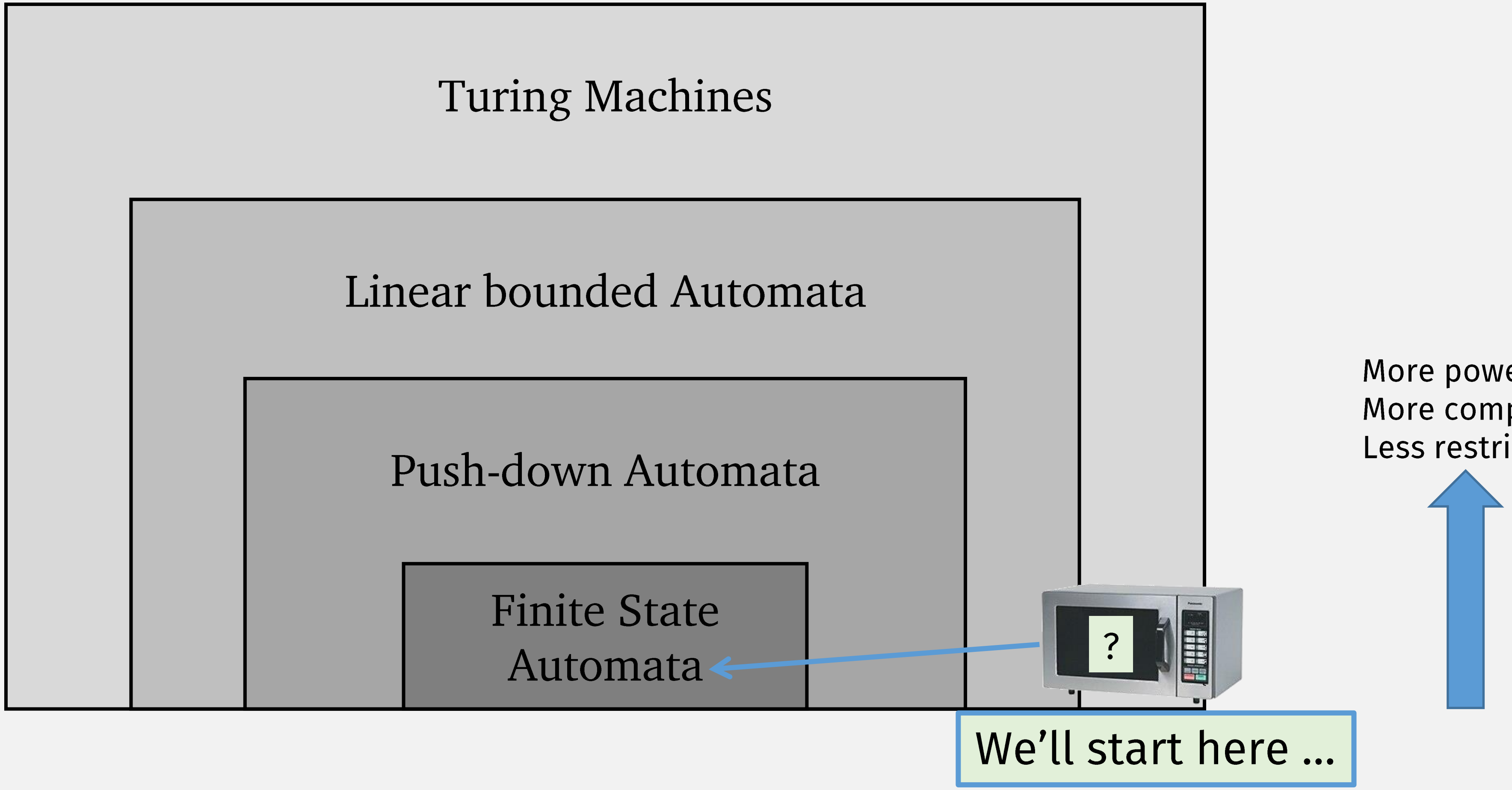
Useful question examples (gets useful answers):

- **"I'm don't understand this notation $A \otimes B \ggg C$ … and I couldn't find it in the book"**
- **"I couldn't find this word's definition …"**

Most HW questions can be answered by **looking up the meaning of a word** or **notation** or **definition!**

Last Time

Models of Computation Hierarchy

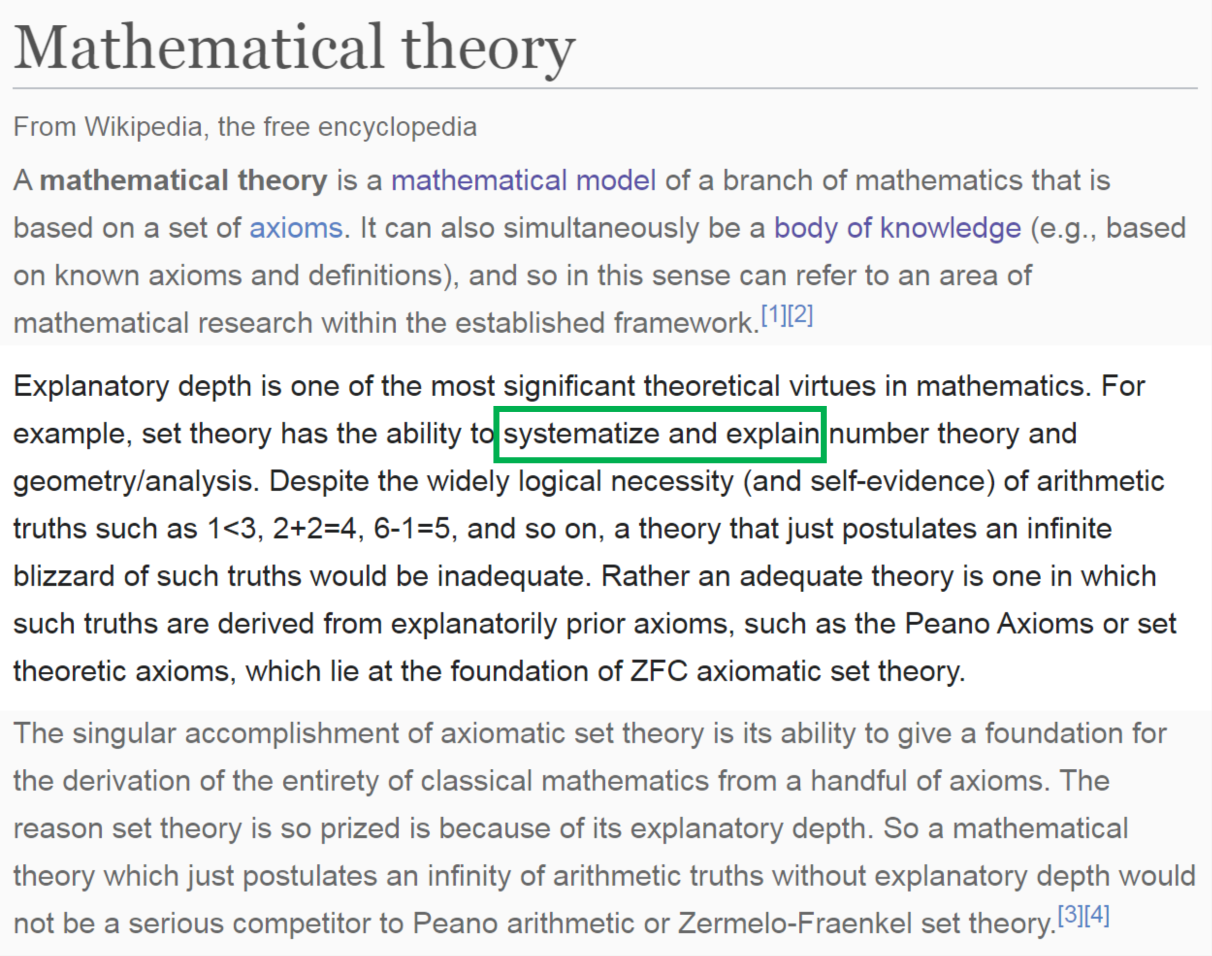


We'll start here …

More powerful
More complex
Less restricted

Last Time

A (Mathematical) Theory …



Mathematical theory

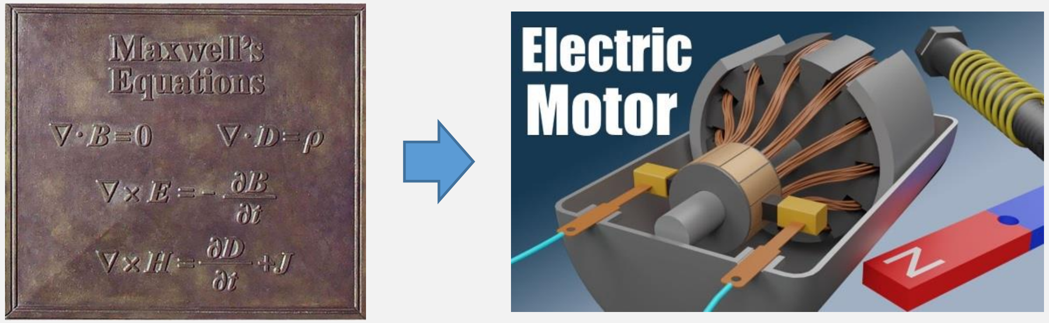
From Wikipedia, the free encyclopedia

A **mathematical theory** is a mathematical model of a branch of mathematics that is based on a set of axioms. It can also simultaneously be a body of knowledge (e.g., based on known axioms and definitions), and so in this sense can refer to an area of mathematical research within the established framework.[1][2]

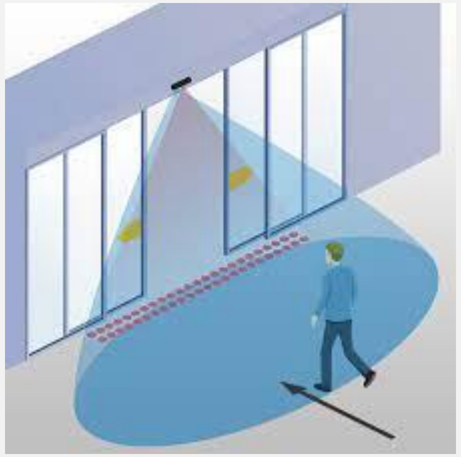
Explanatory depth is one of the most significant theoretical virtues in mathematics. For example, set theory has the ability to systematize and explain number theory and geometry/analysis. Despite the widely logical necessity (and self-evidence) of arithmetic truths such as 1<3, 2+2=4, 6-1=5, and so on, a theory that just postulates an infinite blizzard of such truths would be inadequate. Rather an adequate theory is one in which such truths are derived from explanatorily prior axioms, such as the Peano Axioms or set theoretic axioms, which lie at the foundation of ZFC axiomatic set theory.

The singular accomplishment of axiomatic set theory is its ability to give a foundation for the derivation of the entirety of classical mathematics from a handful of axioms. The reason set theory is so prized is because of its explanatory depth. So a mathematical theory which just postulates an infinity of arithmetic truths without explanatory depth would not be a serious competitor to Peano arithmetic or Zermelo-Fraenkel set theory.[3][4]

… must **explain (predict)** some real-world phenomena …

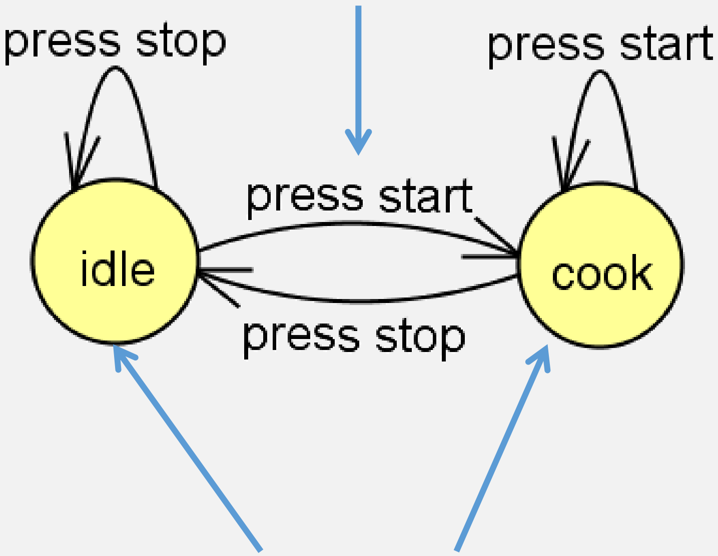

Maxwell's Equations
$\nabla\cdot B=0$
$\nabla\cdot D=\rho$
$\nabla\times E=-\frac{\partial B}{\partial t}$
$\nabla\times H=\frac{\partial D}{\partial t}+J$
Electric Motor
N

Finite Automata: “Simple” Computation / “Programs”



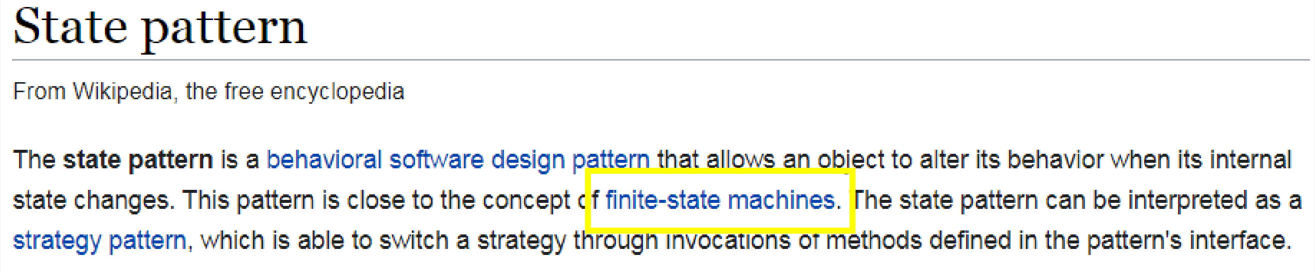
A Microwave Finite Automata

Input "symbols" change states
(possibly)



States

Finite Automata: Not Just for Appliances



State pattern

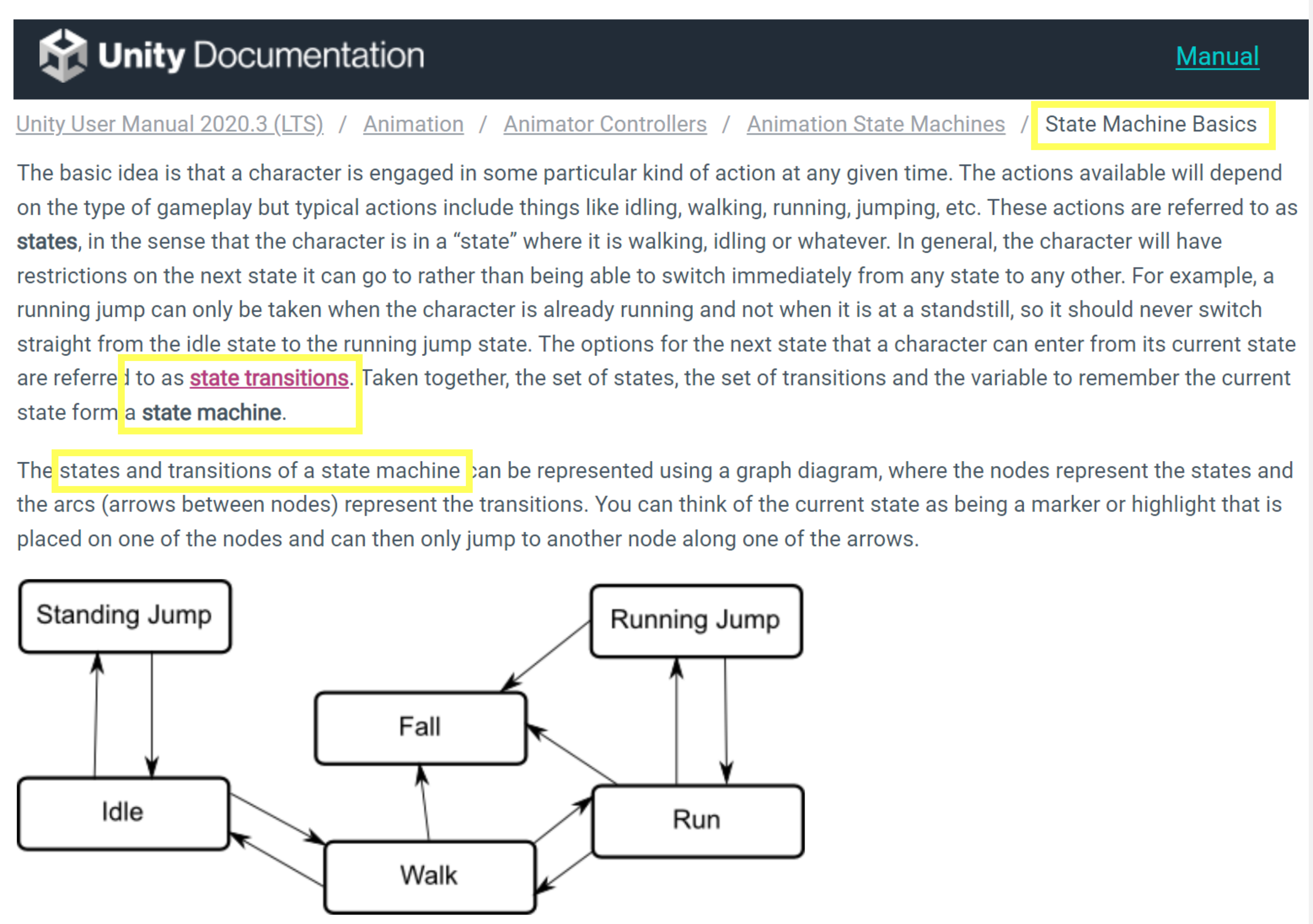
From Wikipedia, the free encyclopedia

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

(More powerful?) **Computation**
"Simulating" other (weaker?) **Computation**
(a common theme this semester)

Video Games Love Finite Automata



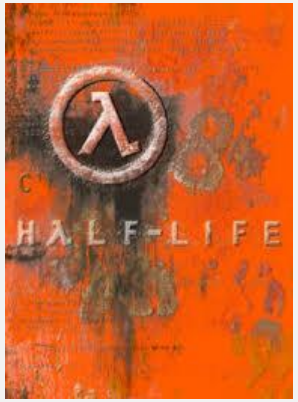
Unity Documentation Manual

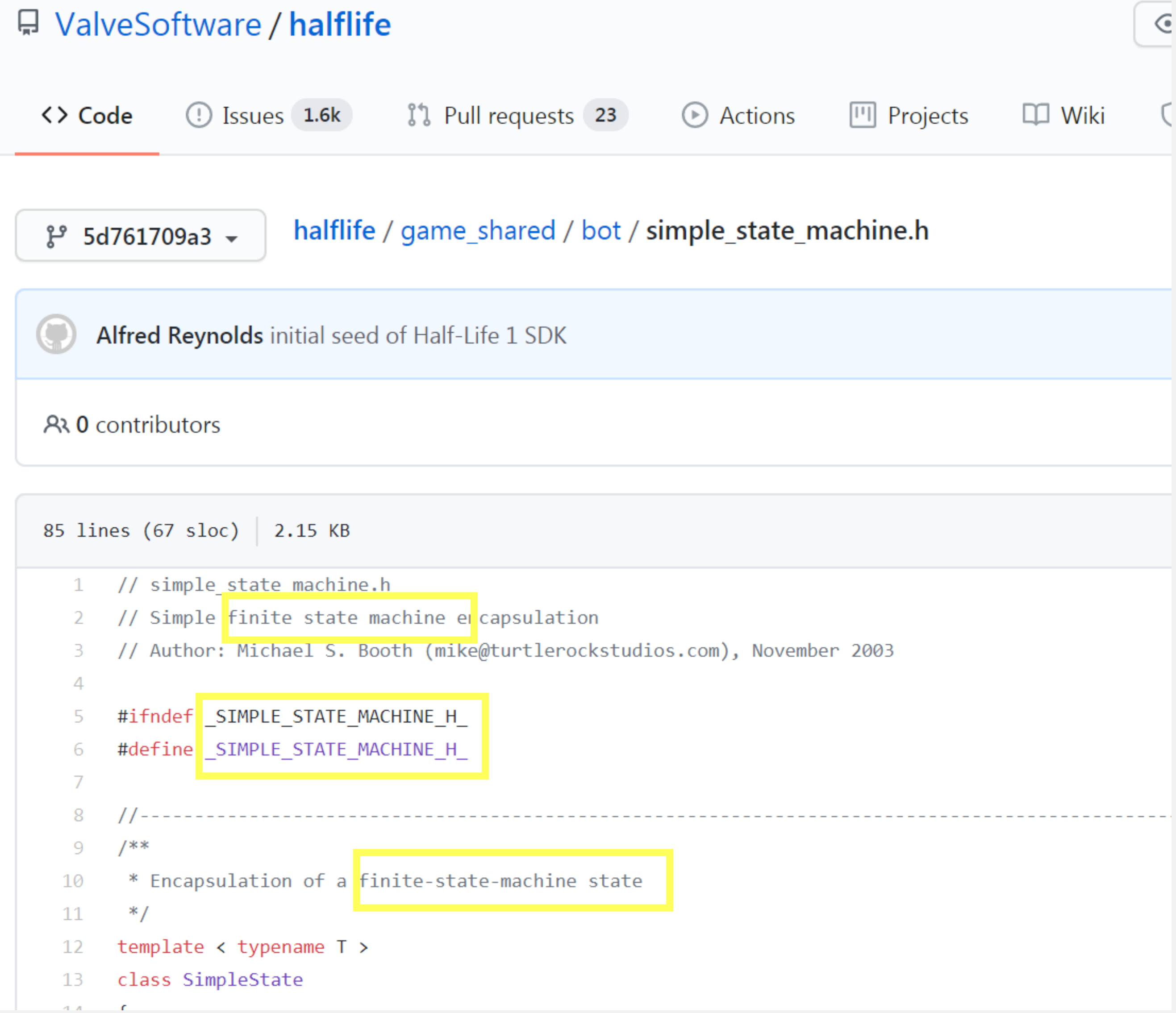
Unity User Manual 2020.3 (LTS) / Animation / Animator Controllers / Animation State Machines / State Machine Basics

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





ValveSoftware / halflife

halflife / game_shared / bot / simple_state_machine.h

Alfred Reynolds initial seed of Half-Life 1 SDK

0 contributors

85 lines (67 sloc) | 2.15 KB

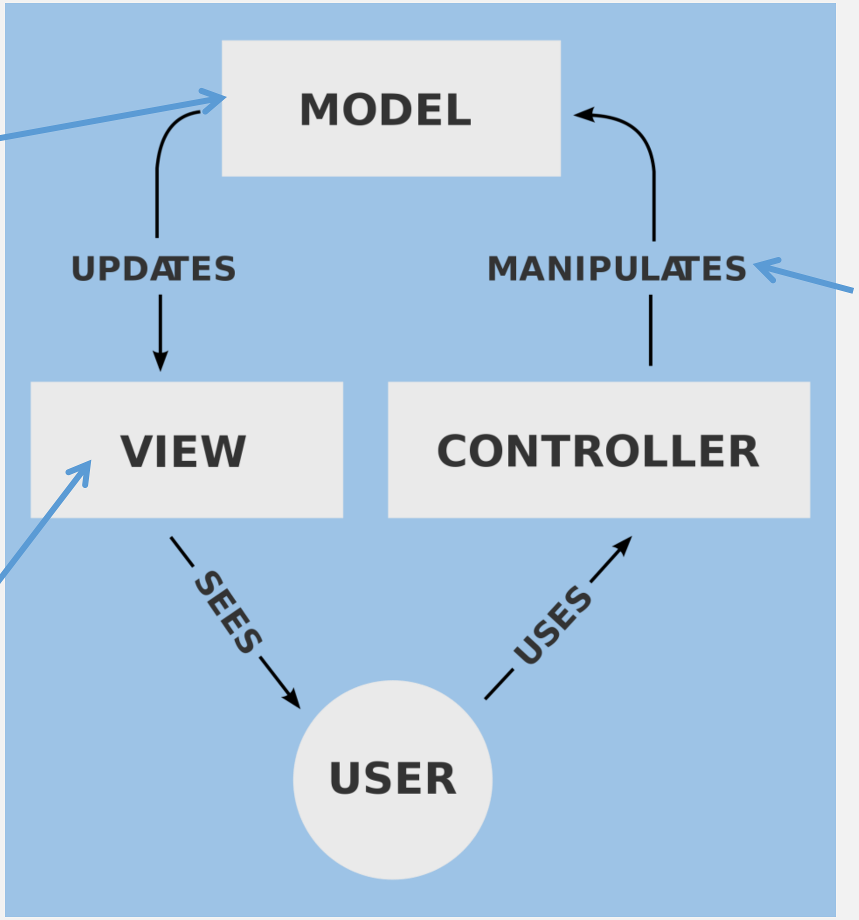
```
// simple_state_machine.h
// Simple finite state machine encapsulation
// Author: Michael S. Booth (mike@turtlerockstudios.com), November 2003

#ifndef _SIMPLE_STATE_MACHINE_H_
#define _SIMPLE_STATE_MACHINE_H_

//----------------------------------------------------------------------------------------------
/**
 * Encapsulation of a finite-state-machine state
 */
template < typename T >
class SimpleState
{
```

Model-view-controller (MVC) is an FSM

States



Input events change states

```
onclick
onload
onmouseover
onmousedown
…
```

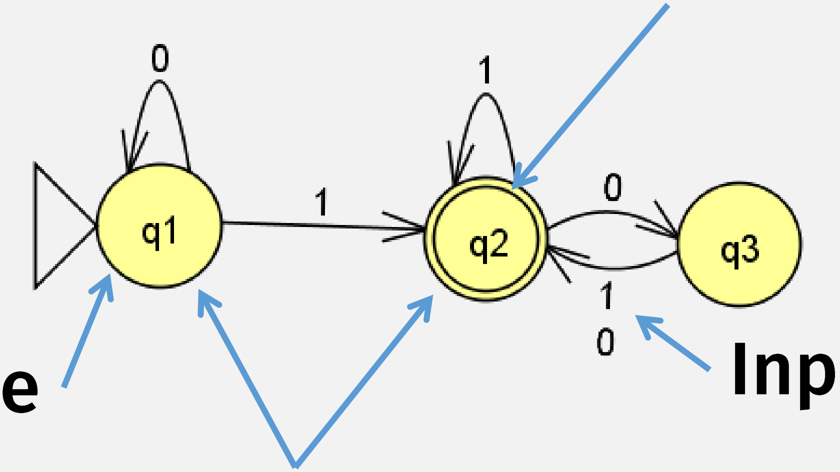
The **View** draws states

Analogy: Finite Automata is a “Program”

- A **restricted “program”** with access to <u>**finite**</u> **memory**
  - Actually, only <u>**1 “cell”** of **memory**</u>**!**
  - Possible **contents** of **memory = # of “states”**

- **Finite Automata has different representations:**
  - **Code** (won’t use in this class)
  - ➢**State diagrams**

Finite Automata state diagram



Accept State

Start State

States

Inputs specify **state transitions**

Analogy: Finite Automata is a “Program”

- A **restricted “program”** with access to <u>**finite**</u> **memory**
  - Only <u>**1 “cell”** of **memory**</u>**!**
  - Possible **contents** of **memory = # of states**

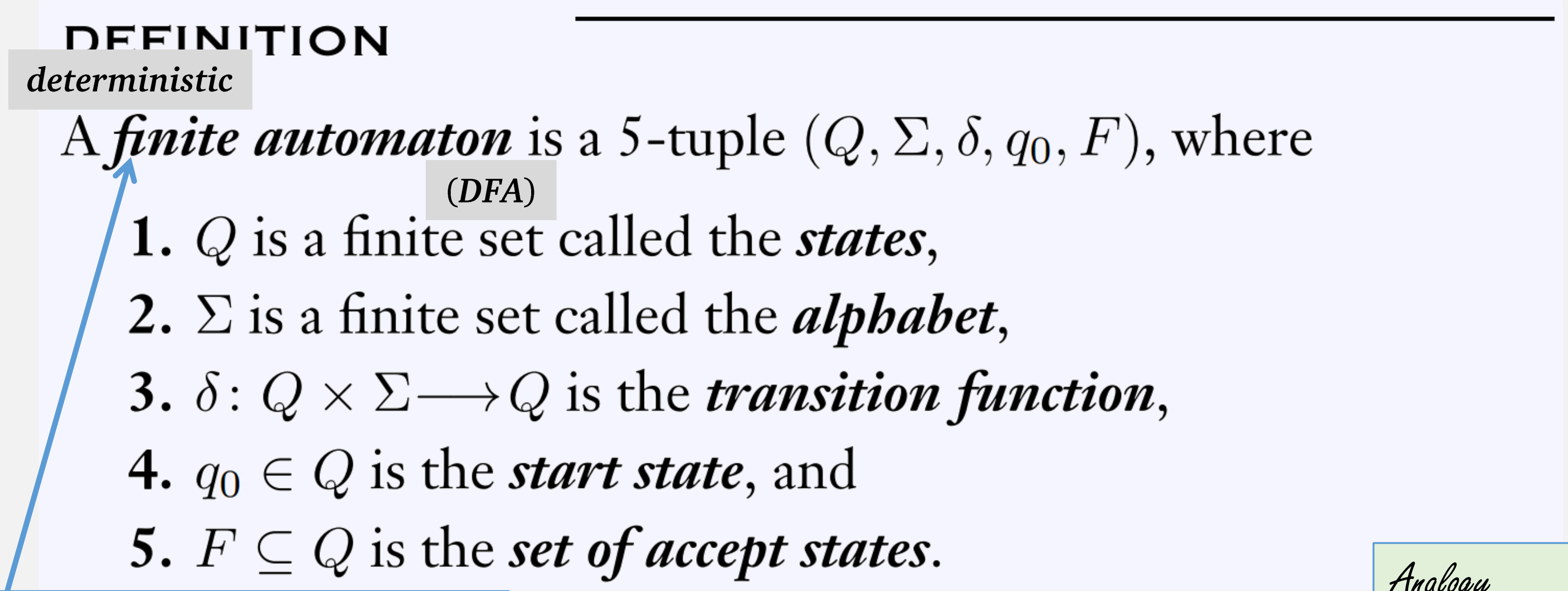
- **Finite Automata has different representations:**
  - **Code** (won’t use in this class)
  - ➢**State diagrams**

Analogy: Finite Automata is a "Program"

- A **restricted "program"** with access to finite **memory**
  - Only 1 "cell" of **memory**!
  - Possible **contents** of **memory = # of states**

- **Finite Automata has different representations:**
  - **Code** (won't use in this class)
  - **State diagrams**
  - ➢ **Formal math description**
    (essentially same as **code** but **in a very different "programming language"**)

Finite Automata: The Formal Definition



DEFINITION

deterministic

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

(DFA)

1. Q is a finite set called the ***states***,

2. Σ 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***.

This semester

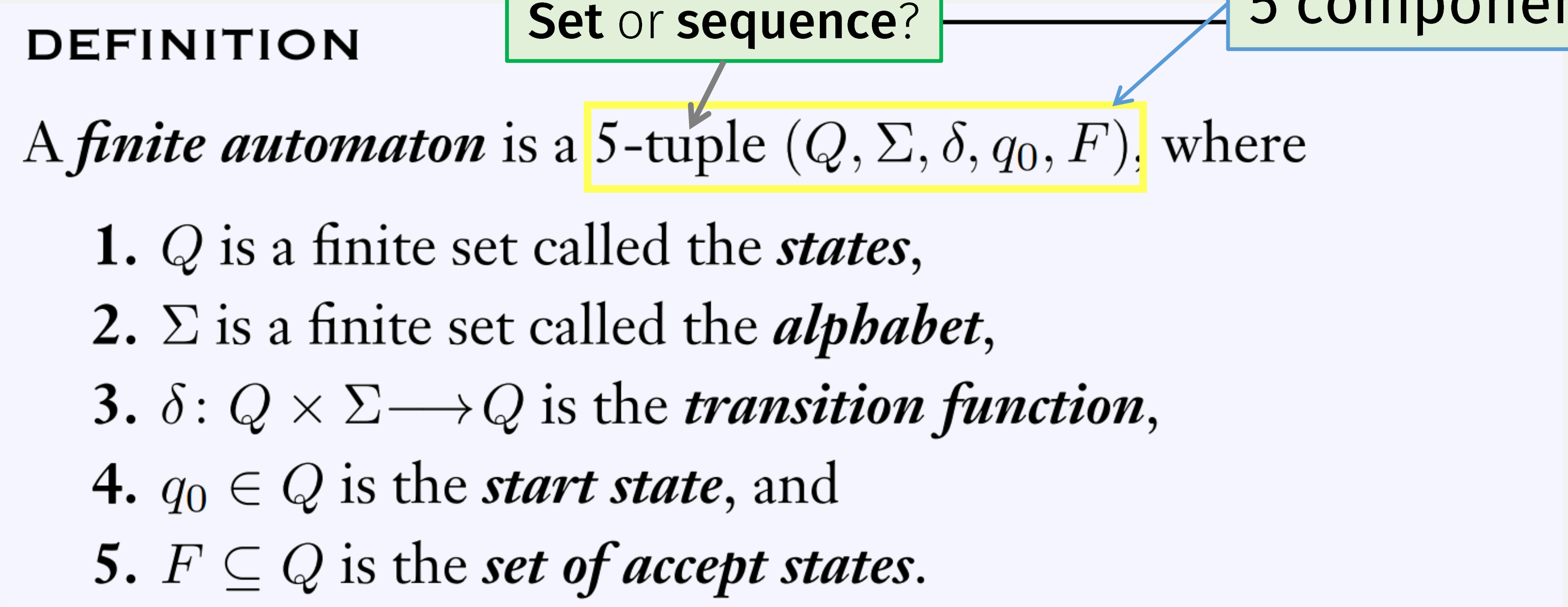
Things in **bold** have **precise formal definitions.**

(be sure to look up and review the definition whenever you are unsure)

Analogy

This is the "**programming language**" for **(deterministic) finite automata** "programs"

Finite Automata: The Formal Definition



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. Σ 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***.

Set or **sequence**?

5 components

Interlude: Sets and Sequences

- Both are: **mathematical objects that group other objects**
- **Members** of the group are called **elements**
- Can be: **empty, finite,** or **infinite**
- Can contain: **other sets** or **sequences**

Sets

- Unordered
- Duplicates not allowed
- Notation: { }
- **Empty set** written: Ø or { }
- A **language** is a (possibly infinite) set of strings

A **set** used a lot in this course

Sequences

- Ordered
- Duplicates ok
- Notation: **varies:** ( ), **comma,** or **concat**
- **Empty sequence:** ( )
- A **tuple** is a finite sequence
- A **string** is a finite sequence of characters

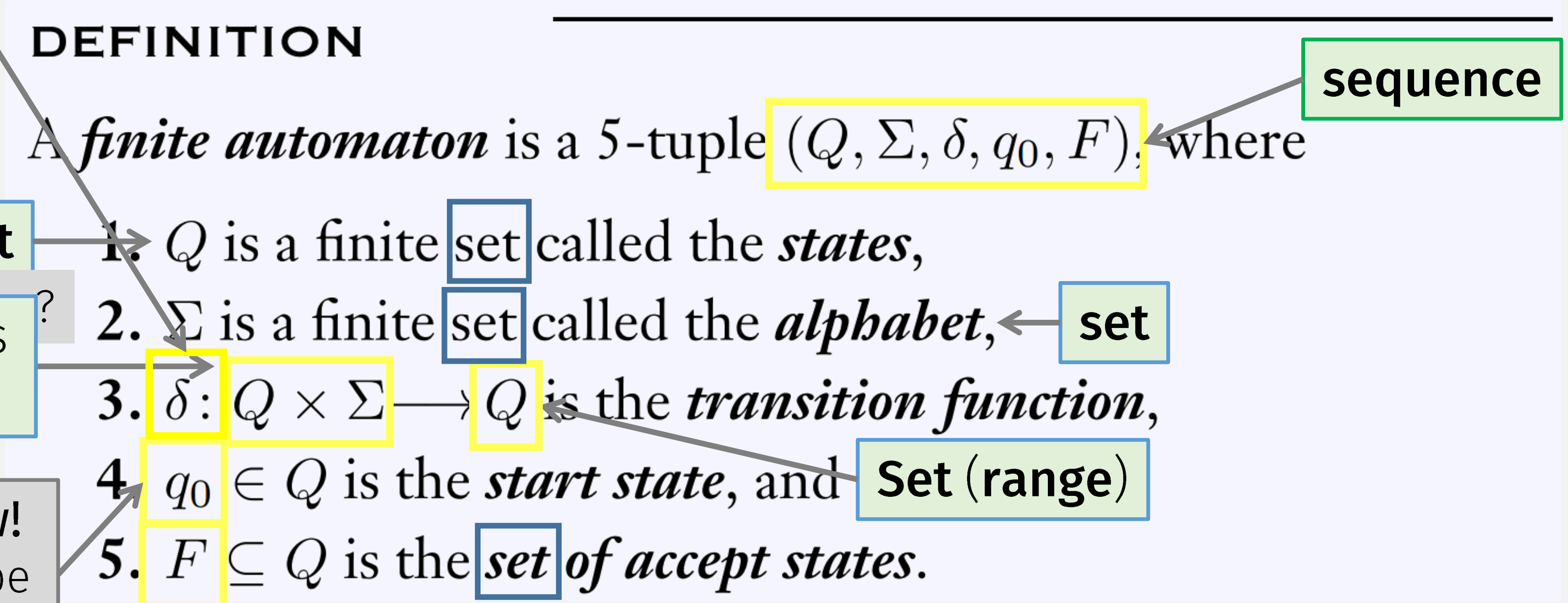
sequences used a lot in this course

Set or Sequence ?

A **function** is ...

... a **set** of **pairs**
(1st of each pair **from domain**, 2nd **from range**)

... has many representations:
a <u>mapping</u>, a <u>table</u>, ...



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. Σ 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***.

sequence

set

set

Set of pairs
(**domain**)

Set (**range**)

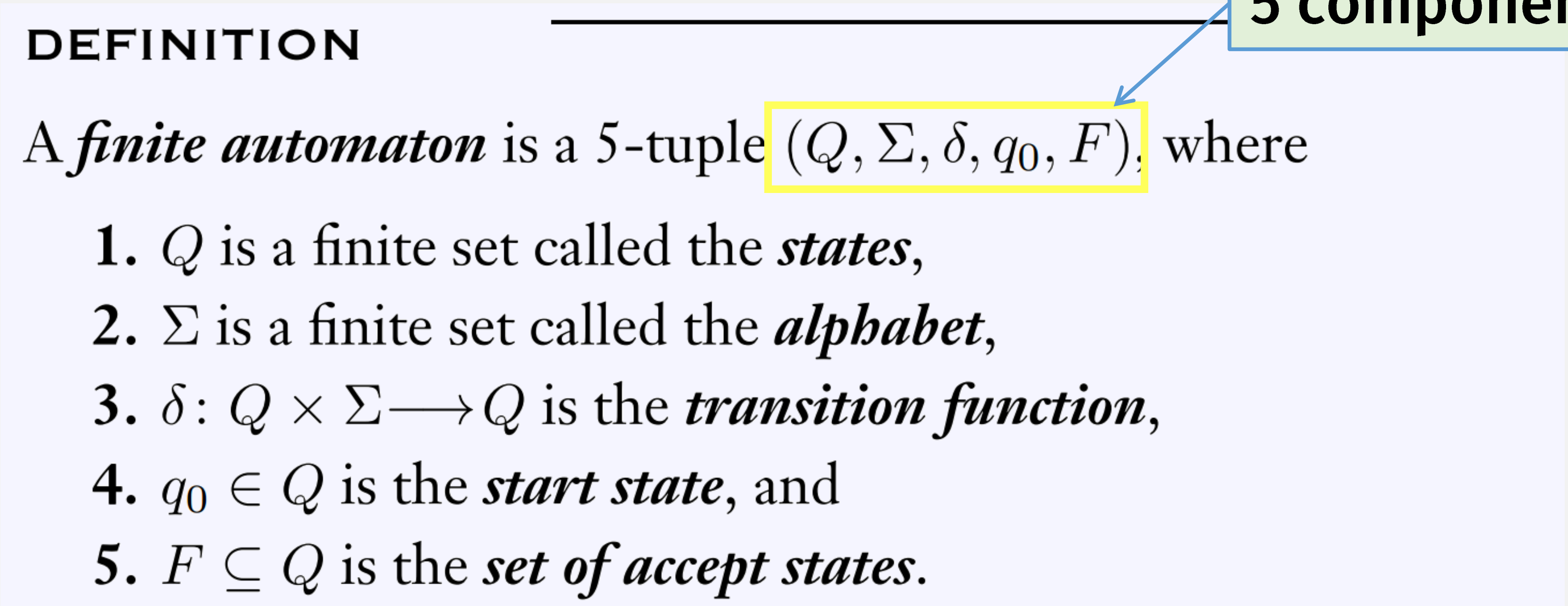
Don't know!
(states can be anything)

set

A **pair** is ...

a **sequence** of 2 elements

Finite Automata: The Formal Definition



DEFINITION

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. Σ 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***.

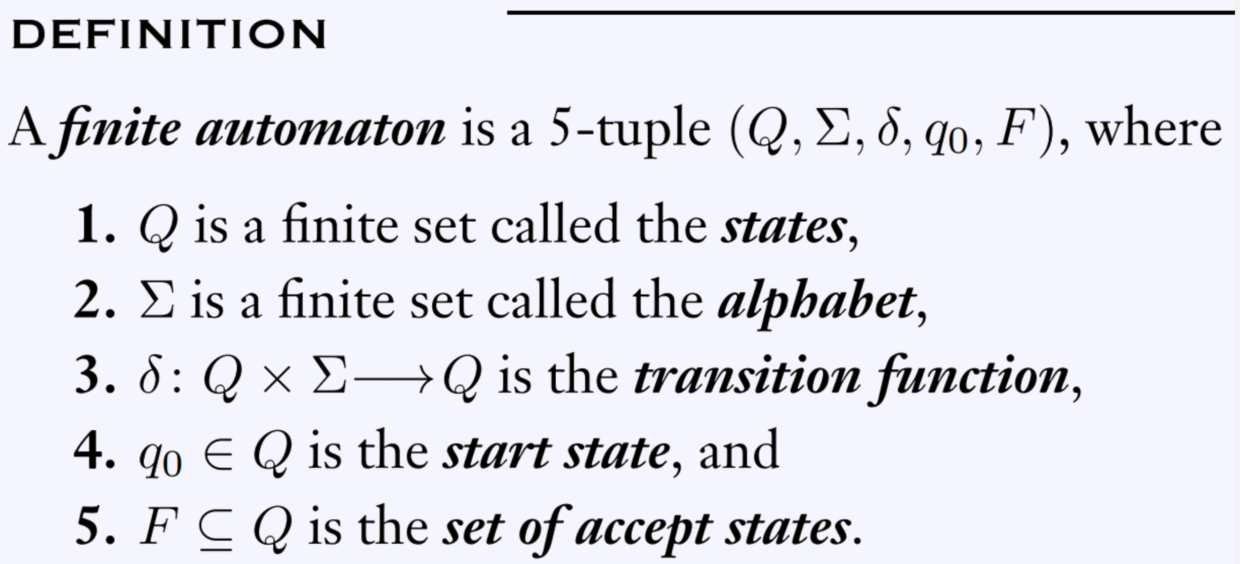
5 components

Analogy: Finite Automata is a "Program"

- A **restricted "program"** with access to <u>finite</u> **memory**
  - Only <u>1 **"cell"** of **memory**</u>!
  - Possible **contents** of **memory = # of states**

- Finite Automata has different <u>equivalent</u> representations:
  - **Code** (won't use in this class)
  - ➢State diagrams
  - ➢Formal math description
    (think of it as **code in a very different "programming language"**)

Finite Automata: State Diagram

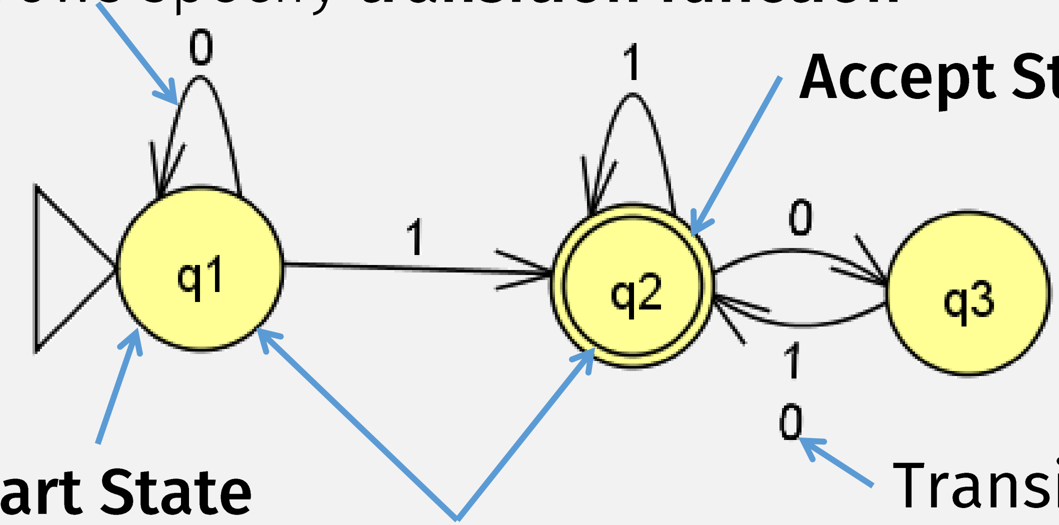


DEFINITION

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

1. Q is a finite set called the ***states***,
2. Σ 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***.

Arrows specify **transition function**

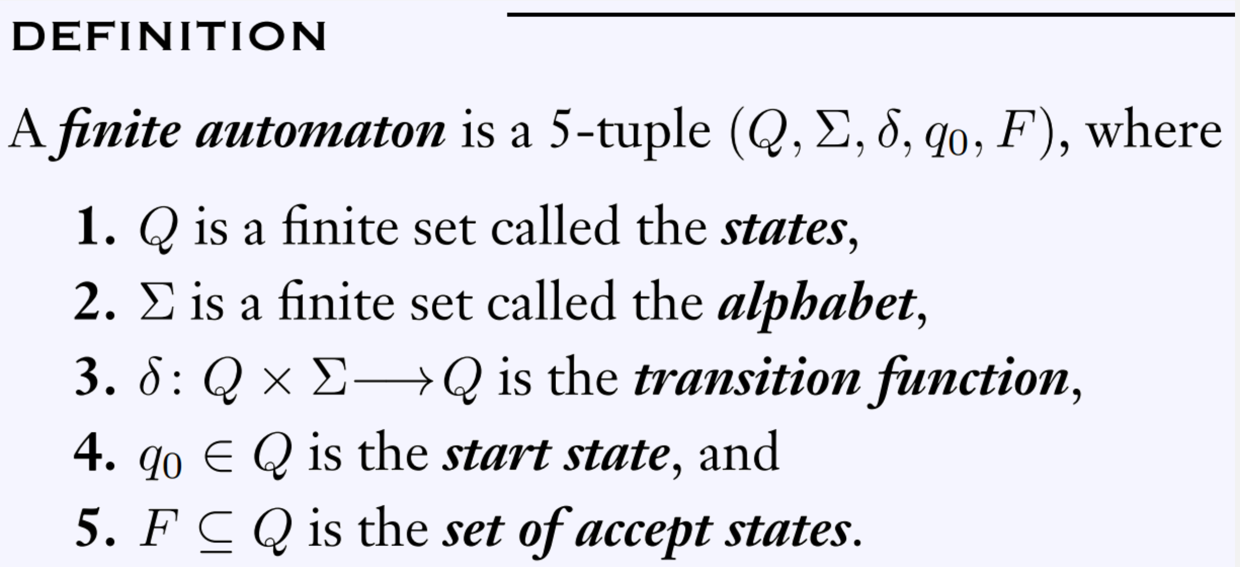


Accept State(s)

Start State

States

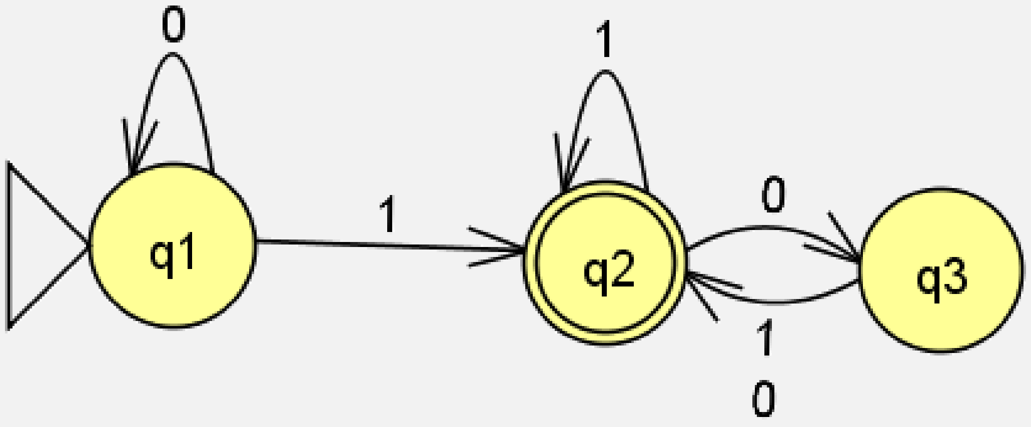
Transition labels must be from **alphabet**



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. Σ 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***.



An **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. Σ 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***.

Note:
Not the same Q

An **Example** (as **formal description**)

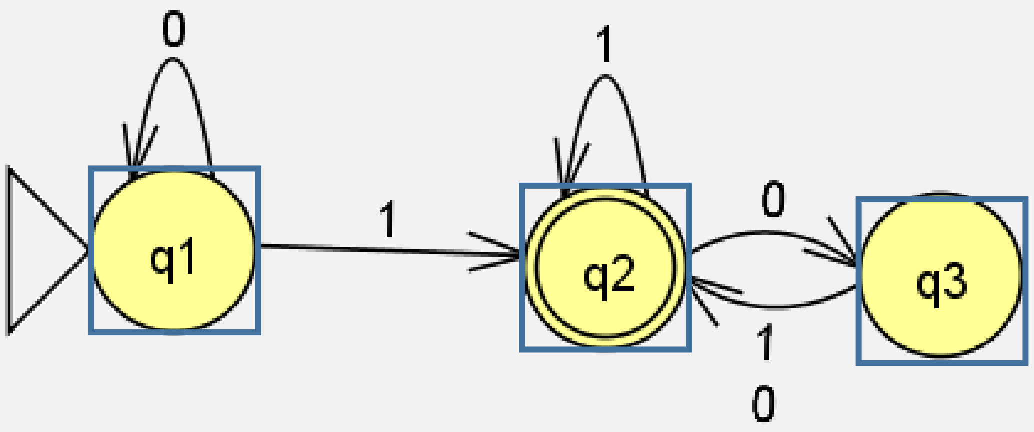
$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ 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)

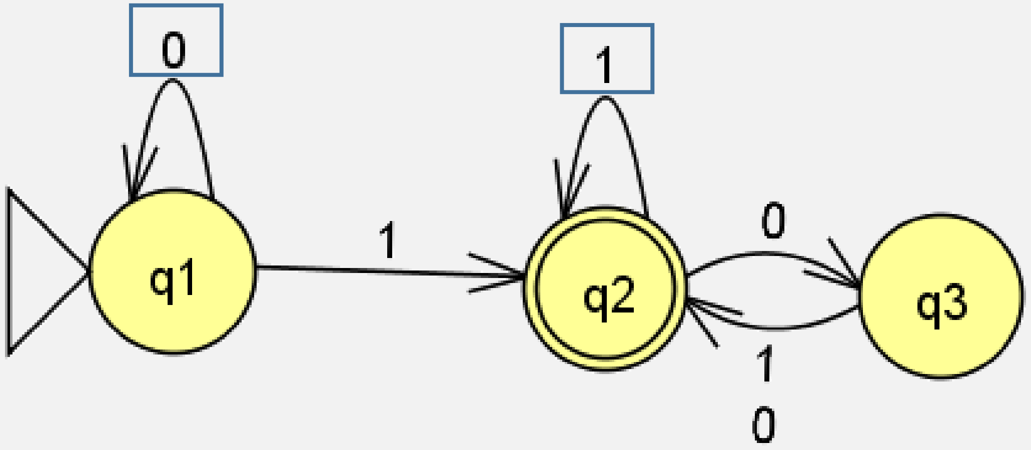


An **Example** (as **state diagram**)

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. Σ 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***.



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

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,

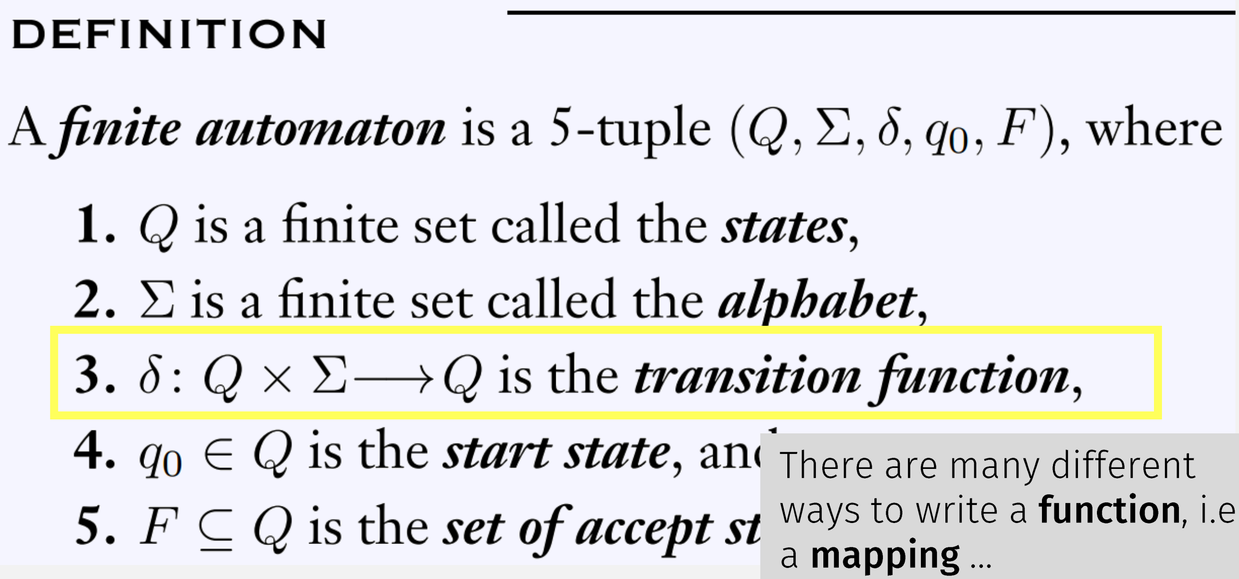
Possible chars of input

3. δ is described

Alphabet defines **all** possible input strings for the machine

	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. $F = \{q_2\}$.



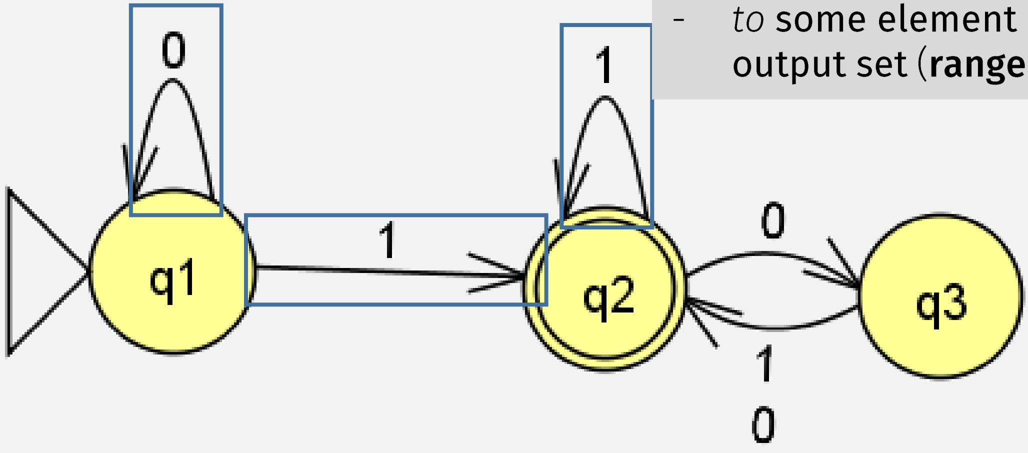
DEFINITION

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

1. Q is a finite set called the ***states***,
2. Σ 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 st***

There are many different ways to write a **function**, i.e., a **mapping** …

- *from* <u>every</u> element in the input set(s) (**domain**)
- *to* some element in the output set (**range**)



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

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

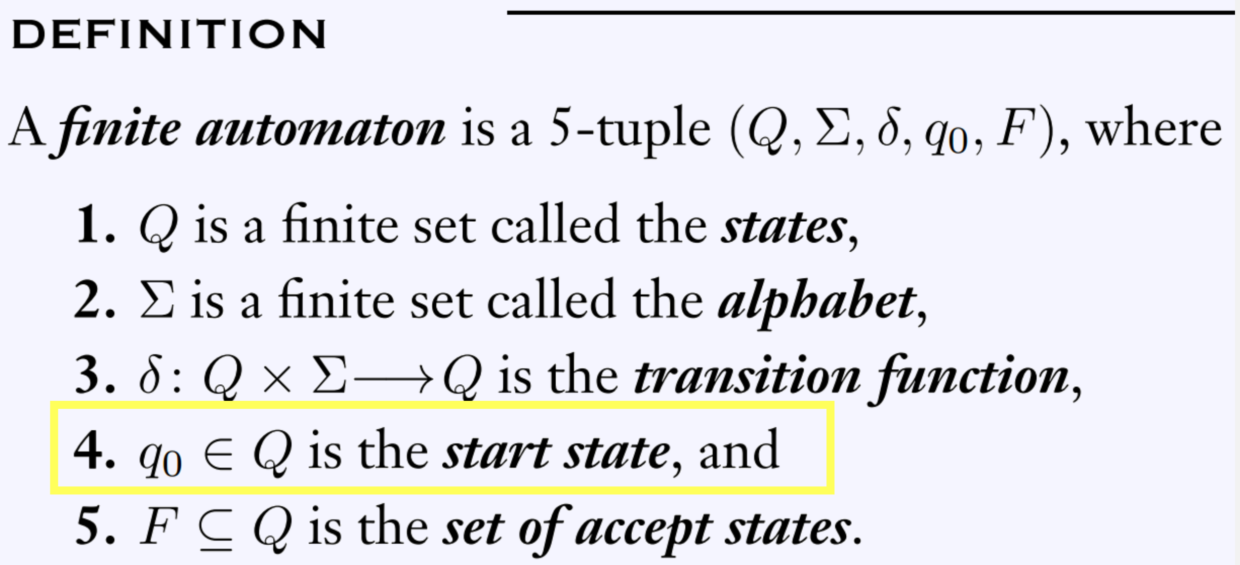
“And this is next input symbol”

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

“If in this state”

“Then go to this state”

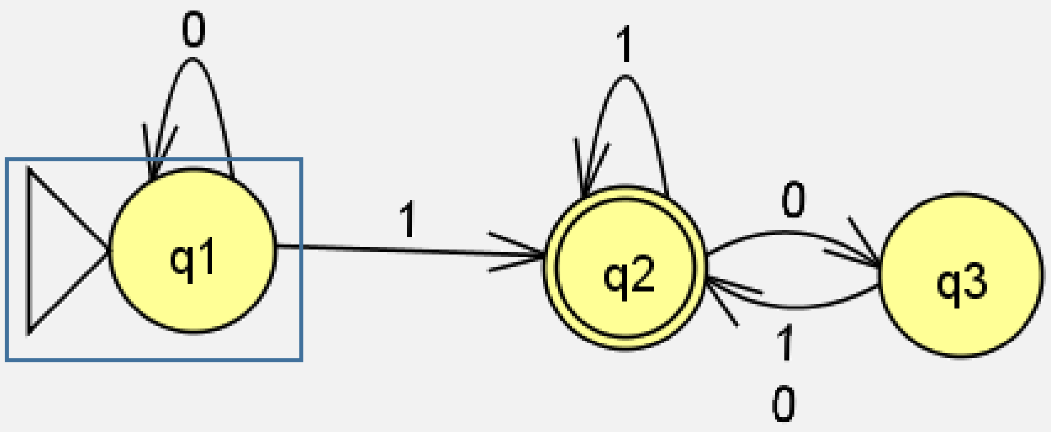
4. q_1 is the start state, and
5. $F = \{q_2\}$.



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. Σ 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***.

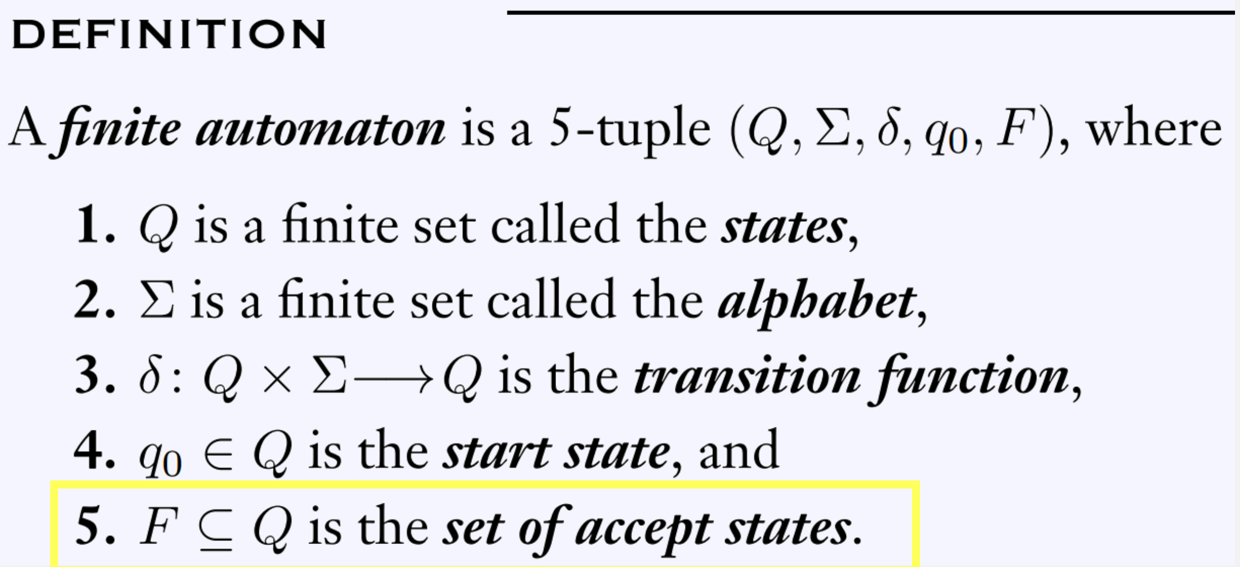


$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ 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. $F = \{q_2\}$.

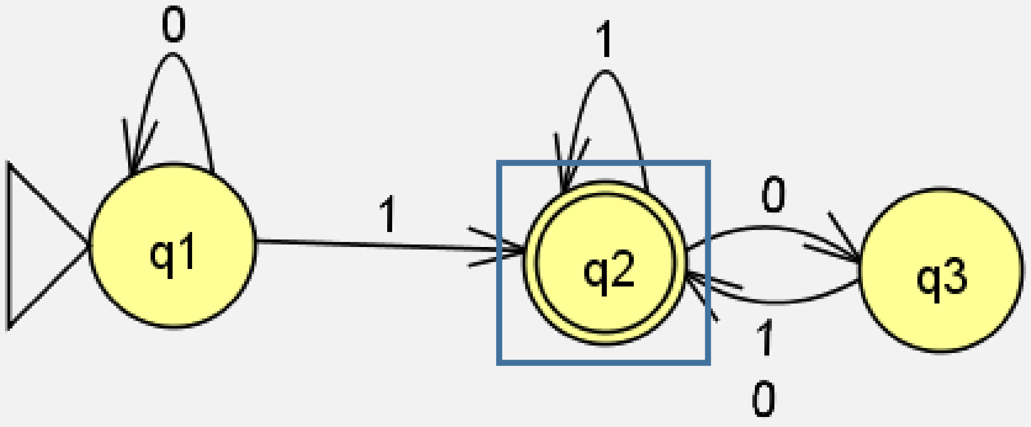


DEFINITION

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

1. Q is a finite set called the ***states***,
2. Σ 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***.

WARNING: This is a **set**!



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

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ 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. $F = \{q_2\}$.

WARNING: This is a **set**!

Writing a non-set here makes the whole thing **an invalid DFA**

A “Programming Language”

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. Σ 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***.

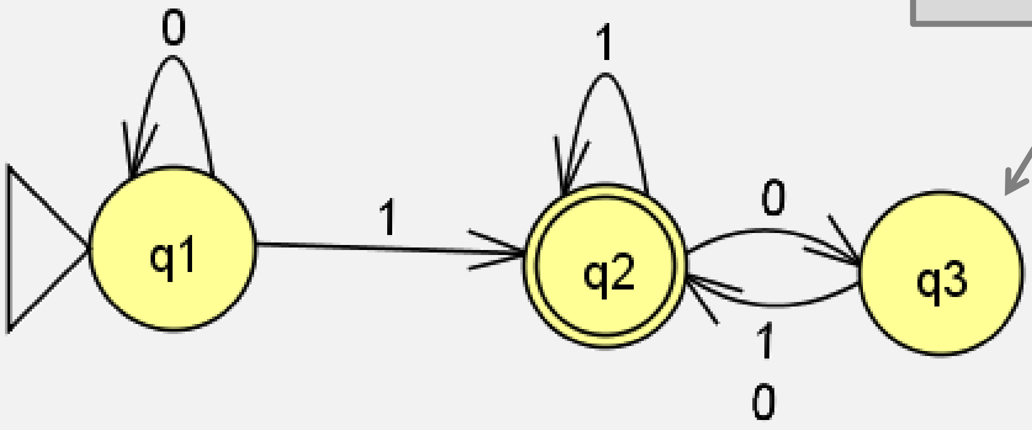
An **Example** (as **formal description**)

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

1. $Q = \{q_1, q_2, q_3\}$,
2. $\Sigma = \{0,1\}$,
3. δ 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. $F = \{q_2\}$.



A “Program”

“Programming” Analogy

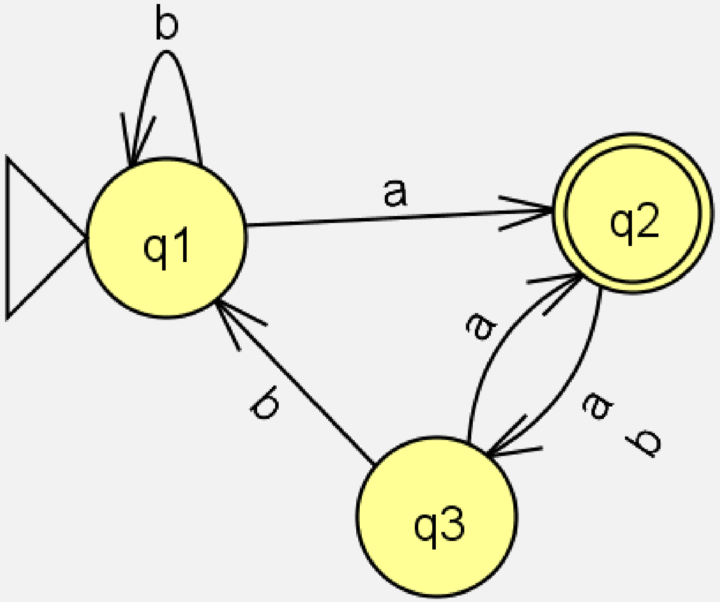
This “analogy” is meant to help your intuition

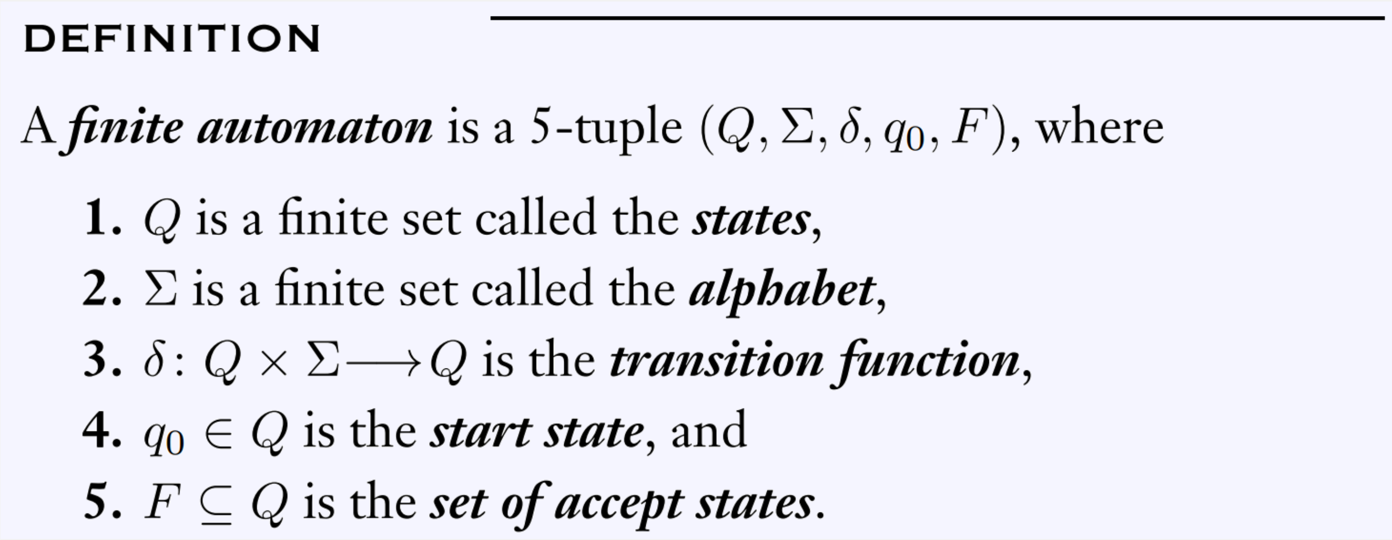
But it’s important not to confuse with **formal definitions**.

Submit in-class work in gradescope!

In-class Exercise (5min)

Come up with a formal description of the following machine:





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. Σ 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***.

In-class Exercise: solution

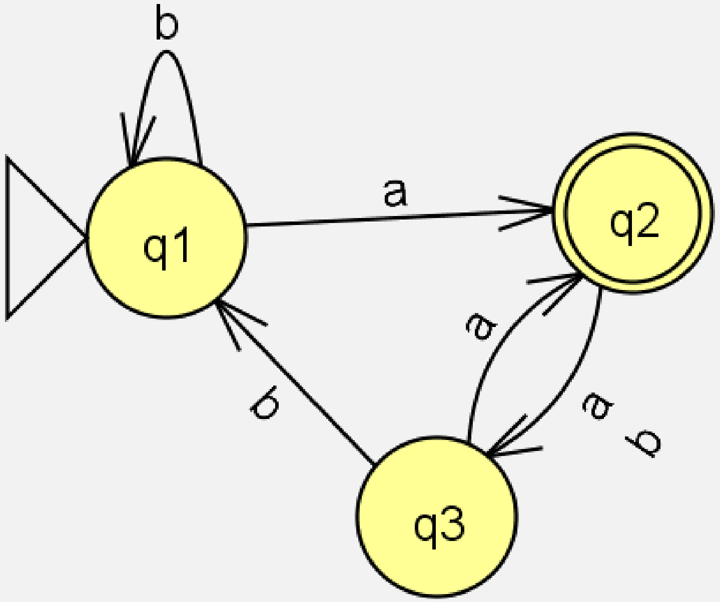
- Q = {q1, q2, q3}
- Σ= { **a**, **b** }
- δ
  - δ(q1, **a**) = q2
  - δ(q1, **b**) = q1
  - δ(q2, **a**) = q3
  - δ(q2, **b**) = q3
  - δ(q3, **a**) = q2
  - δ(q3, **b**) = q1
- q_0 = q1
- F = ~~q2~~ {q2}

???

There are many different ways to write a **function**, i.e., a **mapping** …

- *from* every element in the input set(s) (**domain**)
- *to* some element in the output set (**range**)

$$M = (Q, \Sigma, \delta, q_0, F)$$



A Computation Model is … (from lecture 1)

- Some **definitions** …

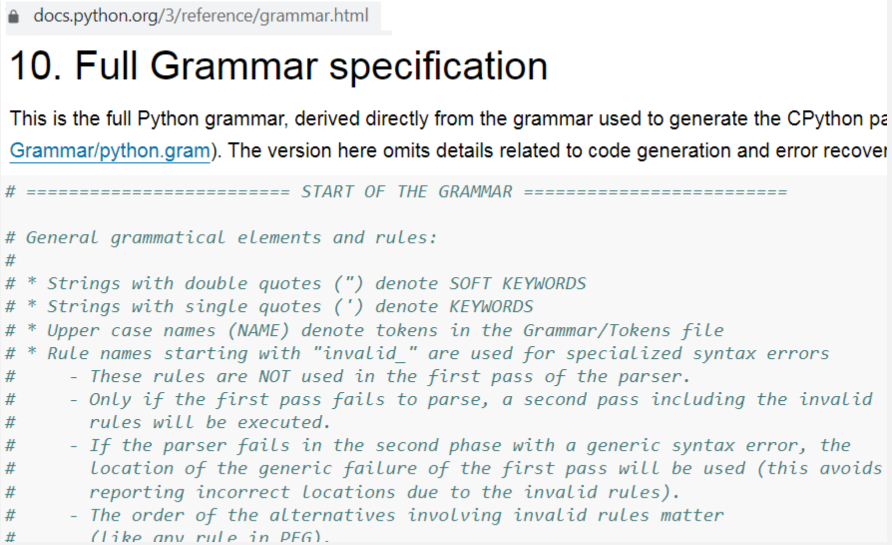
e.g., A **Natural Number** is either
- Zero
- a Natural Number + 1

- And **rules** that describe how to **compute** with the **definitions** …

To **add** two **Natural Numbers:**
- Add the ones place of each num
- Carry anything over 10
- Repeat for each of remaining digits …

A Computation Model is … (from lecture 1)

- Some **definitions** …



docs.python.org/3/reference/grammar.html

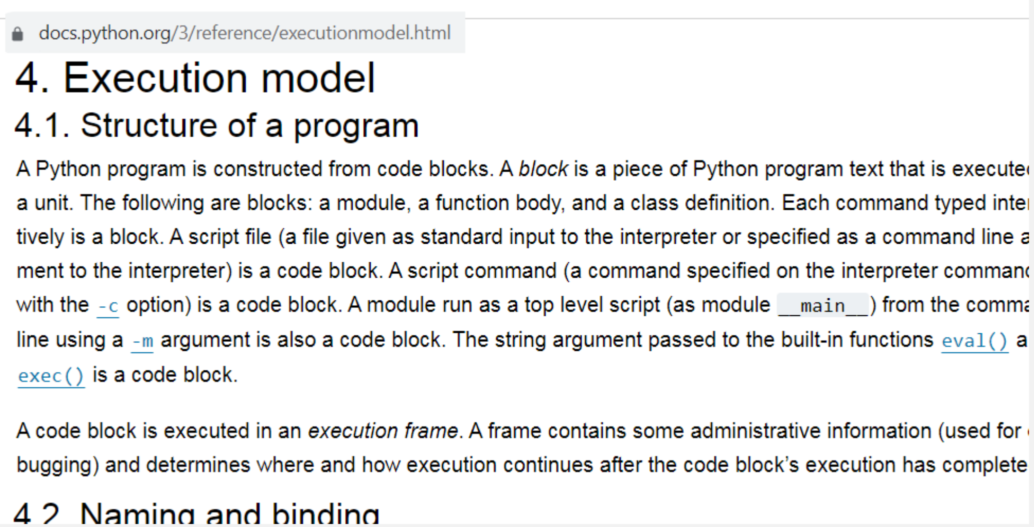
10. Full Grammar specification

This is the full Python grammar, derived directly from the grammar used to generate the CPython pa Grammar/python.gram). The version here omits details related to code generation and error recover

```
# ========================= START OF THE GRAMMAR =========================

# General grammatical elements and rules:
#
# * Strings with double quotes (") denote SOFT KEYWORDS
# * Strings with single quotes (') denote KEYWORDS
# * Upper case names (NAME) denote tokens in the Grammar/Tokens file
# * Rule names starting with "invalid_" are used for specialized syntax errors
#     - These rules are NOT used in the first pass of the parser.
#     - Only if the first pass fails to parse, a second pass including the invalid
#       rules will be executed.
#     - If the parser fails in the second phase with a generic syntax error, the
#       location of the generic failure of the first pass will be used (this avoids
#       reporting incorrect locations due to the invalid rules).
#     - The order of the alternatives involving invalid rules matter
#       (like any rule in PEG).
```

- And **rules** that describe how to **compute** with the **definitions** …



docs.python.org/3/reference/executionmodel.html

4. Execution model

4.1. Structure of a program

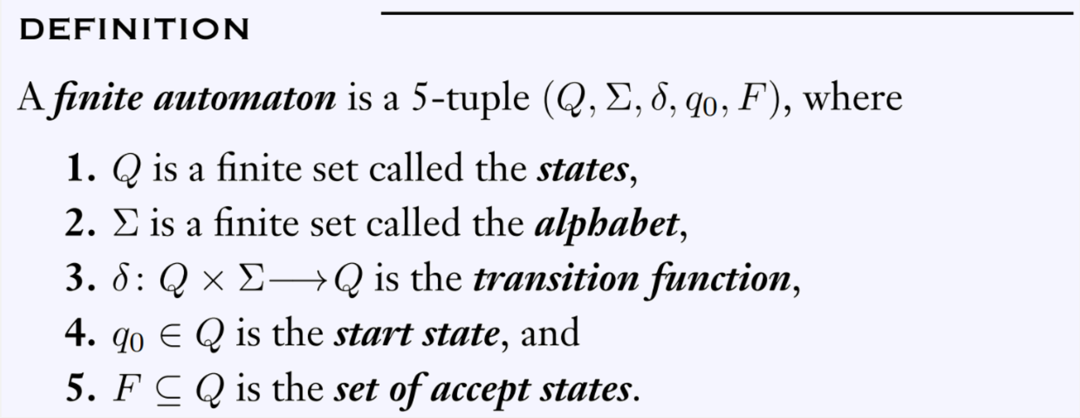
A Python program is constructed from code blocks. A *block* is a piece of Python program text that is execute a unit. The following are blocks: a module, a function body, and a class definition. Each command typed inte tively is a block. A script file (a file given as standard input to the interpreter or specified as a command line a ment to the interpreter) is a code block. A script command (a command specified on the interpreter comman with the `-c` option) is a code block. A module run as a top level script (as module `__main__`) from the comma line using a `-m` argument is also a code block. The string argument passed to the built-in functions `eval()` a `exec()` is a code block.

A code block is executed in an *execution frame*. A frame contains some administrative information (used for bugging) and determines where and how execution continues after the code block's execution has complete

4.2. Naming and binding

A Computation Model is … (from lecture 1)

- Some **definitions** …



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. Σ 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***.

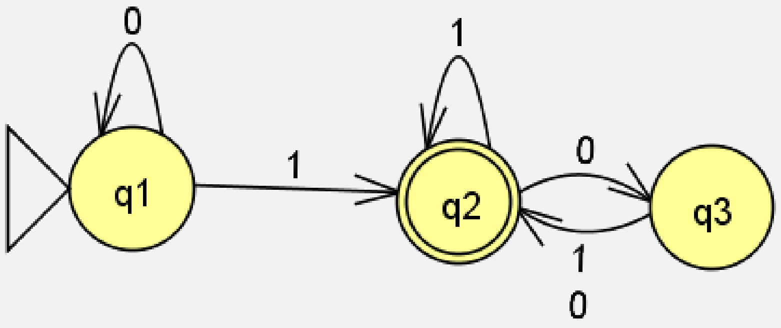
- And **rules** that describe how to **compute** with the **definitions** …

???

Sipser Fig 1.4

Computation with DFAs (JFLAP demo)

- DFA:



- Input: “`1101`”

HINT: always work out concrete examples to understand how a machine works

DFA Computation Rules

Informally

Given

- A **DFA** (~ a “Program”)
- and **Input = string of chars**, e.g. “`1101`”

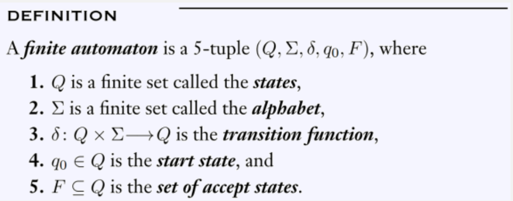
A **DFA** **computation** (~ “Program run”):

- **Starts** in *start state*

- **Repeats:**
  - **Read 1 char** from **Input,** and
  - **Change state** according to ***transition rules***

Result of **computation:**

- **Accept** if **last state** is ***Accept state***
- **Reject** otherwise



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. Σ 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***.

DFA Computation Rules

Informally

Given

- A **DFA** (~ a “Program”)
- and **Input** = **string of chars**, e.g. “1101”

Formally (i.e., mathematically)

- $M =$
- $w =$

A **DFA** computation (~ “Program run”):

- Starts in *start state*
- Repeats:
  - Read **1 char** from **Input**, and
  - Change state according to *transition rules*

Result of **computation**:

- Accept if **last state** is *Accept state*
- Reject otherwise

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and **Input = string of chars**, e.g. "`1101`"

A **DFA computation** (~ "Program run"):

- Starts in *start state*
- Repeats:
  - Read **1 char** from **Input**, and
  - Change state according to *transition rules*

Result of **computation:**

- Accept if **last state** is *Accept state*
- Reject otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A DFA **computation** is a sequence of states $r_0, \ldots, r_n \in Q$ where:

- $r_0 = q_0$

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

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and **Input = string of chars**, e.g. "1101"

A **DFA computation** (~ "Program run"):

- Starts in *start state*
- Repeats:
  - Read 1 char from **Input**, and
  - Change state according to *transition rules*

Result of **computation**:

- **Accept** if **last state** is *Accept state*
- **Reject** otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \boxed{\delta}, q_0, F)$
- $w = \boxed{w_1} w_2 \cdots w_n$

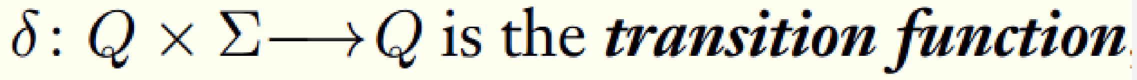
A DFA **computation** is a sequence of states $r_0, \ldots, r_n \in Q$ where:

- $\boxed{r_0} = q_0$
- $\boxed{r_i = \delta(r_{i-1}, w_i), \text{ for } i = 1, \ldots, n}$

if i=1, $r_1 = \delta(r_0, w_1)$

if i=2, $r_2 = \delta(r_1, w_2)$

if i=3, $r_3 = \delta(r_2, w_3)$



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

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and **Input = string of chars**, e.g. "1101"

A **DFA computation** (~ "Program run"):

- Starts in *start state*

- Repeats:
  - Read **1 char** from **Input**, and
  - Change state according to *transition rules*

Result of **computation:**

- Accept if **last state** is *Accept state*
- Reject otherwise

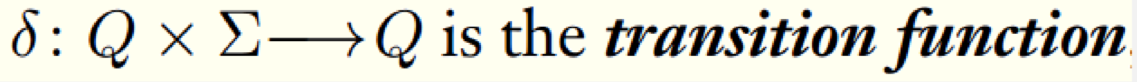
Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A DFA **computation** is a sequence of states $r_0, \ldots, r_n \in Q$ where:

- $r_0 = q_0$

- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$


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

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and **Input** = **string of chars**, e.g. "1101"

A **DFA** computation (~ "Program run"):

- Starts in *start state*
- Repeats:
  - Read **1 char** from **Input**, and
  - Change state according to *transition rules*

Result of computation:

- **Accept** if last state is *Accept state*
- **Reject** otherwise

***Formally** (i.e., mathematically)*

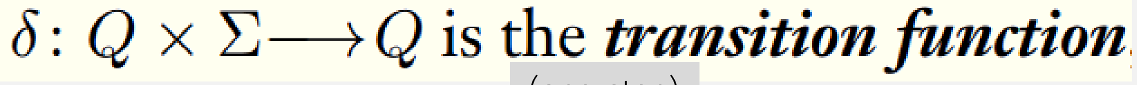
- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A DFA **computation** is a sequence of states $r_0, \ldots, r_n \in Q$ where:

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

This is still pretty verbose …

- M **accepts** w if $r_n \in F$
- M **rejects** w if $r_n \notin F$



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

(one-step)

A Multi-Step Transition Function

set of pairs

$*$ = “0 or more”

Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

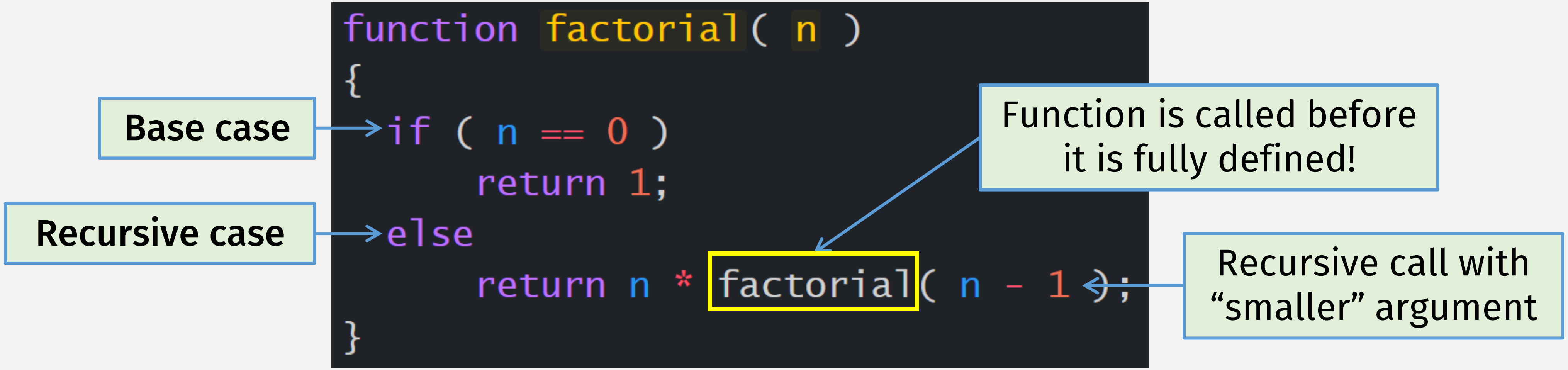
Σ^* = set of all possible strings!

- <u>Domain</u>:
  - Input state $q \in Q$ (doesn’t have to be start state)
  - Input string $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- <u>Range</u>:
  - Output state (doesn’t have to be an accept state)

(Defined recursively)

- <u>Base</u> case: …

Interlude: Recursive Definitions



```
function factorial( n )
{
  if ( n == 0 )
        return 1;
  else
        return n * factorial( n - 1 );
}
```

Base case

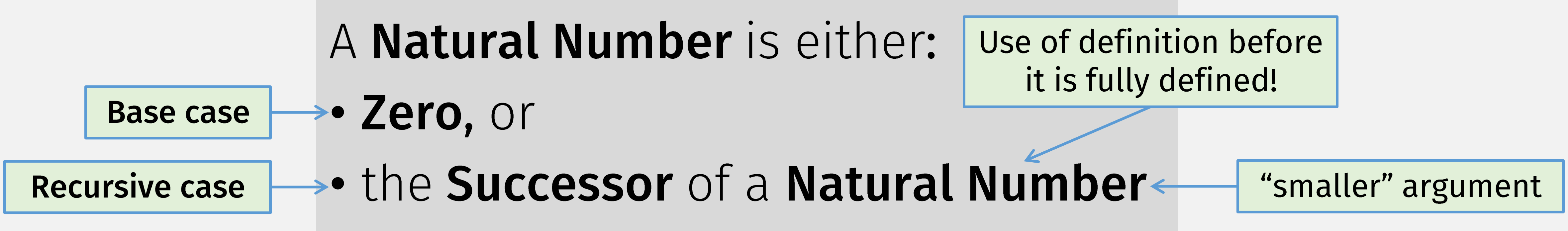
Recursive case

Function is called before it is fully defined!

Recursive call with “smaller” argument

- Why is this allowed?
  - It’s a “**feature**” (i.e., an **axiom!**) of the **programming language**
- Why does this “work”? (Why doesn’t it loop forever?)
  - Because the **recursive call always has** a **“smaller” argument …**
  - … and so **eventually reaches** the **base case** and **stops**

Recursive Definitions



A **Natural Number** is either:

- **Zero**, or
- the **Successor** of a **Natural Number**

Base case

Recursive case

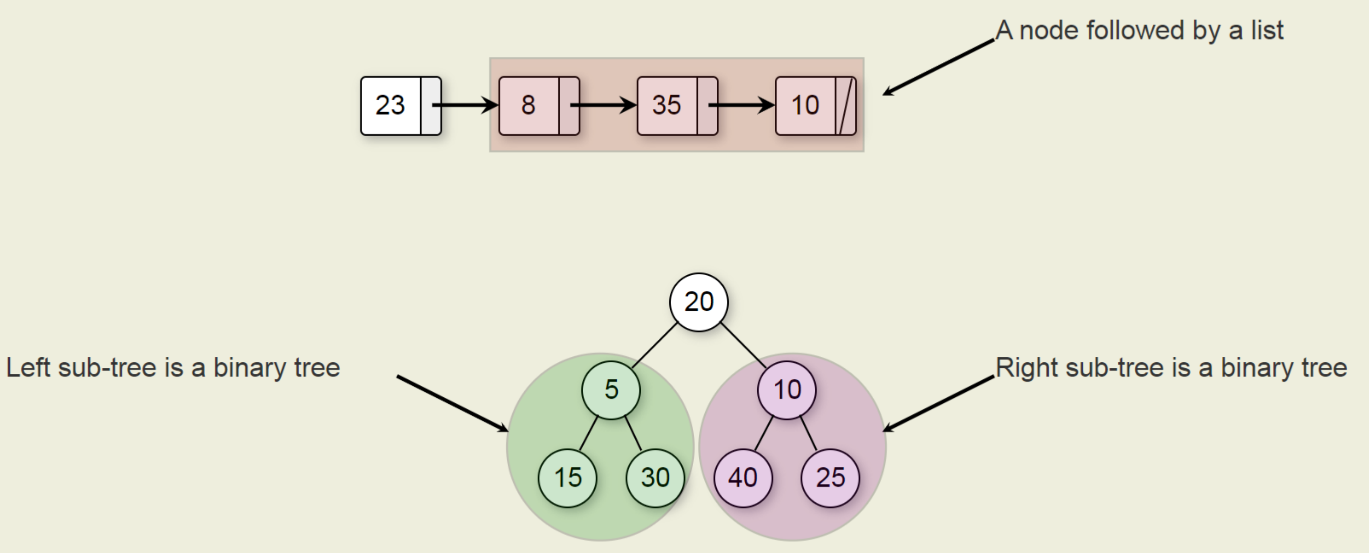
Use of definition before it is fully defined!

"smaller" argument

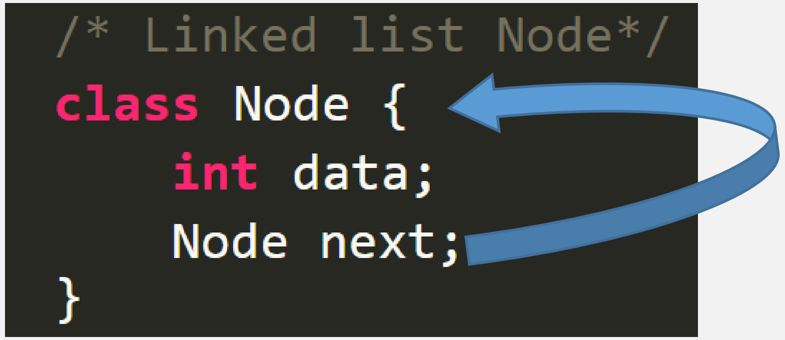
<u>Examples</u>

- **Zero**
- **Successor** of **Zero** (= "one")
- **Successor** of **Successor** of **Zero** (= "two")
- **Successor** of **Successor** of **Successor** of **Zero** (= "three") …

Recursive Data Definitions


A node followed by a list
23
8
35
10
20
5
10
15
30
40
25
Left sub-tree is a binary tree
Right sub-tree is a binary tree

Recursive definitions have:
- base case and
- recursive case
  (with a "smaller" object)



```
/* Linked list Node*/
class Node {
    int data;
    Node next;
}
```

This is a recursive definition: **Node is used before it is fully defined** (but must be "smaller")

Strings Are Defined Recursively

A **String** is either:

- the **empty string** (ε), or
- xa **(non-empty string)** where
  - x is a **string**
  - a is a **“char”** in Σ

Base case

Recursive case

“smaller” argument

Remember: **all strings** are **formed with “chars”** from some **alphabet** set Σ

Σ^* = set of all possible strings!

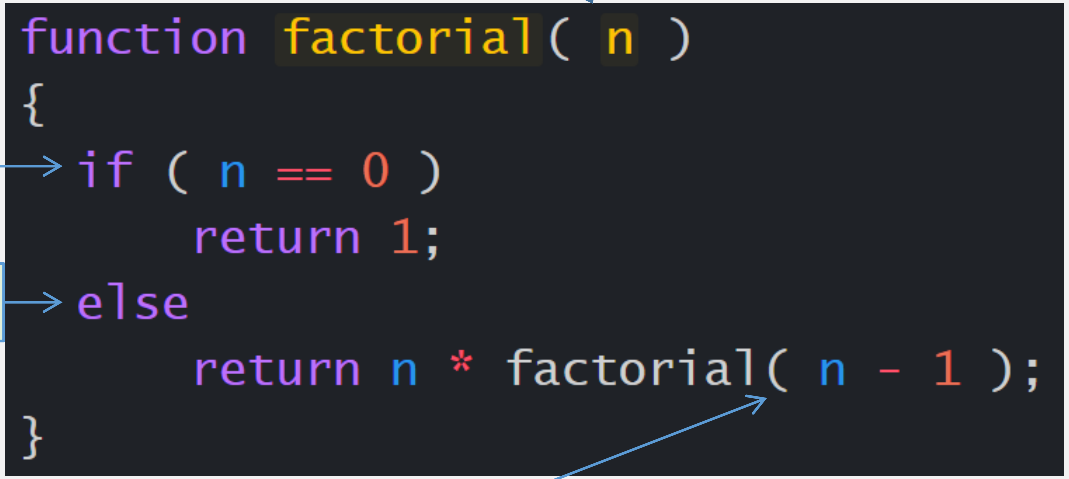
Recursive Data ⇒ Recursive Functions

A **Natural Number** is either:

- **Zero**, or
- the **Successor** of a **Natural Number**

Base case

Recursive case



```
function factorial( n )
{
    if ( n == 0 )
        return 1;
    else
        return n * factorial( n - 1 );
}
```

Recursive case must have “smaller” argument

Recursive functions are **recursive** because … its **input data** is **recursively defined!**

A Multi-Step Transition Function

Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

- Domain:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n \text{ where } w_i \in \Sigma$
- Range:
  - **Output state** (doesn't have to be an accept state)

Recursive Input Data
needs
Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "char" in Σ

Base case

(Defined recursively)

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

A Multi-Step Transition Function

Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \to Q$

- Domain:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range:
  - **Output state** (doesn't have to be an accept state)

Recursive Input Data needs Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a "**char**" in Σ

(Defined recursively)

- Base case $\hat{\delta}(q, \varepsilon) = q$
- Recursive Case $\hat{\delta}(q, w' w_n) = \delta(\hat{\delta}(q, w'), w_n)$

where $w' = w_1 \cdots w_{n-1}$

string

char

Recursive call

"smaller" argument

Recursive case

(or “extended transition function” --- HMU 2.2.4)

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

A Multi-Step Transition Function

Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \to Q$

- Domain:
  - **Input state** $q \in Q$ (doesn't have to be start state)
  - **Input string** $w = w_1 w_2 \cdots w_n$ where $w_i \in \Sigma$
- Range:
  - **Output state** (doesn't have to be an accept state)

Recursive Input Data needs Recursive Function

A **String** is either:

- the **empty string** (ε), or
- *xa* (non-empty string) where
  - *x* is a **string**
  - *a* is a “**char**” in Σ

(Defined recursively)

- Base case $\hat{\delta}(q, \varepsilon) = q$
- Recursive Case $\hat{\delta}(q, w'w_n) = \delta(\hat{\delta}(q, w'), w_n)$

where $w' = w_1 \cdots w_{n-1}$

Previously

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and **Input** = **string of chars**, e.g. "1101"

A **DFA** computation (~ "Program run"):

- Starts in *start state*
- Repeats:
  - Read **1 char** from **Input**, and
  - Change state according to *transition rules*

Result of **computation**:

- Accept if **last state** is *Accept state*
- Reject otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A DFA **computation** is a sequence of states $r_0, \ldots, r_n \in Q$ where:

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

This is still pretty verbose ...

- M **accepts** w if $r_n \in F$
- M **rejects** w if $r_n \notin F$

DFA Computation Rules

Informally

Given

- A **DFA** (~ a "Program")
- and **Input = string of chars**, e.g. "1101"

A **DFA** computation (~ "Program run"):

- Starts in *start state*
- Repeats:
  - Read **1 char** from **Input,** and
  - Change state according to *transition rules*

Result of **computation:**

- Accept if **last state** is *Accept state*
- Reject otherwise

Formally (i.e., mathematically)

- $M = (Q, \Sigma, \delta, q_0, F)$
- $w = w_1 w_2 \cdots w_n$

A DFA **computation** is a sequence of states $r_0, \ldots, r_n \in Q$ where:

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$

- M **accepts** w if $\hat{\delta}(q_0, w) \in F$
- M **rejects** w if $r_n \notin F$

Alphabets, Strings, Languages

An **alphabet** defines **"all possible strings"**

(strings with non-alphabet symbols are impossible)

- An **alphabet** is a non-empty finite set of **symbols**

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

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

- A **string** is a finite sequence of **symbols** from an **alphabet**

01001 abracadabra ε

Empty string (length 0)

(ε symbol is not in the alphabet!)

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

Languages can be **infinite**

$$A = \{\mathtt{good}, \mathtt{bad}\}$$

$$\emptyset \quad \{\ \}$$

The **Empty set** is a **language**

$$A = \{w \mid w \text{ contains at least one 1 and an even number of 0s follow the last 1}\}$$

"the set of all …"

"such that …"

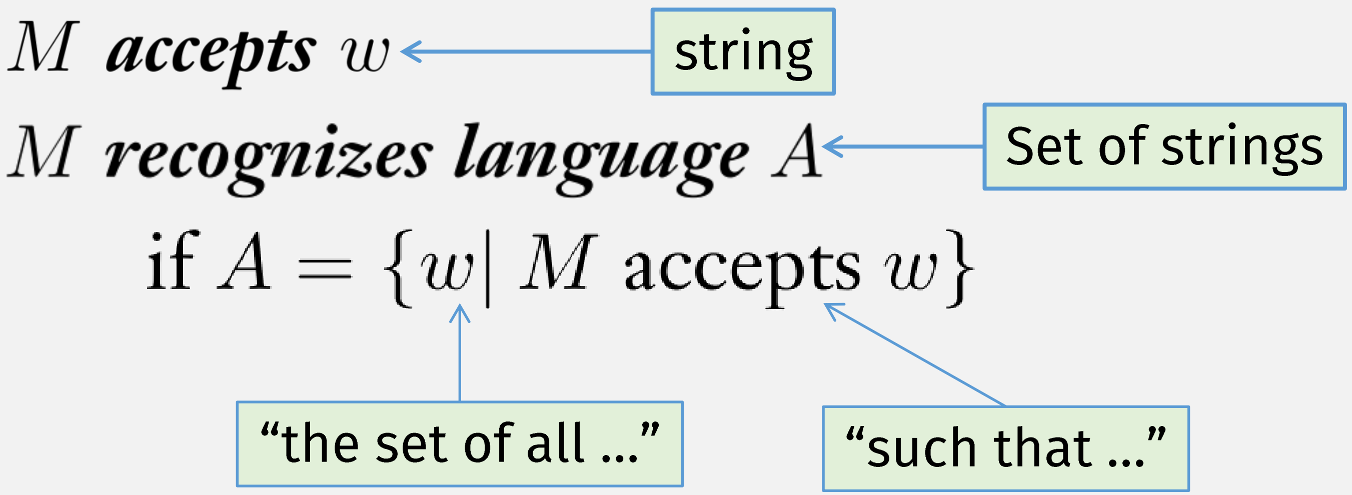
Machine and Language Terminology

- The **language** of a **machine** = set of strings that it **accepts**

- E.g., A DFA M ***accepts*** w

  M ***recognizes language*** A

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



string

Set of strings

“the set of all …”

“such that …”

Machine and Language Terminology

- The **language** of a **machine** = set of strings that it **accepts**

- E.g., A DFA M ***accepts*** w

  M ***recognizes language*** $L(M)$

  $L(M) = \{w \mid M \text{ accepts } w\}$

Using L as function mapping **Machine** → **Language** is common notation

Machine and Language Terminology

- The **language** of a **machine** = set of strings that it **accepts**

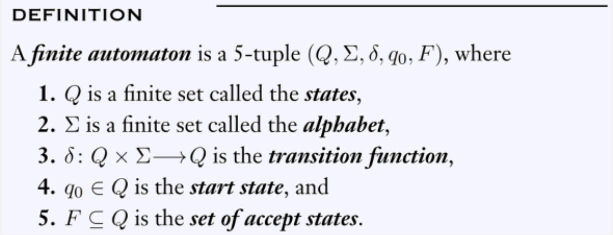
- E.g., A DFA M ***accepts*** w

  M ***recognizes language*** $L(M)$

- **Language** of $M = L(M) = \{w \mid M \text{ accepts } w\}$

Languages Are Computation Models

- The **language** of a **machine** = set of strings that it **accepts**
  - E.g., **a DFA recognizes** a **language**
- A **computation model** = set of machines it **defines**
  - E.g., **all possible DFAs** are a **computation model**



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. Σ 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***.

= set of set of strings

Thus: a **computation model** equivalently = a set of languages

This class is really about studying **sets of languages!**

Regular Languages

- first **set of languages** we will study: **regular languages**

This class is really about studying **sets of languages!**

Regular Languages: Definition

If a **deterministic finite automata** (**DFA**) recognizes a **language,**
then **that language** is called a **regular language**.

A Language, Regular or Not?

- If given: **a DFA M**
  - We know: $L(M)$, the **language recognized by** M, is a **regular language**

Proof : If a **DFA** recognizes a **language**,
then **that language** is called a **regular language**.

(modus ponens)

- If given: **a Language A**
  - Is A a **regular language?**
    - **Not necessarily!**

Proof : ??????