CS420

(Deterministic) Finite Automata

Wednesday, January 31, 2024

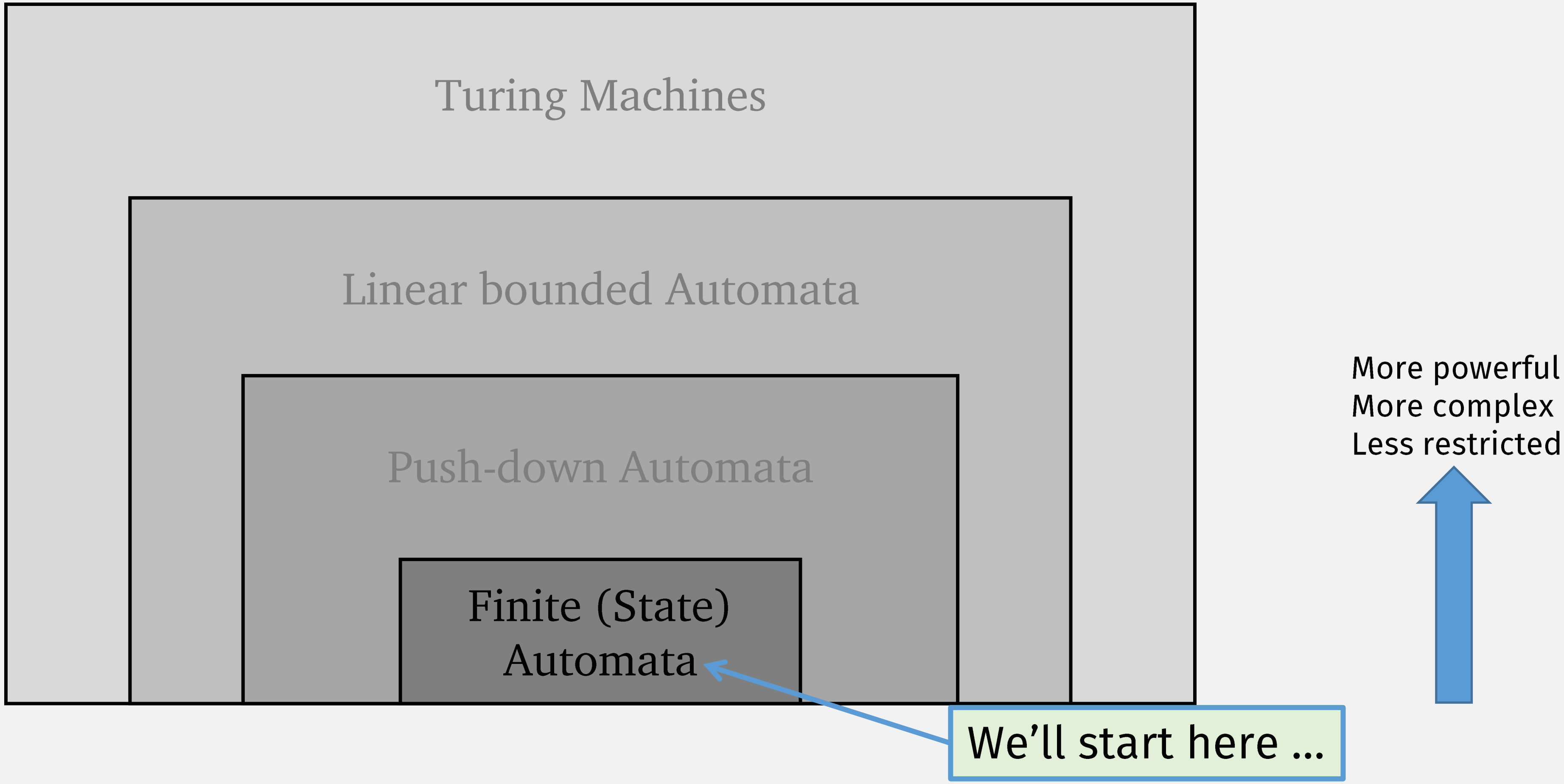
UMass Boston Computer Science

Announcements

- HW 1
  - due date extended: Wed 2/7, 12pm EST (noon)

- Please ask all HW questions on Piazza!
  - So all course staff can see,
  - and entire class can benefit
  - Please: do not email course staff with HW questions

Last Time: Models of Computation Hierarchy



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** (wont use in this class)

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** (wont use in this class)
  - ➢ **Formal math description** (like code, just a different "programming lang")

Finite Automata: The Formal Definition

DEFINITION

deterministic

A ***finite automaton*** *(DFA)* 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***.

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

Set or sequence?

5 components

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***.

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	**Sequences**
• Unordered	• Ordered
• Duplicates not allowed	• Duplicates ok
• Notation: { }	• Notation: **varies:** ( ), **comma,** or **append**
• **Empty set** written: ∅ or { }	• **Empty sequence:** ( )
• A **language** is a (possibly infinite) set of strings	• A **tuple** is a finite sequence
	• A **string** is a finite sequence of characters

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

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 mapping, a table, …

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 → $(Q, \Sigma, \delta, q_0, F)$

set → Q

set → Σ

Set of pairs (**domain**) → $Q \times \Sigma$

Set (**range**) → Q

Don't know! (states can be anything) → q_0

set → F

A **pair** is … a **sequence** of 2 elements

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** (wont use in this class)
  - Formal math description (like code, just a different "programming lang")
  - ➢ State Diagrams

Finite Automata: State Diagram

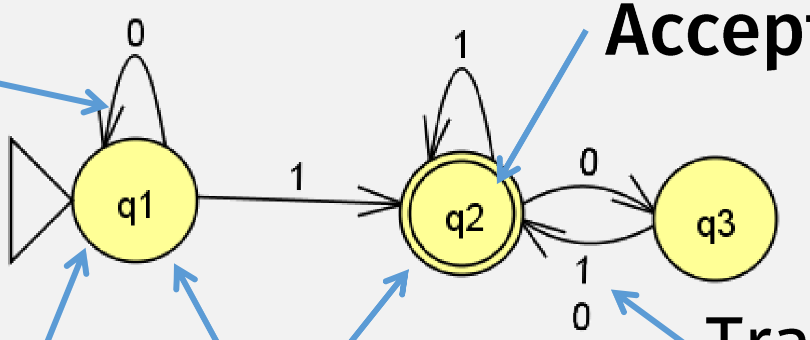
DEFINITION

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. $\delta : Q \times \Sigma \longrightarrow Q$ 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



Start State

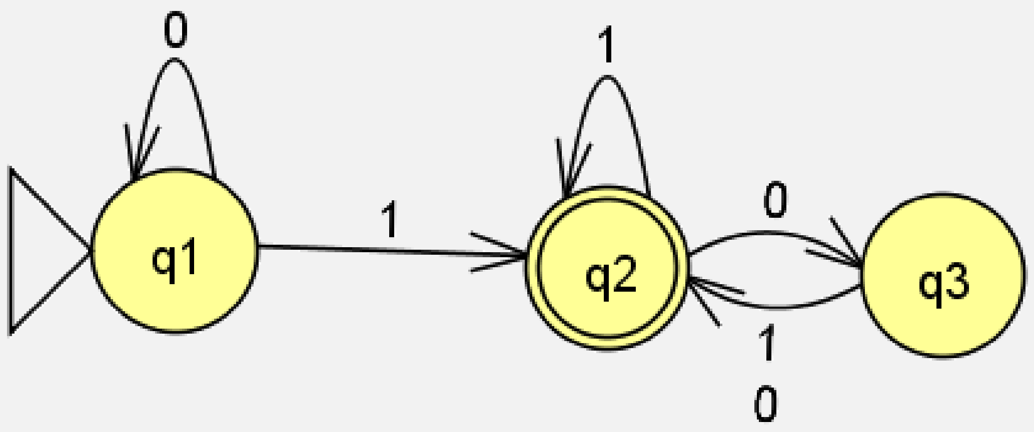
Transition labels from **alphabet**

States

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**)

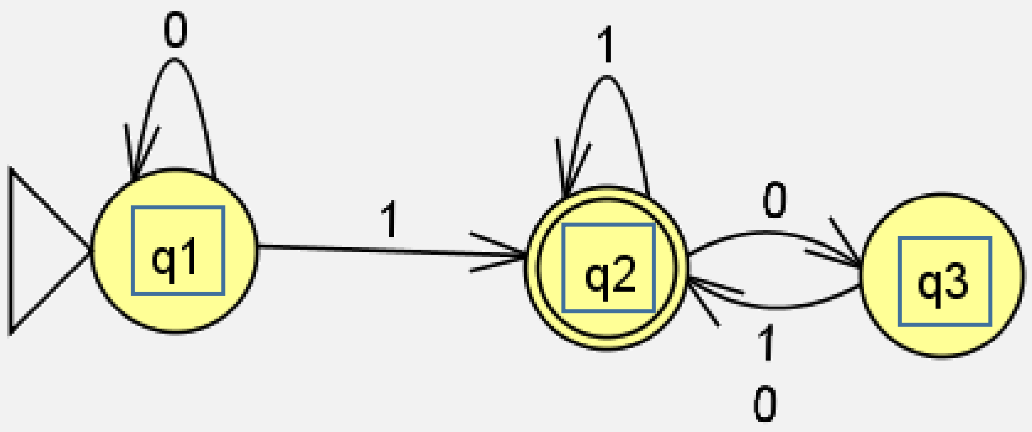
$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

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)

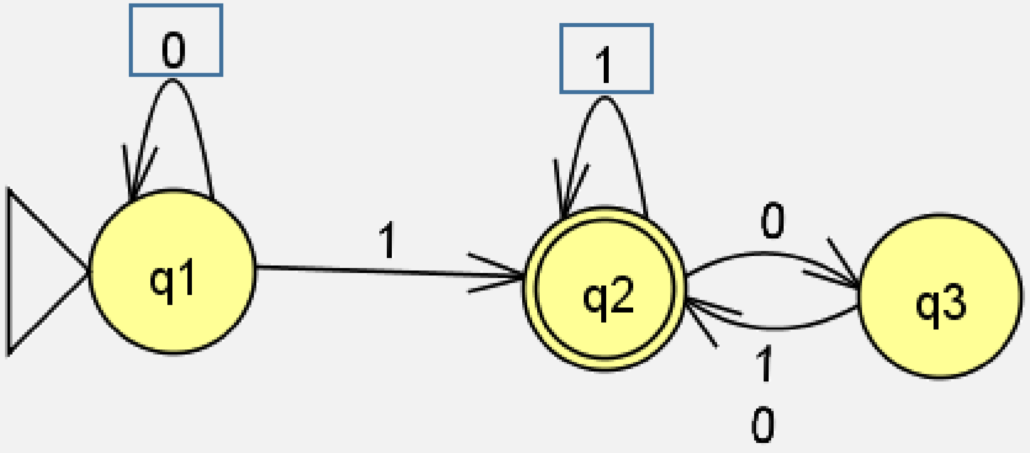


An **Example** (as **state diagram**)

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 as

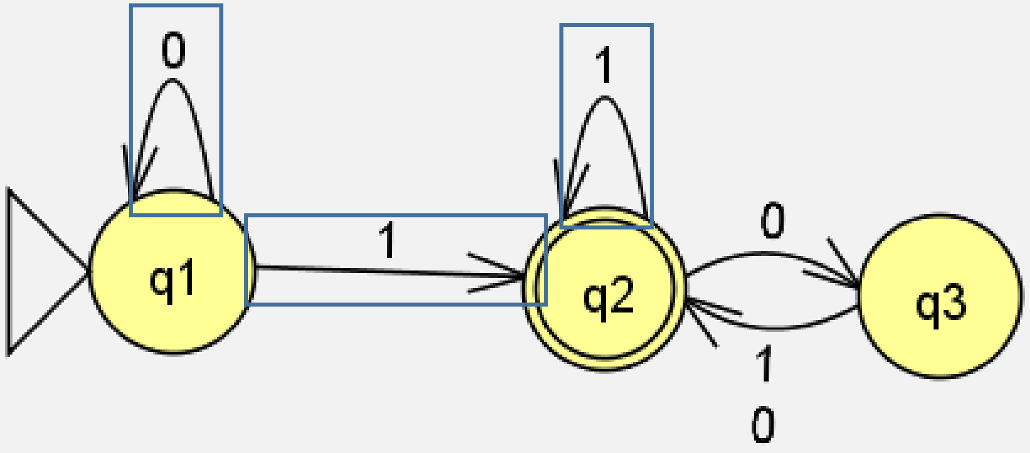
	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

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***.



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

"And this is next input symbol"

"If in this state"

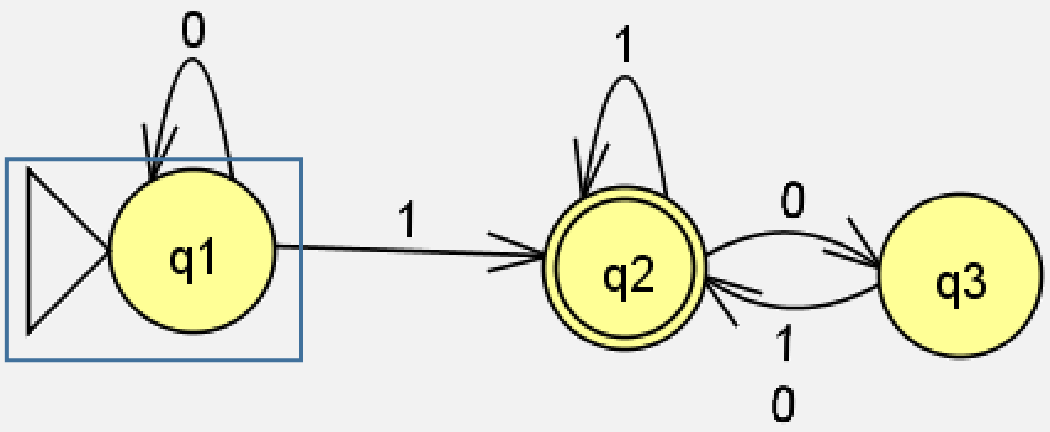
"Then go to this state"

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

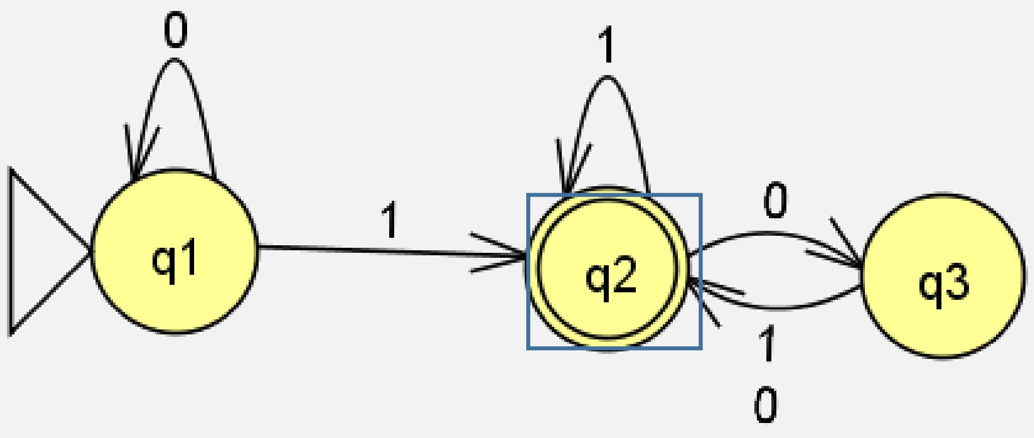
	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2,

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. $\delta: Q \times \Sigma \longrightarrow Q$ 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. $F = \{q_2\}$.

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**)

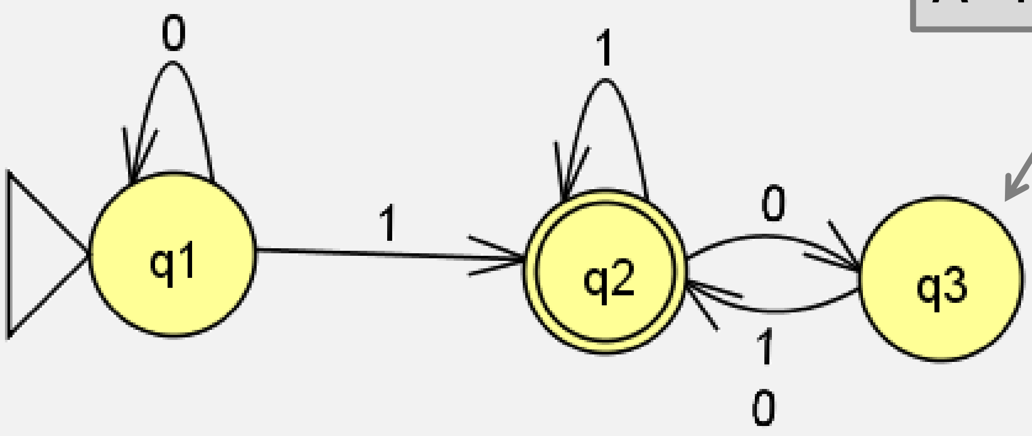
$M_1 = (Q, \Sigma, \delta, q_1, F)$, where

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

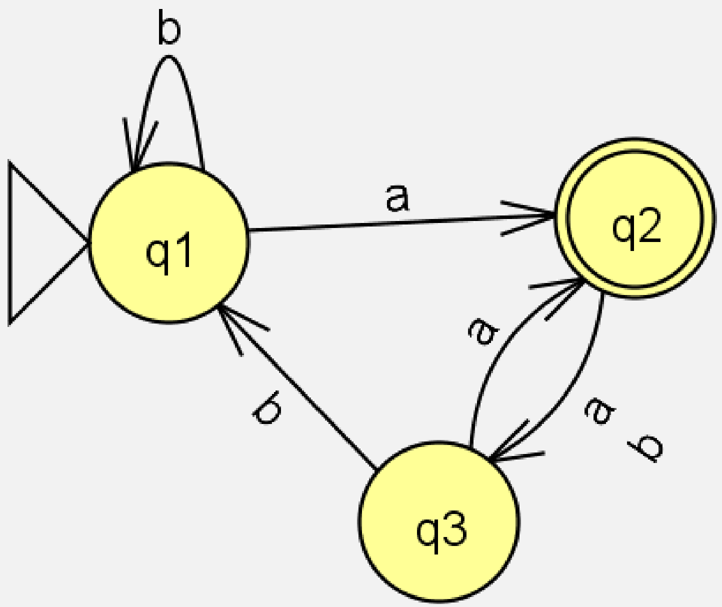
This "analogy" is meant to help your intuition

But it's <u>important</u> not to confuse with **formal definitions**.

Submit in-class work in gradescope!

In-class Exercise

Come up with a formal description of the following machine:



DEFINITION

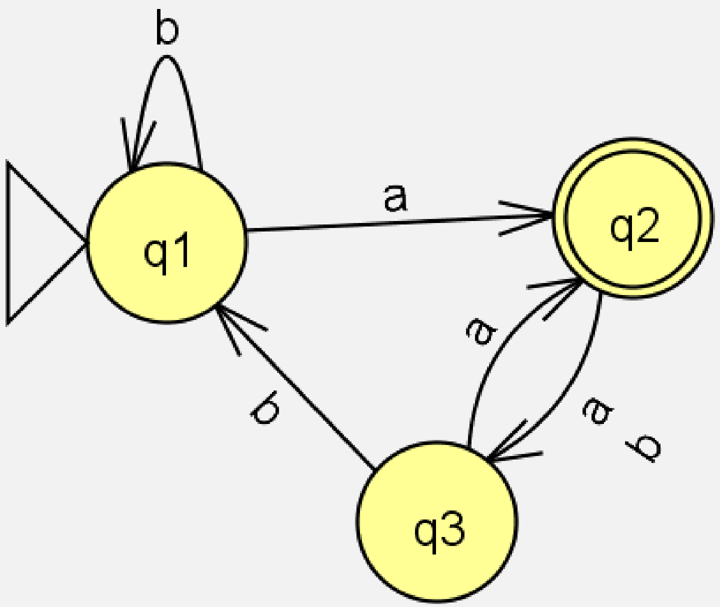
A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

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}

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



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

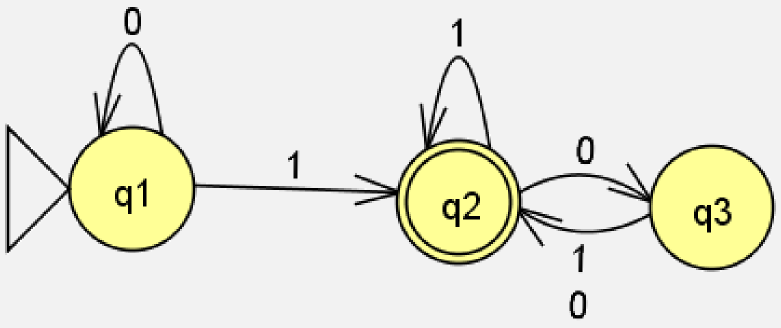
A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

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** …

???

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`"

To **run** the automata / "program":

- Start in **"start state"**
- Repeat:
  - Read **1 char** from **input;**
  - Change **state** according to the transition **table**
- **Result of computation =**
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

DFA Computation Rules

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

Given

- A **DFA** (~ a "Program") → • $M = (Q, \Sigma, \delta, q_0, F)$
- and <u>**Input**</u> **= string of chars**, e.g. "1101" → • $w = w_1 w_2 \cdots w_n$

To **run** the automata / "program":

- <u>Start</u> in **"start state"**
- <u>Repeat</u>:
  - <u>Read</u> **1 char** from **input;**
  - <u>Change</u> **state** according to the <u>**transition**</u> **table**
- <u>Result</u> **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`"

To **run** the automata / "program":

- Start in "start state"
- Repeat:
  - Read **1 char** from **input;**
  - Change **state** according to the transition **table**
- Result of computation =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally (i.e., mathematically)

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

Define: **variables** $r_0, \ldots, r_n$
representing **sequence of states** in the **computation**

- $r_0 = q_0$
- M ***accepts*** w if
  sequence of states $r_0, r_1, \ldots, r_n$ in Q exists . . .
  with $r_n \in F$

DFA Computation Rules

Informally

Given

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

To **run** the automata / "program":

- <u>**Start**</u> in **"start state"**
- <u>**Repeat**</u>:
  - <u>**Read**</u> **1 char** from **input;**
  - <u>**Change**</u> **state** according to the <u>**transition**</u> **table**
- <u>**Result**</u> **of computation** =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally (i.e., mathematically)

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

Define: **variables** r_0, …, r_n representing <u>**sequence of states**</u> in the **computation**

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, 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, $r_3 = \delta(r_2, w_3)$

- M ***accepts*** w if sequence of states $r_0, r_1, \ldots, r_n$ in Q exists … with $r_n \in F$

DFA Computation Rules

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

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

To **run** the automata / "program":
- Start in **"start state"**
- Repeat:
  - Read **1 char** from **input;**
  - Change **state** according to the transition **table**
- Result **of computation** =
  - **Accept** if last state is **Accept state**
  - **Reject** otherwise

Formally *(i.e., mathematically)*

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

Define: **variables** r_0, …, r_n
representing sequence of states in the **computation**

- $r_0 = q_0$
- $r_i = \delta(r_{i-1}, w_i)$, for $i = 1, \ldots, n$
- M ***accepts*** w if sequence of states $r_0, r_1, \ldots, r_n$ in Q exists … with $r_n \in F$

This is still a little "informal"

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

An Extended Transition Function

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

set of pairs

* = “0 or more”

Σ^* = set of all possible strings!

- 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)

(Defined recursively)

- Base case: …

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:

Use of definition before it is fully defined!

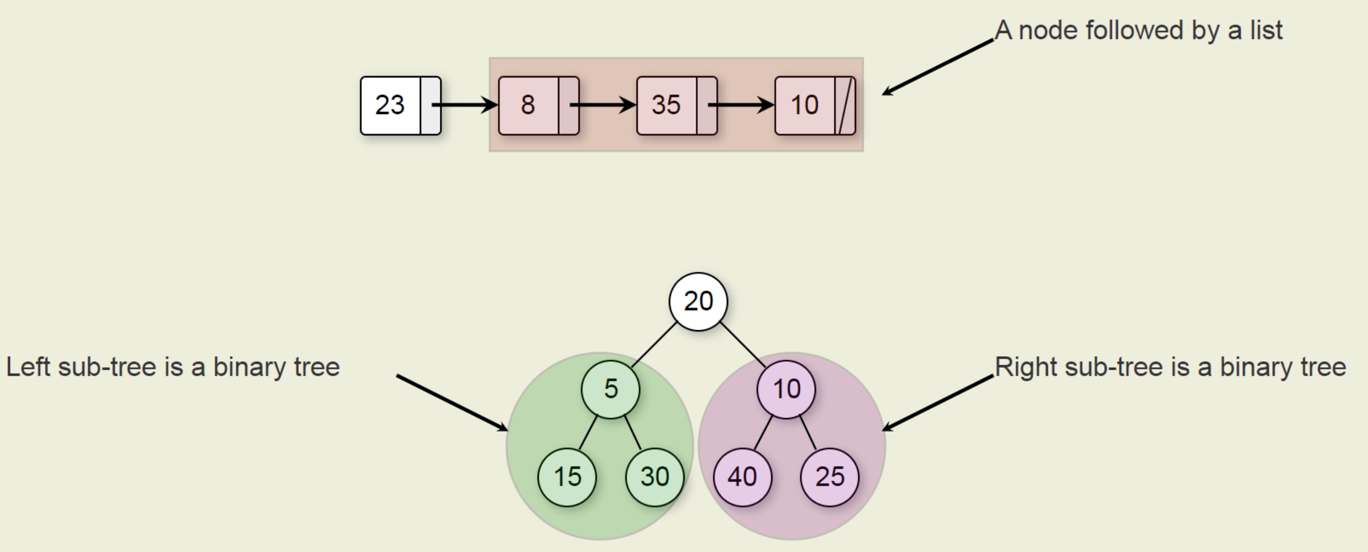
Base case → • **Zero**, or

Recursive case → • the **Successor** of a **Natural Number** ← "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 Definitions



Recursive definitions have:
- <u>base case</u> and
- <u>recursive case</u>
  (with a “smaller” object)

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

This is a <u>recursive definition:</u>
Node is used before it is fully defined (but must be “smaller”)