BU CS 332 – Theory of Computation

Lecture 2:

- Deterministic Finite Automata,
  Regular languages
- Non-deterministic FAs

Reading:

Sipser Ch 1.1-1.2

Ran Canetti

September 8, 2020

Review: What is a Computational Problem?

A computational problem is represented by way of a function

$$f : D \to R$$

(D is the domain, R is the range).

$$f : \Sigma^* \to \{0,1\}$$

$$f_L(x) = \begin{cases} 1 & \text{if } x \in L \\ 0 & \text{ow} \end{cases}$$

A naïve representation of f : via a table

Elements of D	**Corresponding value of f**
x	$f(x)$
$\vdots$	

Note: $D \to R$ can be infinite! (That's the interesting case…)

What is a Computational Problem?

We will concentrate on functions from *strings* to {0,1}.

(Or: recognizing whether a *string* is in a *language.*)

- **Alphabet**: A finite set Σ

  Ex. $\Sigma = \{a, b, \ldots, z\}$
- **String:** A finite concatenation of alphabet symbols (order matters)

  Ex. $bqr, ababb$

  The length of a string is the number of symbols.

  ε denotes empty string, length 0

  Σ^n Σ^* = set of all finite strings over Σ
- **Language:** A (possibly infinite) set L of strings : $L \subseteq \Sigma^*$

Examples of Languages (Computational problems)

Parity: Given a string consisting of a's and b's, does it contain an even number of a's?

$\Sigma = \{a, b\}$ $\quad L = \{x \mid x \text{ has an even \# of } a\text{'s}\}$

Primality: Given a natural number x (represented in binary), is x prime?

$\Sigma = \{0, 1\}^*$ $\quad L = \{x \mid x \text{ represents a number which is prime}\}$

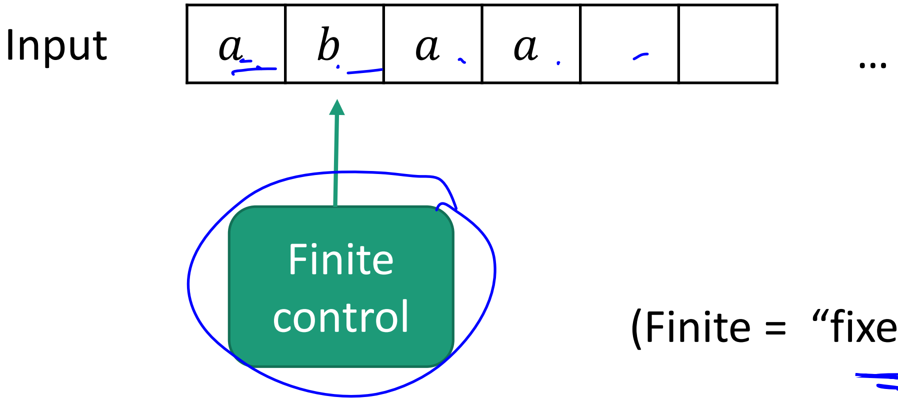
Halting Problem: Given a C program, can it ever get stuck in an infinite loop?

$\Sigma = \{\text{ascii chars}\}$ $\quad L = \{x \mid \text{there exists input s.t. } \langle x\rangle(a) \text{ gets stuck in i.l.}\}$

Models of computation: Machines

Computation is the processing of information by the **repeated application** of a **small set** of **Simple** operations.

➔ What is the simplest "machine model" that can capture computation?

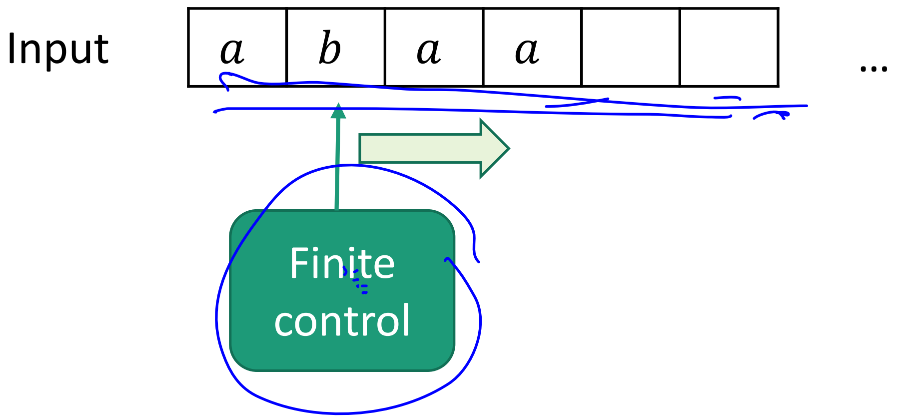


(Finite = "fixed size, independent of input length")

<u>Abstraction:</u> We don't care how the control is implemented. We simply consider "states" of the control. We want the possible number of states to be small/bounded, and to transition between states using fixed rules.

Machine Models

- Finite Automata (FAs): Machine with a finite amount of unstructured memory



Control scans input left-to-right

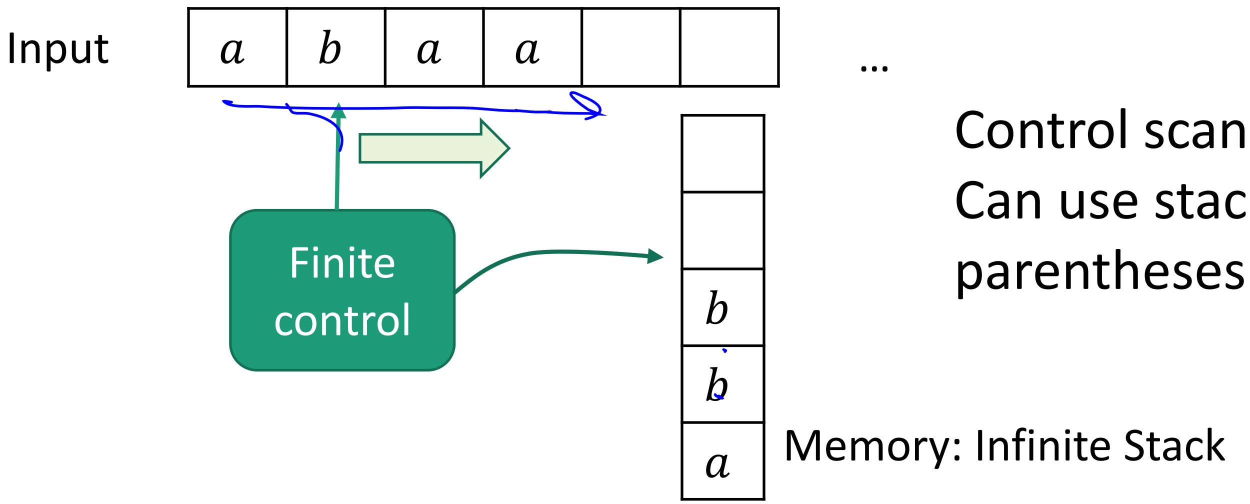
Can check simple patterns

Can't perform unlimited counting

Useful for modeling chips, simple control systems, choose-your-own adventure games, streaming algorithms…

Machine Models

- Pushdown Automata (PDAs): Machine with unbounded structured memory in the form of a stack

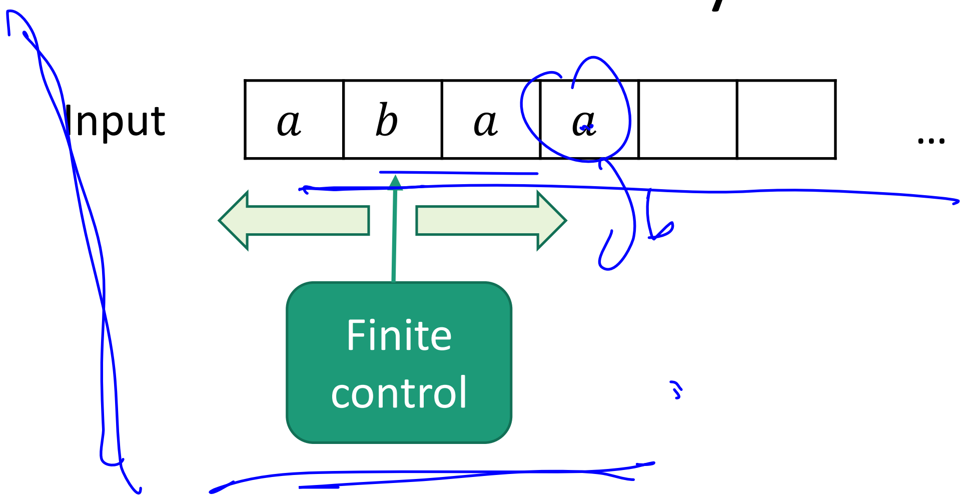


Control scans input left-to-right
Can use stack to count, balance parentheses

Useful for modeling parsers, compilers, some math calculations

Machine Models

- **Turing Machines (TMs):** Machine with unbounded, unstructured memory



Control can scan in both directions
Control can both read and write

Model for general sequential computation
Church-Turing Thesis: Everything we intuitively think of as "computable" is computable by a Turing Machine

What would we like to know?

We will classify languages (computational problems) based on which types of machines can recognize them

Then we will show thing like:

Inclusion: Every language recognizable by a FA is also recognizable by a TM

Non-inclusion: There exist languages recognizable by TMs which are not recognizable by FAs

Hardness: Identify a “hard” and “easy” languages
Robustness: Alternative definitions of the same class

Ex. Languages recognizable by FAs = regular expressions

Why study theory of computation?

- You will learn how to formally reason about computation
- You will learn the technology-independent foundations of CS

Philosophically interesting questions:

  - Are there well-defined problems which cannot be solved by computers?
  - Can we always find the solution to a puzzle faster than trying all possibilities?
  - Can we say what it means for one problem to be "harder" than another?

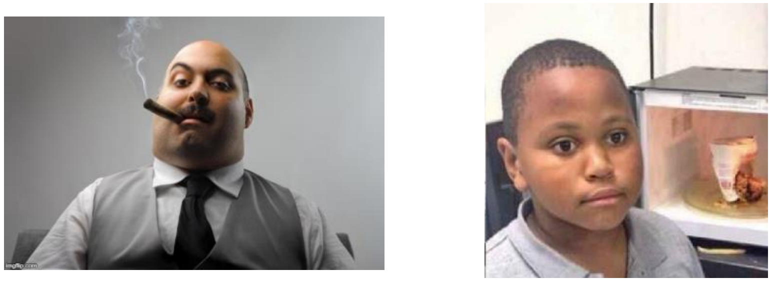
  - This is the core of CS!

Why study theory of computation?

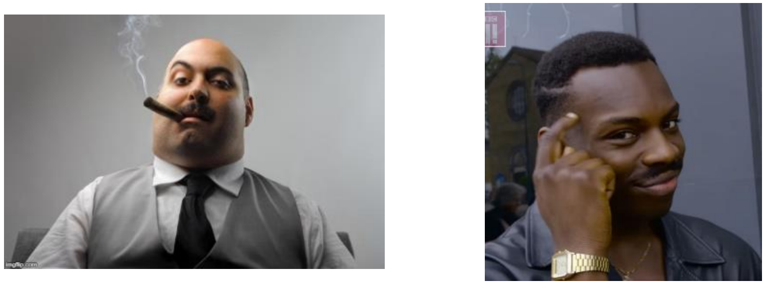
- You will learn how to formally reason about computation
- You will learn the technology-independent foundations of CS

Why study theory of computation?

Practical knowledge for developers



“Boss, I can’t find an efficient algorithm.
I guess I’m just too dumb.”

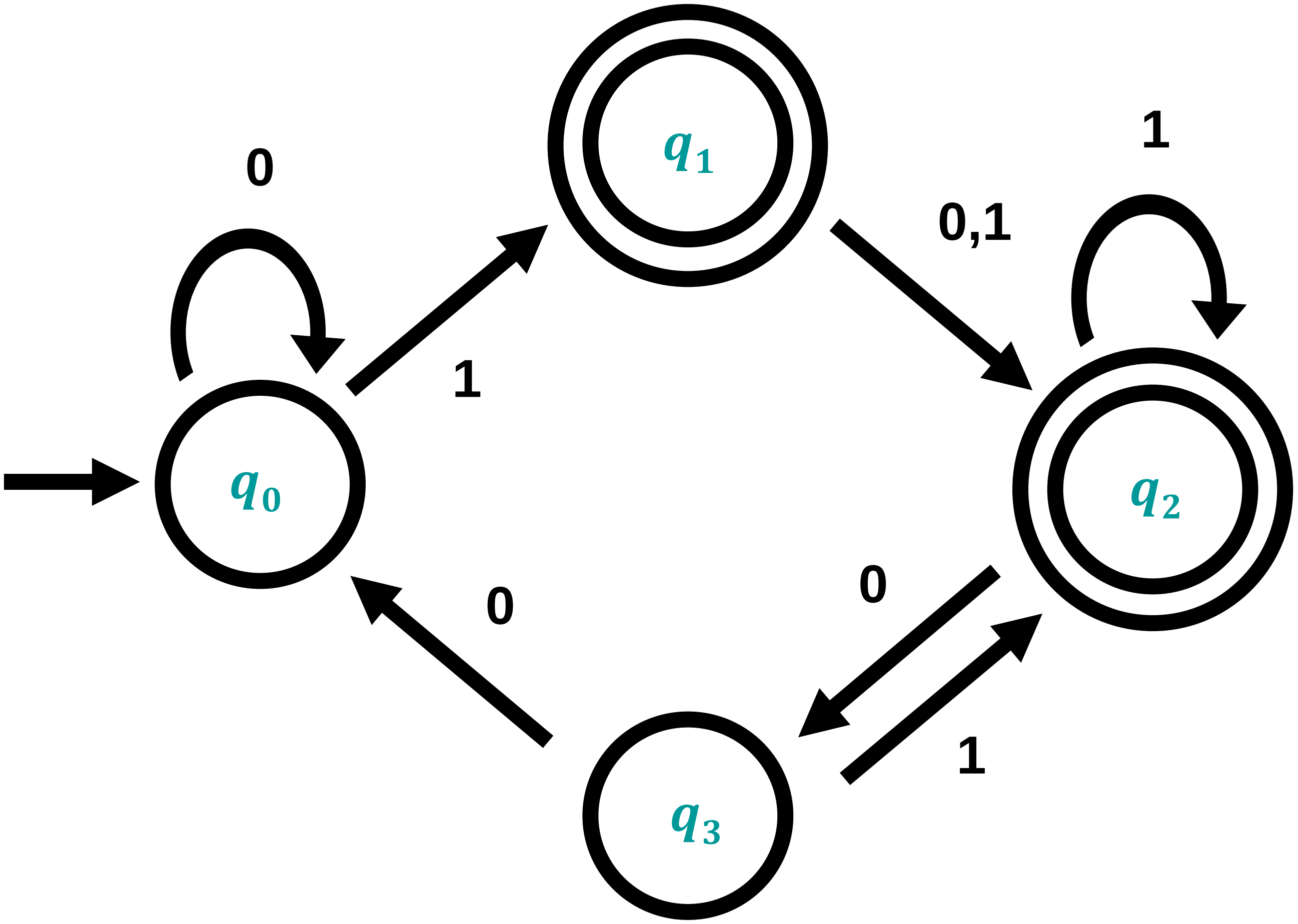


“Boss, I can’t find an efficient algorithm
because no such algorithm exists.”

Will you be asked about this material on job interviews?

No promises, but a true story…

Anatomy of a DFA



Formal Definition of a DFA

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

- Q is the set of states
- Σ is the alphabet
- $\underline{\delta} : \underline{Q} \times \underline{\Sigma} \rightarrow Q$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accept states

$Q = \{q_0, q_1\}$

$\Sigma = \{a, b\}$

	a	b
q_0	q_1	q_0
q_1	q_0	q_1

$F = \{q_0\}$

Formal Definition of DFA Computation

A DFA $M = (Q, \Sigma, \delta, q_0, F)$ **accepts** a string $w = w_1 w_2 \cdots w_n \in \Sigma^*$ (where each $w_i \in \Sigma$) if there exist $r_0, \ldots, r_n \in Q$ such that

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for each $i = 0, \ldots, n-1$, and
3. $r_n \in F$

$L(M) =$ [illegible]

$L(M)$ = the **language** of machine M

= set of all (finite) strings machine M accepts

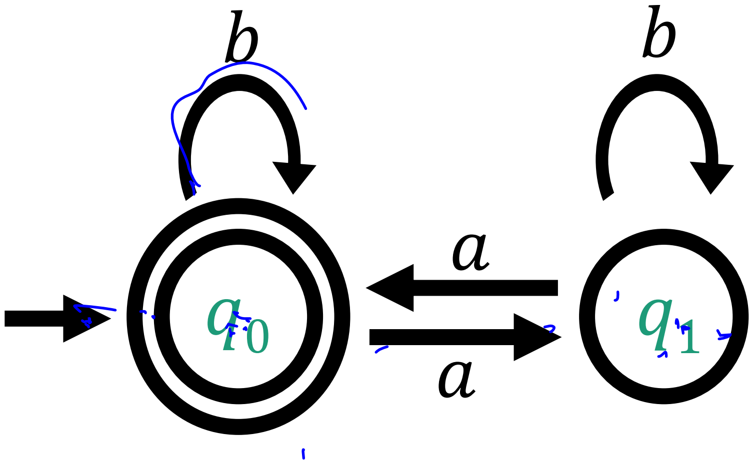
M **recognizes** the language $L(M)$

A DFA for Parity

Parity: Given a string consisting of a's and b's, does it contain an even number of a's?

$\Sigma = \{a, b\}$ $L = \{w \mid w \text{ contains an even number of } a\text{'s}\}$

a a b b a b a



State set Q =

Alphabet Σ =

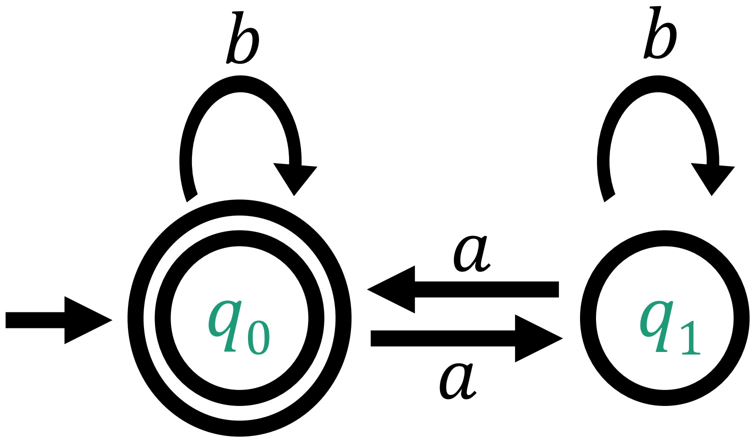
Transition function δ

δ	a	b
q_0		
q_1		

Start state q_0

Set of accept states F =

Example: Computing with the Parity DFA



Let $w = abba$

Does M accept w?

A DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepts a string $w = w_1 w_2 \cdots w_n \in \Sigma^*$ (where each $w_i \in \Sigma$) if there exist $r_0, \ldots, r_n \in Q$ such that

1. $r_0 = q_0$
2. $\delta(r_i, w_{i+1}) = r_{i+1}$ for each $i = 0, \ldots, n-1$, and
3. $r_n \in F$

Automata Tutor

http://automatatutor.com/

Regular Languages

L is regular iff $\exists$ DFA M s.t. $L = L(M)$

<u>Definition:</u> A language is regular if it is recognized by a DFA

- $L = \{ w \in \{a, b\}^* \mid w$ has an even number of a's $\}$ is regular
- $L = \{ w \in \{0, 1\}^* \mid w$ contains $001 \}$ is regular

Many interesting programs recognize regular languages

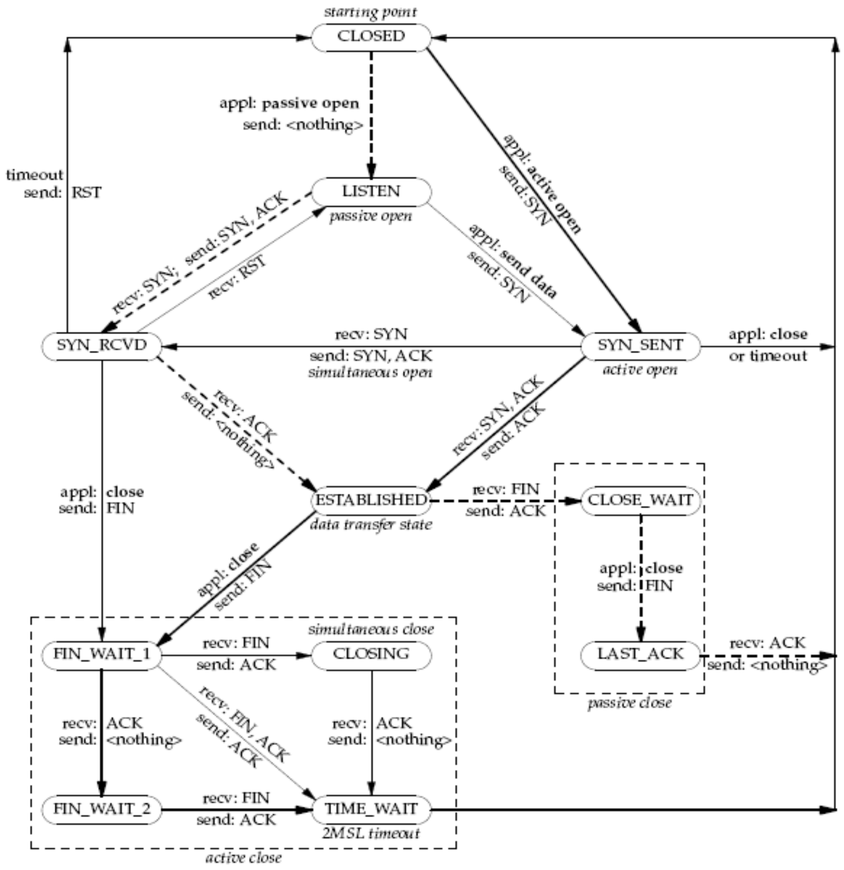
NETWORK PROTOCOLS

COMPILERS

GENETIC TESTING

ARITHMETIC

Internet Transmission Control Protocol



Let TCPS = { w | w is a complete TCP Session}

Theorem. TCPS is regular

Compilers

Comments :

Are delimited by /* */

Cannot have nested /* */

Must be closed by */

***/ is illegal outside a comment**

COMMENTS = {strings over {0,1, /, *} with legal comments}

Theorem. COMMENTS is regular.

Genetic Testing

DNA sequences are strings over the alphabet $\{A, C, G, T\}$.

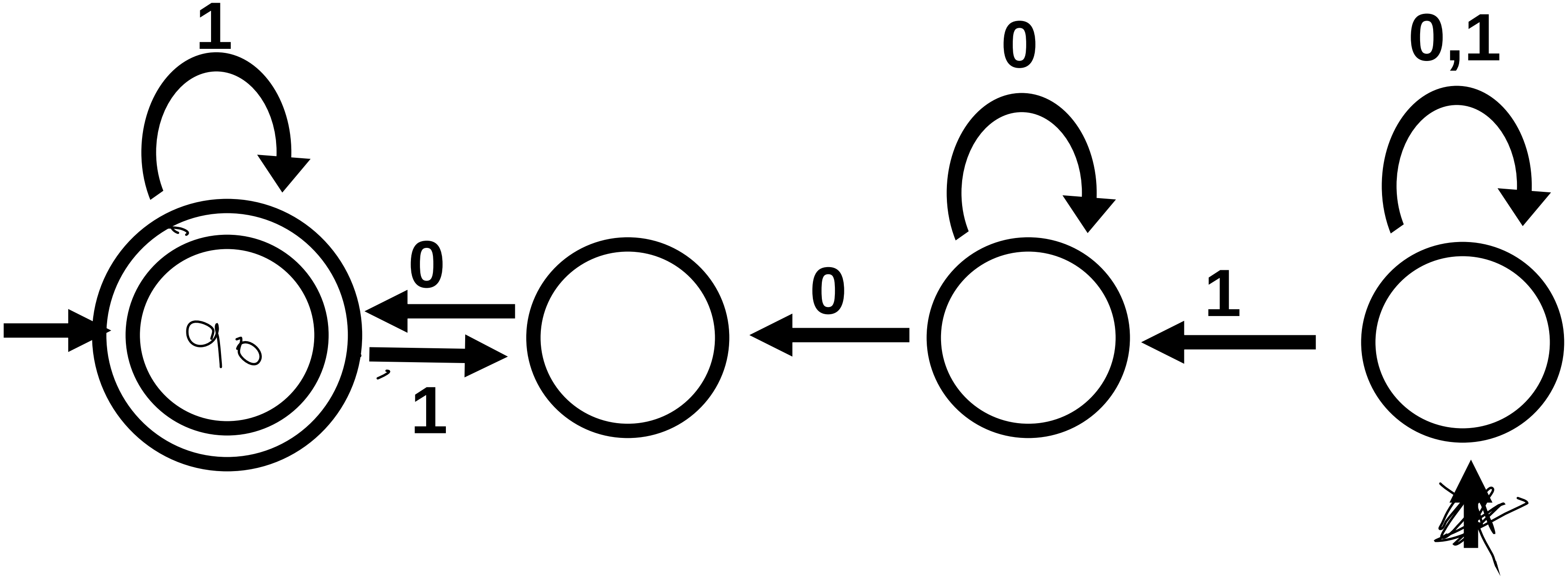
A gene g is a special substring over this alphabet.

A genetic test searches a DNA sequence for a gene.

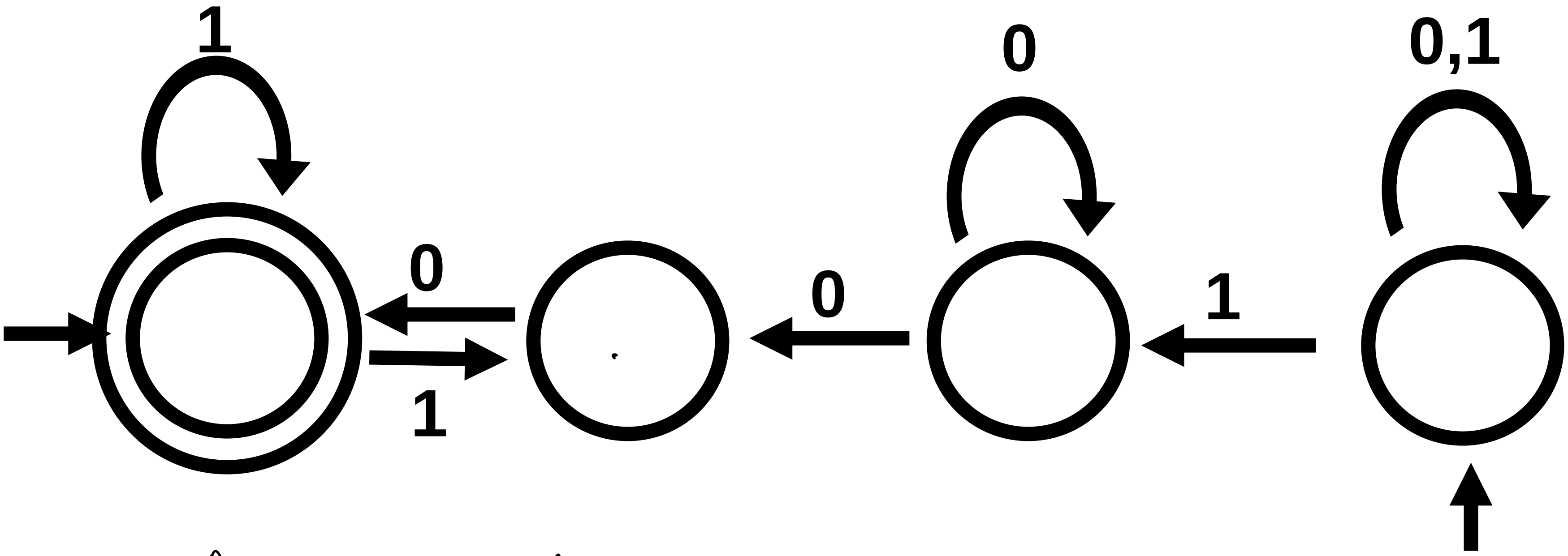
GENETICTEST$_g$ = {strings over $\{A, C, G, T\}$ containing g as a substring}

Theorem. GENETICTEST$_g$ is regular for every gene g.

Non-deterministic Finite Automata


1
0
0,1
q0
0
0
1
1

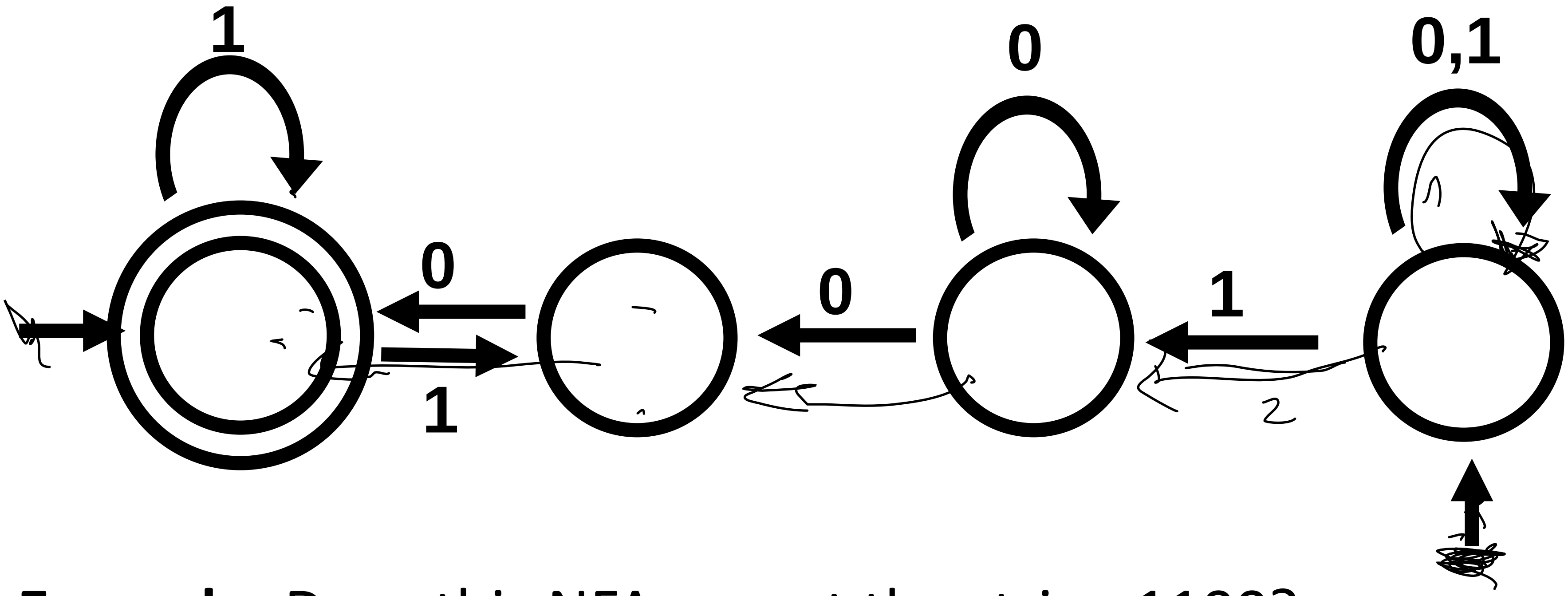
Nondeterminism



1 0 1 1 0 1

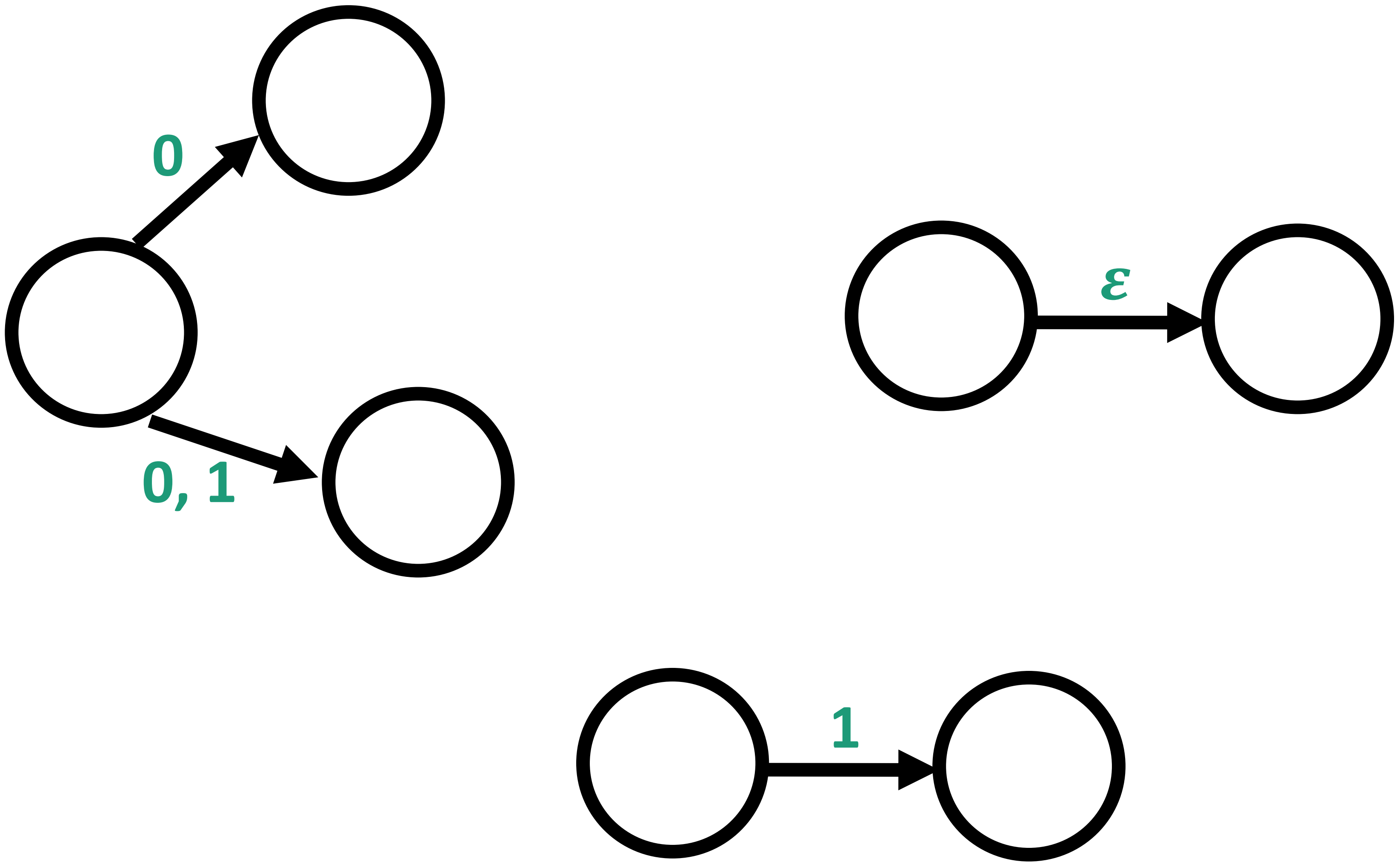
A **Nondeterministic Finite Automaton** (NFA) accepts if there is a way to make it reach an accept state.

Nondeterminism

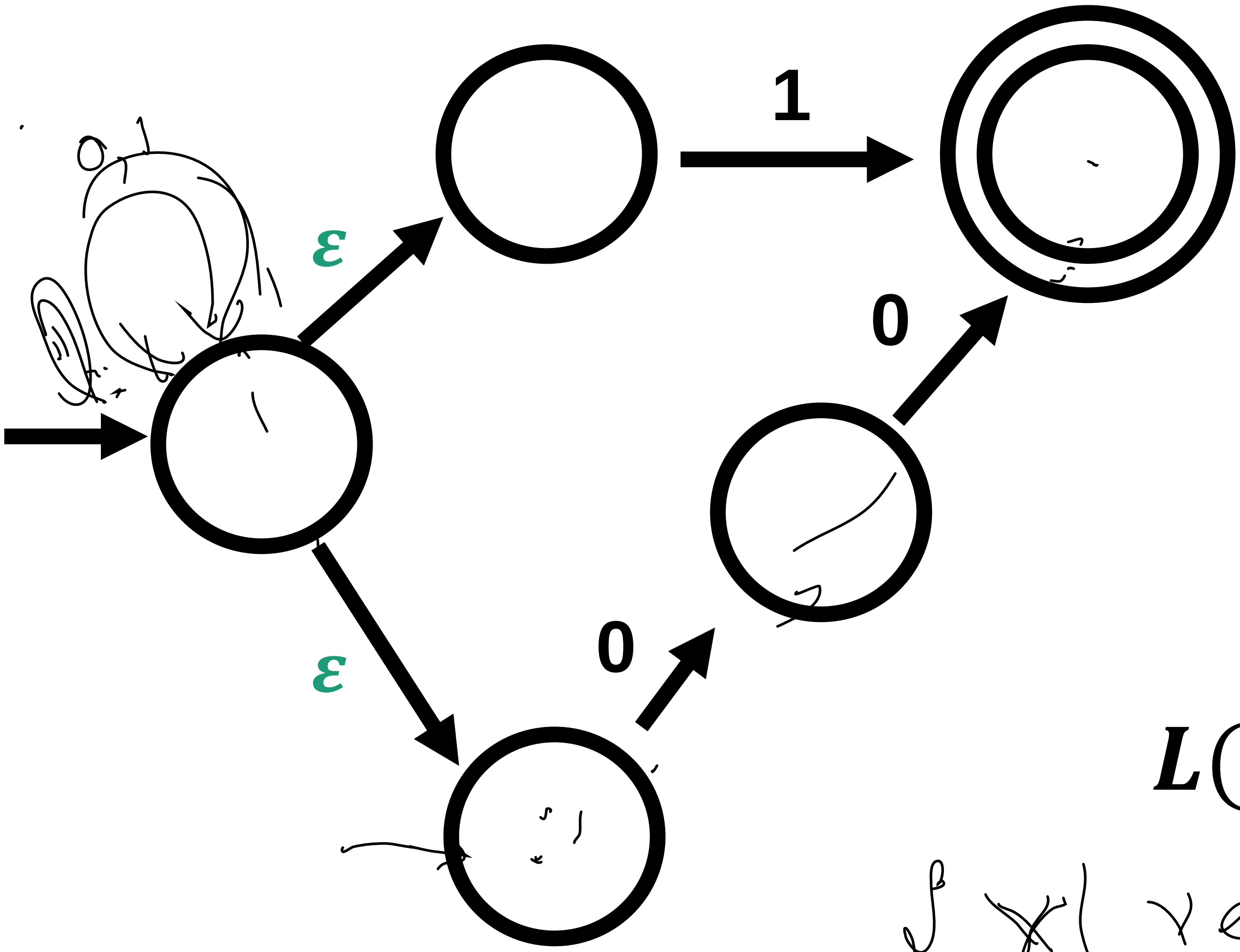


Example: Does this NFA accept the string 1100?

Some special transitions



Example

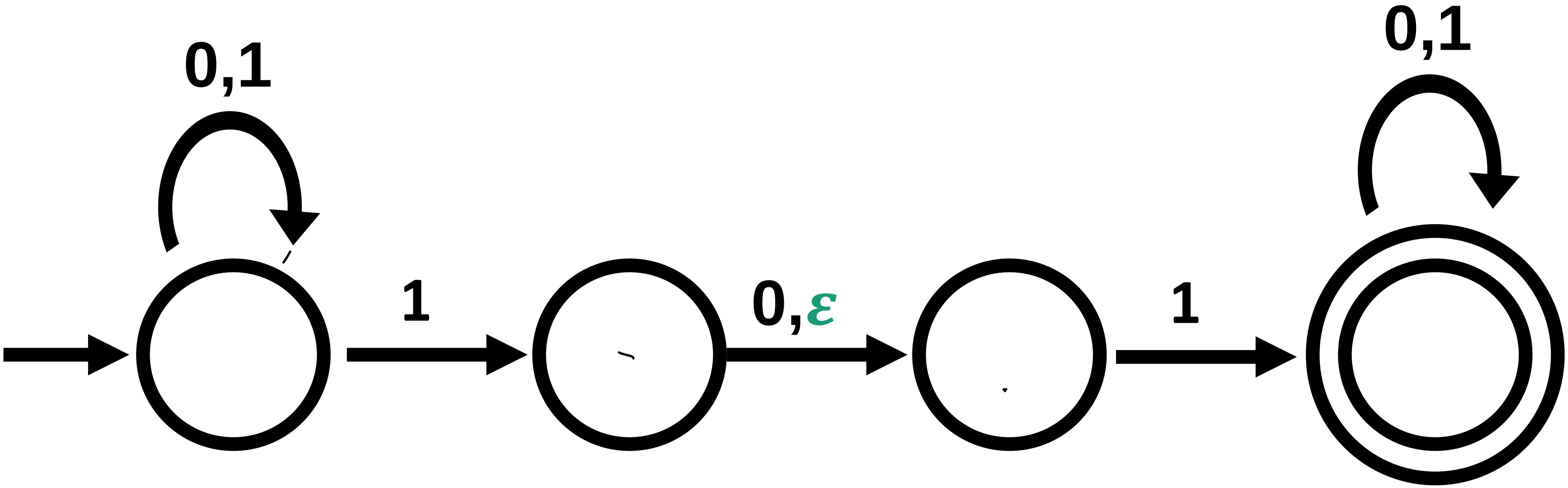


1011 00

$L(M) =$ {1, 00}

{x | x ends with 00 or 1}

Example



$L(M) = \{ x \mid$ x has either 11 or 101 as substring $\}$

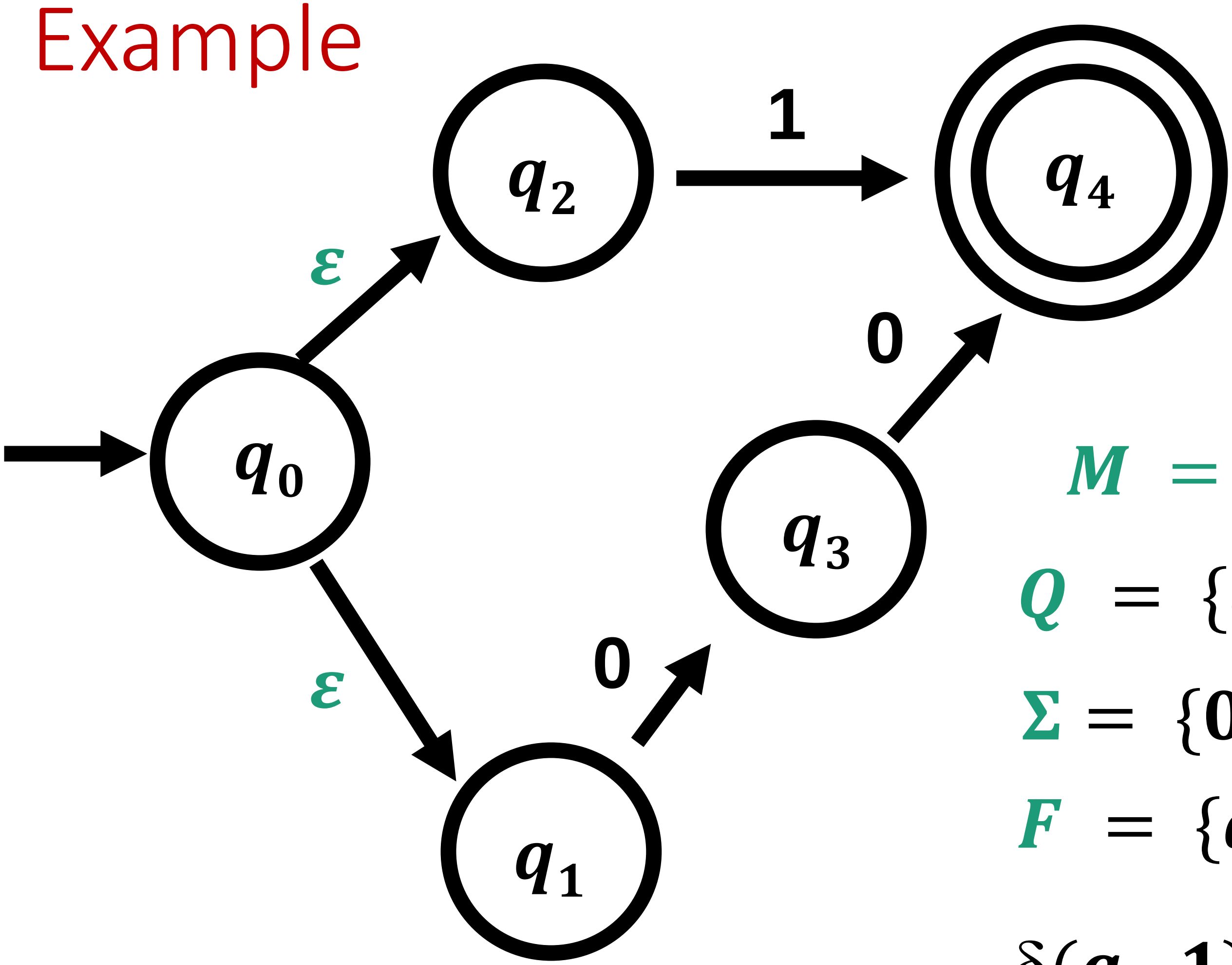
Formal Definition of a NFA

An NFA is a 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$

- Q is the set of states
- Σ is the alphabet $\qquad \Sigma \cup \{\varepsilon\}$
- $\delta : \underline{Q} \times \underline{\Sigma_\varepsilon} \rightarrow \underline{P(Q)}$ is the transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is the set of accept states

M ***accepts*** a string w if **there exists** a path from q_0 to an accept state that can be followed by reading w.

Example



$$M = (Q, \Sigma, \delta, Q_0, F)$$

$$Q = \{q_0, q_1, q_2, q_3, q_4\}$$

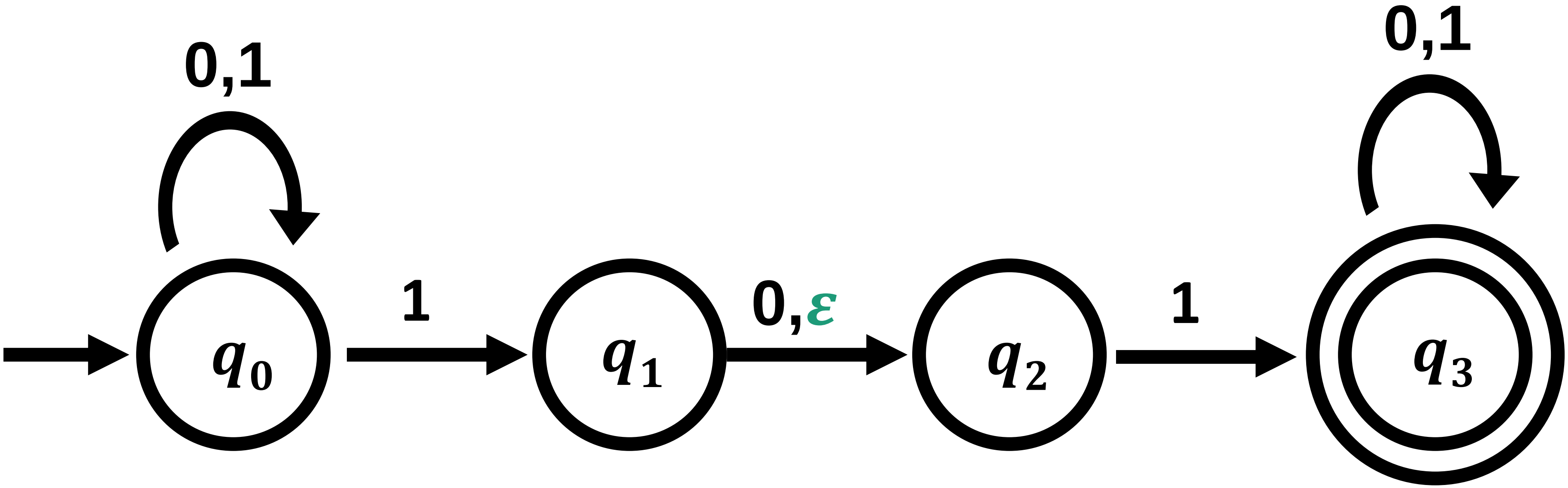
$$\Sigma = \{0, 1\}$$

$$F = \{q_4\}$$

$$\delta(q_2, 1) =$$

$$\delta(q_3, 1) =$$

Example



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

$Q = \{q_0, q_1, q_2, q_3\}$

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

$F = \{q_3\}$

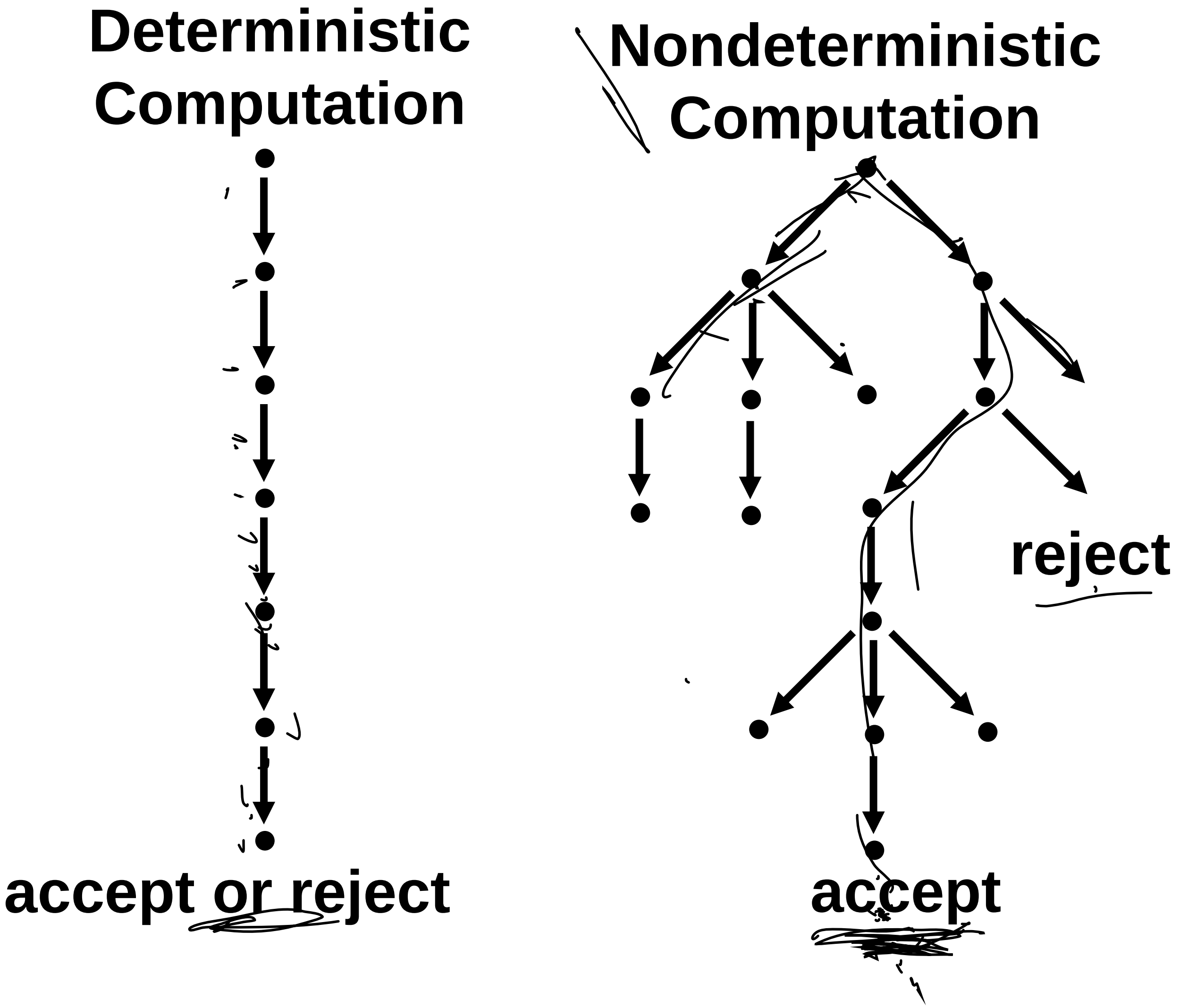
$\delta(q_0, 0) =$

$\delta(q_0, 1) =$

$\delta(q_1, \varepsilon) =$

$\delta(q_2, 0) =$

Nondeterminism



Ways to think about nondeterminism

- **(restricted) parallel computation**
- **tree of possible computations**
- **guessing and verifying the "right" choice**

Why study NFAs?

- Not really a realistic model of computation: Real computing devices can't really try many possibilities in parallel

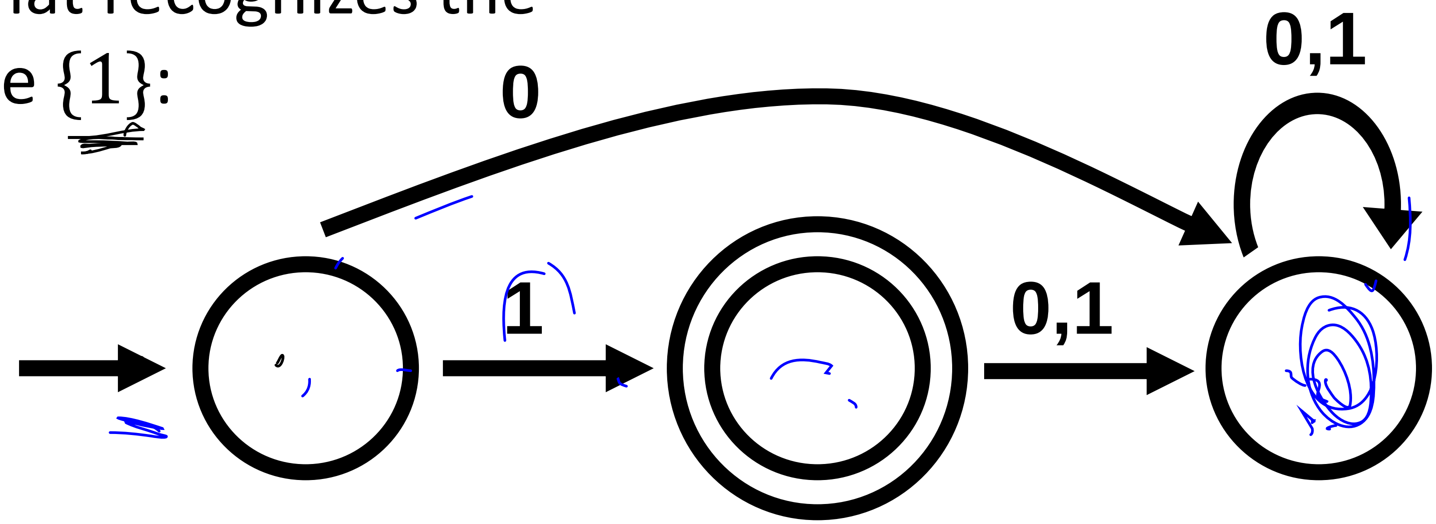
But:

- Useful tool for understanding power of DFAs/regular languages
- NFAs can be simpler than DFAs
- Lets us study "nondeterminism" as a resource

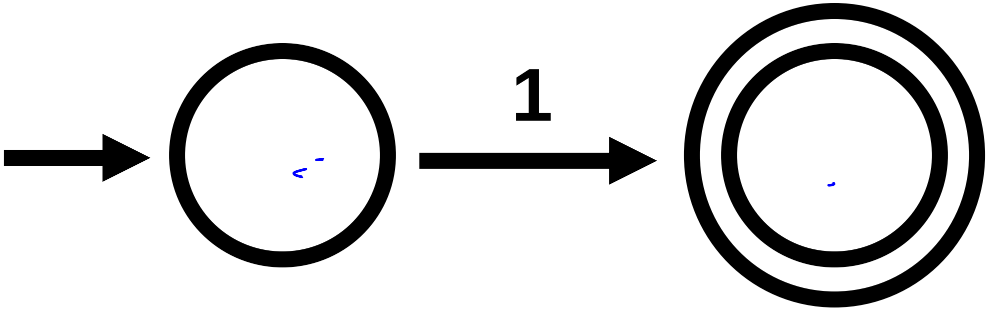
(cf. P vs. NP)

NFAs can be simpler than DFAs

A DFA that recognizes the language {1}:



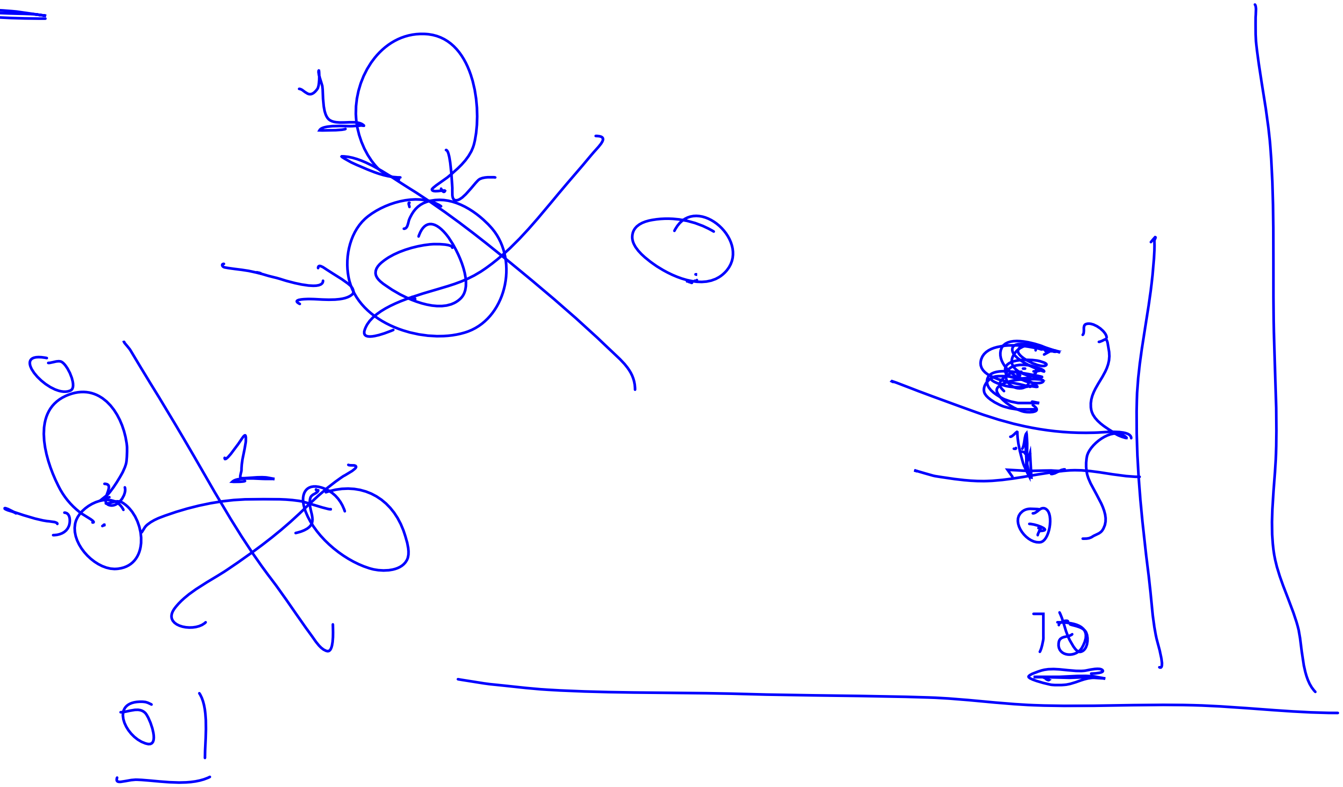
An NFA that recognizes the language {1}:



Sometimes DFAs **must** be larger

Theorem. Every DFA for the language $\{1\}$ must have at least 3 states.

Proof:



Equivalence of NFAs and DFAs

Equivalence of NFAs and DFAs

Every DFA *is* an NFA, so NFAs are *at least* as powerful as DFAs

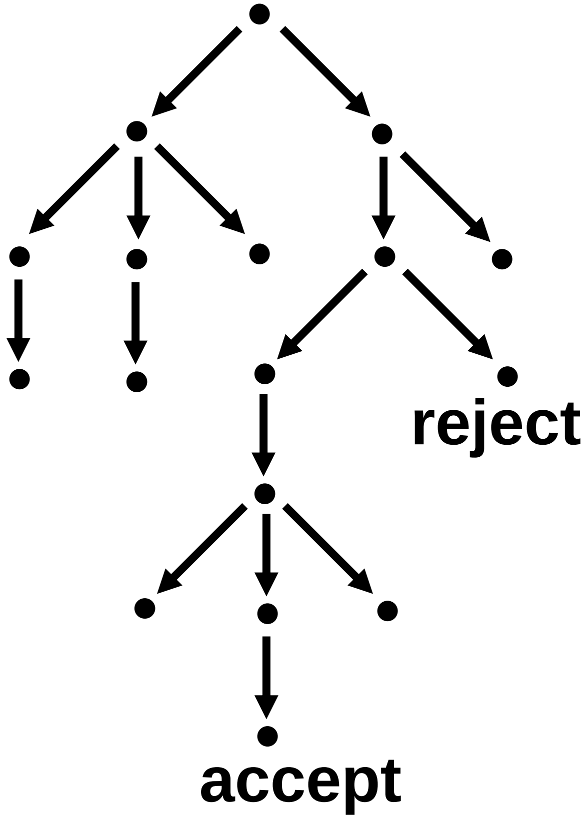
Theorem: For every NFA N, there is a DFA M such that $L(M) = L(N)$

Corollary: A language is regular if and only if it is recognized by an NFA

Equivalence of NFAs and DFAs (Proof)

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA

Goal: Construct DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizing $L(N)$

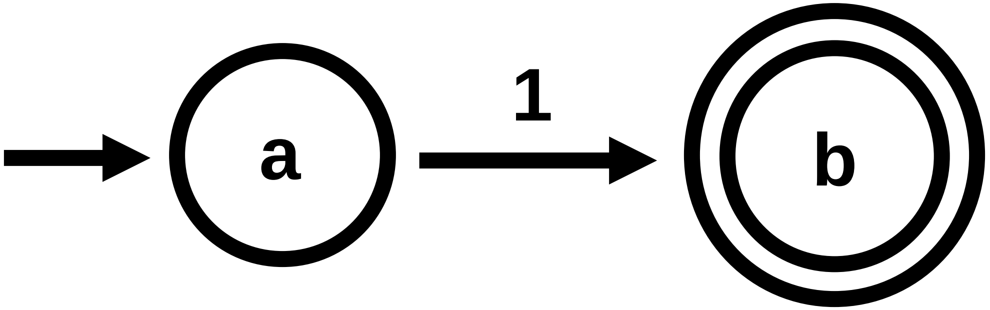


Intuition: Run all threads of N in parallel, maintaining the set of states where all threads are.

Formally: $Q' = P(Q)$

“The Subset Construction”

NFA -> DFA Example



Subset Construction (Formally)

Input: NFA $N = (Q, \Sigma, \delta, q_0, F)$

Output: DFA $M = (Q', \Sigma, \delta', q_0', F')$

Q'

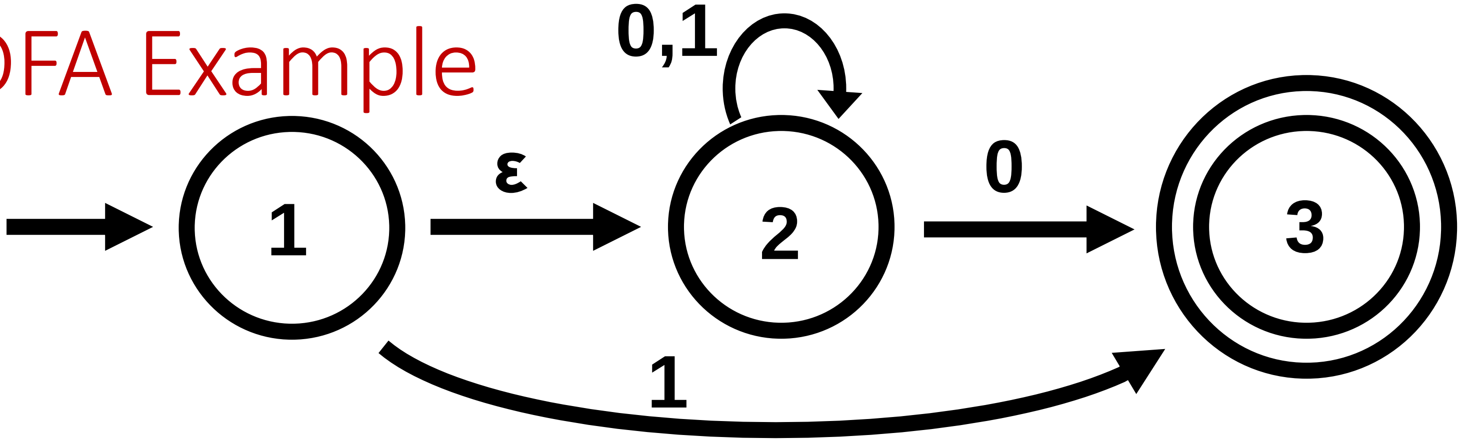
$\delta' : Q' \times \Sigma \rightarrow Q'$

$\delta'(R, \sigma) =$ for all $R \subseteq Q$ and $\sigma \in \Sigma$.

$q_0' =$

$F' =$

NFA -> DFA Example



Subset Construction (Formally)

Input: NFA $N = (Q, \Sigma, \delta, q_0, F)$

Output: DFA $M = (Q', \Sigma, \delta', q_0', F')$

$$Q' = P(Q)$$

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

$\delta'(R, \sigma) = \bigcup_{r \in R} \delta(r, \sigma)$ for all $R \subseteq Q$ and $\sigma \in \Sigma$.

$$q_0' = \{q_0\}$$

$F' = \{ R \in Q' \mid R$ contains some accept state of $N\}$

Proving the Construction Works

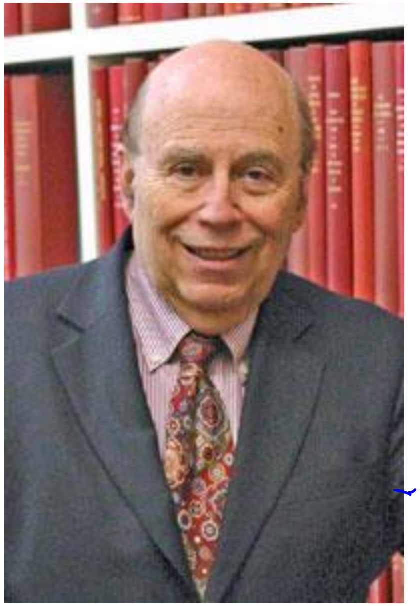
Claim: For every string w, running M on w leads to state

$$\{q \in Q \mid \text{There exists a computation of } N \text{ on input } w \text{ ending at } q\}$$

Proof idea: By induction on $|w|$

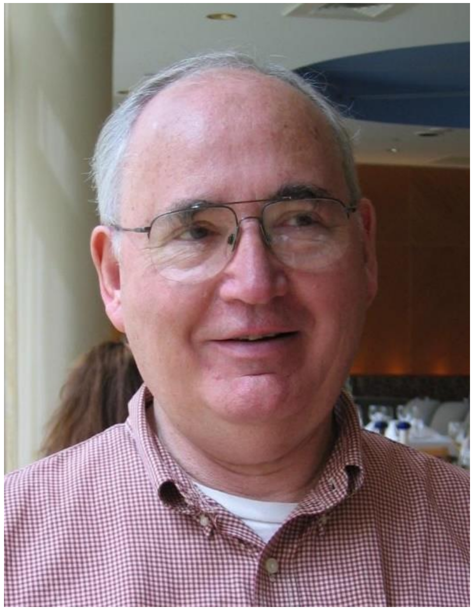
Historical Note

Subset Construction introduced in Rabin & Scott’s 1959 paper “Finite Automata and their Decision Problems”



1976 ACM Turing Award citation

For their joint paper "Finite Automata and Their Decision Problem," which introduced the idea of nondeterministic machines, which has proved to be an enormously valuable concept. Their (Scott & Rabin) classic paper has been a continuous source of inspiration for subsequent work in this field.



Is this construction the best we can do?

Subset construction converts an n state NFA into a 2^n-state DFA

Could there be a construction that always produces, say, an n^2-state DFA?

Theorem: For every $n \geq 1$, there is a language L_n such that

1. There is an $(n+1)$-state NFA recognizing L_n.
2. There is no DFA recognizing L_n with fewer than 2^n states.

Conclusion: For finite automata, nondeterminism provides an exponential savings over determinism (in the worst case).