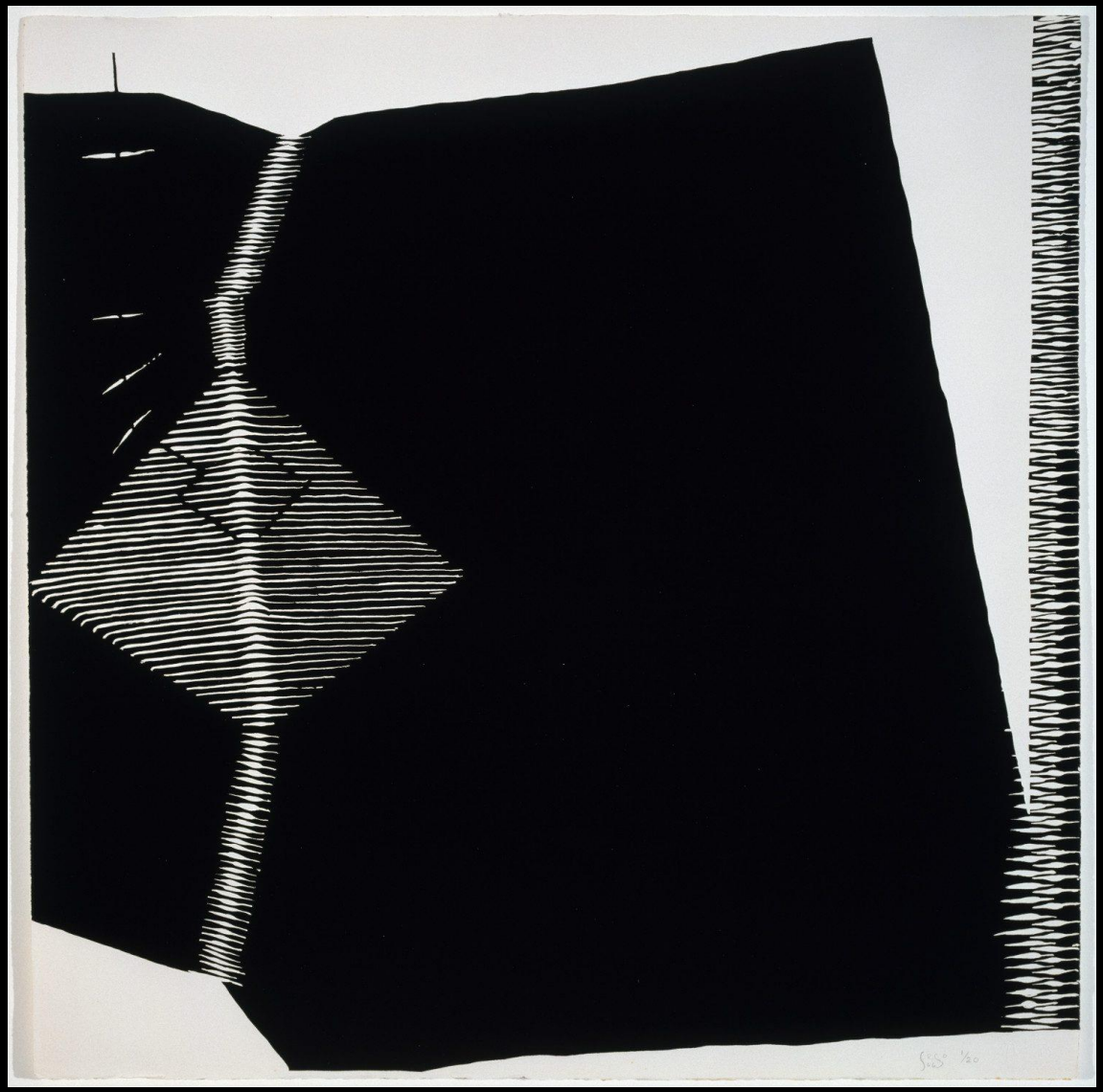


Regular Languages

Contents

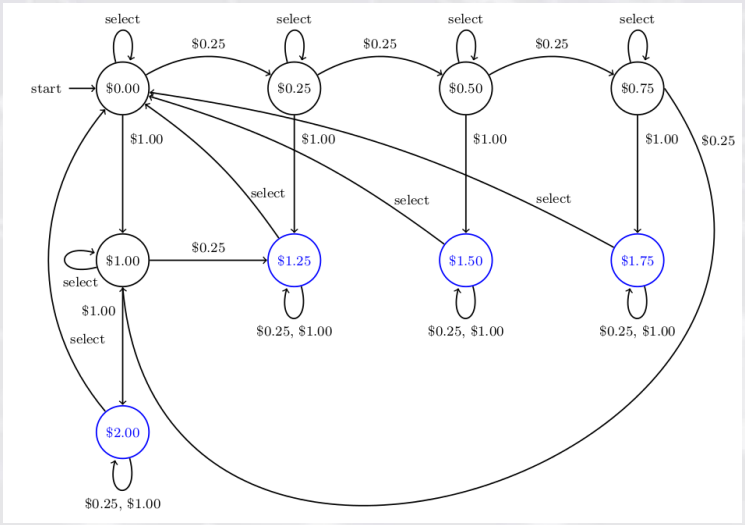
- Finite Automata
- Non-Determinism
- Regular Expressions
- Non-Regular Languages

Finite Automata

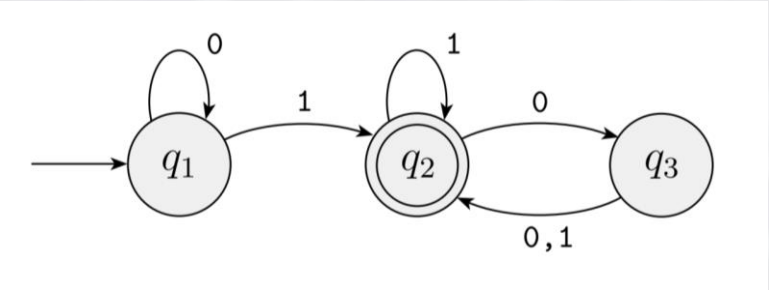
• The theory of computation begins with the basic question: *What is a computer*?

• In our subsequent discussion we introduce various types of abstract, computational models. We begin with the simplest such model: a **finite state automaton** (**FA**), plural: automata.

• Finite automata are good models for computers with an <u>extremely limited amount of memory</u> (e.g. a basic controller). As we will show, this limited memory significantly limits what is computable by a FA.



Finite Automata

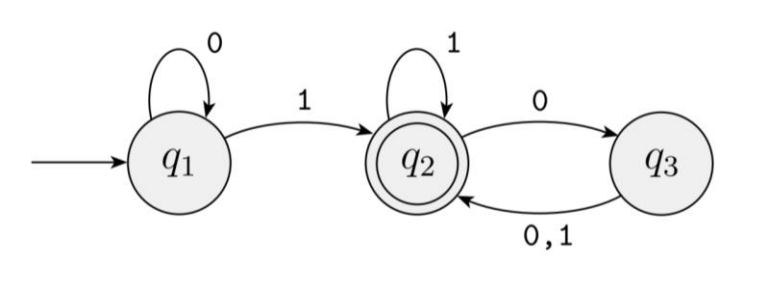


• The figure above is called a **state diagram** of an automaton, which we will call *M*.

• *M* has three **states**, labeled q_1, q_2 and q_3. The **start state**, q_1, is indicated by the arrow pointing at it from nowhere. The **accept state**, q_2, is the one with a double circle. The arrows going from one state to another are called **transitions**.

• When the automaton receives an input string such as 1101, it processes that string and produces an output; the output is either **accept** or **reject**.

Finite Automata



• For example, when we feed the input string **1101** into *M*, the processing proceeds as follows:

1. Start in state q_1
2. Read 1, follow transition from q_1 to q_2.
3. Read 1, follow transition from q_2 to q_2.
4. Read 0, follow transition from q_2 to q_3.
5. Read 1, follow transition from q_3 to q_2.
6. **Accept** because *M* is in accept state q_2 at the end of the input.

Finite Automata

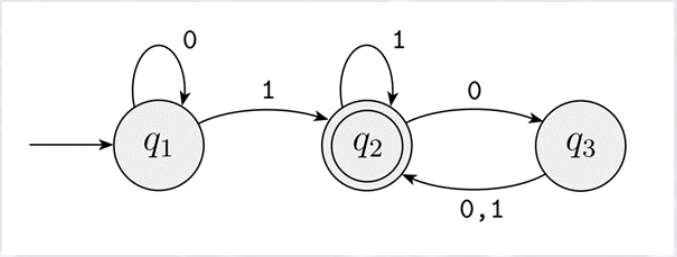
- Formally, a finite automaton is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where:

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 that $F = \emptyset$ is permitted).

Finite Automata

- Formally, 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. $\delta : \Sigma \longrightarrow Q$ 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 that $F = \emptyset$ is permitted).

- From the previous example,
  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_0 is the start state, and
  5. $F = \{q_2\}$.

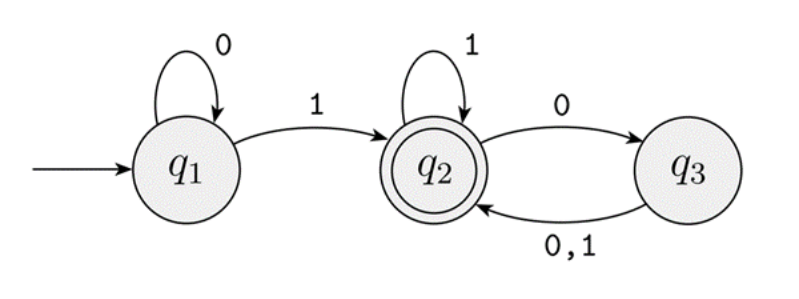
Finite Automata

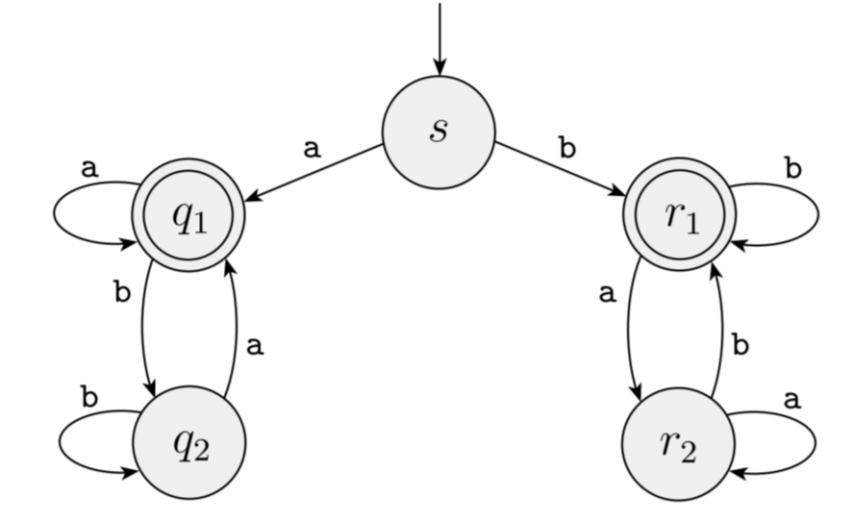
• If A is the set of all strings that machine M accepts, we say that **A is the language of machine M** and write $\mathrm{L}(M) = A$.

We say that **M recognizes A** or that **M accepts A**. Note that a machine may accept many stings, but it always recognizes only one language. If M accepts no strings, then $\mathrm{L}(M) = \emptyset$.

Finite Automata

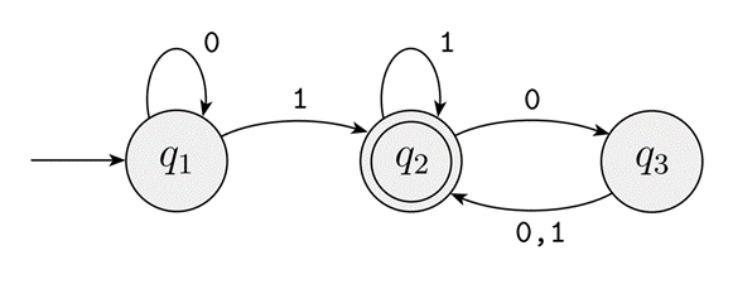
- Determine the languages accepted by the following FA.



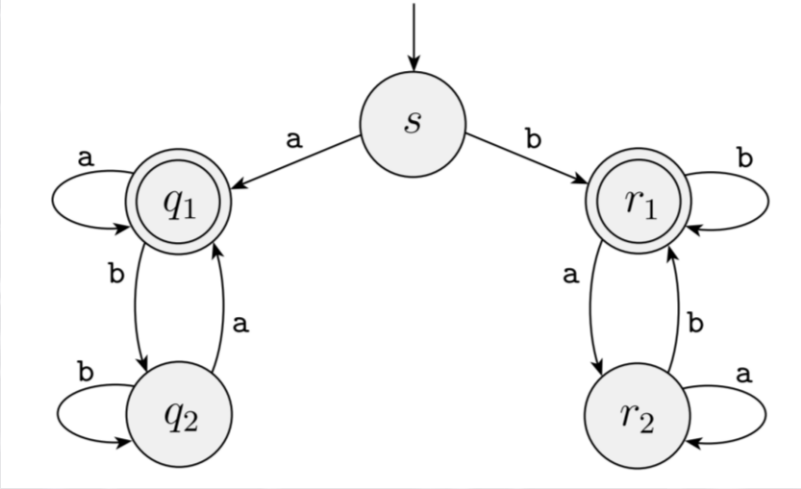


Finite Automata

- Determine the languages accepted by the following FA.



$$L(A) = \left\{ \begin{matrix} w | w \text{ contains at least one} \\ 1 \text{ and an even numbers of } 0s \text{ follow the last } 1 \end{matrix} \right\}$$



$$L(A) = \{w | w \text{ start and end with either } a \text{ or start and end with } b\}$$

Finite Automata

- We are now ready to formally define **computation**:

- Let $\mathrm{M} = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \ldots w_n$ be a string where each $w_i \in \Sigma$. Then **M accepts** w if a sequences of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

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

- We say that **M recognizes language A** if $\mathrm{A} = \{w | M \; accepts \; w\}$.

Finite Automata

A language is called **regular** if some finite automaton recognizes it.

- Consequently, both:

$$L(A) = \left\{ \begin{matrix} w | w \ contains \ at \ least \ one \\ 1 \ and \ an \ even \ numbers \ of \ 0s \ follow \ the \ last \ 1 \end{matrix} \right\}$$

and

$$L(A) = \{w | w \ start \ and \ end \ with \ either \ a \ or \ start \ and \ end \ with \ b\}$$

are regular languages.

Finite Automata

• So far, we have only been given an automaton and then we determine the language that it accepts.

• Now we explore the reverse process, i.e. were are given a regular language and then construct an appropriate FA that accepts this language.

• Consider the problem of building a FA, *M*, that accepts all strings that contain the string 001 as a substring.

The are four essential case considerations here:

1. *M* hasn't seen any of the symbols in the pattern yet,
2. *M* has only just seen a 0
3. *M* has just seen 00, or
4. M has seen the entire pattern.

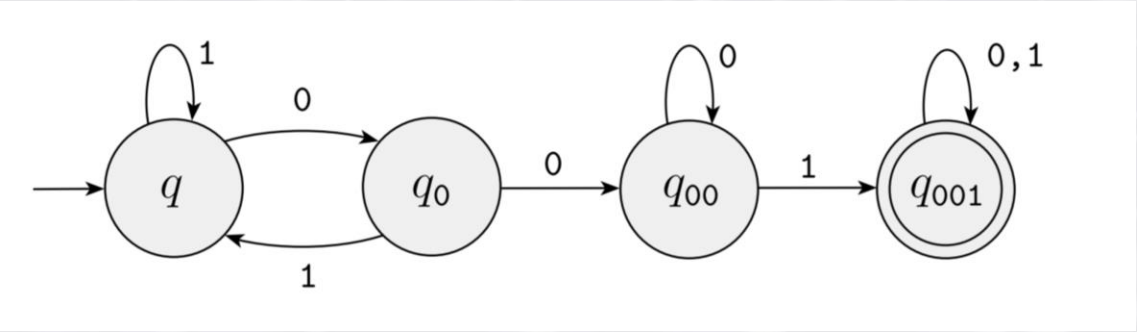
Finite Automata

- Consider the problem of building a FA, *M*, that accepts all strings that contain the string 001 as a substring.

The are four essential case considerations here:

1. *M* hasn't seen any of the symbols in the pattern yet,
2. *M* has only just seen a 0
3. *M* has just seen 00, or
4. M has seen the entire pattern.

Given these observations, we can construct an appropriate FA, M (note that more than one solution is possible – but we have nevertheless constructed the *simplest* such M, in the sense that it contains the least number of states possible).



Finite Automata

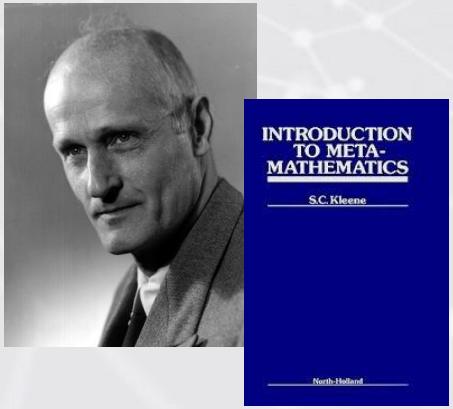
- The **regular operations** consist of **union**, **concatenation** and **star operations**:

- Let A and B be languages. We define the regular operations union, concatenation, and star as follows:

Union: $A \cup B = \{x | x \in A \text{ or } x \in B\}$

Concatenation: $A \circ B = \{xy | x \in A \text{ and } x \in B\}$

(Kleene) Star: $A^* = \{x_1 x_2 \ldots x_k | k \geq 0 \text{ and each } x_i \in A\}$



Finite Automata

- Notice that the *star operation* is a unary operation (unlike union and concatenation which are binary operations); it works by attaching any number of symbols in A together (**including no symbols**).

- For example, consider $\mathrm{A} = \{wild, stallion\}$ and $\mathrm{B} = \{Bill, Ted\}$, then:

$$A \cup B = \{wild, stallion, Bill, Ted\}$$

$A \circ B = \{wildBill, wildTed, stllionBill, stallionTed\}$, and

$$A^* = \begin{Bmatrix} \varepsilon, wild, stallion, wildwild, wildstallion, \\ stallionwild, stallionstallion, \ldots \end{Bmatrix}$$

Finite Automata

- Generally speaking, a collection of objects is **closed under a given operation** if applying that operation to members of the collection returns an object still in the collection.

- For instance, $\mathbb{Z}$ is closed under multiplication, but $\mathbb{Z}$ is not closed under division.

Theorem: The class of regular languages is closed under the union operation.

- Next, we prove this statement.

Finite Automata

Theorem: The class of regular languages is closed under the union operation.

• Proof sketch: We need to show that (in all generality), for any two regular languages A_1 and A_2, $A_1 \cup A_2$ is also regular.

• The proof will be **by construction**, meaning that we will explicitly show how to render a FA, *M*, that recognizes $A_1 \cup A_2$.

• How does this work? We construct *M* from M_1 (the FA for A_1) and M_2 (the FA for A_2). *M* must accept its input exactly when either M_1 or M_2 would accept it in order to recognize $A_1 \cup A_2$.

Bottom line: We construct *M* to simulate M_1 and M_2 *simultaneously*; the key is to consider the states in *M* with respect to the Cartesian product of the states in M_1 and M_2, i.e. $Q = Q_1 \times Q_2$.

Finite Automata

Theorem: The class of regular languages is closed under the union operation.

Pf.

Let M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, and
M_2 recognize A_2, where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.

Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) | \, r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
   This set is the ***Cartesian product*** of sets Q_1 and Q_2 and is written $Q_1 \times Q_2$. It is the set of all pairs of states, the first from Q_1 and the second from Q_2.
2. Σ, the alphabet, is the same as in M_1 and M_2. In this theorem and in all subsequent similar theorems, we assume for simplicity that both M_1 and M_2 have the same input alphabet Σ. The theorem remains true if they have different alphabets, Σ_1 and Σ_2. We would then modify the proof to let $\Sigma = \Sigma_1 \cup \Sigma_2$.
3. δ, the transition function, is defined as follows. For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let

   $$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$

   Hence δ gets a state of M (which actually is a pair of states from M_1 and M_2), together with an input symbol, and returns M's next state.
4. q_0 is the pair (q_1, q_2).
5. F is the set of pairs in which either member is an accept state of M_1 or M_2. We can write it as

   $$F = \{(r_1, r_2) | \, r_1 \in F_1 \text{ or } r_2 \in F_2\}.$$

   This expression is the same as $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. (Note that it is *not* the same as $F = F_1 \times F_2$. What would that give us instead?[3])

(*) In fact, one can show that **regular languages are closed under the union, concatenation, star, and complement operations.** These proofs are much more concise and elegant using the concept of **non-determinism**, which we introduce next.

Non-Determinism

• Until now, we have only considered **deterministic** computations (i.e. computations that are performed sequentially, moving from state to state). In a **non-deterministic** machine, several choices may exist for the next state at any point.

• We will distinguish between deterministic and non-deterministic FA using the notation: **DFA** and **NFA**, respectively. Notice that every DFA is automatically an NFA.

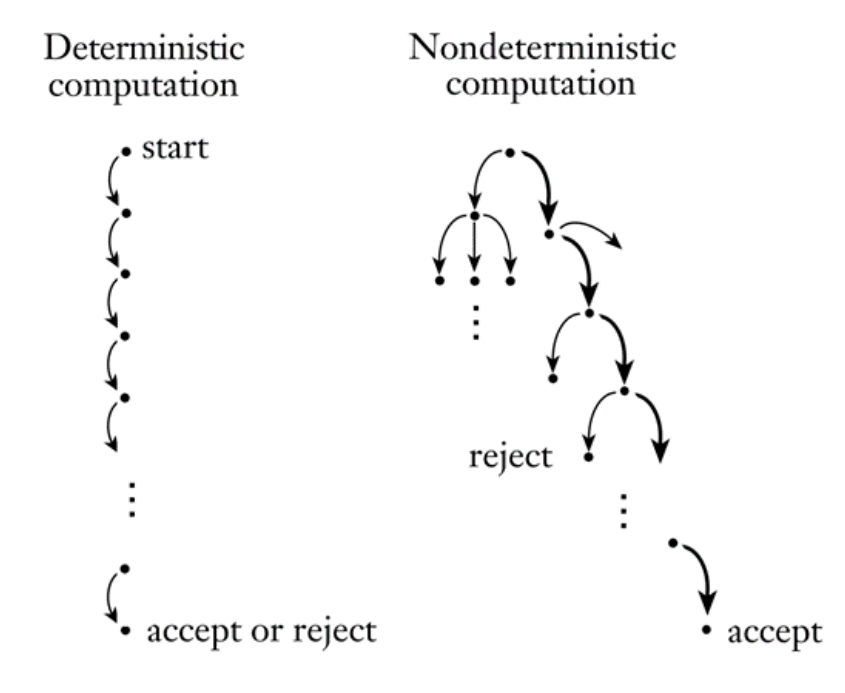
• In an NFA, **a state may have zero, one, or many exiting arrows** for each alphabet symbol (**or** ε).

One can think of the NFA as splitting into multiple copies of itself and following all possibilities in parallel (in this way the NFA runs multiple independent "processes" or "threads" independently).

Non-Determinism

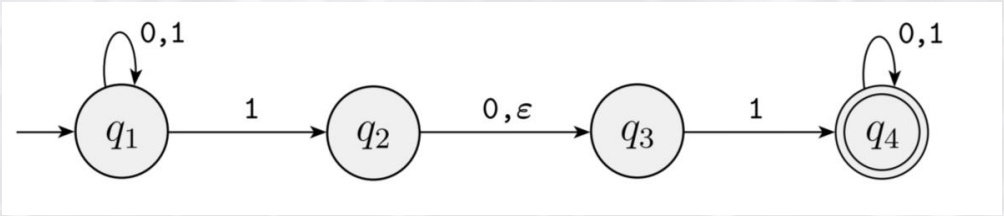
• One can think of the NFA as splitting into multiple copies of itself and following all possibilities in parallel (in this way the NFA runs multiple independent "processes" or "threads" independently).

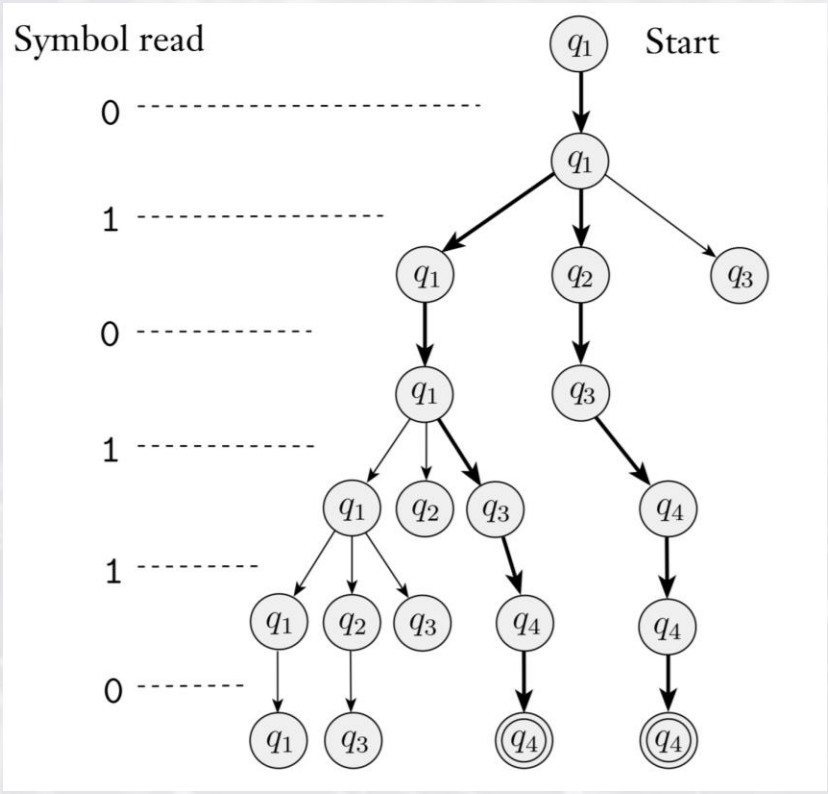
• If **at least one** of these processes accepts, the entire computation accepts.



Non-Determinism

• Below is an example of an NFA and its corresponding computation "tree" for the input 010110 (which is accepted).

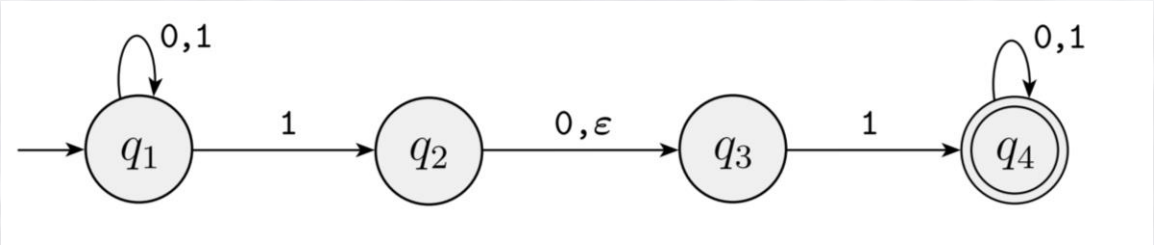




• A few comments: notice that **ε-labeled arrows generate a fork**: one branch corresponds with staying in the state, and the other corresponds with exiting the state along the ε-labeled arrow (note the computation tree at step "010"). Also notice that **threads "die"** when they receive an input symbol but the current state has no corresponding exit arrow.

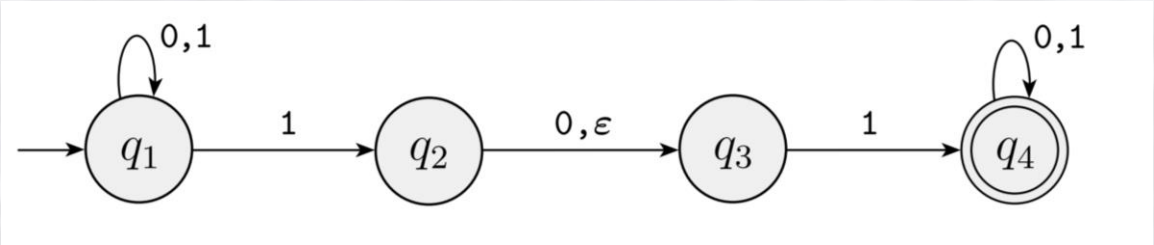
Non-Determinism

Q: Which language is accepted by the previous NFA?



Non-Determinism

Q: Which language is accepted by the previous NFA?



A: Any strings containing 101 or 11 as a substring.

Non-Determinism

• NFA are useful in several respects: As we show, **every NFA can be converted into an equivalent DFA**; NFA are often much simpler and easier to understand than their corresponding DFA.

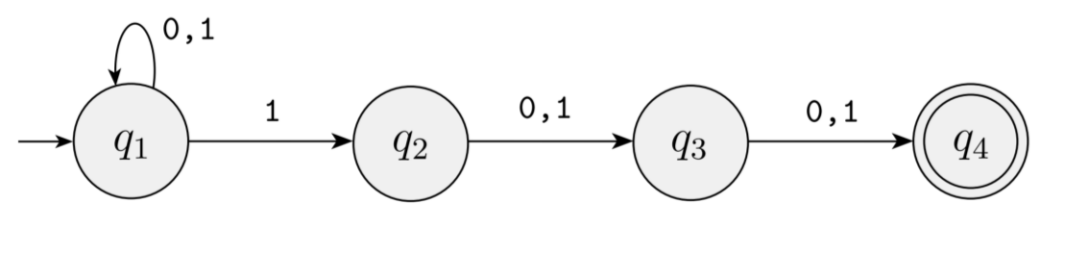
Consider the language A of all strings over $\{0,1\}$ containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not).

Non-Determinism

• NFA are useful in several respects: As we show, **every NFA can be converted into an equivalent DFA**; NFA are often much simpler and easier to understand than their corresponding DFA.

Consider the language A of all strings over $\{0,1\}$ containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not).

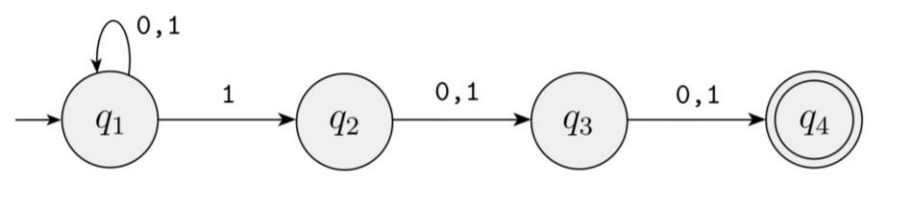
• Here is an NFA recognizing A (notice the elegance of this construction).



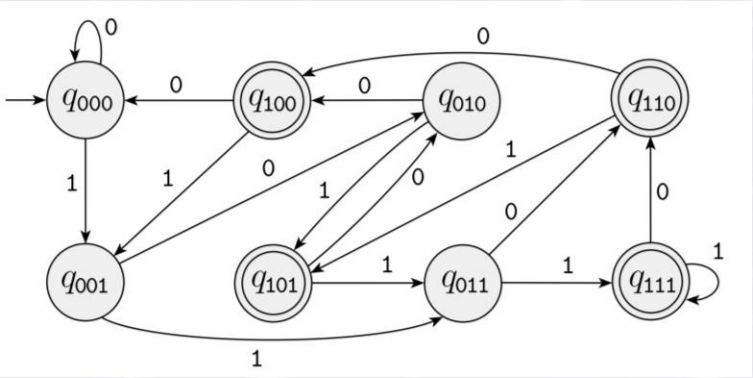
Non-Determinism

Consider the language A of all strings over $\{0,1\}$ containing a 1 in the third position from the end (e.g., 000100 is in A but 0011 is not).

• Here is an NFA recognizing A (notice the elegance of this construction, only 4 states required).



• By contrast, here is the equivalent DFA recognizing A (8 states required).



Non-Determinism

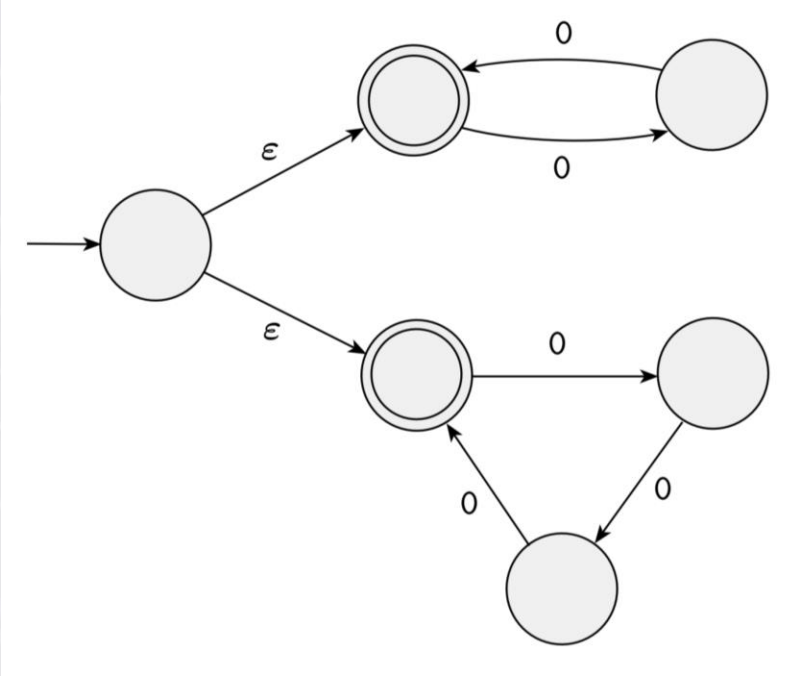
• DFA can be particularly effective for generating languages that admit of a construction using "cases." In these instances, we simply add ε-arrows to handle the different cases.

• For example, consider a DFA that accepts the **unary language** over $\{0\}$ consisting of strings with length $0 \ mod \ 2$ or $0 \ mod \ 3$.

Non-Determinism

• DFA can be particularly effective for generating languages that admit of a construction using "cases." In these instances, we simply add ε-arrows to handle the different cases.

• For example, consider a DFA that accepts the **unary language** over $\{0\}$ consisting of strings with length $0 \ mod \ 2$ or $0 \ mod \ 3$.



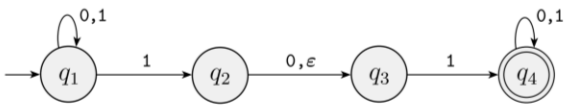
Non-Determinism

• The formal definition of an NFA is very similar to the formal definition of DFA; the two models differ in only one essential way: with an NFA, the transition function maps a state symbol pair to a set $(\delta : Q \times \Sigma_\varepsilon \longrightarrow P(Q))$ to account for non-determinism.

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

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta: Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

• The formal description of our previous NFA example is given by:



The formal description of N_1 is $(Q, \Sigma, \delta, q_1, F)$, where

1. $Q = \{q_1, q_2, q_3, q_4\}$,
2. $\Sigma = \{0,1\}$,
3. δ is given as

	0	1	ε
q_1	$\{q_1\}$	$\{q_1, q_2\}$	$\emptyset$
q_2	$\{q_3\}$	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\{q_4\}$	$\emptyset$
q_4	$\{q_4\}$	$\{q_4\}$	$\emptyset$,

4. q_1 is the start state, and
5. $F = \{q_4\}$.

Non-Determinism

• Let $\mathrm{N} = (Q, \Sigma, \delta, q_0, F)$ and $w = w_1 w_2 \ldots w_n$ be a string where each $w_i \in \Sigma$. Then **N accepts** w if a sequences of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

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

• We say that **N recognizes language A** if $\mathrm{A} = \{w | N \; accepts \; w\}$.

Non-Determinism

Equivalence of NFA and DFA

Theorem. Every non-deterministic finite automaton has an equivalent deterministic finite automaton.

Proof idea: If a language is recognized by an NFA, then we must show the existence of a DFA that also recognizes; to this end, we convert the NFA into an equivalent DFA that simulates the NFA.

Key point: If k is the number of states of the NFA, it has 2^k subsets of states. Thus, the DFA simulating the NFA will have 2^k states. Lastly, we need to be mindful of converting the ε-arrows in the NFA to the DFA.

Non-Determinism

Equivalence of NFA and DFA

Theorem. Every non-deterministic finite automaton has an equivalent deterministic finite automaton.

- For simplicity, consider the case where N contains no ε-arrows.

1. $Q' = \mathcal{P}(Q)$.
Every state of M is a set of states of N. Recall that $\mathcal{P}(Q)$ is the set of subsets of Q.

2. For $R \in Q'$ and $a \in \Sigma$, let $\delta'(R, a) = \{q \in Q | \ q \in \delta(r, a) \text{ for some } r \in R\}$.
If R is a state of M, it is also a set of states of N. When M reads a symbol a in state R, it shows where a takes each state in R. Because each state may go to a set of states, we take the union of all these sets. Another way to write this expression is

$$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).^4$$

3. $q_0' = \{q_0\}$.
M starts in the state corresponding to the collection containing just the start state of N.

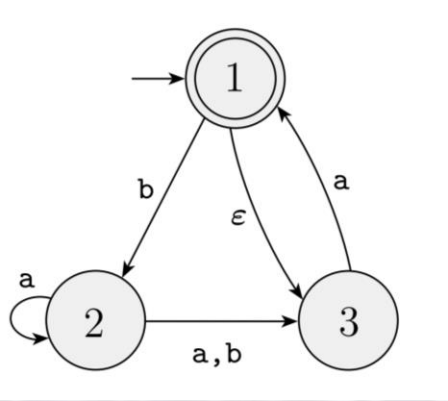
4. $F' = \{R \in Q' | \ R \text{ contains an accept state of } N\}$.
The machine M accepts if one of the possible states that N could be in at this point is an accept state.

*For details on the case where N contains ε-arrows, see Sipser 1.2.

Non-Determinism

Equivalence of NFA and DFA

• We use the previous construction to convert the following NFA into a DFA.

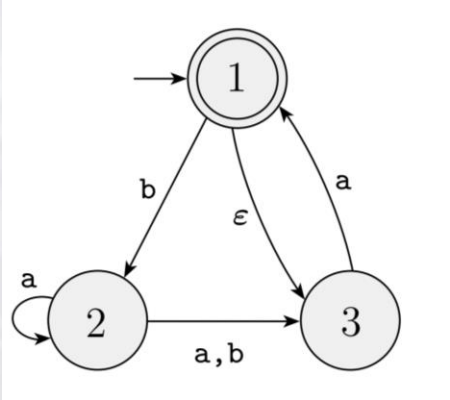


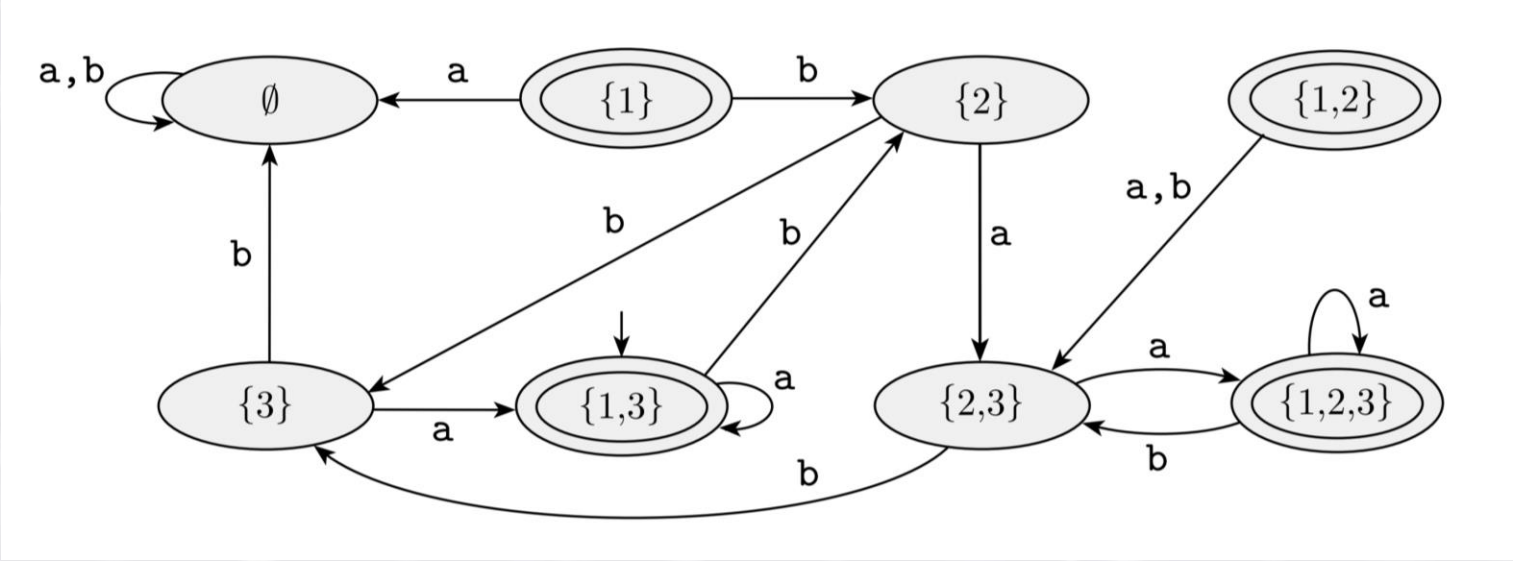
• A few comments: the states of the DFA will consist of the power set of the states of the NFA, namely: $\{\emptyset, \{1\}, \{2\}, \{3\}, \{1,2\}, \{1,3\}, \{2,3\}, \{1,2,3\}\}$. The start state of the DFA needs to account for the fact that the start state of the NFA includes (1) but this state has an ε-arrow to (3); for this reason the start state of the DFA will be labeled {1,3}. Lastly, the accept states of the DFA are those containing the NFA's accept states, i.e., $\{\{1\}, \{1,2\}, \{1,3\}, \{1,2,3\}\}$.

Non-Determinism

Equivalence of NFA and DFA

- We use the previous construction to convert the following NFA into a DFA.





Non-Determinism

• From the theorem that every finite automaton has an equivalent deterministic finite automaton, it follows that:

Corollary. A language is regular *iff* some non-deterministic finite automaton recognizes it.

Why?

Non-Determinism

• From the theorem that every finite automaton has an equivalent deterministic finite automaton, it follows that:

Corollary. A language is regular *iff* some non-deterministic finite automaton recognizes it.

Why?

A: (→)Suppose we have a regular language, then there is a corresponding DFA, and by the theorem: NFA →DFA.

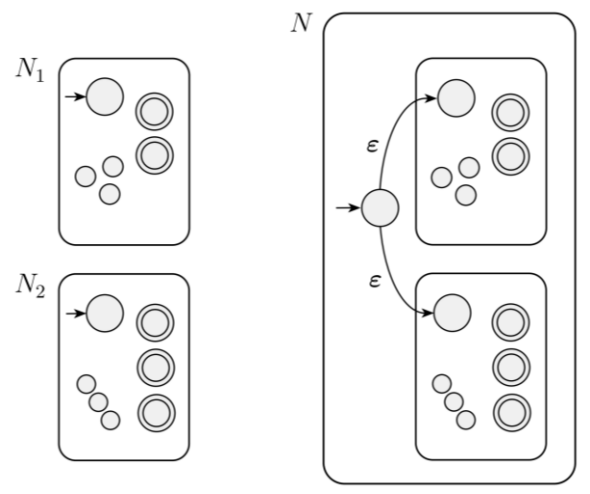
(←) Conversely, suppose we have an DFA; this is automatically an NFA, implying the corresponding accepted language is regular.

Non-Determinism

Theorem. The class of regular languages is closed under the union operation.

Basic idea of proof: We have regular languages A_1 and A_2 and want to prove that $A_1 \cup A_2$ is regular. The idea is to take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into one new NFA, N.

We achieve this by constructing a start state from which two ε-arrows emanate; each branch corresponds with one of N_1 and N_2 respectively.



Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.

Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2, with the addition of a new start state q_0.
2. The state q_0 is the start state of N.
3. The set of accept states $F = F_1 \cup F_2$.
   The accept states of N are all the accept states of N_1 and N_2. That way, N accepts if either N_1 accepts or N_2 accepts.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

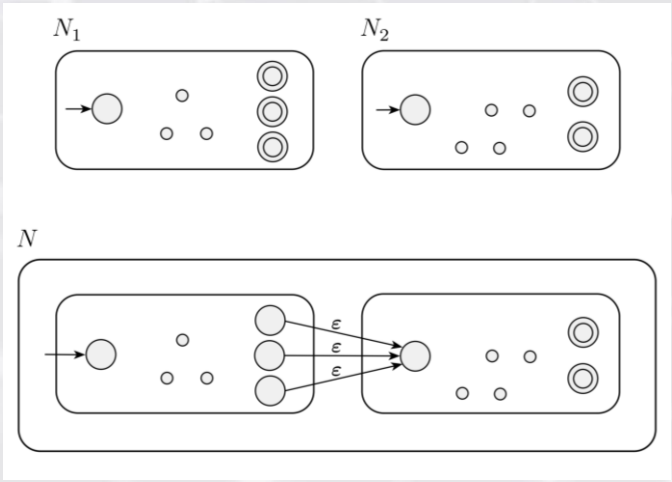
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

Non-Determinism

Theorem. The class of regular languages is closed under the concatenation operation.

Basic idea of proof: We have regular languages A_1 and A_2 and want to prove that $A_1 \circ A_2$ is regular. The idea, again, is to take two NFAs, N_1 and N_2 for A_1 and A_2, and combine them into one new NFA, N.

We achieve this by constructing ε-arrows between the accept states of N_1 and the start states of N_2; the accept states of N are now designated as the accept states of N_2.



Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.

Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2.
2. The state q_1 is the same as the start state of N_1.
3. The accept states F_2 are the same as the accept states of N_2.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

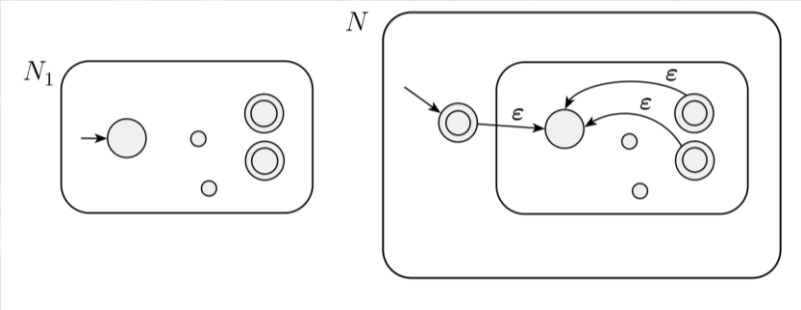
$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \varepsilon \\ \delta_2(q,a) & q \in Q_2. \end{cases}$$

Non-Determinism

Theorem. The class of regular languages is closed under the star operation.

Basic idea of proof: We have regular languages A_1 and want to prove that A_1* also is regular. We take an NFA N_1 recognizing A_1 and modify it to recognize A_1*.

We achieve this by constructing ε-arrows returning to the start state from accept states. Notice that **it is necessary to add a new start state** with an ε-arrow connecting it to the previous start state (why?).



PROOF Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1.
Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^*.

1. $Q = \{q_0\} \cup Q_1$.
   The states of N are the states of N_1 plus a new start state.
2. The state q_0 is the new start state.
3. $F = \{q_0\} \cup F_1$.
   The accept states are the old accept states plus the new start state.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

Regular Expressions

• **Regular expressions** (regex) are sequences of characters that define a search pattern; they have a special role in computer science applications and are commonly used in text editor and search engines.

• The concept is due to Kleene and came into prominent use with the introduction of the *UNIX* OS.

• The formal (inductive) definition of a regular expression is given by:

Say that R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

*Note: Do not confuse the regexs ε and ∅. ε represents the language containing a single string (namely, the empty string), whereas ∅ denotes the empty language.

Regular Expressions

Say that R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

• In simple terms, a regex is a set of strings built from the three regular operations, union, concatenation and star; in addition, they use the four special symbols: + * ( ); precedence order: star, concatenation and then union.

• We denote a *regular expression* R and the *language* that it describes as L(R).

Regular Expressions

•In the following examples, we assume that the alphabet is $\Sigma = \{0,1\}$.

1. $0^*10^* = \{w \mid w \text{ contains a single } 1\}$.
2. $\Sigma^*1\Sigma^* = \{w \mid w \text{ has at least one } 1\}$.
3. $\Sigma^*001\Sigma^* = \{w \mid w \text{ contains the string } 001 \text{ as a substring}\}$.
4. $1^*(01^+)^* = \{w \mid \text{every } 0 \text{ in } w \text{ is followed by at least one } 1\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w \text{ is a string of even length}\}$.[5]
6. $(\Sigma\Sigma\Sigma)^* = \{w \mid \text{the length of } w \text{ is a multiple of } 3\}$.
7. $01 \cup 10 = \{01, 10\}$.
8. $0\Sigma^*0 \cup 1\Sigma^*1 \cup 0 \cup 1 = \{w \mid w \text{ starts and ends with the same symbol}\}$.
9. $(0 \cup \varepsilon)1^* = 01^* \cup 1^*$.
   The expression $0 \cup \varepsilon$ describes the language $\{0, \varepsilon\}$, so the concatenation operation adds either 0 or ε before every string in 1^*.
10. $(0 \cup \varepsilon)(1 \cup \varepsilon) = \{\varepsilon, 0, 1, 01\}$.
11. $1^*\emptyset = \emptyset$.
    Concatenating the empty set to any set yields the empty set.
12. $\emptyset^* = \{\varepsilon\}$.
    The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

Regular Expressions

- Recall that a regular language is one that is recognized by some finite automaton.

Theorem. A language is regular *iff* some regular expression describes it.

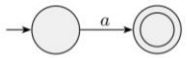
Pf. First, we show that if a language is described by a regular expression, then it is regular. To this end, suppose we have a regular expression R describing some language A; now we show how to convert R into an NFA recognizing A.

Say that R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

PROOF Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.

1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.



Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here; but an NFA is all we need for now, and it is easier to describe.

Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.

Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where $\delta(r, b) = \emptyset$ for any r and b.

3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.

Formally, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$ for any r and b.

4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

For the last three cases, we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. In other words, we construct the NFA for R from the NFAs for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.

Regular Expressions

- Recall that a regular language is one that is recognized by some finite automaton.

Theorem. A language is regular *iff* some regular expression describes it.

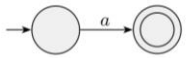
Pf. First, we show that if a language is described by a regular expression, then it is regular. To this end, suppose we have a regular expression R describing some language A; now we show how to convert R into an NFA recognizing A.

Say that R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

PROOF Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.

1. $R = a$ for some $a \in \Sigma$. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.



Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here; but an NFA is all we need for now, and it is easier to describe.

Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.

2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.

Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where $\delta(r, b) = \emptyset$ for any r and b.

3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.

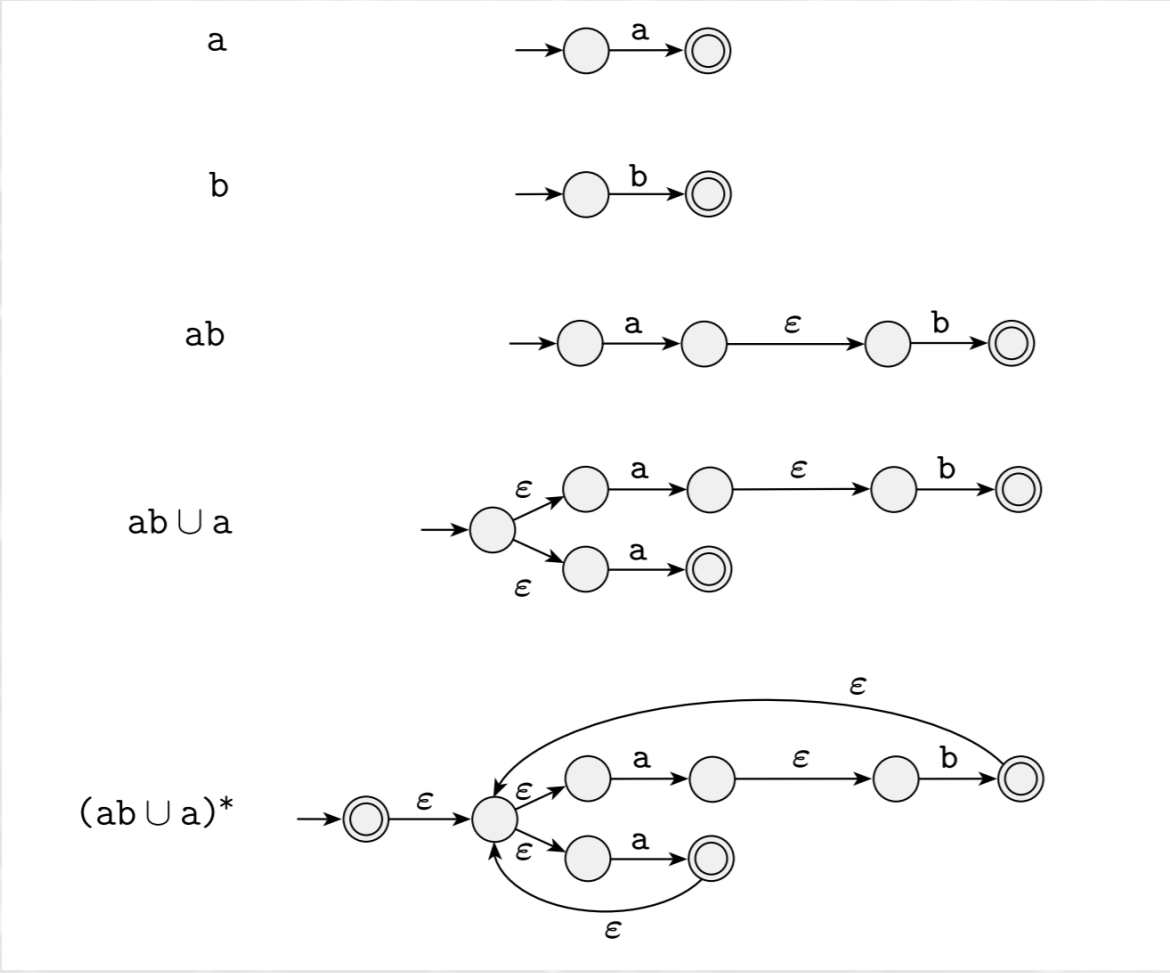
Formally, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$ for any r and b.

4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

For the last three cases, we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. In other words, we construct the NFA for R from the NFAs for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.

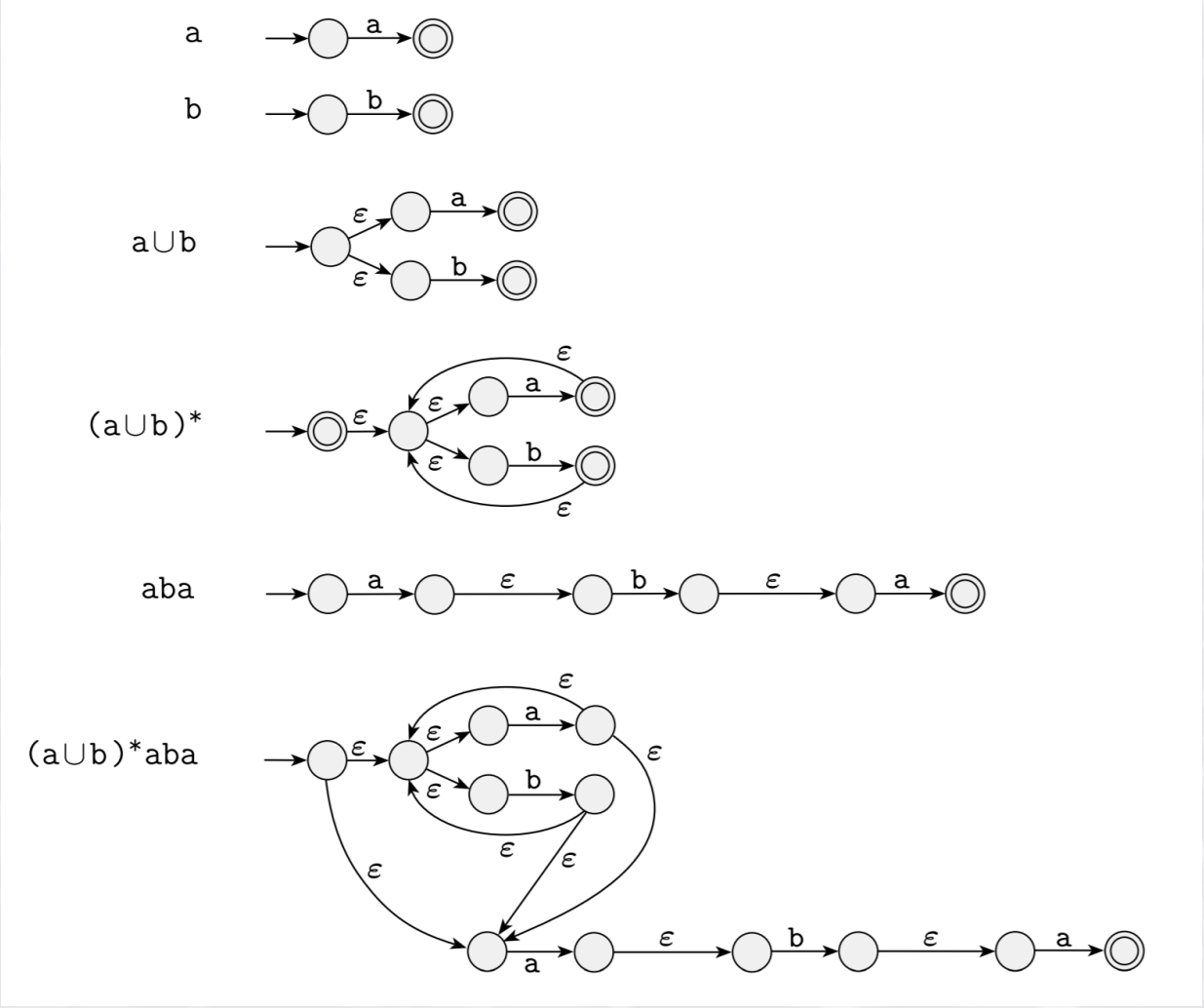
Regular Expressions

Example. We convert the regular expression $(ab \cup a)^*$ to an NFA.



Regular Expressions

Example. We convert the regular expression $(a \cup b)^* aba$ to an NFA.



Regular Expressions

Theorem. A language is regular *iff* some regular expression describes it.

• Recall that we have already proven the implication: (1) if a language is described by a regular expression, then it is regular.

• It still remains to prove the second necessary implication for the theorem, namely: (2) if a language is regular, then it is described by a regular expression.

We prove (2) in two steps:

(a) We first define a new type of finite automaton called a **generalized nondeterministic finite automaton** (GNFA); it can be shown (we omit the proof for brevity) that any DFA can be converted into a GNFA.

(b) Next, we show that any GNFA can be converted into a regular expression.

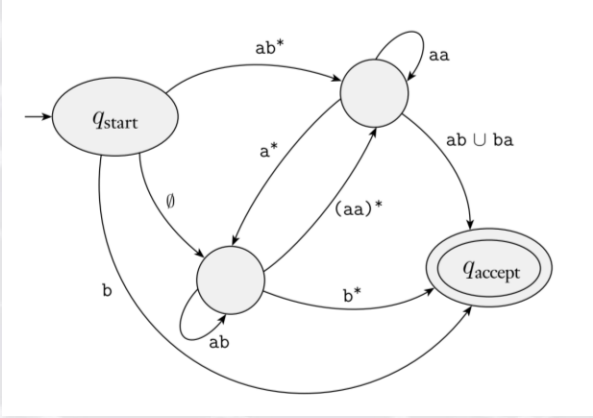
Together (1) and (2) prove the required theorem.

Regular Expressions

(a) We first define a new type of finite automaton called a **generalized nondeterministic finite automaton** (GNFA); it can be shown (we omit the proof for brevity) that any DFA can be converted into a GNFA.

• A GNFA is simply an NFA wherein the transition arrows may have regular expression as labels; in this way the GNFA can read blocks of symbols at a time.

• Here is an example of a GNFA.



• The formal definition of a GNFA is given by:

A ***generalized nondeterministic finite automaton*** is a 5-tuple, $(Q, \Sigma, \delta, q_{start}, q_{accept})$, where

1. Q is the finite set of states,
2. Σ is the input alphabet,
3. $\delta: (Q - \{q_{accept}\}) \times (Q - \{q_{start}\}) \longrightarrow \mathcal{R}$ is the transition function,
4. q_{start} is the start state, and
5. q_{accept} is the accept state.

Regular Expressions

A ***generalized nondeterministic finite automaton*** is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

1. Q is the finite set of states,
2. Σ is the input alphabet,
3. $\delta: (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \longrightarrow \mathcal{R}$ is the transition function,
4. q_{start} is the start state, and
5. q_{accept} is the accept state.

• A GNFA **accepts a string w** in Σ^* if $w = w_1 w_2 \ldots w_n$ where each $w_i \in \Sigma^*$ and a sequence of states $q_0, q_1, \ldots, q_n$ exists such that:

1. $q_0 = q_{start}$ is the start state,
2. $q_n = q_{accept}$ is the accept state, and
3. for each I, we have $w_i \in \mathrm{L}(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; in other words, R_i is the expression on the arrow from q_{i-1} to q_i.

Regular Expressions

(b) Next, we claim that any GNFA can be converted into a regular expression.

This claim follows naturally from the constructive procedure: DFA ⟶ GNFA (use *induction* on the number of states).

- Putting this together, we have:

  (a) Any DFA can be converted into a GNFA.

  (b) Any GNFA can be converted into a regular expression.

(a) and (b) prove the required theorem:

Theorem. A language is regular *iff* some regular expression describes it.

Non-Regular Languages

• Summarizing our major results to date, we have shown the equivalences:

DFA↔NFA, Regular Language ↔ REGEX

• Recall that finite automata (FA) possess severe memory limitations (think of the finite states as imposing an explicit memory restriction). These memory limits mean that there are some languages that FA cannot recognize; these are called **non-regular languages**.

• A classic example of a non-regular (and yet simple) language is given by:

$$B = \{0^n 1^n | n \geq 0\}$$

Careful: although it is tempting to think that the reason that B is non-regular is because the number, say of zeros, is unlimited – the reason is in fact more subtle.

Notice, for instance that while B is non-regular, C, defined below is in fact regular!

$$C = \{w | w \; has \; an \; equal \; number \; of \; occurences \; of \; 01 \; and \; 10 \; substrings\}$$

Non-Regular Languages

• Summarizing our major results to date, we have shown the equivalences:

DFA↔NFA, Regular Language ↔ REGEX

• Recall that finite automata (FA) possess severe memory limitations (think of the finite states as imposing an explicit memory restriction). These memory limits mean that there are some languages that FA cannot recognize; these are called **non-regular languages**.

• A classic example of a non-regular (and yet simple) language is given by:

$$B = \{0^n 1^n | n \geq 0\}$$

Careful: although it is tempting to think that the reason that B is non-regular is because the number, say of zeros, is unlimited – the reason is in fact more subtle.

Notice, for instance that while B is non-regular, C, defined below is in fact regular!

$$C = \{w | w \; has \; an \; equal \; number \; of \; occurences \; of \; 01 \; and \; 10 \; substrings\}$$

Non-Regular Languages

• Thus we see that determining whether a language is regular or non-regular is in general not trivial. Our key tool for determining this distinction is the so-called **pumping lemma** for regular languages, which we subsequently prove.

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Recall the notation where $|s|$ represents the length of string s, y^i means that i copies of y are concatenated together, and y^0 equals ε.

When s is divided into xyz, either x or z may be ε, but condition 2 says that $y \neq \varepsilon$. Observe that without condition 2 the theorem would be trivially true. Condition 3 states that the pieces x and y together have length at most p. It is an extra technical condition that we occasionally find useful when proving certain languages to be nonregular.

Non-Regular Languages

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

• The pumping lemma provides us with a practical tool to show that a language is non-regular (notice the "negative use" of the lemma).

• Thus, to show that a A is non-regular, we must provide a string s in A of length at least p (choose an arbitrary, fixed p) where the conditions of the pumping lemma fail; because there exists a string in A for which the lemma fails, this proves that the language is non-regular (since the lemma makes a claim about all strings in A of length at least p).

Non-Regular Languages

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^i z \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

• First, let's get an intuitive sense of the statement of the lemma, and why it holds for regular languages.

Proof Sketch: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing A. We assign the pumping length p to be the number of state in M.

• We show that any string s of length at least p can be broken into three pieces $s = xyz$, satisfying the (3) conditions of the lemma. (If no strings in A are of length at least p then the lemma is *vacuously true*, why?)

Non-Regular Languages

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

• First, let's get an intuitive sense of the statement of the lemma, and why it holds for regular languages.

Proof Sketch: Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing A. We assign the pumping length p to be the number of state in M.

• We show that any string s of length at least p can be broken into three pieces $s = xyz$, satisfying the (3) conditions of the lemma. (If no strings in A are of length at least p then the lemma is *vacuously true*, why?)

• Suppose that the sequence of states M executes for s is given by $q_1, q_2, \ldots, q_n$, where q_1 is a start state for M and q_n is an accept state. Since $|s| > p$, it stands to reason that this sequence must contain a repeated state, why?

Non-Regular Languages

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Proof Sketch: Let $\mathrm{M} = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing A. We assign the pumping length p to be the number of state in M.

• We show that any string s of length at least p can be broken into three pieces $s = xyz$, satisfying the (3) conditions of the lemma.

• Suppose that the sequence of states M executes for s is given by $q_1, q_2, \ldots, q_n$, where q_1 is a start state for M and q_n is an accept state. Since $|s| > \mathrm{p}$, it stands to reason that this sequence must contain a repeated state, by the **pigeonhole principle**.

• WLOG (*without loss of generality*) call the **repeated state** q_k with $1 \leq k \leq n$, so that the sequence of states M executes for s is:

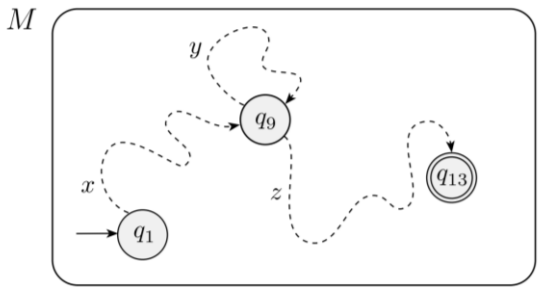
In particular, we define $s = xyz$ so that y corresponds with the execution "loop" of states: $q_k \longrightarrow q_k$.

$$\underbrace{q_1, q_2, \ldots}_{x}, \underbrace{q_k, \ldots, q_k}_{y}, \ldots, q_n$$

Non-Regular Languages

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.



Proof Sketch: Let $\mathrm{M} = (Q, \Sigma, \delta, q_0, F)$ be a DFA recognizing A. We assign the pumping length p to be the number of state in M.

• We show that any string s of length at least p can be broken into three pieces $s = xyz$, satisfying the (3) conditions of the lemma.

$$\underbrace{q_1, q_2, \ldots}_{x}, \underbrace{q_k, \ldots, q_k}_{y}, \ldots, \underset{z}{q_n}$$

• From this construction, it is not hard to see that condition (1) is met, namely: $\mathrm{x}y^i\mathrm{z} \in A \ \forall i \geq 0$. Furthermore, condition (2) holds, $|y| > 0$, as it was the part of s that occurred between two different occurrences of state q_k.

Lastly, by the pigeonhole principle, the first p+1 states in the sequence must contain a repetition, therefore $|xy| \leq p$.

Non-Regular Languages

• To prove a language is not regular, we first assume that it is, and then follow the method of proof by contradiction. Concretely, we construct a string s in the language, and then show that s cannot be "pumped", by way of the pumping lemma, i.e. show one of the (3) conditions fails.

Non-Regular Languages

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Example. Returning to our previous example, show that the language: $B = \{0^n1^n | n \geq 0\}$ is not regular.

Pf. We assume that B is regular and derive a contradiction using the pumping lemma.

Let $s = 0^p1^p$, where p is the pumping length; notice that s is in B and that $|s| >$p, so the pumping lemma applies, which states that $s = xyz$, satisfying the (3) conditions. Consider 3 possible cases, each one resulting in a contradiction.

(1) The string y contains only 0s. Then $xyyz$ has more 0s than 1s, so $xyyz$ is not a member of B. This violates condition (2) of the pumping lemma – a contradiction.

(2) The string y contains only 1s, also a contradiction (why?).

(3) The string y contains both 0s and 1s; notice in this case $xyyz$ will have the same number of 0s and 1s but they will be out of order, so $xyyz$ is not a member of B – a contradiction.

Non-Regular Languages

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Example.

Let $F = \{ww \mid w \in \{0,1\}^*\}$. We show that F is nonregular, using the pumping lemma.

Assume to the contrary that F is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 0^p10^p1. Because s is a member of F and s has length more than p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, satisfying the three conditions of the lemma. We show that this outcome is impossible.

Condition 3 is once again crucial because without it we could pump s if we let x and z be the empty string. With condition 3 the proof follows because y must consist only of 0s, so $xyyz \notin F$.

Observe that we chose $s = 0^p10^p1$ to be a string that exhibits the "essence" of the nonregularity of F, as opposed to, say, the string 0^p0^p. Even though 0^p0^p is a member of F, it fails to demonstrate a contradiction because it can be pumped.

Non-Regular Languages

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

Example.

Here we demonstrate a nonregular unary language. Let $D = \{1^{n^2} \mid n \geq 0\}$. In other words, D contains all strings of 1s whose length is a perfect square. We use the pumping lemma to prove that D is not regular. The proof is by contradiction.

Assume to the contrary that D is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 1^{p^2}. Because s is a member of D and s has length at least p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in D. As in the preceding examples, we show that this outcome is impossible. Doing so in this case requires a little thought about the sequence of perfect squares:

$$0, 1, 4, 9, 16, 25, 36, 49, \ldots$$

Note the growing gap between successive members of this sequence. Large members of this sequence cannot be near each other.

Now consider the two strings xyz and xy^2z. These strings differ from each other by a single repetition of y, and consequently their lengths differ by the length of y. By condition 3 of the pumping lemma, $|xy| \leq p$ and thus $|y| \leq p$. We have $|xyz| = p^2$ and so $|xy^2z| \leq p^2 + p$. But $p^2 + p < p^2 + 2p + 1 = (p+1)^2$. Moreover, condition 2 implies that y is not the empty string and so $|xy^2z| > p^2$. Therefore, the length of xy^2z lies strictly between the consecutive perfect squares p^2 and $(p+1)^2$. Hence this length cannot be a perfect square itself. So we arrive at the contradiction $xy^2z \notin D$ and conclude that D is not regular. ■

Fin