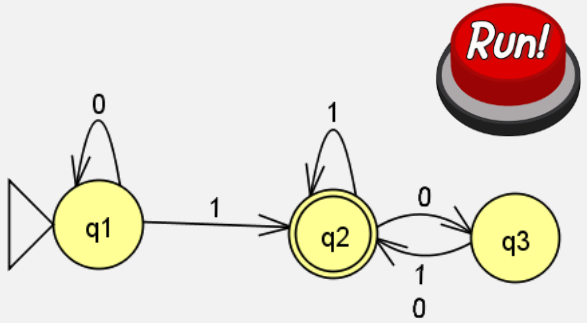
CS 420

Computing With DFAs

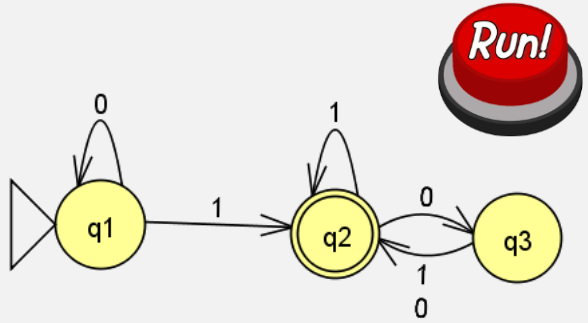
Thursday, September 17, 2025

UMass Boston Computer Science



Announcements

- HW 0
  - ~~Due: 9/15~~
- HW 1
  - Out: 9/15
  - Due: 9/22 12:30pm EDT
- Office hours started
  - check times and locations on course web site



Last Time

Finite Automata: The Formal Definition

DEFINITION

deterministic

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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***.

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, where

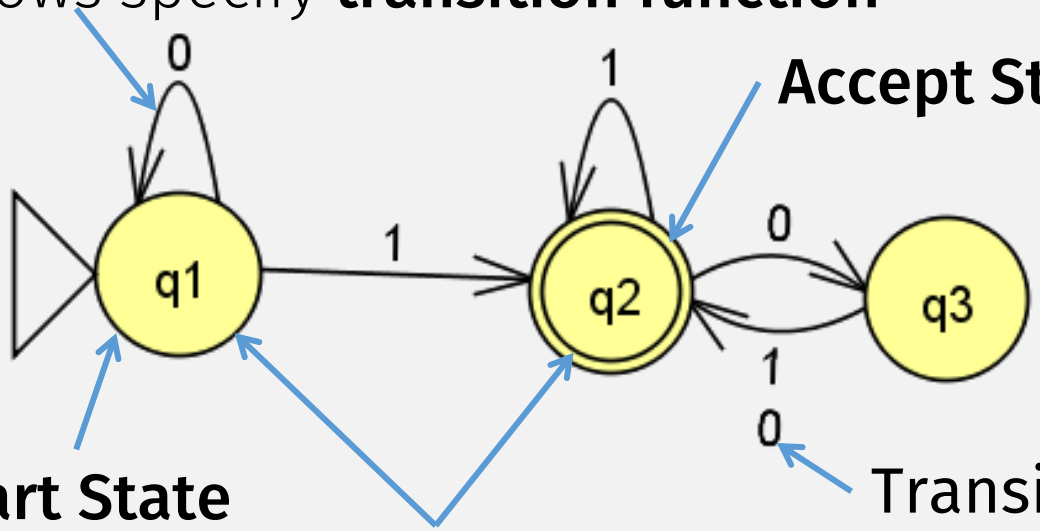
1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta: Q \times \Sigma \longrightarrow Q$ is the ***transition function***,
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.

Finite Automata: Formal Description

vs

Finite Automata: State Diagram

Arrows specify **transition function**



Accept State(s)

Start State

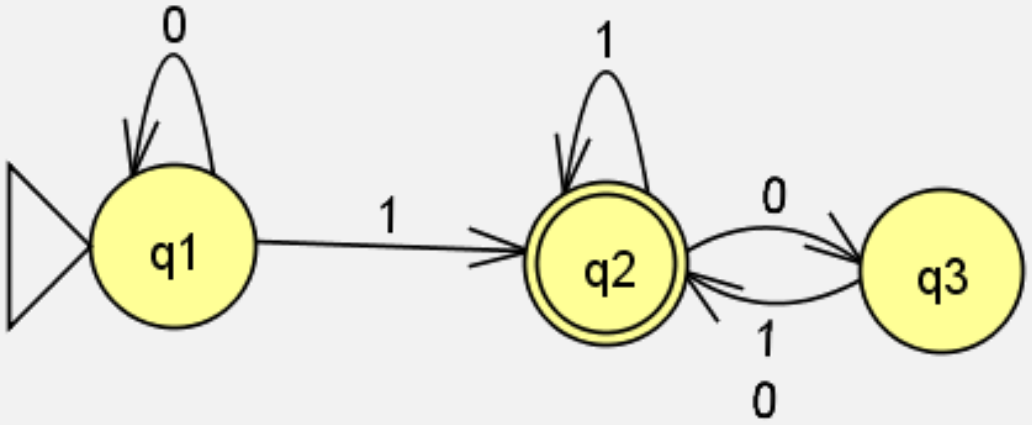
States

Transition labels must be from **alphabet**

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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***.



An Example (as **state diagram**)

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

DEFINITION (constructor function **definition**) – (function input **parameter**)

A ***finite automaton*** is a **mk-DFA**$(Q, \Sigma, \delta, q_0, F)$, 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**): (constructor function **use**) (function input **argument**)

M_1 = **mk-DFA**(Q, Σ, δ, q1, F) where

$Q = \{q_1, q_2, q_3\}$

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

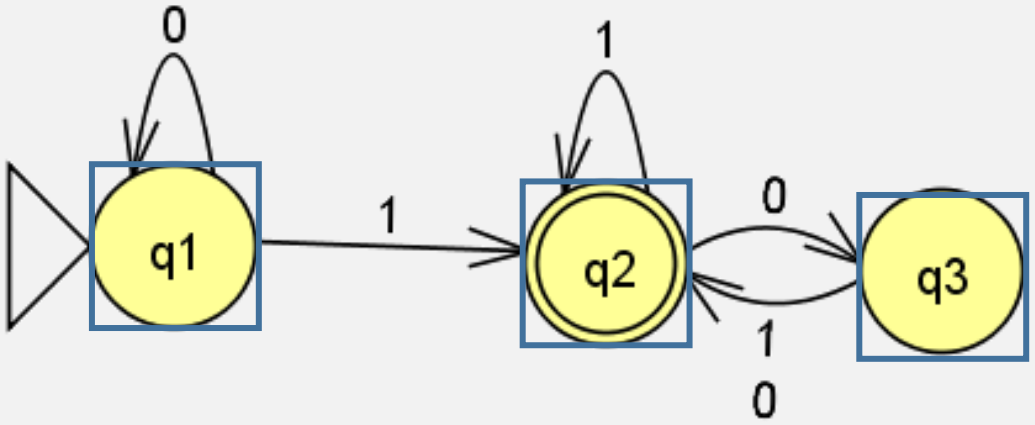
δ

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

q_1 is the start state

$F = \{q_2\}$

braces = set notation (no duplicates)

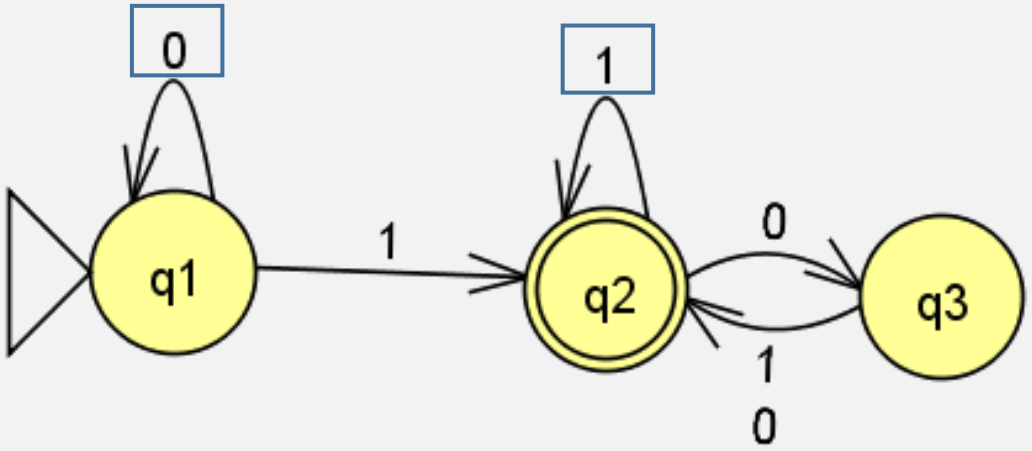


An **Example** (as **state diagram**)

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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 = `mk-DFA`$(\ Q,\ \Sigma,\ \delta,$ `q1`$,\ F\ )$, where

$Q = \{q_1, q_2, q_3\}$

$\Sigma = \{0,1\}$ ← Possible chars of input

δ

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

q_1 is the start state

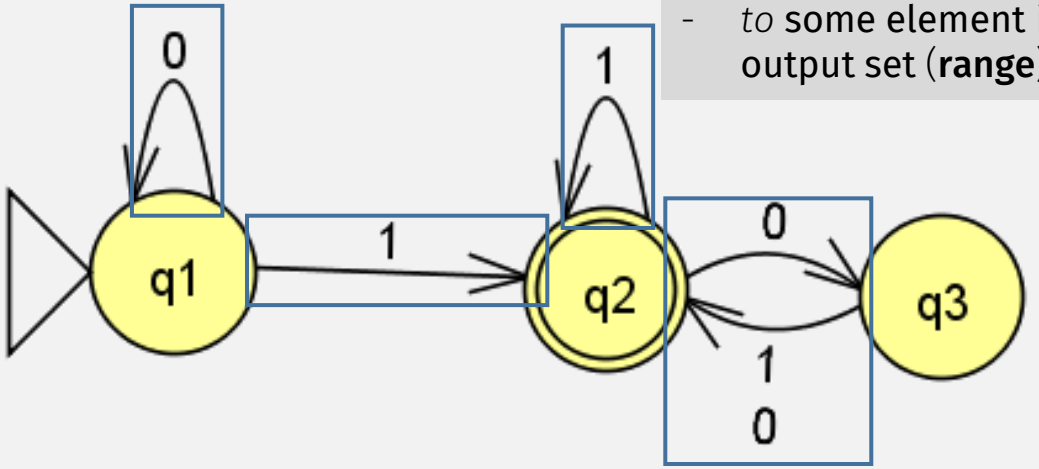
$F = \{q_2\}$

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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 st***

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_1 = `mk-DFA`(Q, Σ, δ, q1, F), where

$$Q = \{q_1, q_2, q_3\}$$

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

Functions must be **defined with signatures** specifying **domain** and **range** sets!

$$\delta : Q \times \Sigma \rightarrow Q$$

	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"

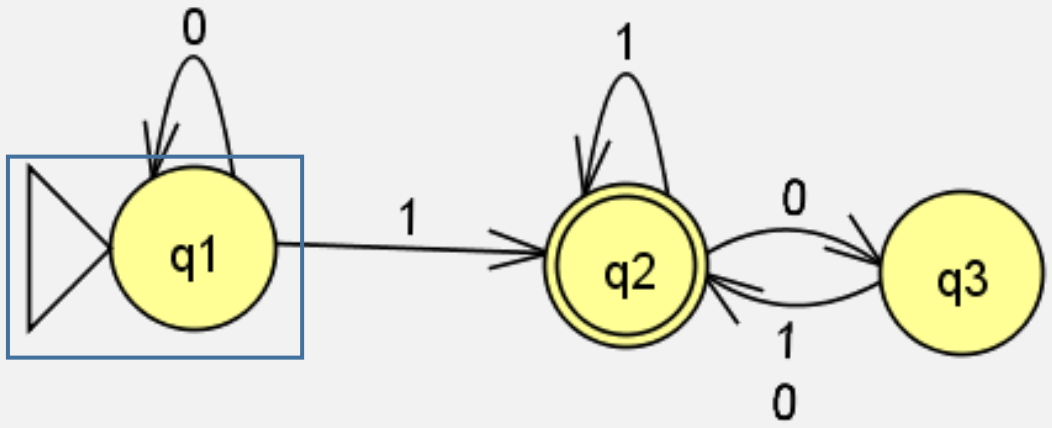
q_1 is the start state

$$F = \{q_2\}$$

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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 = `mk-DFA`$(\ Q,\ \Sigma,\ \delta,$ q1, $F\ )$, where

We already wrote it

$$Q = \{q_1, q_2, q_3\}$$

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

$$\delta : Q \times \Sigma \to Q$$

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

q1 (is the start state)

(do we need to write this?)

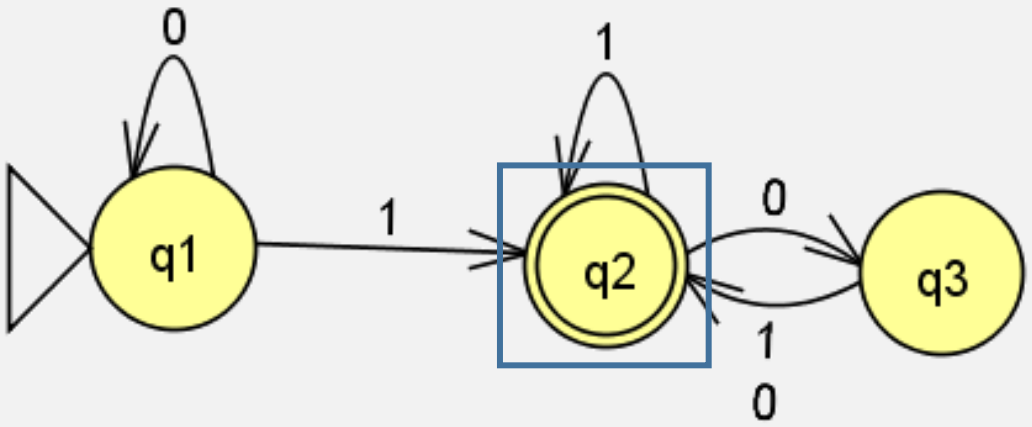
$$F = \{q_2\}$$

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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***.

WARNING: This is a **set**!



M_1 = `mk-DFA`$(\ Q,\ \Sigma,\ \delta,$ `q1`$,\ F\ )$, where

$$Q = \{q_1, q_2, q_3\}$$

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

$$\delta : Q \times \Sigma \to Q$$

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

~~q1 (is the start state)~~

$$F = \{q_2\}$$

WARNING: This is a **set**!

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

DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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***.

A "Programming Language"
specifies how to write computation

M_1 = `mk-DFA`$(\ Q,\ \Sigma,\ \delta,$ q1, $F\ )$, where

$$Q = \{q_1, q_2, q_3\}$$

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

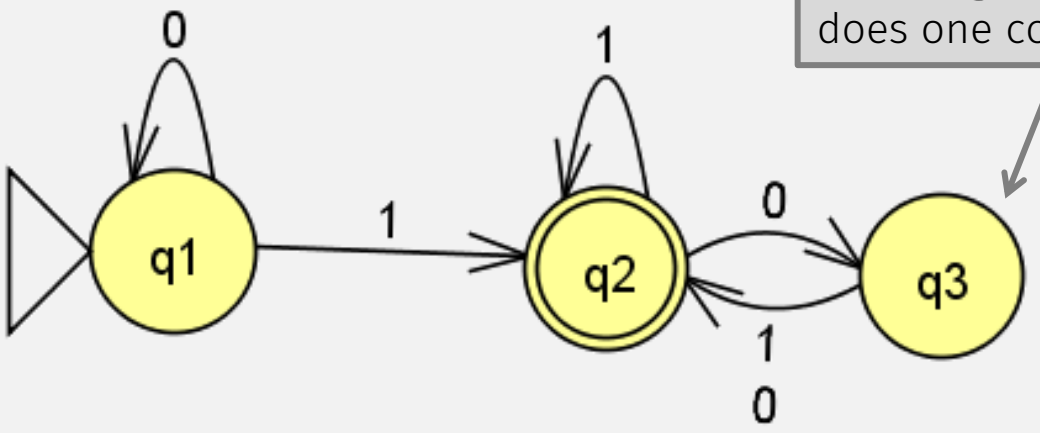
$$\delta : Q \times \Sigma \to Q$$

	0	1
q_1	q_1	q_2
q_2	q_3	q_2
q_3	q_2	q_2

~~q1 (is the start state)~~

$$F = \{q_2\}$$

A "Program"
does one computation



"Programming" Analogy

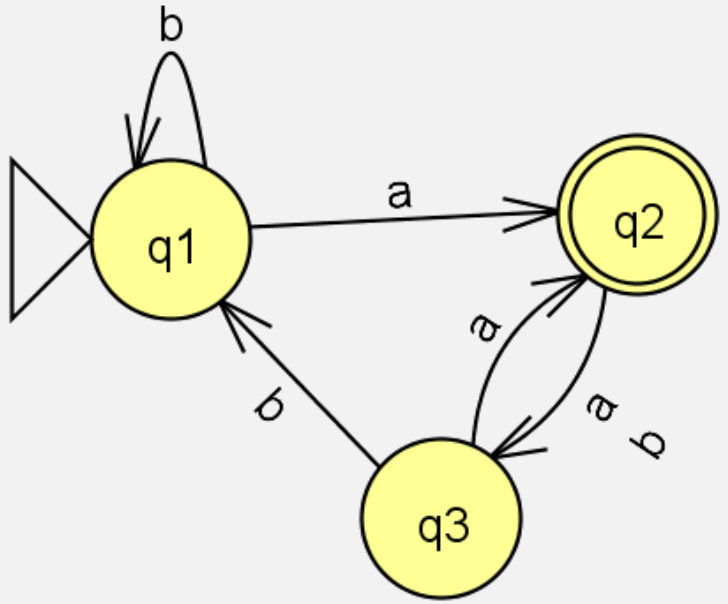
This "analogy" is meant to help your intuition

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

Submit in-class work in gradescope!

In-class Exercise (5min)

Come up with a formal description of the following machine:



DEFINITION

A ***finite automaton*** is a `mk-DFA`$(Q, \Sigma, \delta, q_0, F)$, 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***.

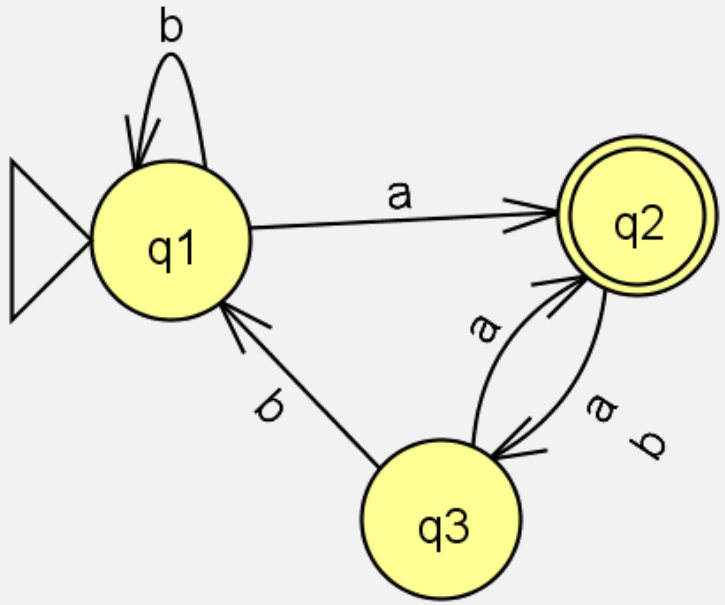
In-class Exercise: solution

- Q = {q1, q2, q3}
- Σ = { **a**, **b** }
- $\delta : Q \times \Sigma \rightarrow Q$
  - 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 (do we need to write this?)
- F = ~~q2~~ {q2}
  ???

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**)

Yes! Otherwise q_0 is an unbound variable!

M = **mk-DFA**(Q, Σ, δ, q_0, F), where



Finite Automata: The Formal Definition

DefDFA

("fact" added to our course proof "library")

DEFINITION

deterministic

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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***.

Programming analogy: "subtype" or "class" — "generic interface" or "abstract class"

Can be used to <u>prove</u> a **new kind of statement:** *M* IS-A **DFA Machine**

DefDFA is a **valid** <u>Justification</u> **...** so long as M was constructed with **`mk-DFA( .... )`**

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**:
1. Add the **ones place of each num**
2. Carry **anything over 10**
3. 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 executed
a unit. The following are blocks: a module, a function body, and a class definition. Each command typed inter
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 command
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 `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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***.

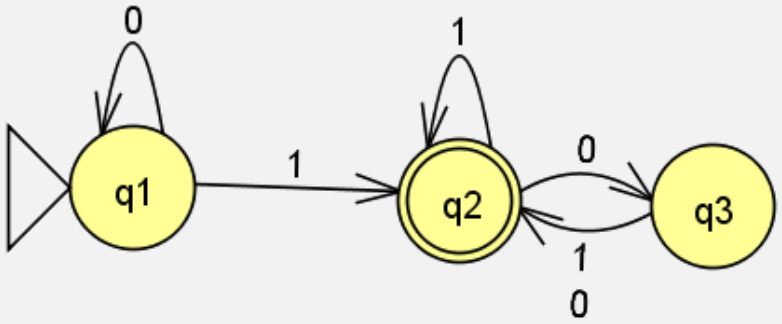
- And **rules** that describe how to **compute** with the **definitions …**

???

Sipser Fig 1.4

Computation with DFAs (JFLAP demo)

- DFA:



- Input: “1101”

HINT: always work out concrete examples to understand how a machine (i.e., “program”) 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

DFA Computation Rules

DEFINITION

A ***finite automaton*** is a `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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***.

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

Given

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

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 = **mk-DFA**(Q, Σ, δ, 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)

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

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

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

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)

- $M =$ **mk-DFA**$(Q, \Sigma, \delta, \mathrm{q}_0, F)$ ...
- $w = w_1 w_2 \cdots w_n$

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)

- $M = \texttt{mk-DFA}(\ Q,\ \Sigma,\ \delta,\ \mathrm{q}_0,\ F\ )$ ...
- $w = w_1 w_2 \cdots w_n$

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 somewhat informal ...

- 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

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

set of pairs

* = “0 or more”

Σ^* = set of all possible strings!

Alphabets, Strings, Languages

An **alphabet** defines **"all possible strings"**
(that a machine can compute with)

- An **alphabet** is a <u>non-empty finite set</u> of **symbols**

(strings with non-alphabet symbols are impossible)

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

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

- A **string** is a <u>finite sequence</u> of **symbols** from an **alphabet**

`01001` `abracadabra` ε

Empty string (length 0)

(ε symbol <u>is not</u> in the alphabet!)

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

(one-step)

A Multi-Step Transition Function

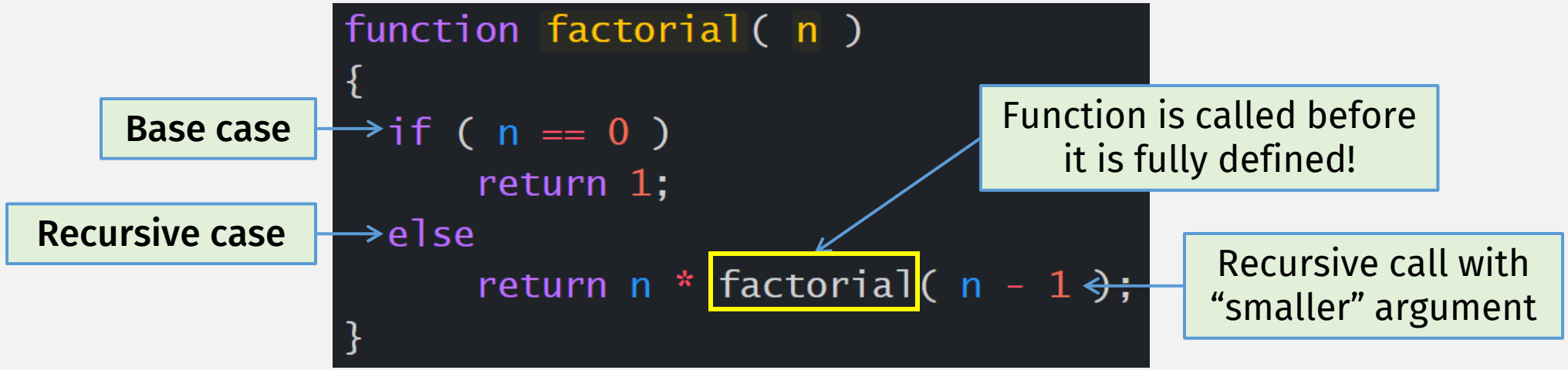
Define a **multi-step transition function:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

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

(Defined recursively)

- Base case: …

Interlude: Recursive Definitions



- 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

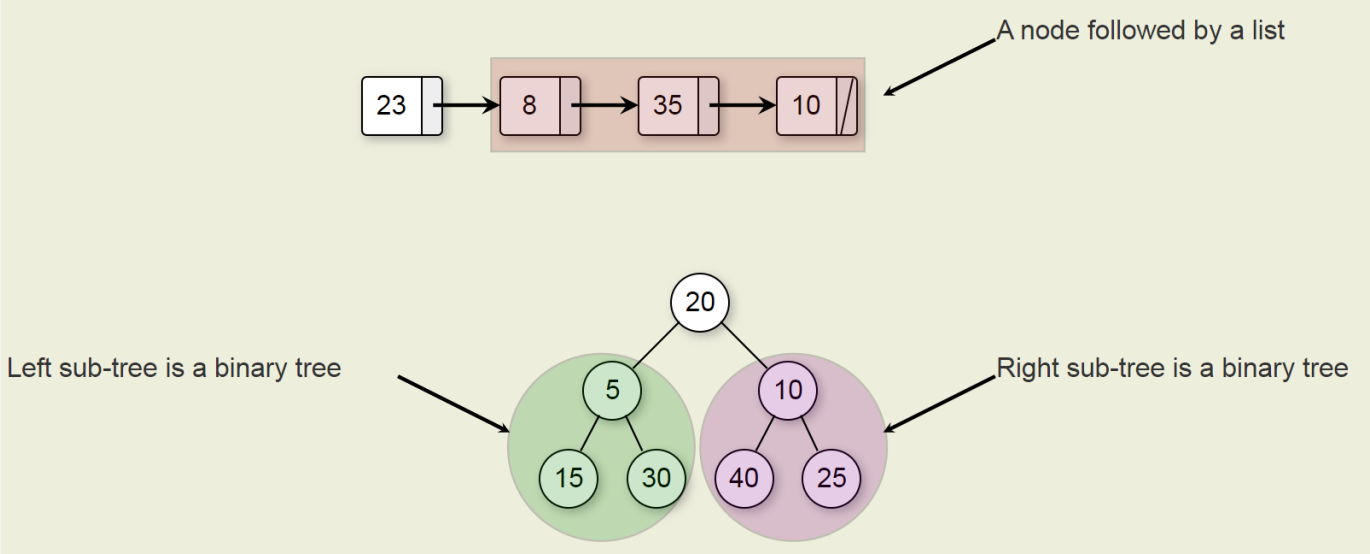
Use of definition before it is fully defined!

"smaller" argument

Examples

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

Recursive Data 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;
}
```

Not good language design!

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

Note: **Where's the base case???**

I call it my billion-dollar mistake. It was the invention of the null reference in 1965.

— Tony Hoare —

Tony Hoare introduced Null references in ALGOL W back in 1965 "simply because it was so easy to implement", says Mr. Hoare. He talks about that decision considering it "my billion-dollar mistake".

Strings Defined Recursively

Base case

Recursive case

A **String** is either:

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

Remember: **strings** are **relative** to some **alphabet** set Σ

Σ^* = set of all possible strings!

Recursive Data needs 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

(The structure of) **Recursive functions** … **match** the **recursively defined input data!**

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

(Defined recursively)

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

Recursive Input Data needs Recursive Function

Base case

A String is either:

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

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 $w = w_1 w_2 \cdots w_n$ 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

Recursive case

"smaller" argument

(Defined recursively)

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

Recursive call

string

char

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

(also called “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:** $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$

- Domain:
  - Input state $q \in Q$ (doesn't have to be start state)
  - Input string $w = w_1 w_2 \cdots w_n$ 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 = **mk-DFA**(Q, Σ, δ, 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 informal ...

- 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 = **mk-DFA**(Q, Σ, δ, 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"**

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

(strings with non-alphabet symbols are impossible)

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

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

- 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**

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

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

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

Languages can be **infinite**

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

"the set of all ..." "such that ..."

Machine and Language Terminology

- The **language** of a **machine** = <u>set of strings</u> that it **accepts**

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

M ***recognizes language*** A ← Set of strings

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

"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*** $\textsc{LangOf}(M)$

$$\textsc{LangOf}(M) = \{w | \ M \text{ accepts } w\}$$

Notation definition:
LangOf is **function mapping**
Machine → Language

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*** $\textsc{LangOf}(M)$

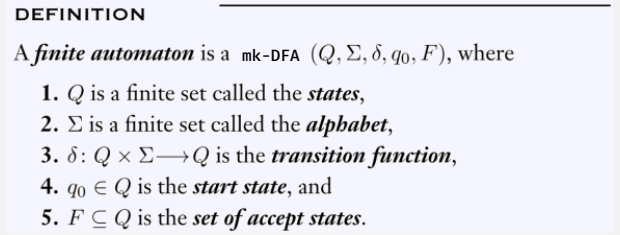
$$\textsc{LangOf}(M) = \{w | \ M \text{ accepts } w\}$$

DefDFACompute

IF M IS-A **DFA Machine**
THEN M **recognizes** $\textsc{LangOf}(M)$

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 `mk-DFA` $(Q, \Sigma, \delta, q_0, F)$, 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 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 <u>really</u> about studying **sets of languages!**