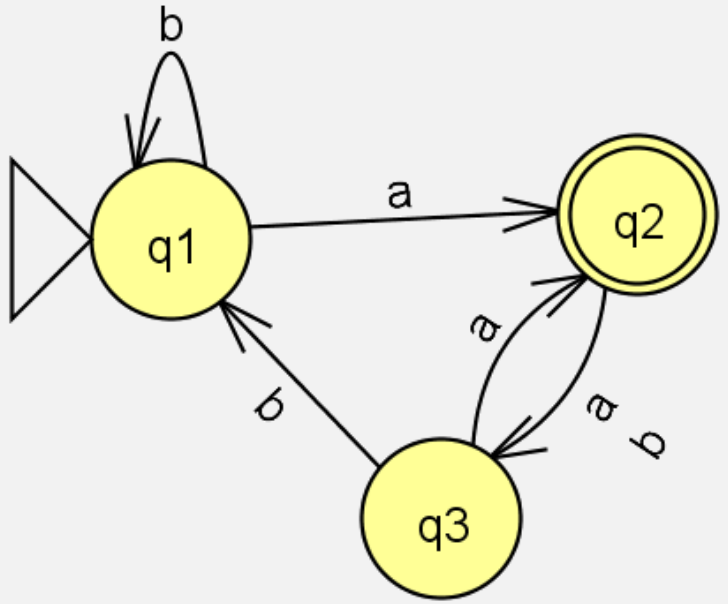
Submit in-class work in gradescope!

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

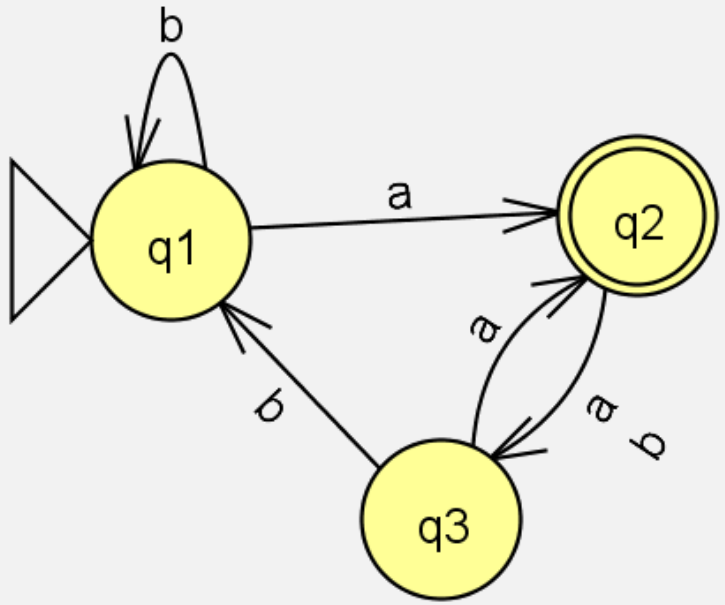
In-class Exercise: solution

- 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}

???

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

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**)



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 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 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***.

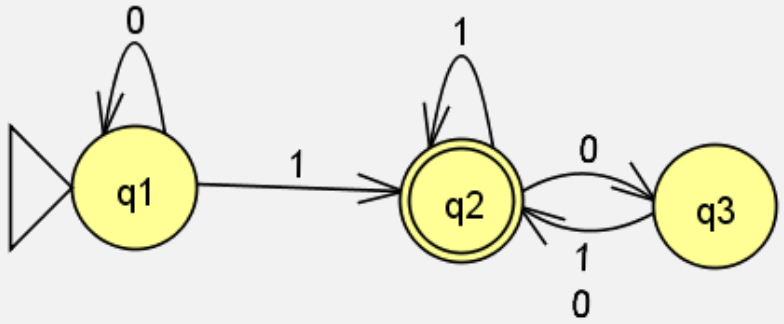
- And **rules** that describe how to **compute** with the **definitions …**

???

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

DFA Computation Rules

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***.

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)

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

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)

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

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 $i = 1, \ldots, n$

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

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

set of pairs

* = "0 or more"

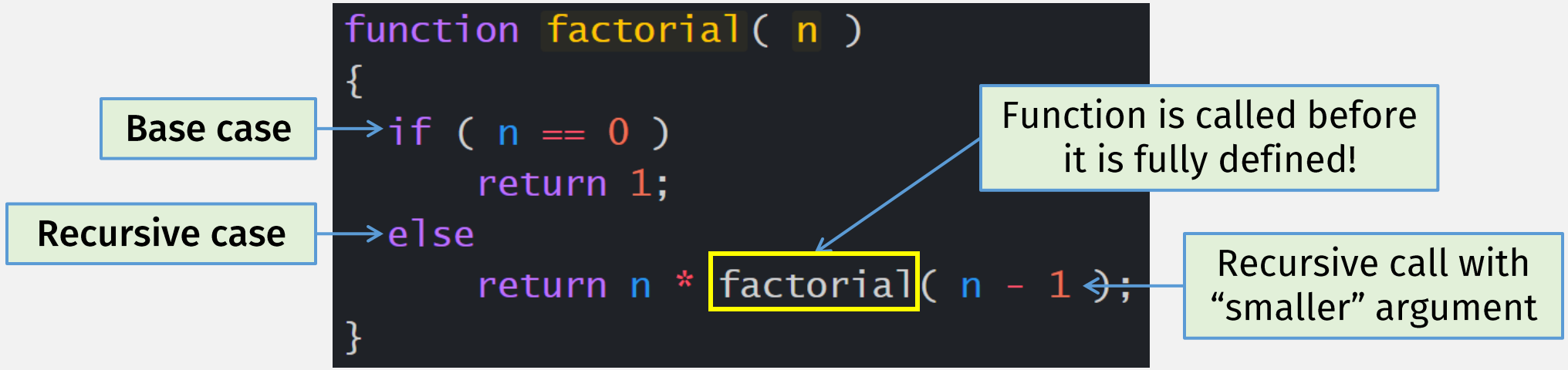
Σ^* = set of all possible strings!

- 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

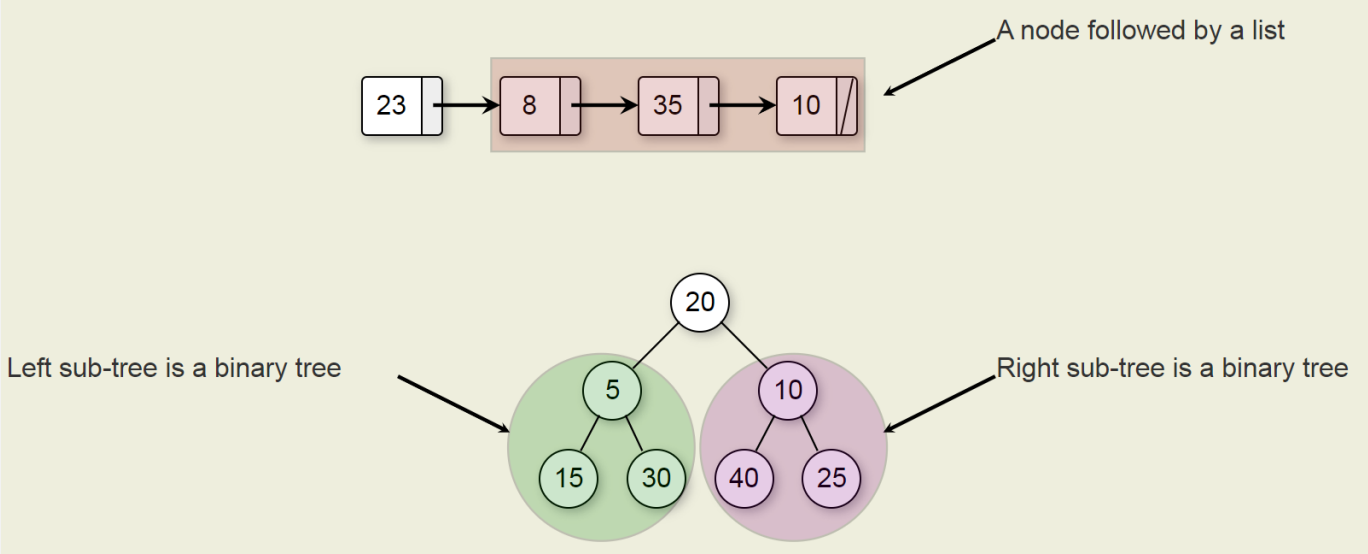
Use of definition before it is fully defined!

"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;
}
```

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

Strings Are 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: **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 $w = w_1 w_2 \cdots w_n$ 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

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

Recursive case

"smaller" argument

Recursive call

string

char

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

- 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)

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

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"**

- 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**

Languages can be **infinite**

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

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

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

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

"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 ← 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*** $L(M)$

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

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**

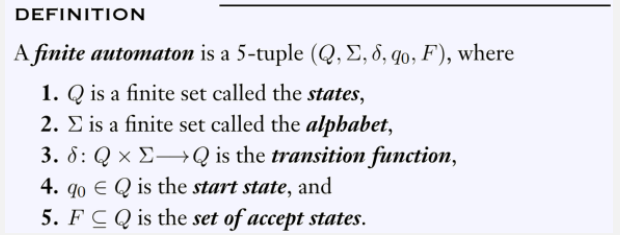
- E.g., A DFA M ***accepts*** w

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

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

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

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 : ??????