COMP S265F Unit 6: Languages, Finite Automata, Regular Expressions

Dr. Keith Lee

School of Science and Technology

The Open University of Hong Kong

Overview

- Basics:
  - Alphabet, String over an alphabet, Language
  - Decision problem = Language acceptance problem
- Finite state machine:
  - States, Input & Output alphabets,
  - Transition & Output functions (representations), Starting state
  - Example: Binary adder
- Finite automaton:
  - States, Input alphabet, Transition function, Starting state, **Final states**
  - Deterministic vs Nondeterministic Finite Automaton (**DFA** vs **NFA**)
  - Pumping lemma
  - NFA with ε moves
- Regular expression

Basics: Formal language

- An alphabet, usually denoted by Σ or Γ, is a set of symbols.

  E.g., $\Sigma = \{0,1\}$; $\Sigma = \{a,b,c,d,...,x,y,z\}$.
- A string over an alphabet is a sequence of symbols from that alphabet.

  E.g., 10111001 is a string over the alphabet {0,1};

  "computers" is a string over the alphabet {a,b,..,y,z}.
- The length of a string is the number of symbols in the string.

  E.g., The length of "computers" is 9.
- The null string or empty string is a string of length 0.

Basics: Formal language (cont')

- **Σ^*** denotes the set of all possible strings over the alphabet Σ, including the empty string.
- **Σ^i**, where $i \geq 1$, denotes the set of strings of length exactly i.

E.g., $\Sigma = \{0, 1\}$, and $\Sigma^2 = \{00, 10, 11, 01\}$

A **language** L over an alphabet Σ is a set of strings over Σ. I.e., $L \subseteq \Sigma^*$.

E.g., $\Sigma = \{a,b,c,d,\ldots,x,y,z\}$;

- L_1 = {algorithms, complexity, computer, PC, unix};
- $L_2 = \{ w \in \Sigma^* \mid w \text{ contains an "a"} \}$

E.g., $\Sigma = \{0, 1\}$;

- **L_3** $= \{ w \in \Sigma^* \mid w$ is a *prime* binary number $\}$

Languages versus Decision problems

- Decision (yes/no) problems: Given a binary string x, determine whether x is prime.

- Language acceptance problem: Given a binary string x, determine whether x ***is an element*** of L_3.

- Note that $x \in L_3$ if and only if x is prime.

Language acceptance problem

In general, any decision problem can be formulated as a language acceptance problem.

- Let **P** be any decision problem; assume the input is a string over an alphabet Σ.

  The answer of **P** with respect to any input $w \in \Sigma^*$ is either "YES" or "NO".

- The corresponding language **L** is

  $\{w \in \Sigma^* \mid$ the answer of **P** w.r.t. input w is "YES" $\}$.

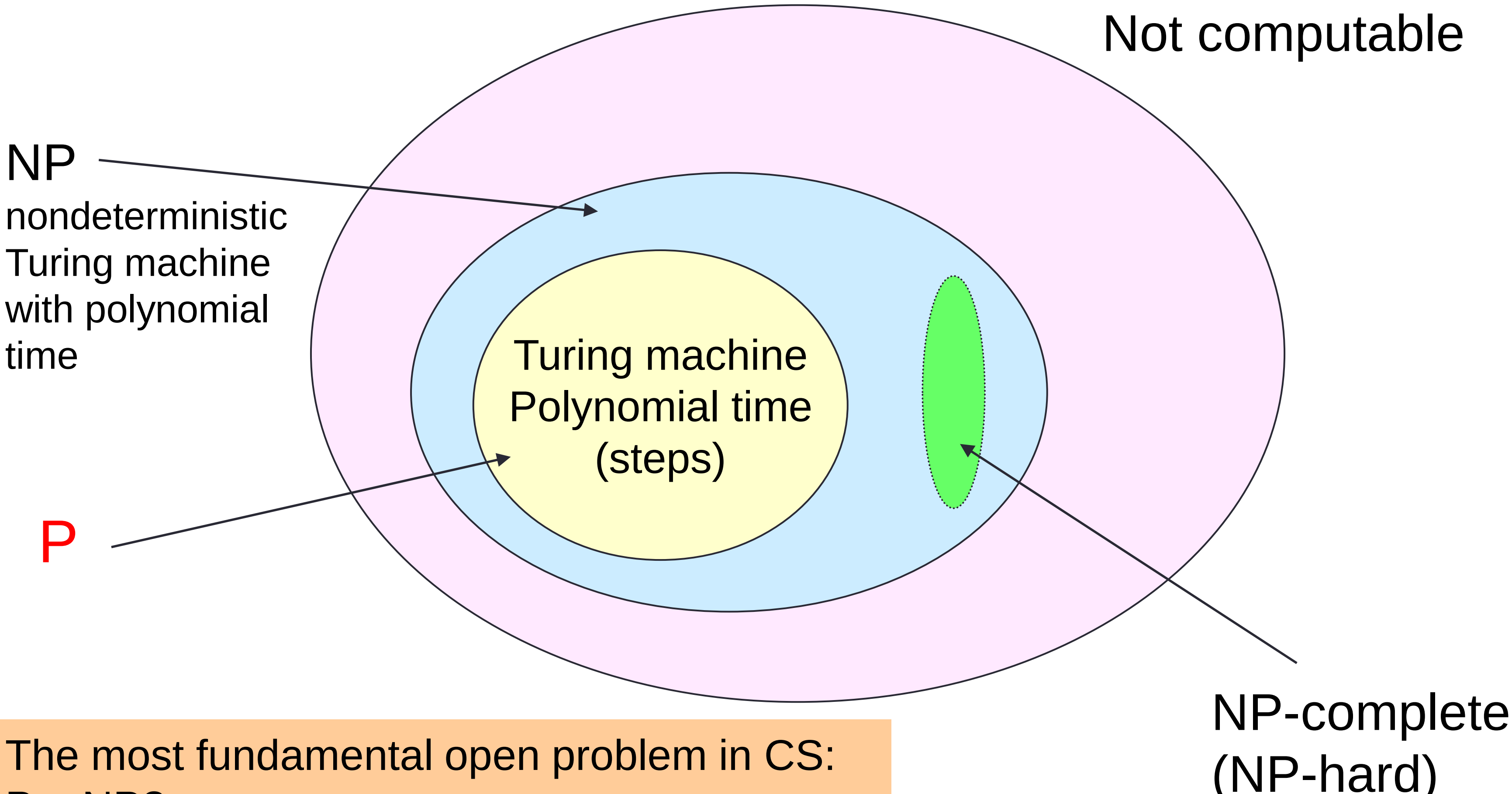
  NB. **L** includes all positive instances of **P**.

- An algorithm that can accept (decide) correctly the elements of **L** also solves the problem **P**.

Modeling Computation

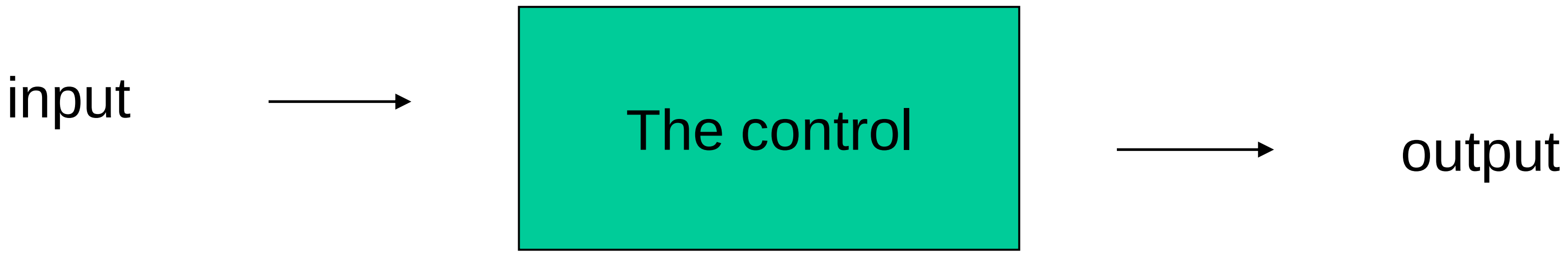
- We will study three models of computation in this course: finite automata, pushdown automata and Turing machines.
- They are very simple, and their computation can be argued mathematically.
- Finite automata are primitive, modelling computers with a very limited memory.
- Turing machines are more powerful and can model the computation of a PC or any computer.
- Based on Turing machines, we can easily study the limitation of computers, showing that some problems cannot be solved by computers.

Computability & Complexity theory: Classification of problems



The most fundamental open problem in CS:
P = NP?

Finite state machines

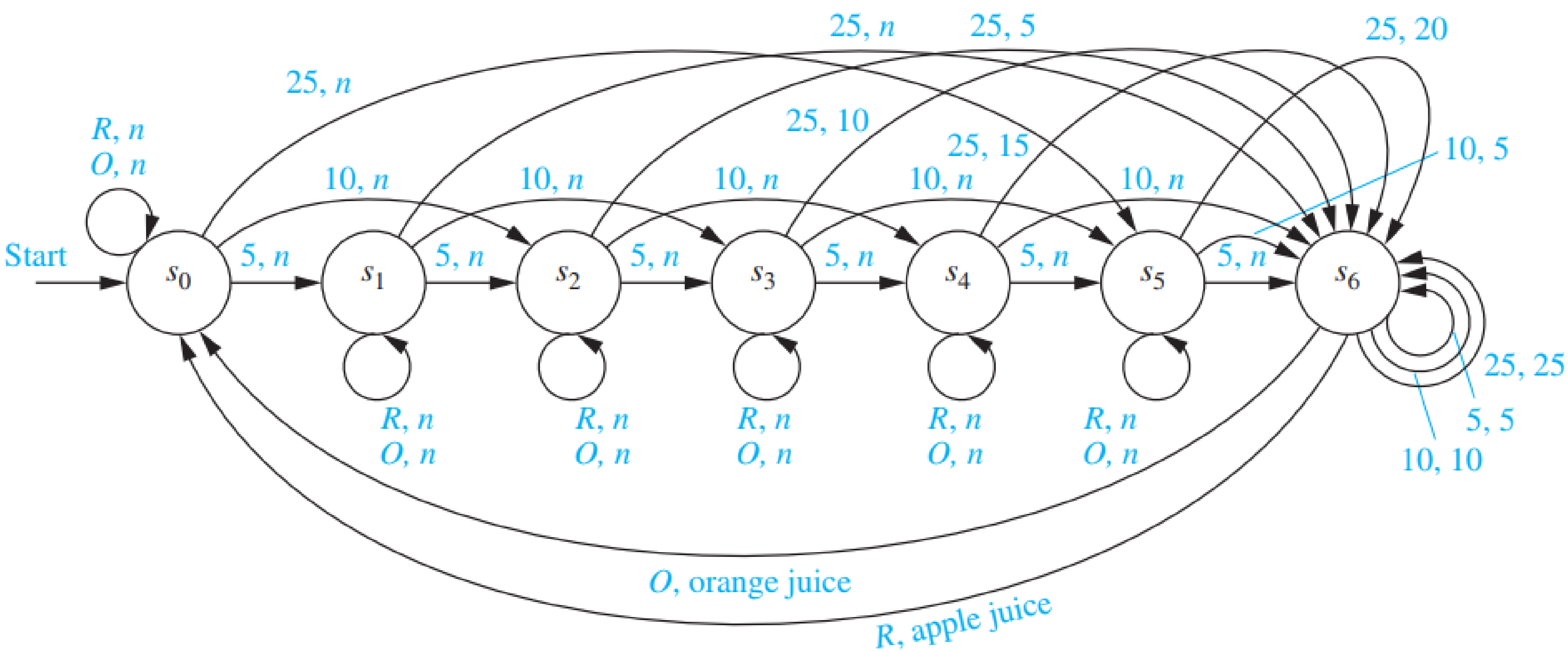


The control - At any time the machine is in a particular **state**. In one step, it reads an input. Depending on <u>what is read</u> and the <u>current state</u>, the machine jumps to another state and outputs something.

- The number of possible states is fixed in advance, i.e., independent of the input.
- The state transition is pre-specified by a function.

Example: Vending machine

- **Input:** 5 cents (nickel), 10 cents (dime), 25 cents (quarter), Orange button, Red button
- **Output:** *change*, Orange juice, Apple juice
- **Control:** 7 possible states



Finite state machine: Formal definition

A finite state machine **M = (S, I, O, f, g, s_0)** consists of

- a finite set **S** of states,
- a finite input alphabet **I**,
- a finite output alphabet **O**,
- a transition function **f**: **S** $\times$ **I** $\rightarrow$ **S** that assigns a new state to each combination of state and input,
- an output function **g**: **S** $\times$ **I** $\rightarrow$ **O** that assigns to each state and input an output,
- a starting state $\mathbf{s_0}$

What are the possible input/output symbols?

NB. Alphabet = set of symbols

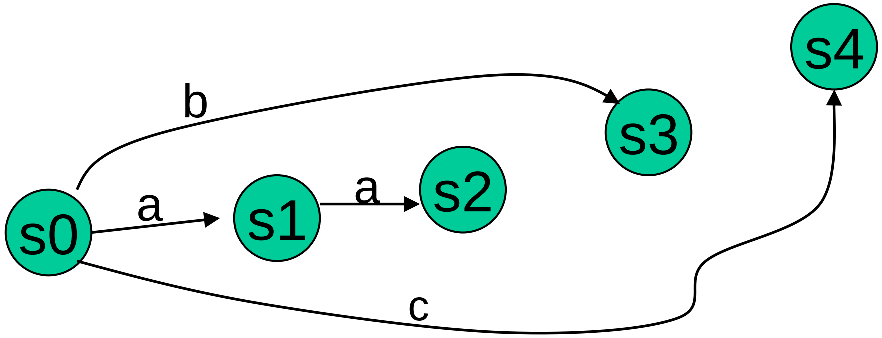
Transition/Output function

- can be represented by a table or a diagram.

	s0	s1	s2	s3	s4
a	s1	s2	s2	s3	s0
b	s3	s1	s0	s4	s1
c	s4	s4	s4	s4	s2

S = {s0, s1, s2, s3, s4}

I = {a, b, c}



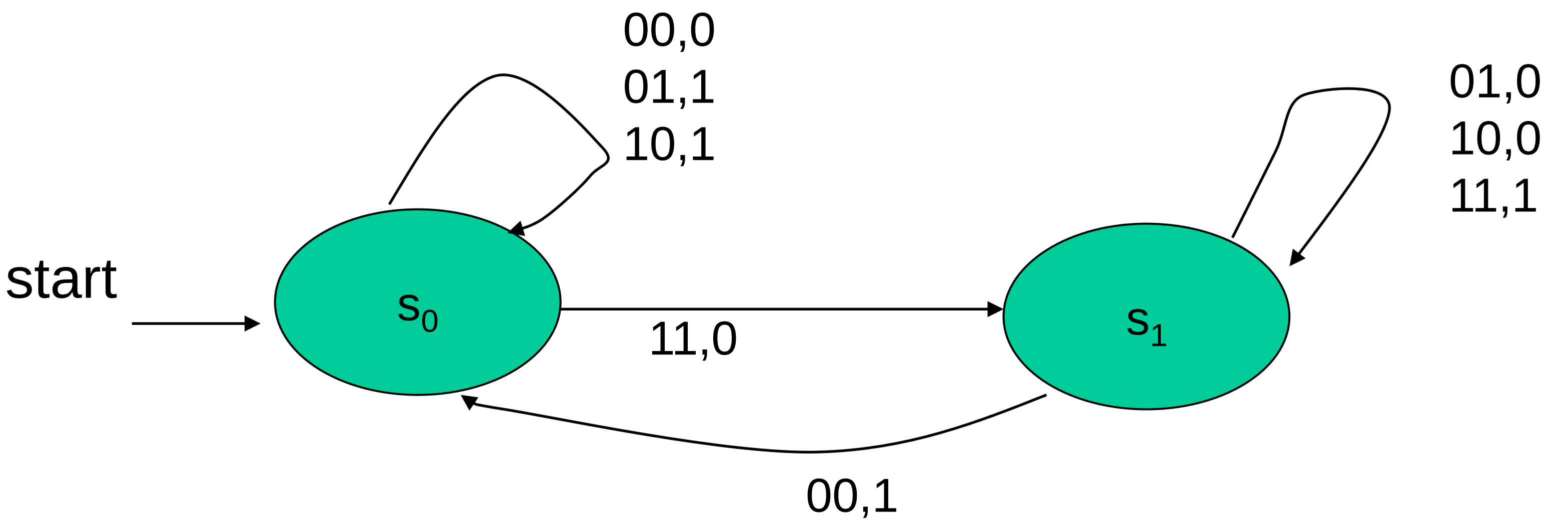
Finite state machine: Computation

Given a finite state machine M, how does it operate?

- First, the machine starts off in the state s_0 .
- Let $x = x_1\ x_2\ x_3 \dots x_n$ be the input, where each $x_i \in I$.
- The machine reads the input symbols one by one.
- After reading x_1, M jumps to state $s = f(s_0, x_1)$ and outputs the symbol $g(s_0, x_1)$.
- Next, M reads x_2 and jumps to state $s' = f(s, x_2)$ and outputs the symbol $g(s, x_2)$.
- Next, M reads x_3 and jumps to state $s'' = f(s', x_3)$ and outputs the symbol $g(s', x_3)$.
- …
- After the whole input is read, the computation ends.

Example: Binary adder

- A finite state machine with 2 states, representing carry=0 and carry=1.
- **S** = {s_0, s_1}, **I** = {00, 01, 10, 11}, **O** = {0,1}
- The input **01** means the first operand is **0** and the second operand is **1**...
  - E.g., if we want to add **0110** & **1000**, the input is **00**, **10**, **10**, **01**.



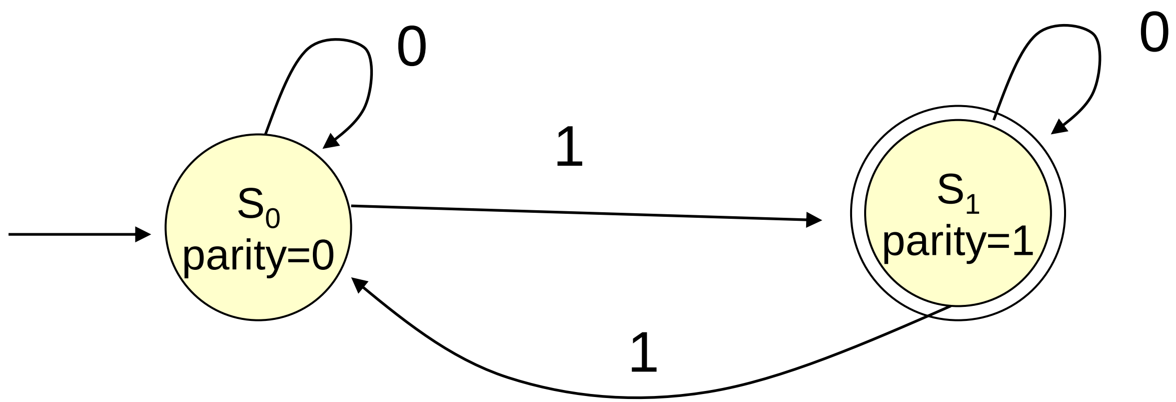
Finite state machine with no output

- In the literature, the studies of finite state machines focus on those which do not output. These machines only tell us whether the input is accepted (yes) or rejected (no).

- E.g., a finite state machine for checking whether the input is a binary string with odd parity.

How to signal the acceptance of input?

- Some of the states in the machine are marked as **final states**.
- If the machine, after reading the input, stops at a final state, the input is said to be ***accepted*** *or recognized (Yes)*;
- if the machine stops at a non-final state, the input is said to be ***rejected*** *(No)*.

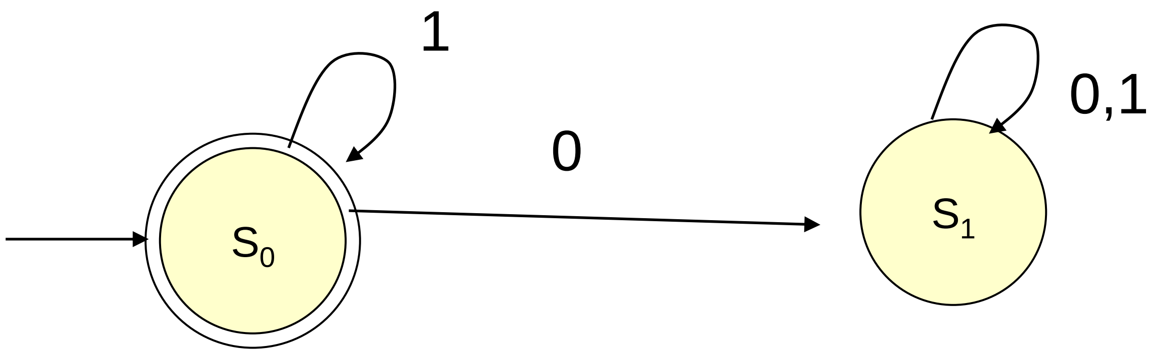


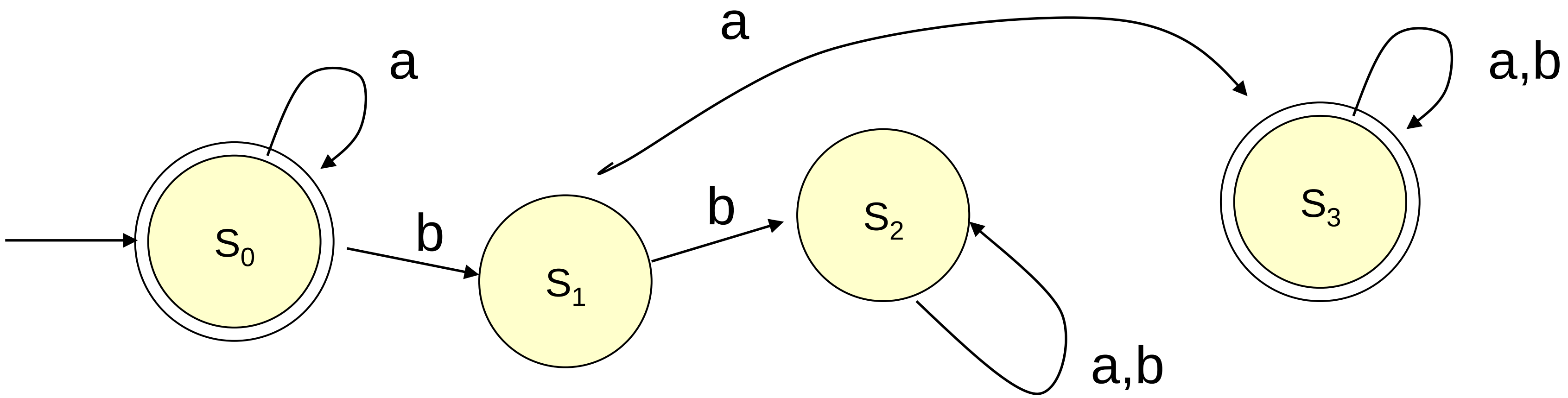
S_1 is the **final state**; final states are represented by **double circles**

E.g., input = 11011 (stops at s_0; rejected)
input = 11011001 (stops at s_1; accepted)

More examples

- What are the inputs accepted by the following machines?





Finite automaton: Formal definition

A finite state machine with no output (also called a **finite automaton**) $M = (S, \Sigma, f, s_0, F)$ consists of

- a finite set S of states,
- an input alphabet Σ,
- a transition function $f: S \times \Sigma \rightarrow S$,
- a starting state s_0, and
- a set of $F \subseteq S$ of **final states.**

No output alphabet & output function!

An input x, which is a string composed of symbols in Σ, is accepted or recognized by M if M on input x stops at a state in F.

Intuitively, **M** solves a **decision** (yes/no) **problem** rather than computing something.

Finite automaton: Languages

- Recall that a language is a subset of strings over a certain alphabet.
- The language accepted (recognized) by M comprises all the input strings over Σ that are accepted by M.
- I.e., **L(M)** = { *w* | M accepts *w* }.

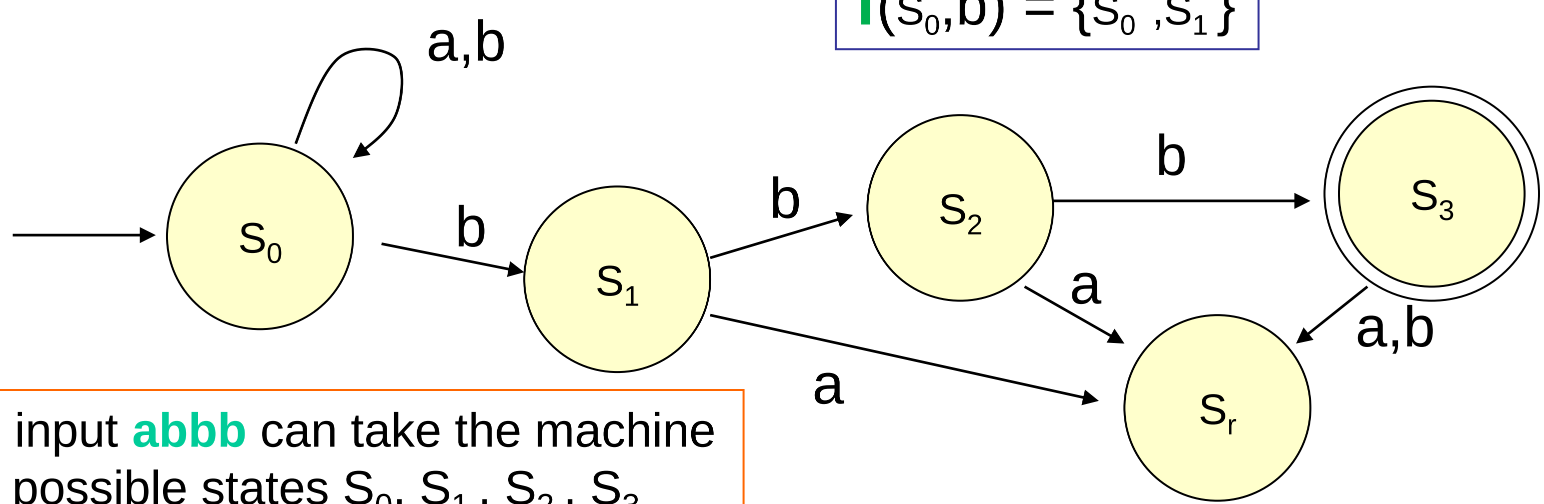
Nondeterministic finite automaton

- The finite automata discussed so far are deterministic in the sense that given any pair of state and input, an automaton goes to a single state in the next step (because **f** is a function).
- A nondeterministic finite automaton is more flexible, allowing more than one possible next state.

DFA

NFA

$\mathbf{f}(s_0,b) = \{s_0, s_1\}$



The input **abbb** can take the machine to 4 possible states S_0, S_1, S_2, S_3

NFA: Formal definition

NFA

A nondeterministic finite automaton $M = (S, \Sigma, \mathbf{f}, s_0, F)$ consists of

- a finite set S of states, an input alphabet Σ,
- a transition function **f** that assigns **a set of states** to each pair of state and input
- a starting state s_0, and a set of **F** $\subseteq$ S of final states.

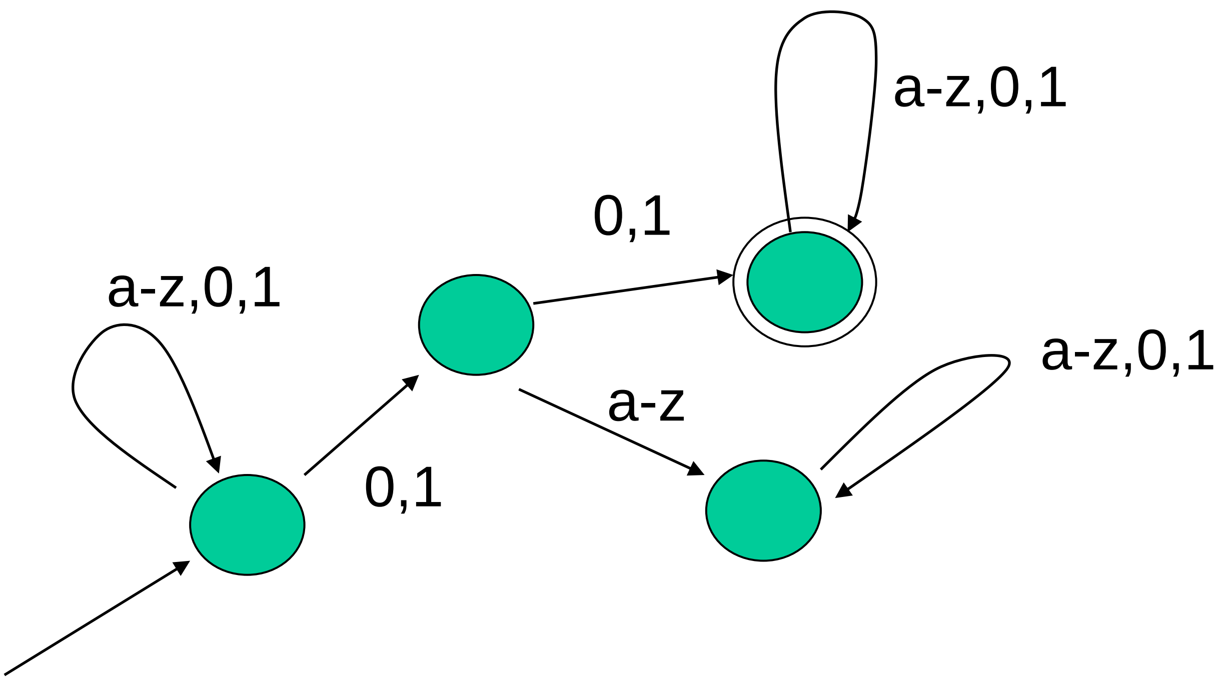
Given an input ***x***, M can ***take different sequence of moves, arriving at different states***. Some of these states may be final and others may not.

We say that ***x*** is **accepted/recognized** if, among all the states at which M can arrive, there is one in **F**.

An NFA also defines a language, which comprises all the input strings it accepts.

NFA: Example

- Design an NFA to accept the set of strings of lower-case letters or bits containing at least two consecutive bits.



Is NFA more powerful than DFA?

- Is there a decision problem that can be solved by an NFA but not by a DFA?
- Is there a set of strings over a certain alphabet that can be accepted by an NFA but not by a DFA?
- The answer is NO.

- **Theorem:** Let **M** be any NFA accepting a set **L** of strings. Then there exists a DFA **M'** that can accept exactly all strings in **L**.

Proof: Subset construction

- Suppose an NFA M=(S,Σ,f,s_0,F) has **n** states, where **n** is a constant.
- After reading some input symbols, M can possibly reach more than one state, more precisely, a certain subset of states.
- On different inputs, M can reach different subsets of states.

How many possible subsets of states M can reach?
Answer: $\mathbf{2^n}$, which is also a constant.

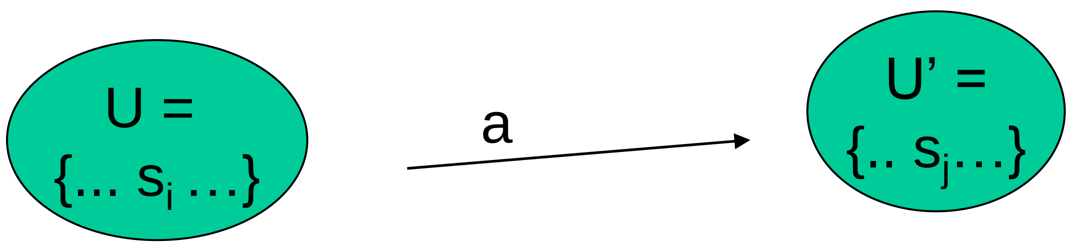
- E.g., **n**=10. There are 1024 different subsets of states. No matter what is the input, the subset of states M can reach is one of these 1024 subsets.
- By definition, an input x is accepted by M if and only if one of the states at which M stops is a final state (i.e., in **F**).
- We construct a DFA **M'** to simulate M in the next slide…

Proof: Subset construction - DFA

- Let $S = \{s_0, s_1, \ldots, s_{n-1}\}$ be the set of states of M.
- **M'** = (**S'**,Σ,**f'**,U_0,F') is a DFA with 2^n "states", each "state" ∈ **S'** is labeled with (and represents) a subset of **S**.

Let U and U' be two "states" of **M'**.

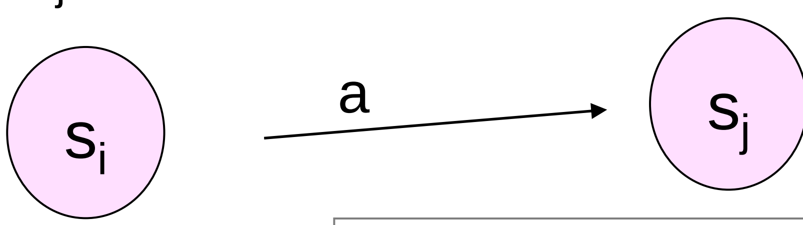
M'



f'(U, a) = U'

if and only if U' represents the subset containing all the states s_j such that s_j is in **f**(s_i, a) for some state s_i in U.

M



f(s_i, a) = {…, s_j, …}

Intuitively, M' uses one state to memorize all the possible states that can be reached in M.

M' *(DFA)* simulates M *(NFA)*

- What is the starting state of M'? $U_0 = \{s_0\}$.
- Which are the final states of M'?

  $U \subseteq S$ is a final state of M' if U contains a state in F.

- On any input x, M can reach a subset U of states

  $\Leftrightarrow$ M' can reach the state U.

  NB. This can be proven using an induction on the length of x.

- M accepts x $\Leftrightarrow$ U contains a state in F

  $\Leftrightarrow$ U is a final state of M'

  $\Leftrightarrow$ M' accepts x.

Limitation of finite automata

Let L be the set of strings 00…00011…111 which contain the same number of 0’s and 1’s.

I.e., L = {01, 0011, 000111, …. }.

Notation: Let **a^i** denote the string with *i* a’s.

Question: Can we construct a DFA or NFA to accept L ?

- In other words, we want a finite automaton to check the number of 0’s and 1’s.

Answer: No, such an automaton doesn’t exist.

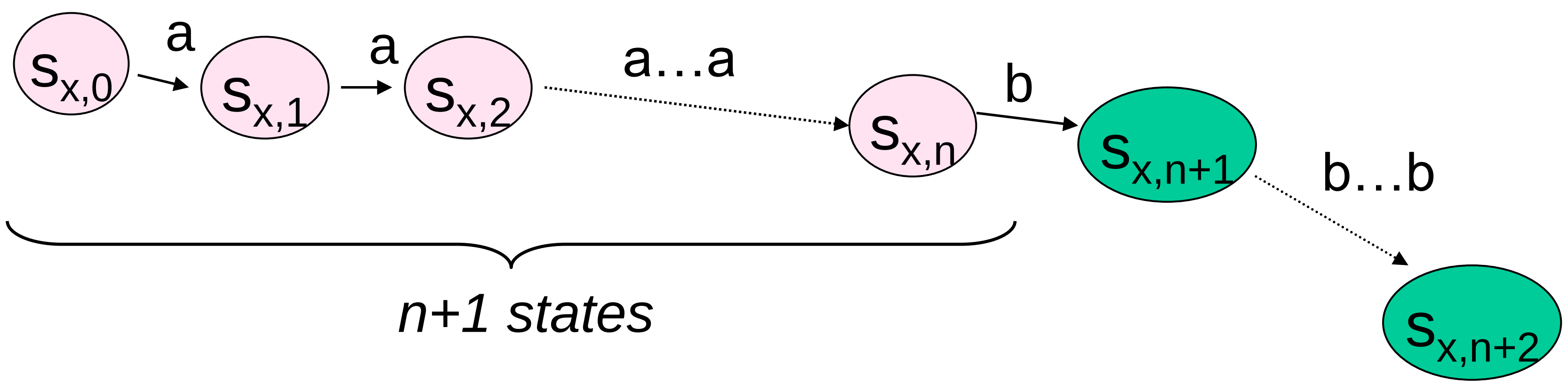
NB. Roughly speaking, DFA has no memory to store a counter.

Proof (by contradiction)

- Suppose that there is a DFA M accepting L.
- Assume that M has n states and the starting state is s_0.
  Note that n is a constant.
- Consider the string $x = a^n b^n$. By definition, M should accept x.
- Denote the state of M after reading the 1st symbol of x as $s_{x,1}$.
- And similarly, $s_{x,2}$, …, $s_{x,k}$ for the 2nd symbol, …, k-th symbol, respectively.
- For convenience, we denote $s_{x,0} = s_0$.

Proof (by contradiction): Pigeonhole principle

- Consider the states $s_0 (= s_{x,0})$, $s_{x,1}$, $s_{x,2}$, …, $s_{x,n}$, $s_{x,n+1}$, …, $s_{x,2n}$.
- Since M accepts x, $\mathbf{s_{x,2n}}$ is a final state of M.



- Note that M has n distinct states. By the pigeonhole principle, there exist $0 \le \mathbf{j < k} \le n$ such that $s_{x,\mathbf{j}} = s_{x,\mathbf{k}}$
- What can we conclude?
- Let m = k-j. M accepts the string $a^{n-m} b^n$. What about $a^{n+m} b^n$?
- A contradiction occurs.

Pumping Lemma

Theorem: Let L be a language that can be accepted by a DFA M with n states. For any string s in L of length at least n, s can be divided into three pieces, s = xyz such that

- $|y| > 0$,
- $|xy| \leq n$, and
- for all $\boldsymbol{i} \geq 0$, $xy^{i}z$ is in L.

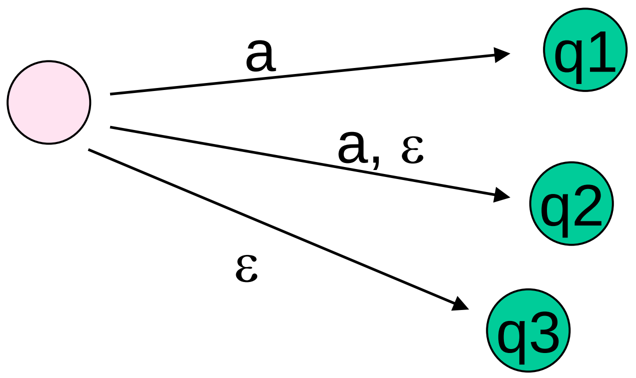
Proof: Let $s = s_1\ s_2\ s_3\ \ldots\ s_m$, where its length $m \geq n$.

- Let $r_1, r_2, \ldots, r_{m+1}$ be the sequence of states M enters in processing s.
- Among $r_1, r_2, \ldots, r_{\mathbf{n+1}}$, by pigeonhole principle, $\exists$ $\mathbf{p < q}$ s.t. $\mathbf{r_p = r_q}$.
- Now, let $\mathbf{x} = s_1\ s_2\ \ldots\ s_{p-1}$, $\mathbf{y} = s_p\ s_{p+1}\ \ldots\ s_{q-1}$, $\mathbf{z} = s_q\ s_{q+1} \ldots\ s_m$.
- $|\mathbf{y}| = (q-1) - p + 1 = q - p > 0$ (as $p < q$).
- $|\mathbf{xy}| = q-1 \leq \mathbf{n+1} - 1 = n$ (as $q \leq \mathbf{n+1}$).
- As **x** takes M from r_1 to r_p, **y** from $\mathbf{r_p}$ to $\mathbf{r_q} = \mathbf{r_p}$, **z** from $\mathbf{r_q}$ to r_{m+1} (which is a final state), M must accept $\mathbf{x\ y^{\mathit{i}}\ z}$ for all $\boldsymbol{i} \geq 0$ (i.e., $\mathbf{x\ y^{\mathit{i}}\ z} \in L$).

NFA with ε moves

power set of S

- Let ε denote the null string.
- Extend the transition function **f:** **$S \times (\Sigma \cup \{\varepsilon\}) \rightarrow P(S)$**
  - E.g., f(a) = {q1, q2}; f(ε) = {q2, q3}



- Are NFA with ε moves more powerful than NFA and DFA?
- Answer: **No.**

NFA with ε moves vs. NFA

Lemma. Given an NFA M_ε with ε moves, we can construct another NFA M (without ε moves) accepting the same language.

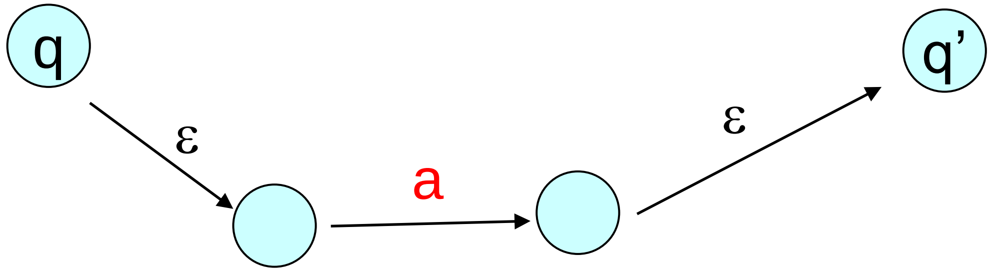
Idea. M uses the same set of states as M_ε.

For every pair of states (q, q'),

M has a transition from q to q' labeled with some a in Σ

if and only if

in M_ε, there is a path from q to q' labeled with all ε except one a.



NFA with ε moves vs. NFA

Lemma. Given an NFA M_ε with ε moves, we can construct another NFA M (without ε moves) accepting the same language.

Idea. M uses the same set of states as M_ε.

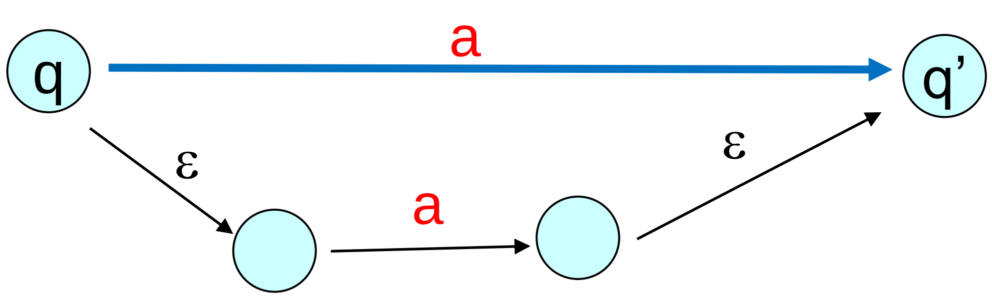
For every pair of states (q, q'),

M has a transition from q to q' labeled with some a in Σ

if and only if

in M_ε, there is a path from q to q' labeled with all ε except

one a.



Non-computational models

Given a language (decision problem) L, we can reason whether there is a finite automaton (pda, or Turing machine) accepting L.

In fact, languages that can be accepted by finite automata can be characterized by some non-computation-based models.

- **Regular expressions**
  - **Theorem.** A language L is accepted by a DFA if and only if L = L(R) of some regular expression R.
- **Right (Left) Linear grammars**
  - **Theorem.** A language L is accepted by a DFA if and only if L = L(G) of some right linear grammar G.

Equivalence

- DFA
- NFA
- NFA with ε moves
- Regular expressions

Regular Expressions

- A simple way to define a set of strings (i.e., a language).
- For example, $(0 \cup 1)0^*$ denotes the set { 0, 1, 00, 10, 000, 100, 0000, 1000, …}

- **A recursive definition:** R is a regular expression over an alphabet Σ if R is
  - *a* for some *a* in Σ, ε, $\emptyset$,
  - $(R1 \cup R2)$, $(R1 \circ R2)$, or $R1^*$, where R1 and R2 are regular expressions.

R1 R2

- **More examples:** 0^*10^*, $(0 \cup 1)^*1$

Regular Expressions: Languages

A regular expression R defines a language, denoted by **L**(R).

- R = a: L(R) = {a}.
- R = **ε**: L(R) = {**ε**} (i.e., the set of null string).
- R = $\varnothing$: L(R) is empty.
- R = ($R_1 \cup R_2$): L(R) = { w | w is in L(R_1) or L(R_2) }.
- R = (R_1 o R_2): L(R) = { w | w = ***xy*** where ***x*** is in L(R_1) and ***y*** is in L(R_2) }.
- R = R_1*: L(R) = { w | w is in L(R1)*}.

Def. w = **ε** or $w_1w_2w_3 \ldots w_n$, where $n \geq 1$ and each $w_i \in L(R1)$.

Regular Expressions: Examples

- $(01^*)^*$:

For convenience: we let $\mathbf{R^+}$ be shorthand for RR^*, i.e., R occurs at least 1 times.

- $(01^+)^*$:

- $1^* \varnothing$:

- ε^*:

Regular Expressions & DFA

Theorem. Let L be a language accepted by a DFA M. Then there exists a regular expression R such that L(R) = L.

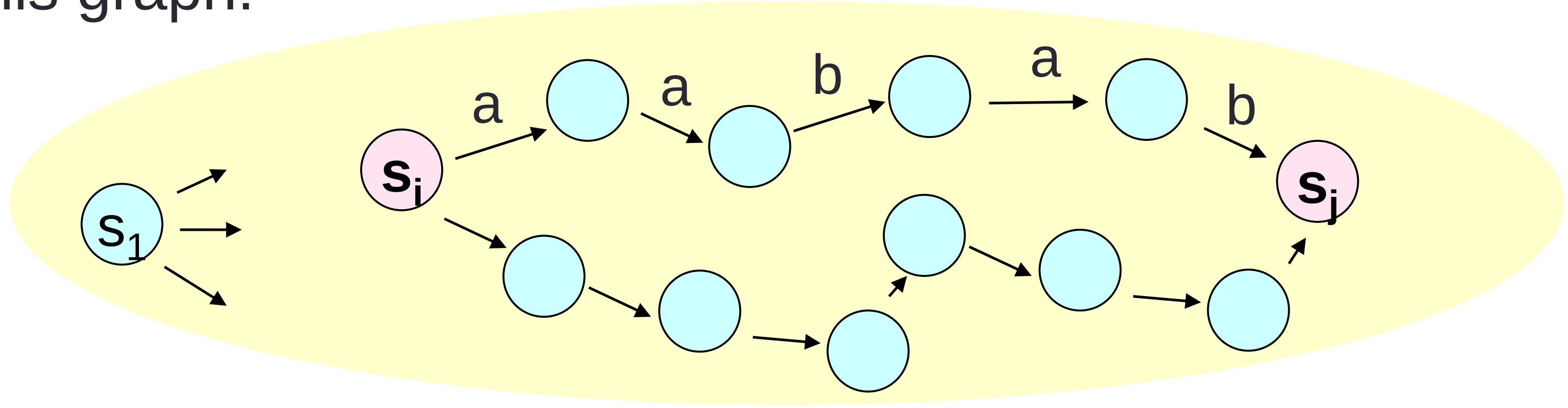
Implication: An NFA or DFA is no more powerful than a regular expression.

Regular Expressions & DFA (con't)

Theorem. Suppose L is accepted by a DFA. Then L = L(R) for some regular expression R.

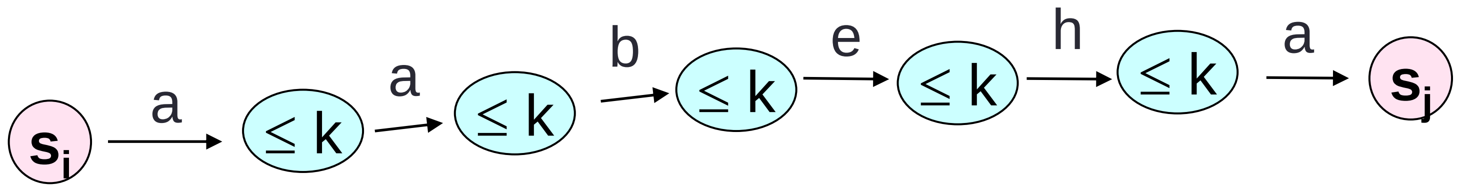
Proof: Suppose L is accepted by a DFA M = (S, Σ, **f**, s_1, F) with n states. Let S = {s_1, s_2, …, s_n}.

- The transition function of M (i.e., **f**) defines a directed graph, in which every vertex is a state and every edge is labeled with a symbol in Σ. The following discussion is based on this graph:



From State $\mathbf{s_i}$ to State $\mathbf{s_j}$

- Consider any k in the range [0, n].
- Let $S_{i,j}$ = { x | x is the string on the path from $\mathbf{s_i}$ to $\mathbf{s_j}$ in M }.
- Let $S_{i,j}(k)$ = { x | x is the string on the path from $\mathbf{s_i}$ to $\mathbf{s_j}$ in M, and excluding the two ends, every state on this path has a label s_b with $b \le k$}.

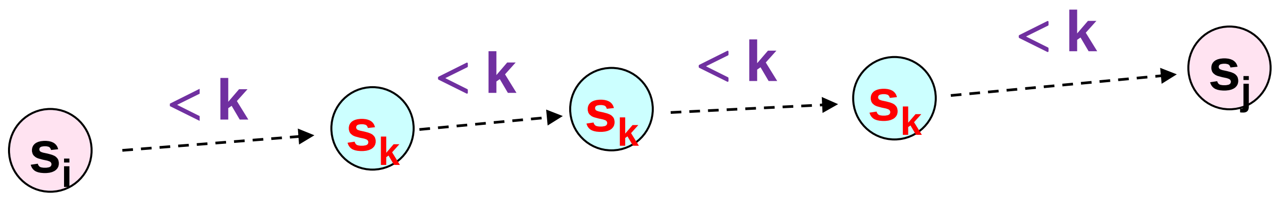


A technical lemma

Lemma. $S_{i,j}(k) = S_{i,j}(k-1) \cup S_{i,k}(k-1)\ (S_{k,k}(k-1))^*\ S_{k,j}(k-1)$

Proof.

- Any string (i.e., path) in $S_{i,j}(k-1)$ is also a string in $S_{i,j}(k)$.
- If a string in $S_{i,j}(k)$ passes state s_k, then
  - it must first go from s_i to s_k via a path with label $< k$,
  - it ***may*** then go from s_k to s_k via a path with label $< k$, and this ***may*** repeat multiple times, and
  - it must finally go from s_k to s_j via a path with label $< k$.



$S_{1,j}(n)$

- Suppose s_j is a state in F.
- What does $S_{1,j}(n) = S_{1,j}$ denote?
- **Ans:** The set of strings that M accepts using the final state s_j.

- Let L be the language accepted by M.
- Then, L is equal to the ***union*** of all $S_{1,j}(n)$, where $s_j \in F$.

Converting DFA to regular expressions

Lemma. For any i, j, k, there is a regular expression R such that $L(R) = S_{i,j}(k)$.

Proof. By induction on k.

- ***Base case:*** $k = 0$.
- Let $a_1, a_2, \ldots, a_h$ be symbols in Σ such that $\mathbf{f}(\mathbf{s_i}, a_1) = \mathbf{s_j}$, $\mathbf{f}(\mathbf{s_i}, a_2) = \mathbf{s_j}$, …, $\mathbf{f}(\mathbf{s_i}, a_h) = \mathbf{s_j}$.
- Let R be the regular expression $a_1 \cup a_2 \cup \ldots \cup a_h$.
- Then $L(R) = \{a_1, a_2, \ldots, a_h\} = S_{\mathbf{i,j}}(0)$.

Lemma. For any i, j, k, there is a regular expression R such that $L(R) = S_{i,j}(k)$.

- ***Induction hypothesis:*** Suppose the lemma is true for k-1.
- Recall that $S_{i,j}(k) = S_{i,j}(k-1) \cup S_{i,k}(k-1)\ (S_{k,k}(k-1))^*\ S_{k,j}(k-1)$.

- By the induction hypothesis, there exists regular expressions R1, R2, R3, and R4 such that
  - $L(R1) = S_{i,j}(k-1)$,
  - $L(R2) = S_{i,k}(k-1)$,
  - $L(R3) = S_{k,k}(k-1)$,
  - $L(R4) = S_{k,j}(k-1)$.

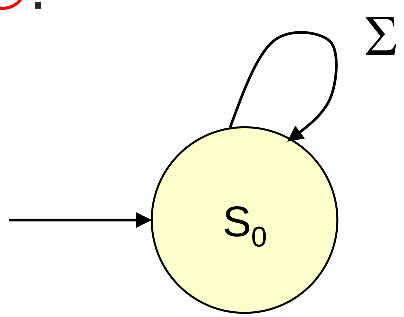
- Then, $R = R1 \cup (\ R2\ (R3)^*\ R4\ )$ is a regular expression such that $L(R) = S_{i,j}(k)$.

From regular expressions to NFA

Theorem. Let R be a regular expression. Then there exists an NFA with ε moves M such that L(M) = L(R).

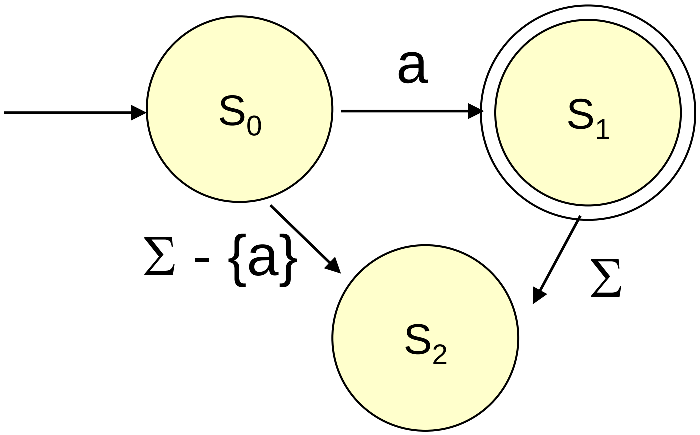
Proof. By induction on the structure of R.

- R = ∅:

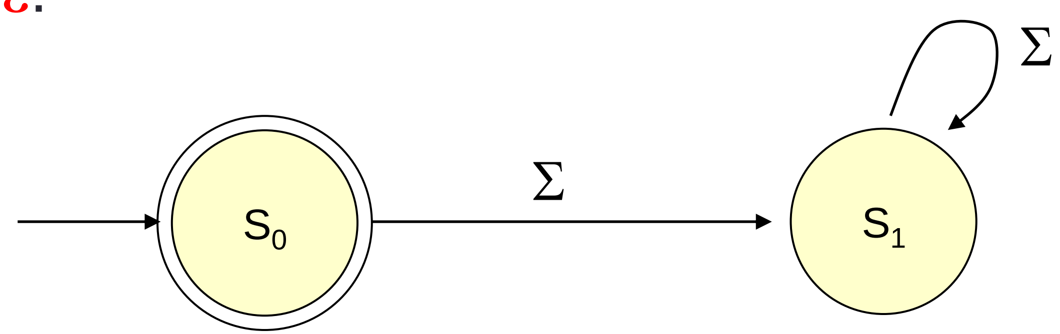


From regular expressions to NFA (cont')

- R = a:



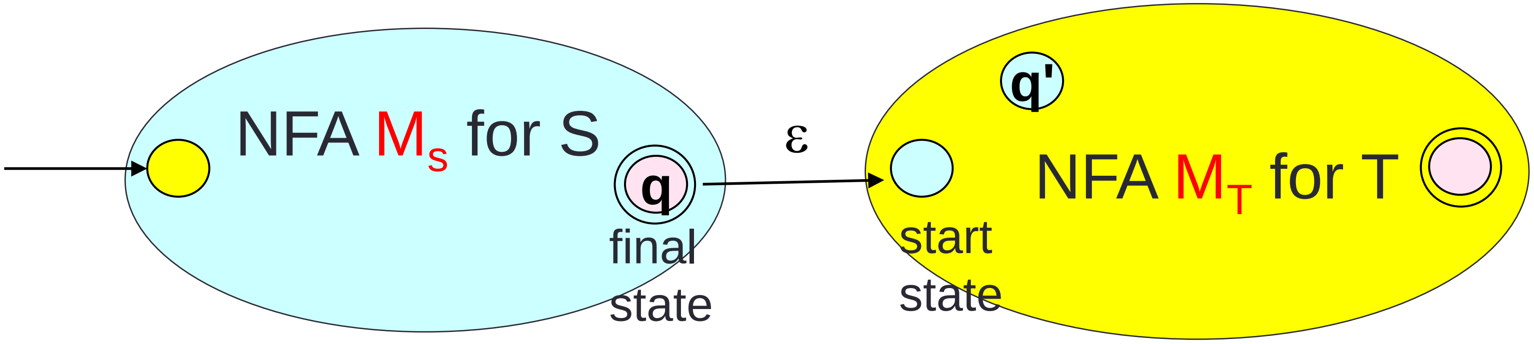
- R = ε:



From regular expressions to NFA: Induction step

Consider a regular expression R. Assume the theorem is true for all sub-expressions of R.

Case 1. R = S T



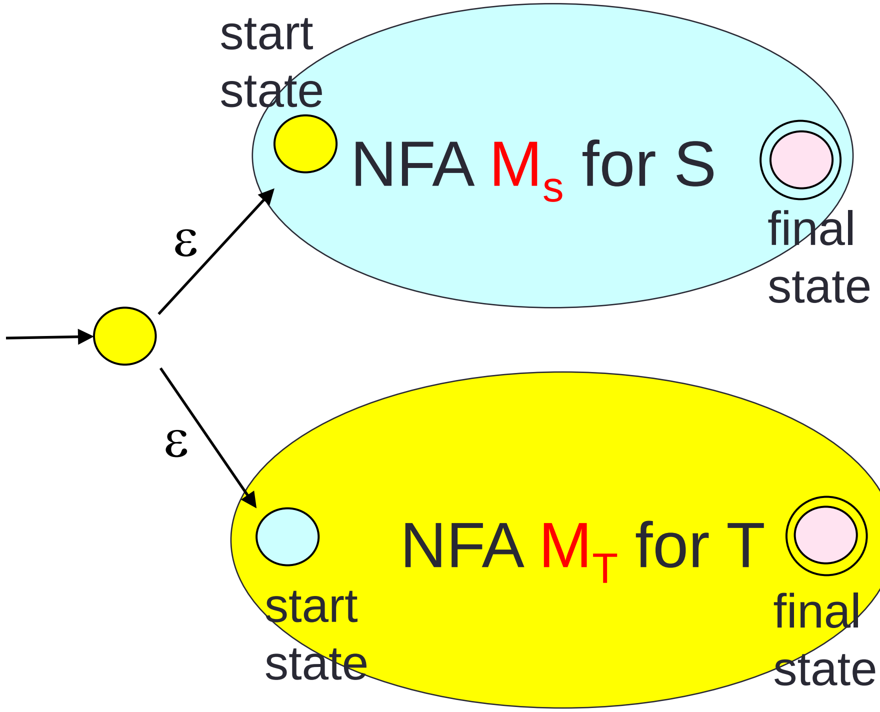
Combine the two NFAs to make a bigger NFA for R.

- For each final state q of M_S, add an ε transition to the start state of M_T.
- New final states: All final states of M_T remain final states.
- What about the final states of M_S?

From regular expressions to NFA: Induction step (cont')

Case 2. $R = S \cup T$

- Create a new start state, which has an ε transition to the start state of M_S and M_T.



- What are the new final states?
- **Ans:** Final states of M_S and M_T.

From regular expressions to NFA: Induction step (cont')

Case 3. R = T*

- Create a new start state, which is also a new final state, and has an ε transition to the original start state.
- Each original final state has an ε transition to the original start state.