




# Finite Automata




Korea Institute of Energy Technology

## Computability Theory



- What problems can we solve with a computer?



Alan Turing (1912 – 1954), *the Father of Computer Science*, was an English mathematician, computer scientist, logician, cryptanalyst, philosopher, and theoretical biologist. Turing was highly influential in the development of theoretical computer science, providing a formalization of the concepts of algorithm and computation with the Turing machine.

We now know that computers can do many things. Before Turing, computers are for simple computations. Researchers wondered the capability and limit of computers.

## Two Challenges



- Computers are dramatically better now than they've ever been, and that trend continues
- Writing proofs on formal definitions is hard, and computers are way more complicated than numbers, sets, graphs, or functions
- Key Question: How can we prove what computers can and can't do...
  - So that our results are still true in 20 years?
  - Without multi-hundred page proofs?

## Automata



- An automaton is a mathematical model of a computing device
  - Automata: Plural of Automaton
- It's an **abstraction** of a real computer, the way that graphs are abstractions of social networks, electrical grids, etc.
 

CS is very abstract. Try to materialize concepts for better understanding of abstract concepts
- The automata are
  - Powerful enough to capture huge classes of computing devices
  - Simple enough that we can reason about them in a small space
  - Fascinating and useful in their own rights

## Computing with Finite Memory



Data stored electronically (DRAM)  
Algorithm is in CPU  
Memory limited by display



Data stored in Beads  
Algorithm is in brain  
Memory limited by beads

Model **“Memory”** and **“Algorithms”** when they take on many different forms

## Common Characteristics



- These devices (machines) receive input from an external source
  - Input is provided sequentially, one discrete unit at a time
- Each input causes the device to change configuration
  - Computation is done through configuration changes
- Once all input is provided, we can read off an answer based on the configuration of the device



## Example



- $52 + 5$

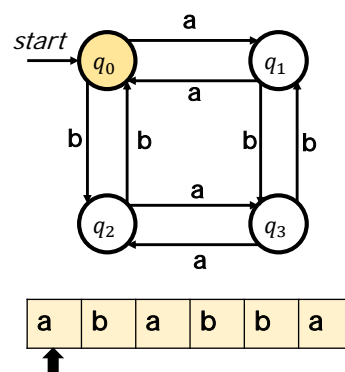


Do the same thing with the abacus

## Modeling Computation



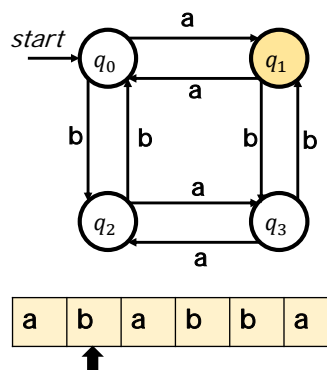
- We will model a finite memory computer as a collection of states linked by transitions
- Each state corresponds to one possible *configuration* of the device's memory
- Each transition indicates how memory changes in response to inputs
- Some state is designated as the start state
  - The computation begins in the start state



## Modeling Computation



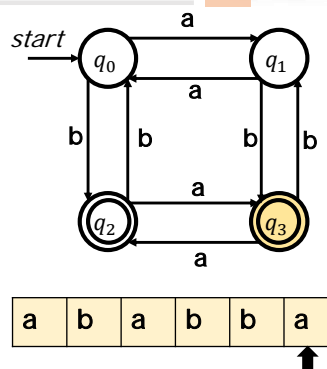
- This device processes strings made of characters
  - Each character represents some external input to the device
  - The string represents the full sequence of inputs to the device
- To run this device, we begin in our start state and scan the input from left to right
- Each time the machine sees a character, it changes state by following the transition labeled with that character



## Modeling Computation



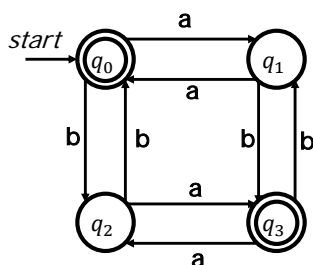
- Once we've finished entering all the input characters, we need to obtain the result of the computation
- Pedagogically, we'll assume that we just need to get a single bit of output
  - That is, the machines will just return YES or NO.
- Some of the states in our computational device will be marked as accepting states
  - Highlighted with a double ring.



# Modeling Computation



- If the machine ends in an accepting state after seeing all the input, accepts the input (says YES)
- If the machine does not end in an accepting state after seeing all the input, it rejects the input (says NO).



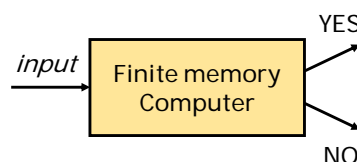
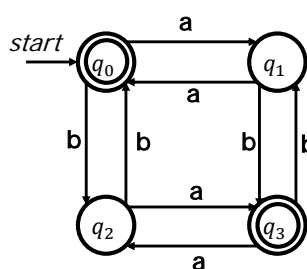
Which of these strings does this machine accept?

aab  
aabb  
abbababba

# Modeling Computation



- This type of computational machine is called a **finite automaton** (plural: finite automata)
- Finite automata model computers where (1) **memory is finite** and (2) the computation produces as **YES/NO** answer
- In other words, finite automata model **predicates**, and do so with a fixed, finite amount of memory.



## Strings



- An **alphabet** is a finite, nonempty set of symbols called **characters**
- Typically, we use the symbol  $\Sigma$  to refer to an alphabet
- A **string** over an alphabet  $\Sigma$  is a **finite** sequence of characters drawn from  $\Sigma$ 
  - Example: Let  $\Sigma = \{a, b\}$
  - Here are some strings over  $\Sigma$ :
    - a
    - aabaaabbabaaabaaaabbb
    - abbababba
- The empty string has no characters and is denoted  $\epsilon$

## Language



- A **formal language** is a **set of strings**
- $L$  is a language over  $\Sigma$  if it is a set of strings over  $\Sigma$ 
  - Example: The language of palindromes over  $\Sigma = \{a, b, c\}$  is the set
  - $\{\epsilon, a, b, c, aa, bb, cc, aaa, aba, aca, bab, \dots\}$
- The set of all strings composed from letters in  $\Sigma$  is denoted  $\Sigma^*$
- Formally, we say that  $L$  is a language over  $\Sigma$  if  $L \subseteq \Sigma^*$ .

## Symbols



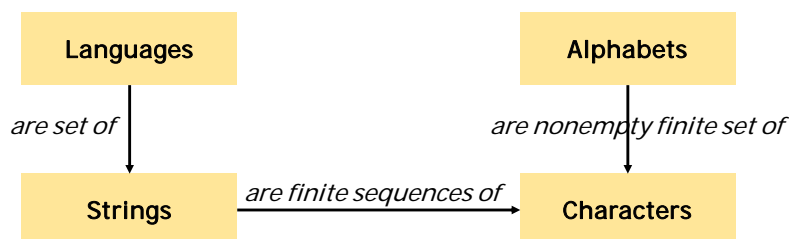
Symbol	
$\epsilon$	Empty string
$\Sigma$	An alphabet
$\Sigma^*$	All strings that can be made from characters in $\Sigma$

- Note:  $\epsilon \in \Sigma^*$ , but  $\epsilon \notin \Sigma$ .

## Relations



- Languages are sets of strings
- Strings are finite sequences of characters
- Characters are individual symbols
- Alphabets are sets of characters.

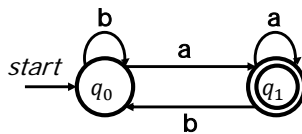




## FA and Languages



- Let  $A$  be an automaton that processes strings drawn from an alphabet  $\Sigma$
- The language of  $A$ , denoted  $\mathcal{L}(A)$ , is the set of strings over  $\Sigma$  that  $A$  accepts:
- $\mathcal{L}(A) = \{ w \in \Sigma^* \mid A \text{ accepts } w \}$

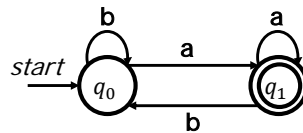


## FA and Languages



- A automaton  $D$  processes strings over  $\{a, b\}$
- Notice that  $D$  accepts all strings of  $a$ 's and  $b$ 's that end in  $a$  and rejects everything else

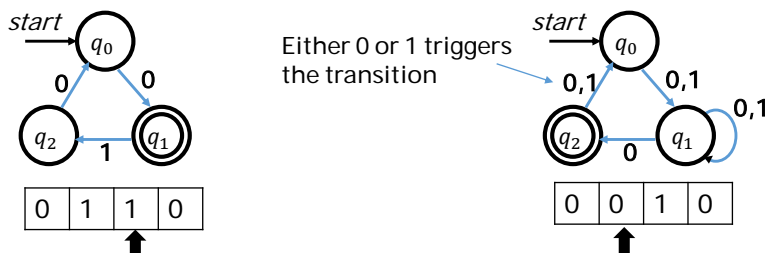
$$\rightarrow \mathcal{L}(D) = \{ w \in \{a, b\}^* \mid w \text{ ends in } a \}$$



$$\mathcal{L}(A) = \{ w \in \Sigma^* \mid A \text{ accepts } w \}$$

Passwords should contains special characters, numbers and alphabets  
Write a program that checks given strings are PW or not (With a single parsing)

## Formalism



- In order to reason about the limits of what finite automata can and cannot do, we need to formally specify their behavior in all cases
- All of the following need to be defined or disallowed
  - What happens if there is no transition out of a state on some input?
  - What happens if there are multiple transitions out of a state on some input?

## DFA



- DFA (Deterministic Finite Automaton)
- DFAs are the simplest type of automaton
- A DFA is defined relative to some alphabet  $\Sigma$
- For each state in the DFA, there must be exactly one transition defined for each symbol in  $\Sigma$ 
  - “Deterministic”
  - A transition function that maps (state, character) ordered pairs to states
  - For each state in the DFA, there must be exactly one transition defined for each symbol in  $\Sigma$
- There is a unique start state
- There are zero or more accepting states.

## Designing DFAs

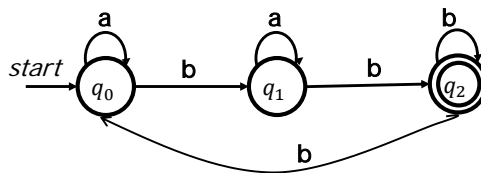


- At each point in its execution, the DFA can only remember what state it is in
- DFA Design Tip: Build each state to correspond to some piece of information you need to remember
  - Each state acts as a “memento” of what you're supposed to do next
- Only finitely many different states means only finitely many different things the machine can remember
  - *Finite memory*

## Recognizing Languages with DFAs

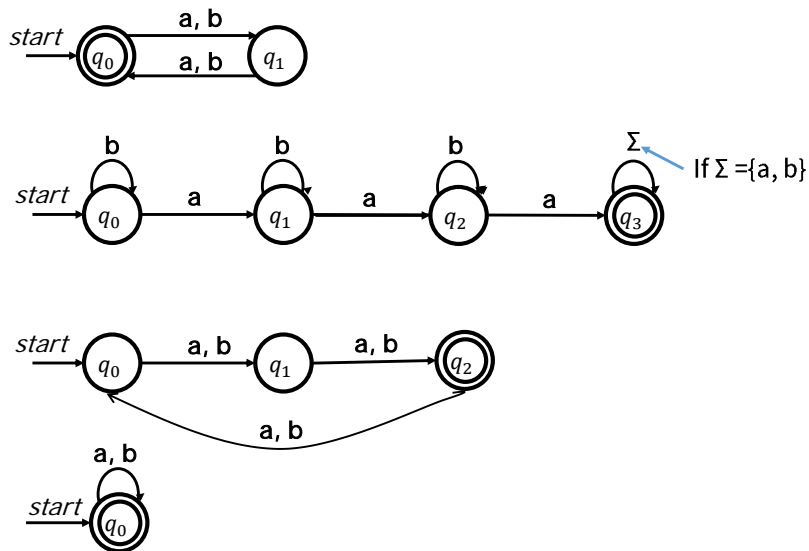


- $L = \{ w \in \{a, b\}^* \mid \text{the number of } b\text{'s in } w \text{ is congruent to two modulo three} \}$ 
  - For example, “bb”, “bab”, “bbababab” are elements of L



Each state remembers the remainder of the number of **b**s seen so far modulo three

# Finite Automata and Languages



Korea Institute of Energy Technology 2022-03-11

23

## DFA Design



Design a DFA for a language

$L = \{ w \in \{a, b\}^* \mid w \text{ contains } aa \text{ as a substring} \}$

Korea Institute of Energy Technology 2022-03-11

24

## DFA Design



Design a DFA for a language

$L = \{ w \in \{a, *, /\}^* \mid w \text{ represents a C-style comment} \}$

A can be any alphanumeric other than \* and /

C-style comment

`/* ... */`

Accepted:

`/*a*/`

`/**/`

`/***/`

`/*aaa*aaa*/`

`/*a/a*/`

Rejected:

`/**`

`/**/a/*aa*/`

`aaa/**/aa`

`/*/`

`/**a/`

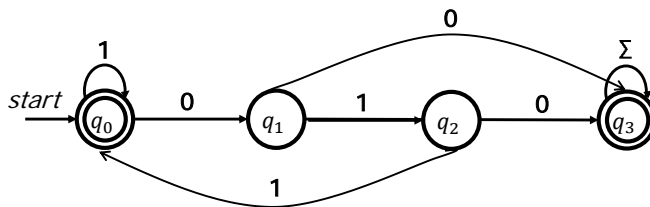
## DFA Design



Design a DFA for a language

$L = \{ w \in \{a, *, /\}^* \mid w \text{ represents a C-style comment} \}$

## Tabular DFAs



	0	1
* <i>q</i> <sub>0</sub>	<i>q</i> <sub>1</sub>	<i>q</i> <sub>0</sub>
<i>q</i> <sub>1</sub>	<i>q</i> <sub>3</sub>	<i>q</i> <sub>2</sub>
<i>q</i> <sub>2</sub>	<i>q</i> <sub>3</sub>	<i>q</i> <sub>0</sub>
* <i>q</i> <sub>3</sub>	<i>q</i> <sub>3</sub>	<i>q</i> <sub>3</sub>

The first row is the start state

## DFA as a C Program



```

int kTransitionTable[kNumStates][kNumSymbols] = {
    {0, 0, 1, 3, 7, 1, ...},
    ...
};

bool kAcceptTable[kNumStates] = {
    false, true, true,
    ...
};

bool SimulateDFA(string input) {
    int state = 0;
    for (char ch: input) {
        state = kTransitionTable[state][ch];
    }
    return kAcceptTable[state];
}

```

## Language & DFA

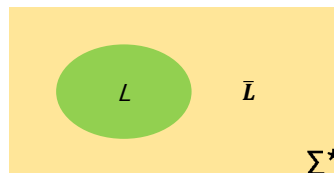


- A language  $L$  is called a *regular language (RL)* if there exists a DFA  $D$  such that  $\mathcal{L}(D) = L$ .
- If  $L$  is a language and  $\mathcal{L}(D) = L$ , we say that  $D$  *recognizes* the language  $L$

## Complement of a Language



- Given a language  $L \subseteq \Sigma^*$ , the complement of that language (denoted  $\bar{L}$ ) is the language of all strings in  $\Sigma^*$  that aren't in  $L$ .
- Formally:  $\bar{L} = \Sigma^* - L$



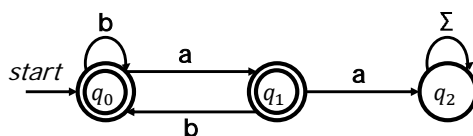
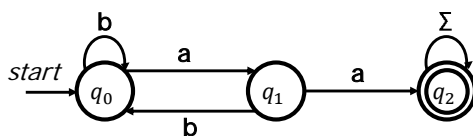
Theorem: For any language  $L$ ,  $\bar{\bar{L}} = L$

## Complementing Regular Languages



$L = \{ w \in \{a, b\}^* \mid w \text{ contains } aa \text{ as a substring} \}$

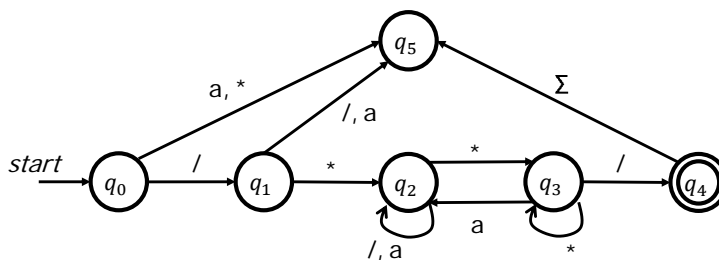
$\bar{L} = \{ w \in \{a, b\}^* \mid w \text{ does not contain } aa \text{ as a substring} \}$



## Complementing Regular Languages



$L = \{ w \in \{a, *, /\}^* \mid w \text{ represents a C-style comment} \}$

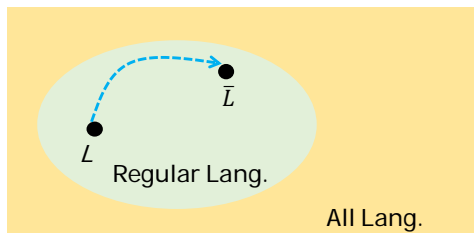




# Closure Property



- Theorem: If  $L$  is a regular language, then  $\bar{L}$  is also a regular language
- As a result, we say that the regular languages are *closed under complementation*



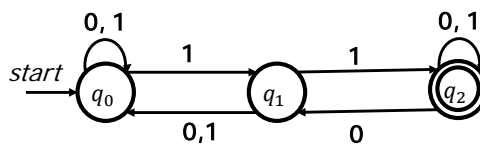
Question: Are the **nonregular** languages closed under complementation?

Languages that require to remember numbers of occurrences  
See: Pumping lemma

# NFA



- **N**ondeterministic **F**inite **A**utomata
- Structurally similar to a DFA, but represents a fundamental shift in how we'll think about computation



## (Non)determinism

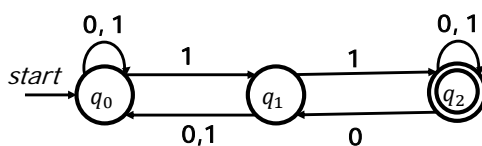


- A model of computation is **deterministic** if at every point in the computation, there is exactly one choice that can make
  - The machine accepts if that series of choices leads to an accepting state
- A model of computation is **nondeterministic** if the computing machine has a finite number of choices available to make at each point, possibly including zero
- The machine accepts if **any** series of choices leads to an accepting state
  - This sort of nondeterminism is technically called existential nondeterminism

Korea Institute of Energy Technology 2022-03-11

35

## A Simple NFA



$q_0$  has two transitions defined on 1

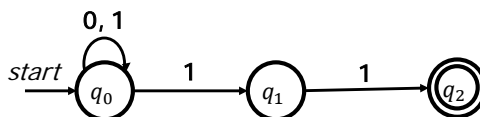
0	1	0	1	1
---	---	---	---	---

There are multiple paths; some lead to the accepting states, others not

Korea Institute of Energy Technology 2022-03-11

36

## Another Nondeterminism



If a NFA needs to make a transition when no transition exists, the automaton dies and that particular path does not accept

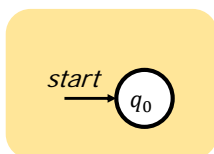
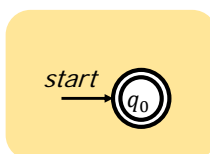
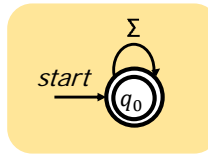
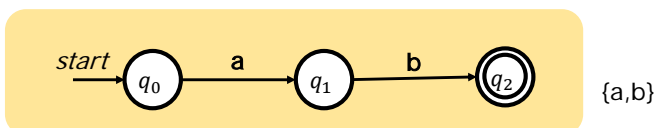
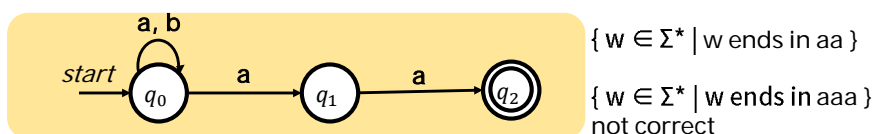
0	1	0	1	1
---	---	---	---	---

Some transition paths fail to parse input completely while some finish parsing

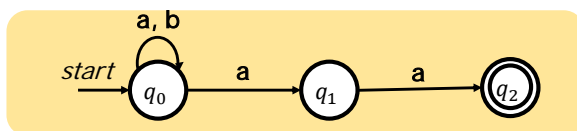
## Language of NFAs



- The language of an NFA is  $\mathcal{L}(N) = \{ w \in \Sigma^* \mid N \text{ accepts } w \}$
- What is the language of each NFA? (Assume  $\Sigma = \{a, b\}$ )


 $\emptyset$ 

 $\{\epsilon\}$ 

 $\Sigma^*$ 

 $\{a, b\}$ 

 $\{ w \in \Sigma^* \mid w \text{ ends in } aa \}$ 
 $\{ w \in \Sigma^* \mid w \text{ ends in } aaa \}$   
not correct

## Language of NFAs



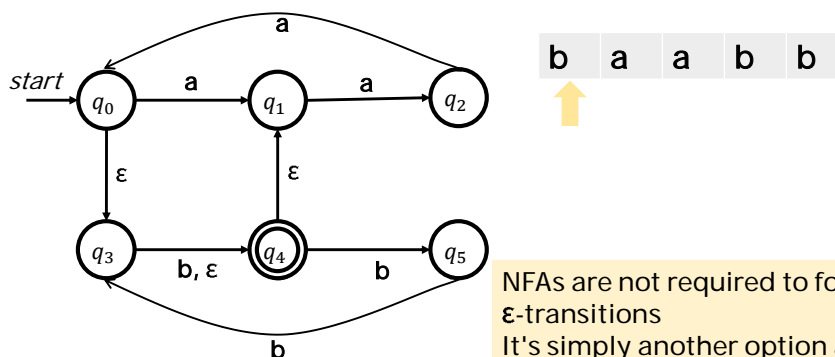
$\{ w \in \Sigma^* \mid w \text{ ends in } aa \}$

$\{ w \in \Sigma^* \mid w \text{ ends in } aaa \}$   
not correct

Flipping the accept and reject states of NFA doesn't always give an NFA for the complement of the original language  
Why?

## $\epsilon$ -Transitions

- NFAs have a special type of transition called the  $\epsilon$ -transition
- An NFA may follow any number of  $\epsilon$ -transitions at any time without consuming any input



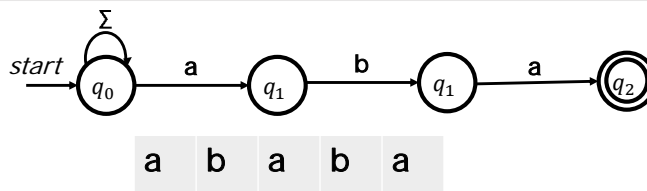
NFAs are not required to follow  $\epsilon$ -transitions  
It's simply another option at the machine's disposal

# Intuiting Nondeterminism



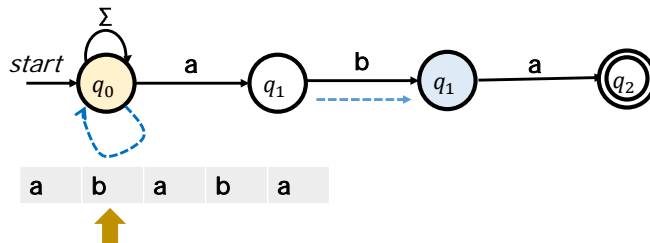
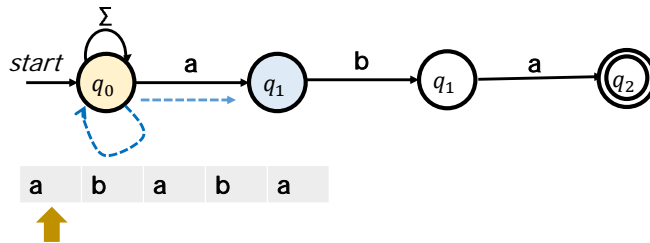
- Nondeterministic machines are a serious departure from physical computers
  - ➔ How can we build up an intuition for them?
- There are two particularly useful frameworks for interpreting nondeterminism:
  - Perfect positive guessing
  - Massive parallelism

## Perfect Positive Guessing

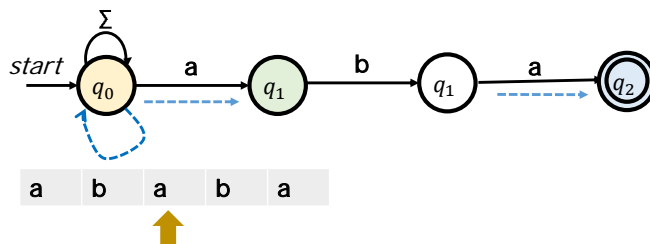


- We can view nondeterministic machines as having **Magic Superpowers** that enable them to guess choices that lead to an accepting state
  - If there is at least one choice that leads to an accepting state, the machine will guess it
- There is no known way to physically model this intuition of nondeterminism

# Massive Parallelism

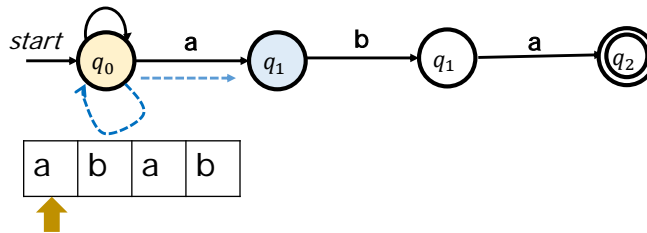


# Massive Parallelism



At least one accepting state, so there's some path that gets us to an accepting state.

## Massive Parallelism



There is no path to the accepting state

## Massive Parallelism



- An NFA can be thought of as a DFA that can be in many states at once
- At each point in time, when the NFA needs to follow a transition, it tries all the options at the same time
- Procedure
  - Start off in the set of all states formed by taking the start state and including each state that can be reached by zero or more  $\epsilon$ -transitions
  - When you read a symbol  $a$  in a set of states  $S$ :
    - Form the set  $S'$  of states that can be reached by following a single  $a$  transition from some state in  $S$
    - Your new set of states is the set of states in  $S'$ , plus the states reachable from  $S'$  by following zero or more  $\epsilon$ -transitions.

## Designing NFAs

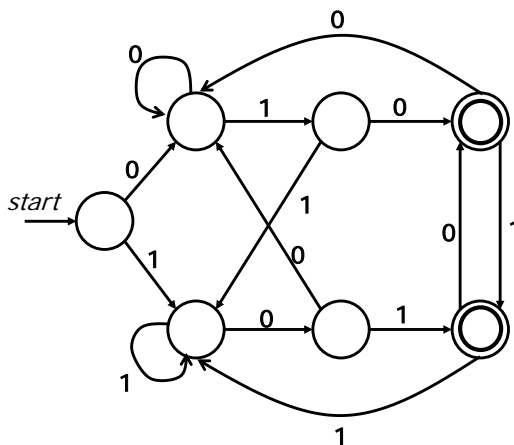


- Embrace the nondeterminism!
- Good model: Guess-and-check:
  - Is there some information that you'd really like to have? Have the machine nondeterministically guess that information
  - Then, have the machine deterministically check that the choice was correct
- The guess phase corresponds to trying lots of different options
- The check phase corresponds to filtering out bad guesses or wrong options.

## Guess & Check

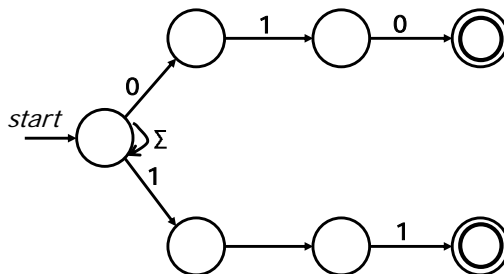


- $L = \{ w \in \{0, 1\}^* \mid w \text{ ends in } 010 \text{ or } 101 \}$



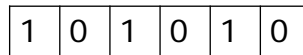


## Guess & Check



Nondeterministically guess when the end of the string is coming up

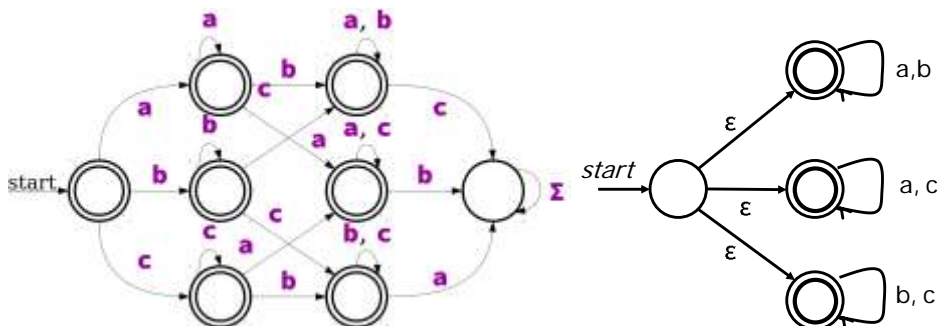
Deterministically check whether you were correct



## Guess & Check



$L = \{ w \in \{a, b, c\}^* \mid \text{at least one of } a, b, \text{ or } c \text{ is not in } w \}$

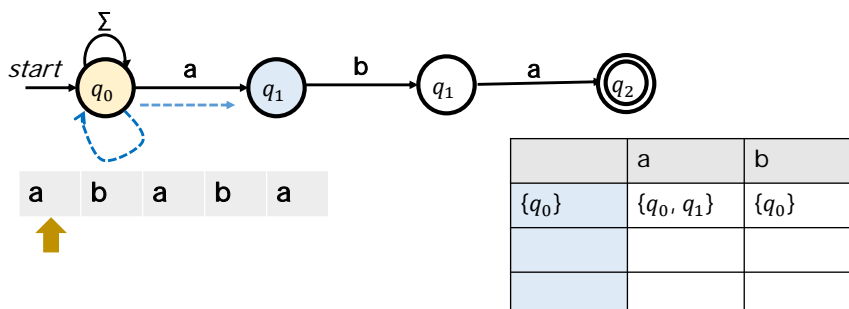
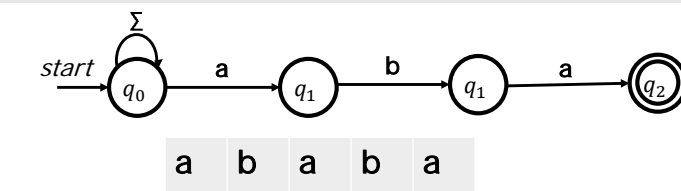


## DFA and NFA

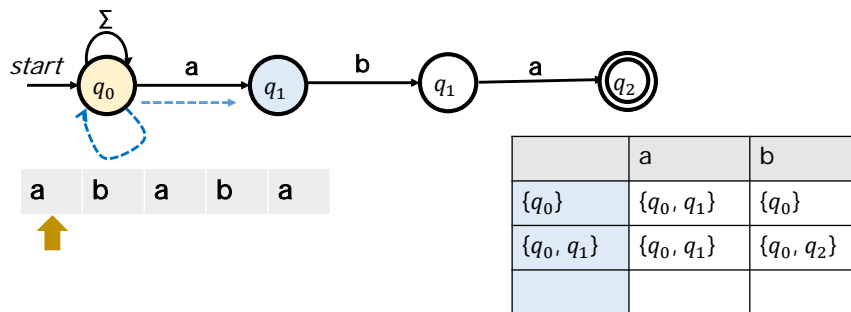


- Any language that can be accepted by a DFA can be accepted by an NFA
- Why?
- Every DFA essentially already is an NFA!
- How about the reverse?
- Can any language accepted by an NFA also be accepted by a DFA?
- Surprisingly, the answer is yes!
- To prove this, we need to:
  - Pick an arbitrary NFA
  - Describe how we would construct a DFA with the same language (in a generalizable way)

## DFA & NFA



## DFA & NFA



## The Subset Construction



- This procedure for turning an NFA for a language  $L$  into a DFA for a language  $L$  is called the *subset construction*
  - Or powerset construction
- Intuitively:
  - Each state in the DFA corresponds to a set of states from the NFA
  - Each transition in the DFA corresponds to what transitions would be taken in the NFA when using the massive parallel
  - The accepting states in the DFA correspond to which sets of states would be considered accepting in the NFA when using the massive parallel
- Explore more on Subset Construction with elaborate examples involving  $\epsilon$ -transitions and cases where the NFA dies

## The Subset Construction



- In converting an NFA to a DFA, the DFA's states correspond to sets of NFA states
- Useful fact:  $|\mathcal{P}(S)| = 2^{|S|}$  for any finite set  $S$

Power sets from a set  $S$

- In the worst-case, the construction can result in a DFA that is exponentially larger than the original NFA
- Question to ponder: Can you find a family of languages that have NFAs of size  $n$ , but no DFAs of size less than  $2^n$ ?

## Important Theorem



- Theorem: A language  $L$  is regular if and only if there is some NFA  $N$  such that  $\mathcal{L}(N) = L$
- Proof
  - part is easy
    - Pick a language  $L$  that is regular
    - That means there's a DFA  $D$  where  $\mathcal{L}(D) = L$
    - Every DFA is "basically" an NFA, so there's an NFA  $(D)$  whose language is  $L$
  - ← Part is also easy
    - Next, assume there's an NFA  $N$  such that  $\mathcal{L}(N) = L$
    - Using the subset construction, we can build a DFA  $D$  where  $\mathcal{L}(D) = \mathcal{L}(N)$
    - Then we have that  $\mathcal{L}(D) = L$ , so  $L$  is regular.

## Perspectives on RL

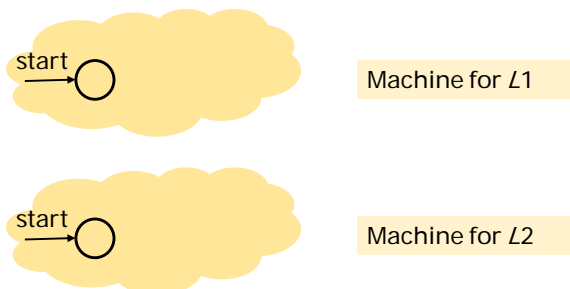


- We now have two perspectives on regular languages
  - Regular languages are languages accepted by DFAs
  - Regular languages are languages accepted by NFAs
- We can now reason about the regular languages in two different ways

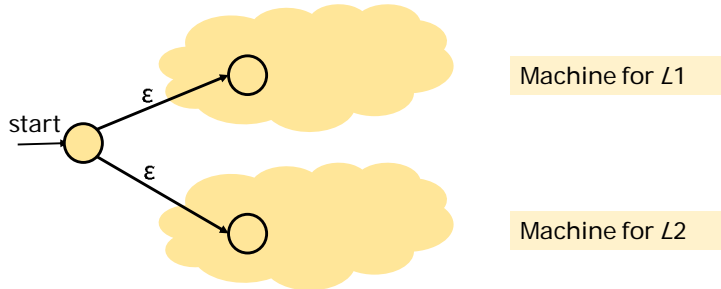
## Union of Two Languages



- If  $L_1$  and  $L_2$  are languages over the alphabet  $\Sigma$ , the language  $L_1 \cup L_2$  is the language of all strings in at least one of the two languages
- If  $L_1$  and  $L_2$  are regular languages, is  $L_1 \cup L_2$ ?

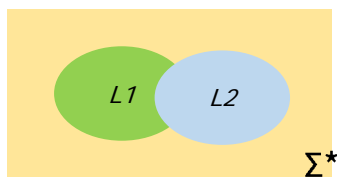


- Machine for  $L_1 \cup L_2$



## Intersection of Two Languages

- If  $L_1$  and  $L_2$  are languages over  $\Sigma$ , then  $L_1 \cap L_2$  is the language of strings in both  $L_1$  and  $L_2$
- Question: If  $L_1$  and  $L_2$  are regular, is  $L_1 \cap L_2$  regular as well?



## String Concatenation



- If  $w \in \Sigma^*$  and  $x \in \Sigma^*$ , the **concatenation** of  $w$  and  $x$ , denoted  $wx$ , is the string formed by tacking all the characters of  $x$  onto the end of  $w$ 
  - Example: if  $w = \text{quo}$  and  $x = \text{kka}$ , the concatenation  $wx = \text{quokka}$
- This is analogous to the  $+$  operator for strings in many programming languages
- Some facts about concatenation:
  - The empty string  $\epsilon$  is the identity element for concatenation:
    - $w\epsilon = \epsilon w = w$
  - Concatenation is associative:  $wxy = w(xy) = (wx)y$

## Language Concatenation



- The concatenation of two languages  $L_1$  and  $L_2$  over the alphabet  $\Sigma$  is the language  $L_1L_2 = \{wx \in \Sigma^* \mid w \in L_1 \wedge x \in L_2\}$
- Example
  - Let  $\Sigma = \{a, b, \dots, z, A, B, \dots, Z\}$  and consider these languages over  $\Sigma$ 
    - **Noun** = { Puppy, Rainbow, Whale, ... }
    - **Verb** = { Hugs, Juggles, Loves, ... }
    - **The** = { The }
  - The language **TheNounVerbTheNoun** is
    - { ThePuppyHugsTheWhale, TheWhaleLovesTheRainbow, TheRainbowJugglesTheRainbow, ... }

## Two Perspectives of Concatenation



- The set of all strings that can be made by concatenating a string in  $L_1$  with a string in  $L_2$
- The set of strings that can be split into two pieces: a piece from  $L_1$  and a piece from  $L_2$ ?

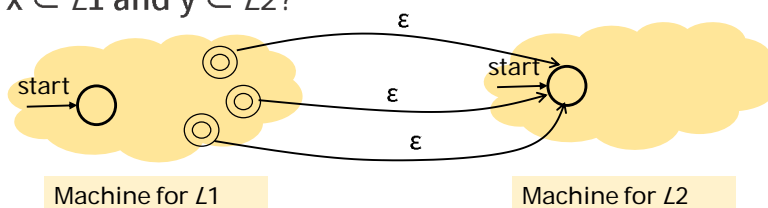
This is closely related to, but different than, the Cartesian product.

Question to ponder: In what ways are concatenations similar to Cartesian products? In what ways are they different?

## Concatenating Regular Languages



- If  $L_1$  and  $L_2$  are regular languages, is  $L_1L_2$ ?
- Intuition – can we split a string  $w$  into two strings  $xy$  such that  $x \in L_1$  and  $y \in L_2$ ?



- Idea:
  - Run a DFA/NFA for  $L_1$  on  $w$ .
  - Whenever it reaches an accepting state, optionally hand the rest of  $w$  to a DFA/NFA for  $L_2$ .
  - If the automaton for  $L_2$  accepts the rest,  $w \in L_1L_2$ .
  - If the automaton for  $L_2$  rejects the remainder, the split was incorrect.



## Concatenation



- Consider the language  $L = \{aa, b\}$
- $LL$  is the set of strings formed by concatenating pairs of strings in  $L$ 
  - $\{aaaa, aab, baa, bb\}$
- $LLL$  is the set of strings formed by concatenating triples of strings in  $L$ 
  - $\{aaaaaa, aaaab, aabaa, aabb, baaaa, baab, bbba, bbb\}$
- $LLLL$  is the set of strings formed by concatenating quadruples of strings in  $L$ 
  - $\{aaaaaaaa, aaaaaab, aaaabaa, aaaabb, aabaaaa, aabaab, aabbaa, aabbb, baaaaab, baaab, baabaa, baabb, bbaaaa, bbaab, bbbba, bbbb\}$

## Language Exponentiation



- We can define what it means to “exponentiate” a language as follows:
- $L^0 = \{\epsilon\}$ 
  - Intuition: The only string you can form by gluing no strings together is the empty string
  - Notice that  $\{\epsilon\} \neq \emptyset$ .
- $L^{n+1} = LL^n$
- Idea:
  - Concatenating  $(n+1)$  strings together works by concatenating  $n$  strings, then concatenating one more

## The Kleene Closure



- An important operation on languages is the **Kleene Closure**, which is defined as  $L^* = \{ w \in \Sigma^* \mid \exists n \in \mathbb{N}. w \in L^n \}$
- Mathematically:  $w \in L^* \leftrightarrow \exists n \in \mathbb{N}. w \in L^n$
- Intuitively,  $L^*$  is the language all possible ways of concatenating zero or more strings in  $L$  together, possibly with repetition
- Question to ponder: What is  $\emptyset^*$ ?

## The Kleene Closure



- If  $L = \{ a, bb \}$ , then  $L^* = \{$   
 $\epsilon,$   
 $a, bb,$   
 $aa, abb, bba, bbbb,$   
 $aaa, aabb, abba, abbbb, bbaa, bbabb, bbbba, bbbbbb,$   
 $\dots$   
 $\}$

Think of  $L^*$  as the set of strings you can make if you have a collection of stamps – one for each string in  $L$  – and you form every possible string that can be made from those stamps

## Reasoning about Infinity



- If  $L$  is regular, is  $L^*$  necessarily regular?

$L^0 = \{ \epsilon \}$  is regular

$L^1 = L$  is regular

$L^2 = LL$  is regular

$L^3 = L(LL)$  is regular

...

- Regular languages are closed under union
- So the union of all these languages is regular.

## Reasoning about the Infinite

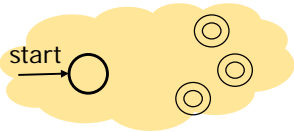


- If a series of finite objects all have some property, the “limit” of that process does not necessarily have that property
- In general, it is not safe to conclude that some property that always holds in the finite case must hold in the infinite case
  - (This is why calculus is interesting)
- So our earlier argument ( $L^* = L^0 \cup L^1 \cup \dots$ ) isn't going to work
- We need a different line of reasoning

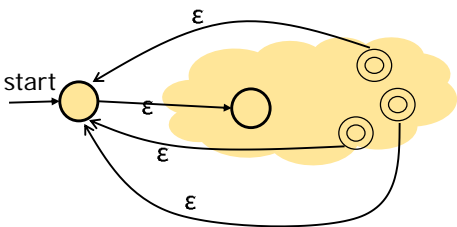
# The Kleene Star



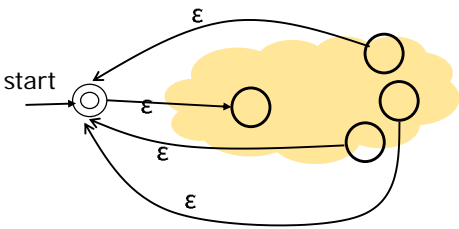
- Idea: Can we directly convert an NFA for language  $L$  to an NFA for language  $L^*$ ?



Machine for  $L_1$



# The Kleene Star



# Closure Properties



- Theorem: If  $L_1$  and  $L_2$  are regular languages over an alphabet  $\Sigma$ , then so are the following languages:

$$\overline{L_1}$$

$$L_1 \cup L_2$$

$$L_1 \cap L_2$$

$$L_1 L_2$$

$$L_1^*$$

- These properties are called *closure properties* of the regular languages