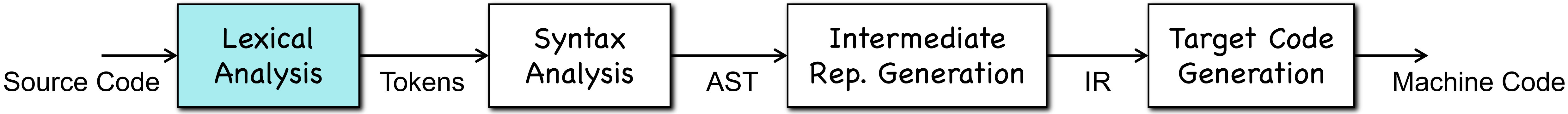
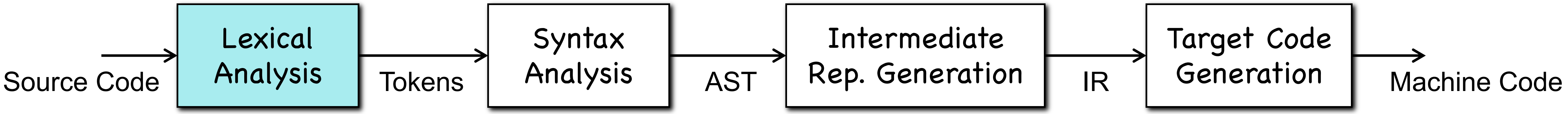
Made by Qingkai Shi
qingkaishi@nju.edu.cn

Chapter 3-1
Finite Automata

Lexical Analysis

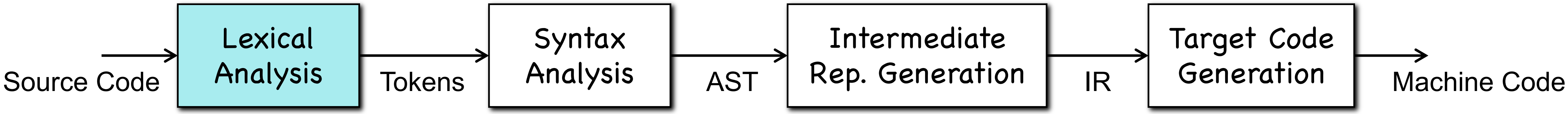


Lexical Analysis

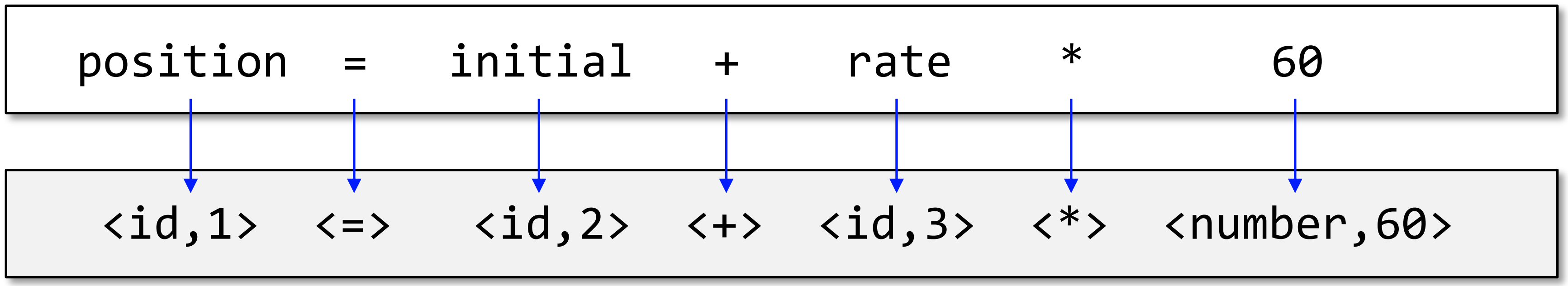


- Find **lexemes** according to **patterns**, and create **tokens**
  - Lexeme – a character string
  - Pattern – regular expression (lexical errors if no patterns matched)
  - Token – <token-class-name, attribute>

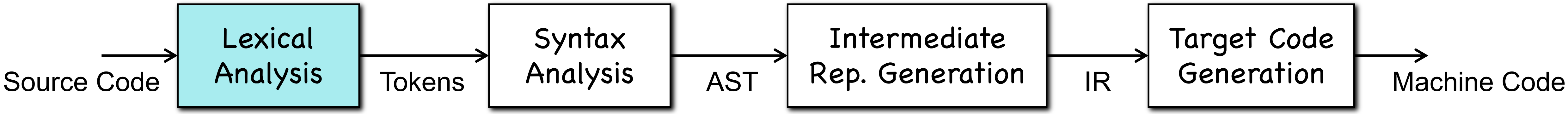
Lexical Analysis



- Find **lexemes** according to **patterns**, and create **tokens**
  - Lexeme – a character string
  - Pattern – regular expression (lexical errors if no patterns matched)
  - Token – <token-class-name, attribute>



Lexical Analysis



- Find **lexemes** according to **patterns**, and create **tokens**
  - Lexeme – a character string
  - Pattern – regular expression (lexical errors if no patterns matched)
  - Token – <token-class-name, attribute>

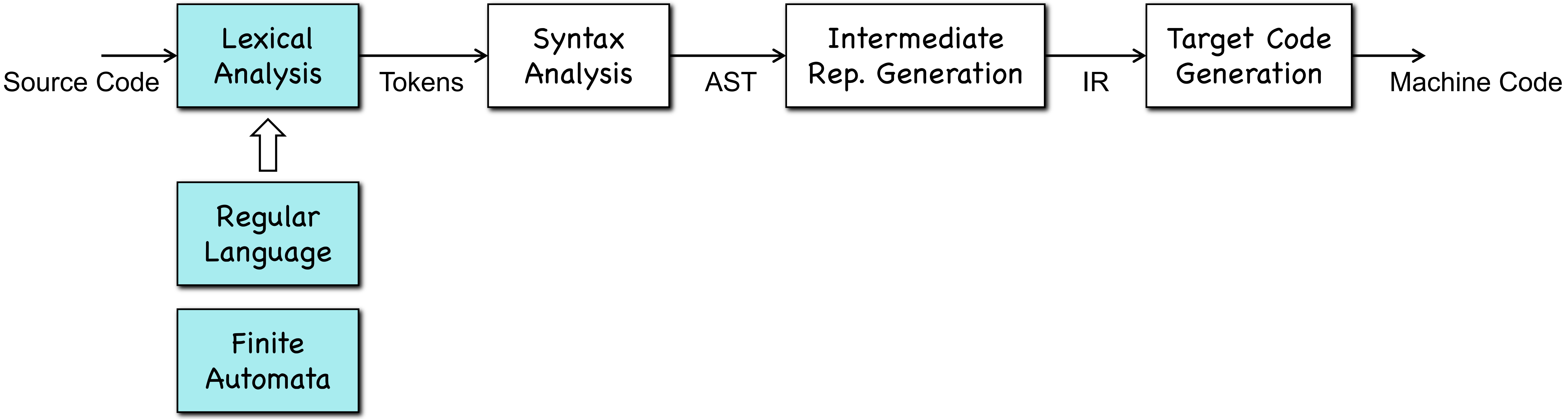
key	name	…
1	position	…
2	initial	…
…		…

symbol table

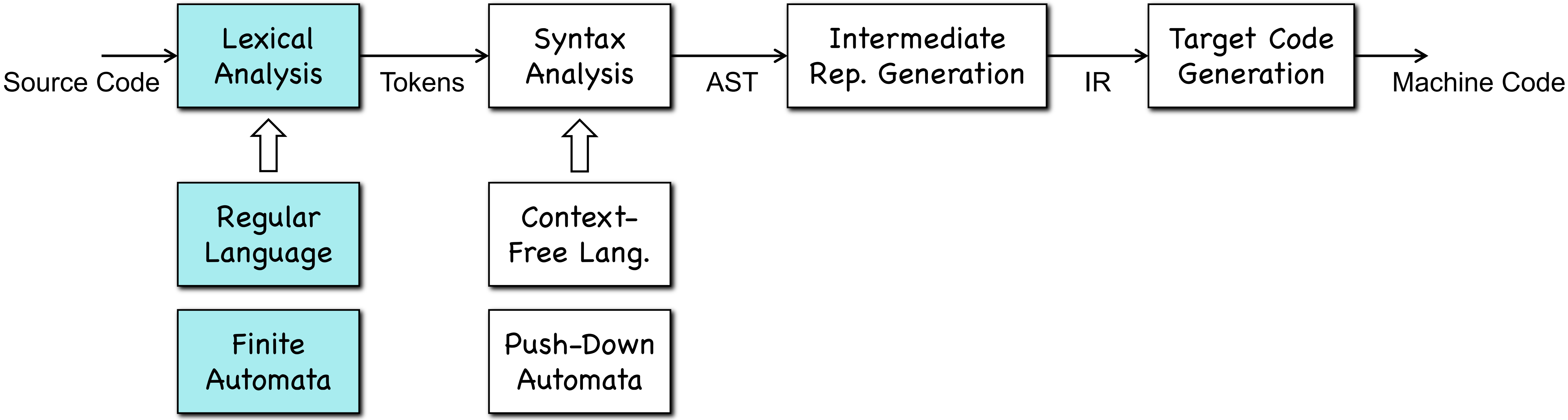
```
position   =   initial   +   rate   *   60

<id,1>  <=>  <id,2>  <+>  <id,3>  <*>  <number,60>
```

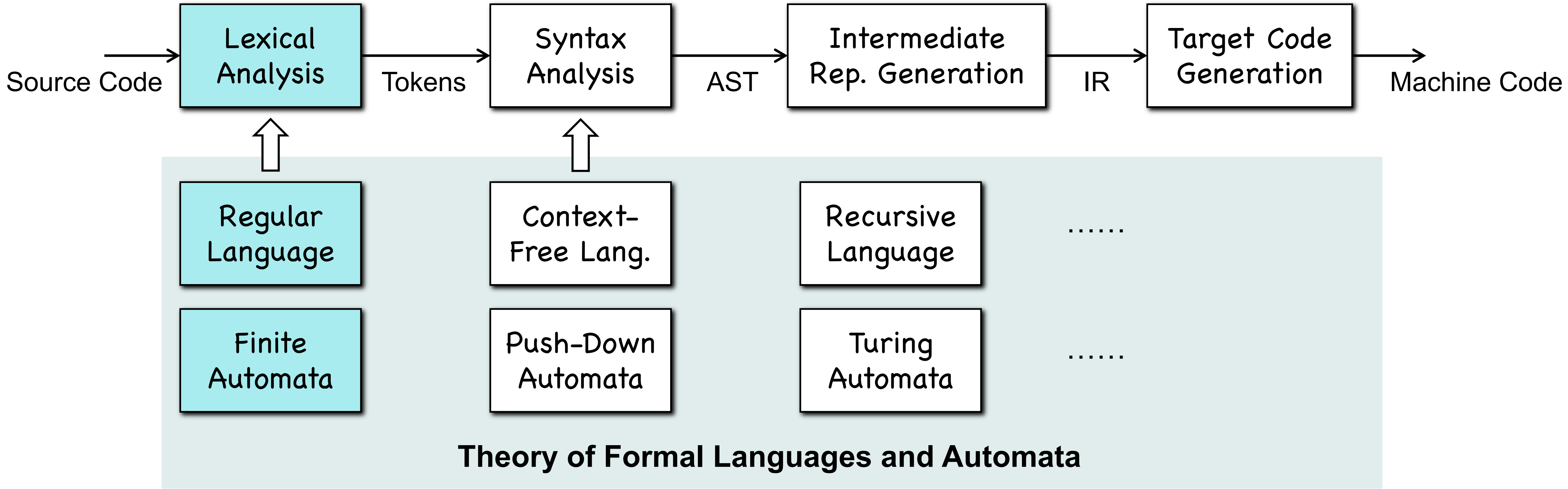
Lexical Analysis



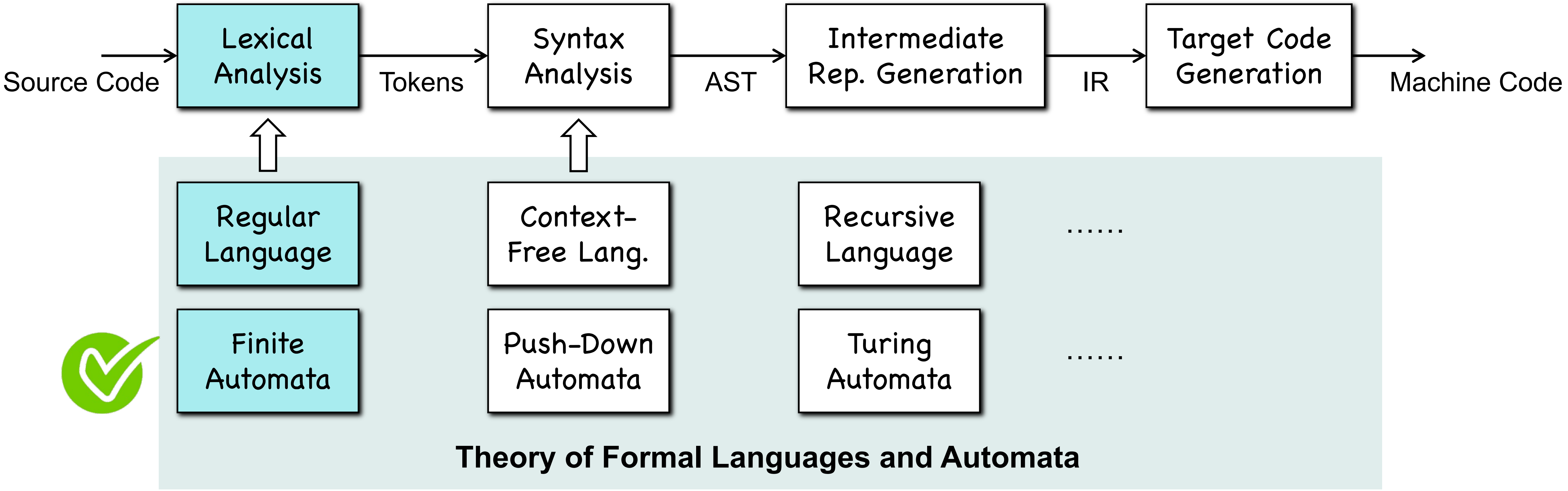
Lexical Analysis



Lexical Analysis



Lexical Analysis



PART I: Math Preliminaries

Alphabet and Strings

- A **language** is a set of strings, e.g., { cat, dog, … }

Alphabet and Strings

- A **language** is a set of strings, e.g., { cat, dog, … }
- A **string** is a sequence of letters defined over an **alphabet**
  - string example: cat, dog, …
  - alphabet example: **Σ** = { a, b, c, d, …, z }

Alphabet and Strings

- A **language** is a set of strings, e.g., { cat, dog, … }
- A **string** is a sequence of letters defined over an **alphabet**
  - string example: cat, dog, …
  - alphabet example: Σ = { a, b, c, d, …, z }

- **Example**: consider a small alphabet $\boldsymbol{\Sigma}$ = { a, b }
  - strings: a, b, ab, aab, aabb, …
  - a language is any subset of { a, b, ab, aab, aabb, … }

String Operations

- Concatenation
  - $w_1 = aabb;\ w_2 = bbaa;$ → $w_1w_2 = aabbbbaa;$

String Operations

- Concatenation
  - $w_1 = aabb;\ w_2 = bbaa;$ → $w_1 w_2 = aabbbbaa;$
- Reverse
  - $w = a_1 a_2 a_3 a_4;$ → $w^R = a_4 a_3 a_2 a_1;$

String Operations

- Concatenation
  - $w_1 = aabb;\ w_2 = bbaa;$ → $w_1w_2 = aabbbbaa;$
- Reverse
  - $w = a_1a_2a_3a_4;$ → $w^R = a_4a_3a_2a_1;$
- Length
  - $w = a_1a_2a_3a_4;$ → $|w| = 4;$

Empty String and Sub-String

- **Empty String**: ϵ
  - $|\epsilon| = 0$
  - $\epsilon aabb = aab\epsilon\epsilon b = aabb\epsilon = aabb$

Empty String and Sub-String

- **Empty String**: ϵ
  - $|\epsilon| = 0$
  - $\epsilon aabb = aab\epsilon\epsilon b = aabb\epsilon = aabb$

- **Substring**: a subsequence of consecutive characters
  - $abbab, abbab, abbab, abbab$

Empty String and Sub-String

- **Empty String**: ϵ
  - $|\epsilon| = 0$
  - $\epsilon aabb = aab\epsilon\epsilon b = aabb\epsilon = aabb$

- **Substring**: a subsequence of consecutive characters
  - $abbab, abbab, abbab, abbab$

- **Prefix and Suffix**: $w = uv$, where u is prefix and v is suffix
  - prefix of abb includes: ϵ, a, ab, abb
  - suffix of abb includes: abb, bb, b, ϵ

Power, Kleene Star, and Plus

- **Power**: $w^n = \underbrace{ww \ldots w}_{n}$; $w^0 = \epsilon$;

Power, Kleene Star, and Plus

- **Power**: $w^n = \underbrace{ww \dots w}_{n}$; $w^0 = \epsilon$;
  - $(abb)^2 = abbabb$
  - $(abb)^0 = \epsilon$

Power, Kleene Star, and Plus

- **Power**: $w^n = \underbrace{ww \ldots w}_{n}$; $w^0 = \epsilon$;
  - $(abb)^2 = abbabb$
  - $(abb)^0 = \epsilon$
- **Kleene Star**: $\Sigma = \{\, a, b \,\} \Rightarrow \Sigma^* = \{\, \epsilon, a, b, ab, abb, aab, \ldots \,\}$

Power, Kleene Star, and Plus

- **Power**: $w^n = \underbrace{ww \dots w}_{n}$; $w^0 = \epsilon$;
  - $(abb)^2 = abbabb$
  - $(abb)^0 = \epsilon$
- **Kleene Star**: $\Sigma = \{ a, b \} \Rightarrow \Sigma^* = \{ \epsilon, a, b, ab, abb, aab, \dots \}$
- **Plus**: $\Sigma^+ = \Sigma^* - \{\epsilon\}$

Power, Kleene Star, and Plus

- **Power**: $w^n = \underbrace{ww \ldots w}_{n}$; $w^0 = \epsilon$;
  - $(abb)^2 = abbabb$
  - $(abb)^0 = \epsilon$
- **Kleene Star**: $\Sigma = \{a, b\} \Rightarrow \Sigma^* = \{\epsilon, a, b, ab, abb, aab, \ldots\}$ …
- **Plus**: $\Sigma^+ = \Sigma^* - \{\epsilon\}$
- **Language:** a language is any subset of $\boldsymbol{\Sigma}^*$, e.g., $\{\epsilon\}$, $\{\epsilon, a, b\}$, …

Power, Kleene Star, and Plus

- **Power**: $w^n = \underbrace{ww \dots w}_{n}$; $w^0 = \epsilon$;
  - $(abb)^2 = abbabb$
  - $(abb)^0 = \epsilon$
- **Kleene Star**: $\Sigma = \{ a, b \} \Rightarrow \Sigma^* = \{ \epsilon, a, b, ab, abb, aab, \dots \} \dots$
- **Plus**: $\Sigma^+ = \Sigma^* - \{\epsilon\}$

Not ∅ !!!

- **Language:** a language is any subset of $\boldsymbol{\Sigma}^*$, e.g., $\{\epsilon\}$, $\{\epsilon, a, b\}$, …

Operations on Languages

- Usual set operations
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 \cup L_2 = \{a, b, ab\}$
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 \cap L_2 = \{a\}$
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 - L_2 = \{b\}$

Operations on Languages

- Usual set operations
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 \cup L_2 = \{a, b, ab\}$
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 \cap L_2 = \{a\}$
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 - L_2 = \{b\}$

- Complement: $\bar{L} = \Sigma^* - L$

Operations on Languages

- Usual set operations
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 \cup L_2 = \{a, b, ab\}$
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 \cap L_2 = \{a\}$
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 - L_2 = \{b\}$

- Complement: $\bar{L} = \Sigma^* - L$
- Reverse: $L^R = \{w^R : w \in L\}$

Operations on Languages

- Usual set operations
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 \cup L_2 = \{a, b, ab\}$
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 \cap L_2 = \{a\}$
  - $L_1 = \{a, b\}; L_2 = \{a, ab\}; \Rightarrow L_1 - L_2 = \{b\}$
- Complement: $\overline{L} = \Sigma^* - L$
- Reverse: $L^R = \{w^R : w \in L\}$
- Concatenation: $L_1 L_2 = \{xy : x \in L_1, y \in L_2\}$

Operations on Languages

- **Power**: $L^n = \underbrace{LL \dots \dots L}_{n}$; $L^0 = \{\epsilon\}$

Operations on Languages

- **Power**: $L^n = \underbrace{LL \ldots \ldots L}_{n}$; $L^0 = \{\epsilon\}$

- **Star-Closure**: $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$

Operations on Languages

- **Power**: $L^n = \underbrace{LL \dots \dots L}_{n}$; $L^0 = \{\epsilon\}$

- **Star-Closure**: $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$
- **Positive-Closure**: $L^+ = L^1 \cup L^2 \cup \cdots = L^* - \{\epsilon\}$

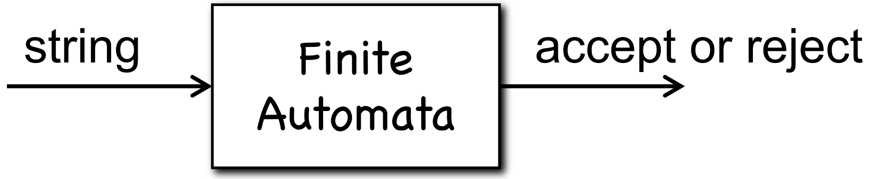
Operations on Languages

- **Power**: $L^n = \underbrace{LL \dots \dots L}_{n}$; $\quad L^0 = \{\epsilon\}$
- **Star-Closure**: $L^* = L^0 \cup L^1 \cup L^2 \cup \cdots$
- **Positive-Closure**: $L^+ = L^1 \cup L^2 \cup \cdots = L^* - \{\epsilon\}$
- **Quiz 1**: $L = \{a, b\}$; $\ L^3 = ?$
- **Quiz 2**: $L = \{a^n b^n : n \geq 0\}$; $L^2 = ?$

PART II:
Deterministic Finite Automata

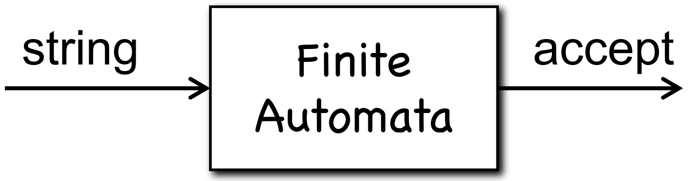
Finite Automata

- Input a string, output “accept” or “reject”

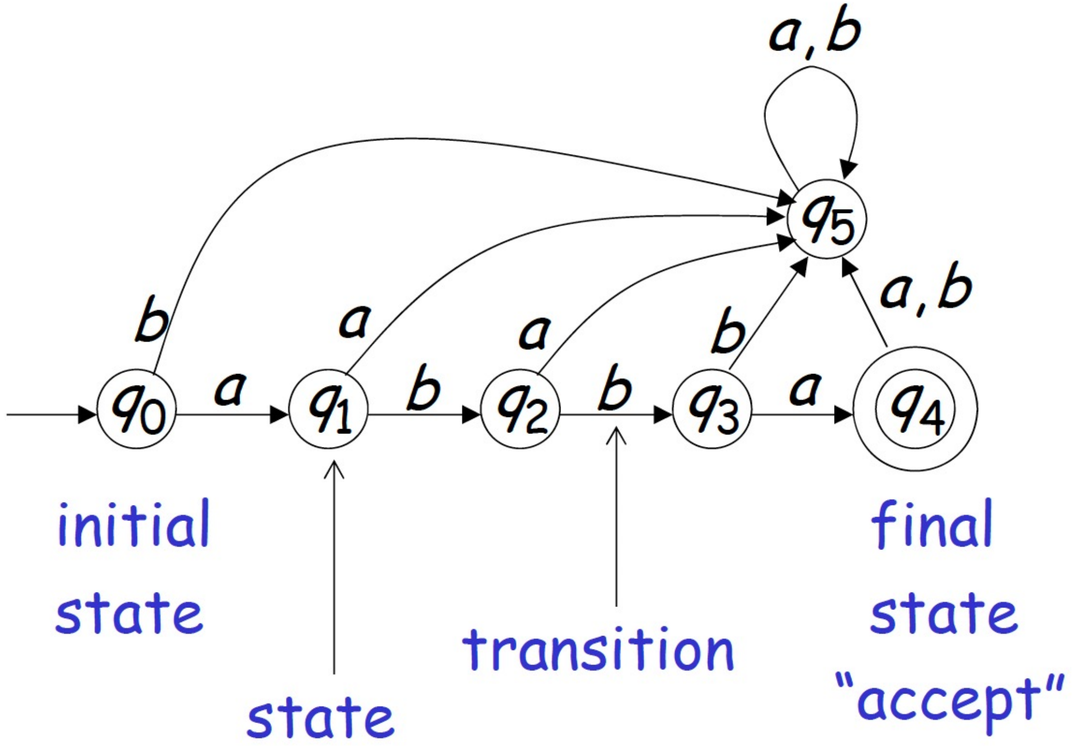


Finite Automata

- Input a string, output “accept” or “reject”

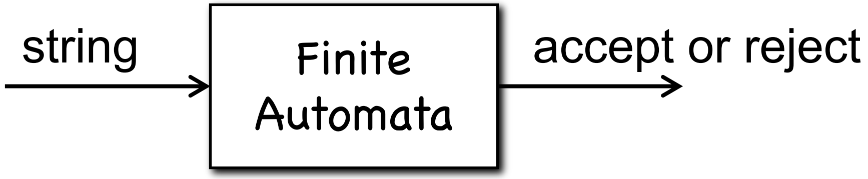


- **Example**: finite automata for *abba*

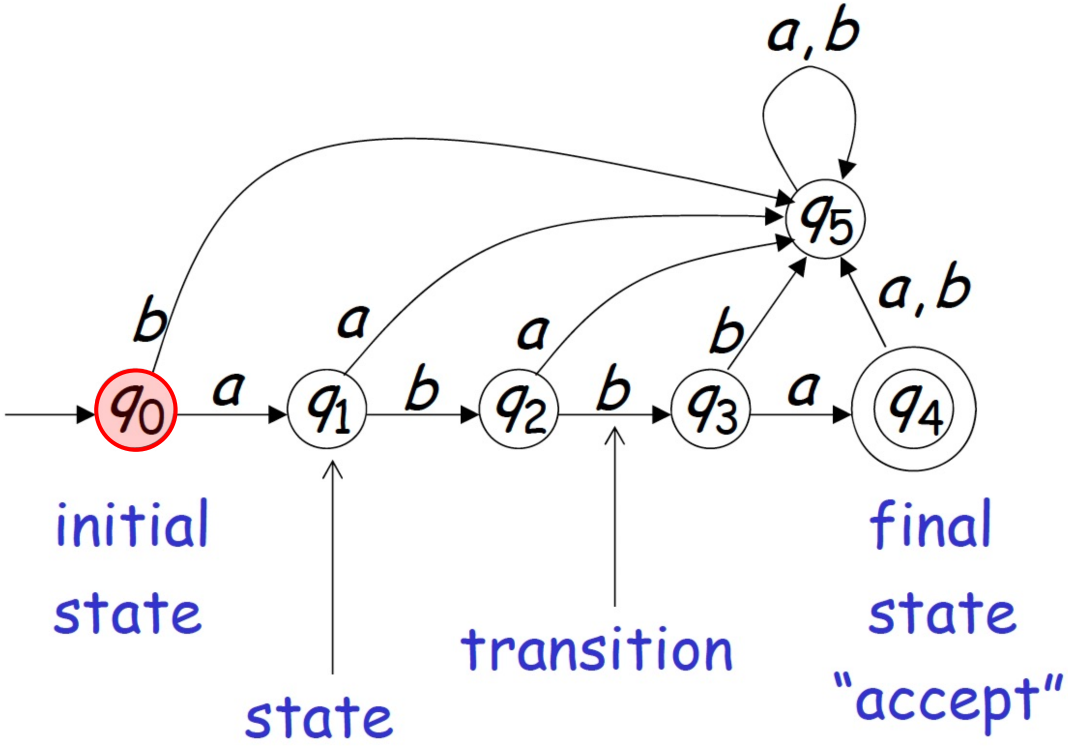


Finite Automata

- Input a string, output “accept” or “reject”

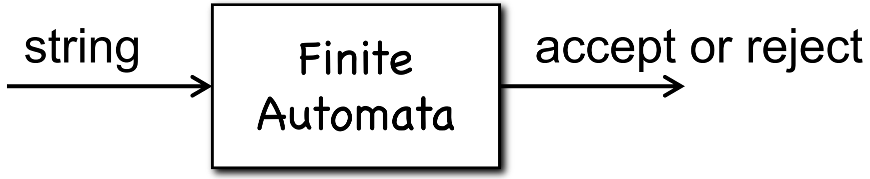


- **Example**: finite automata for *abba*

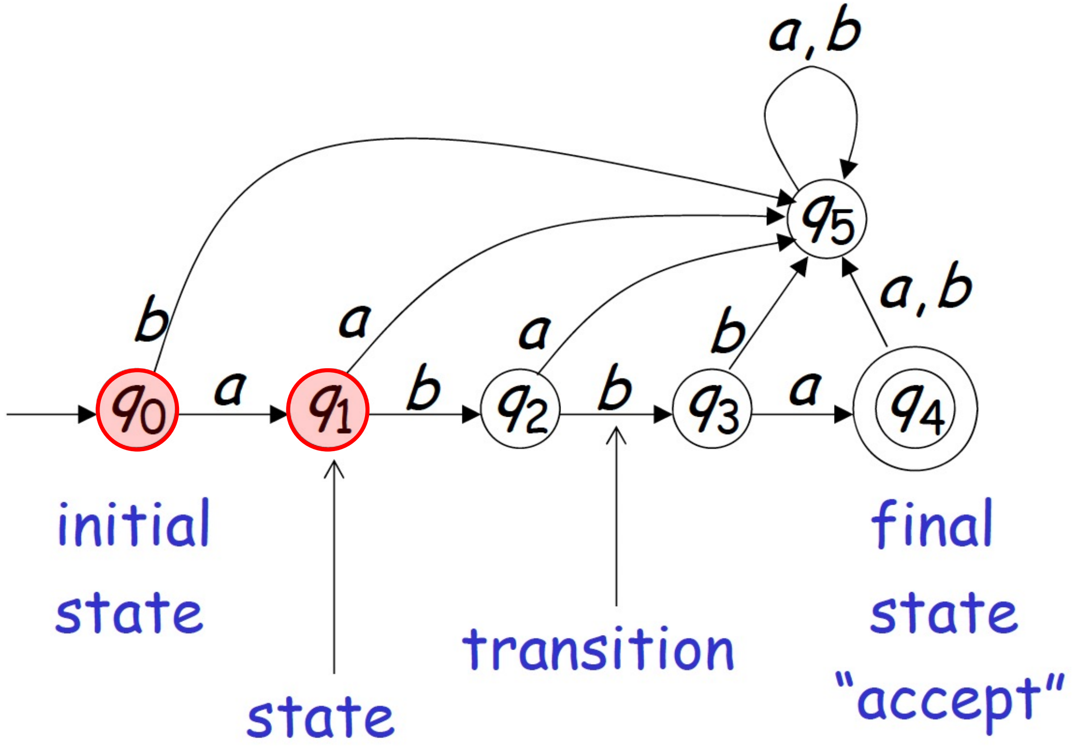


Finite Automata

- Input a string, output “accept” or “reject”

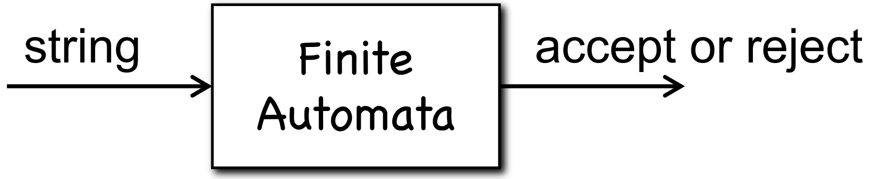


- **Example**: finite automata for *abba*

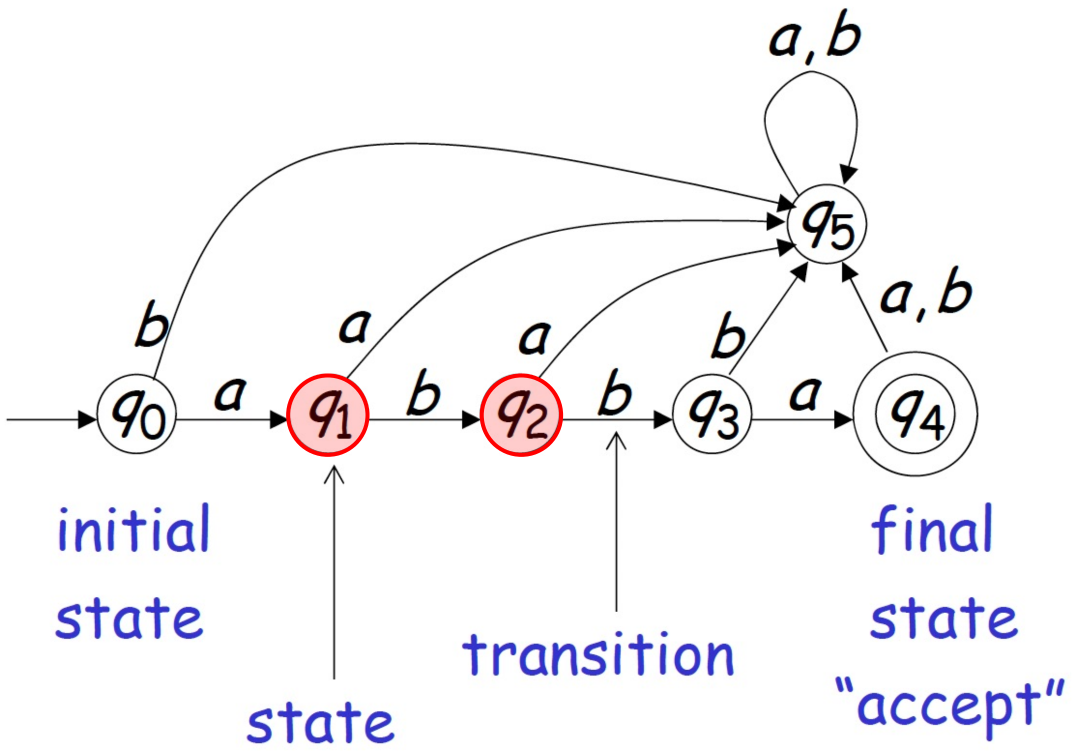


Finite Automata

- Input a string, output "accept" or "reject"

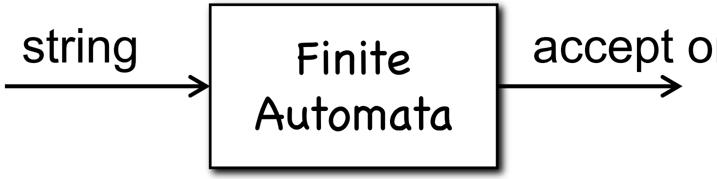


- **Example**: finite automata for *abba*

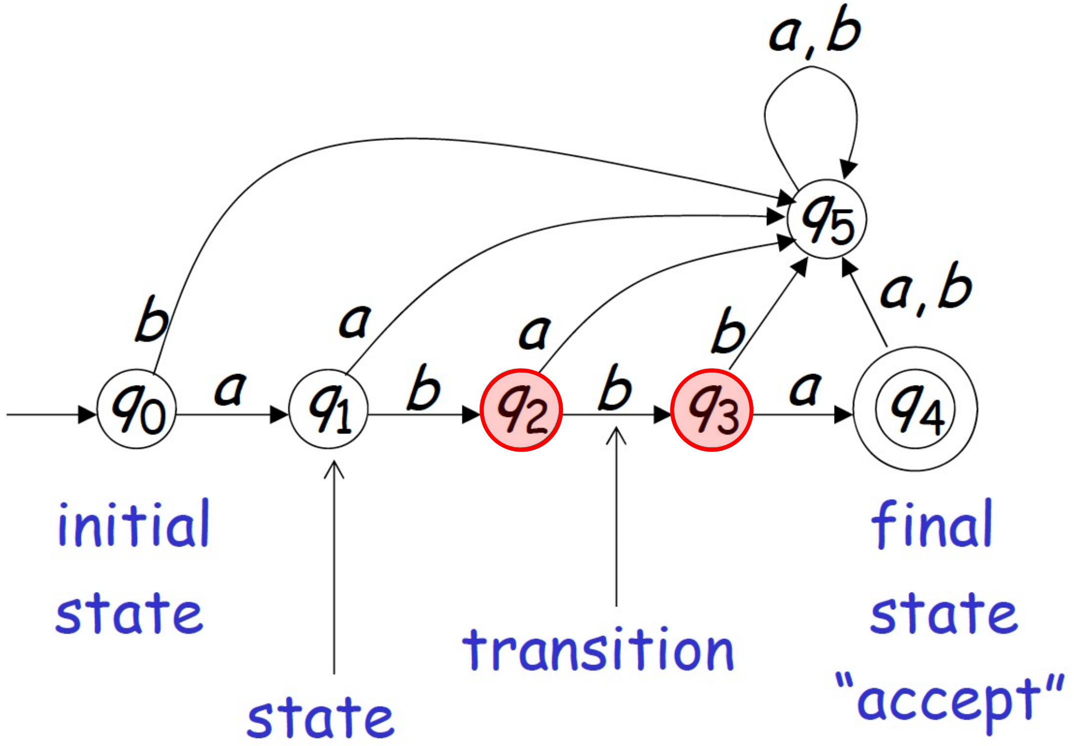


Finite Automata

- Input a string, output "accept" or "reject"

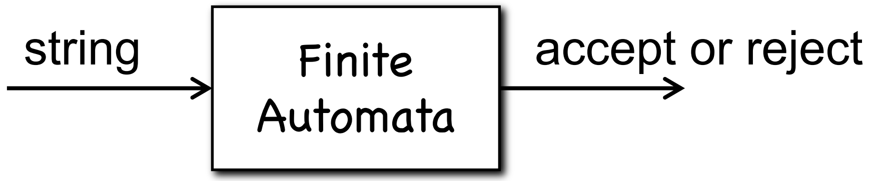


- **Example**: finite automata for *abba*

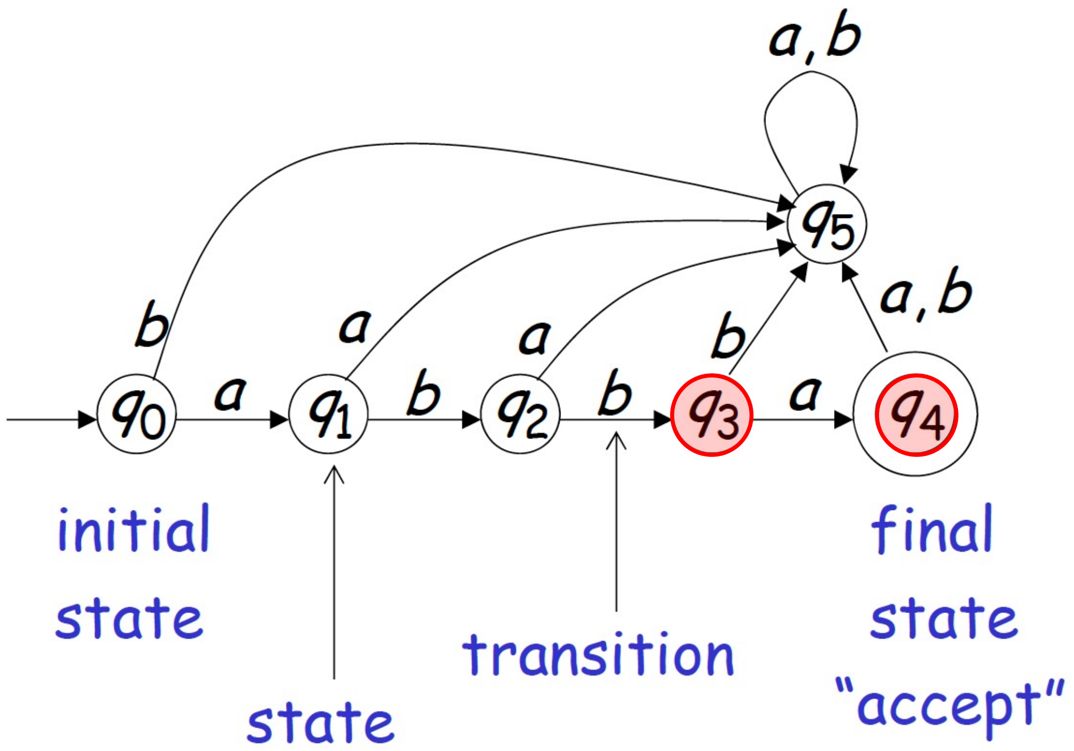


Finite Automata

- Input a string, output “accept” or “reject”

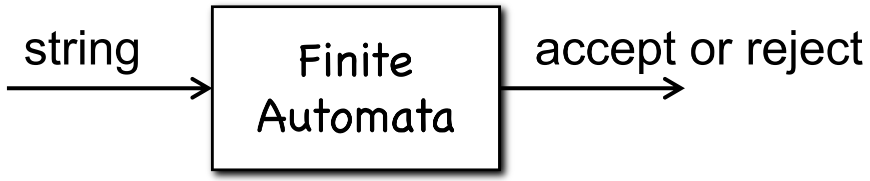


- **Example**: finite automata for *abba*

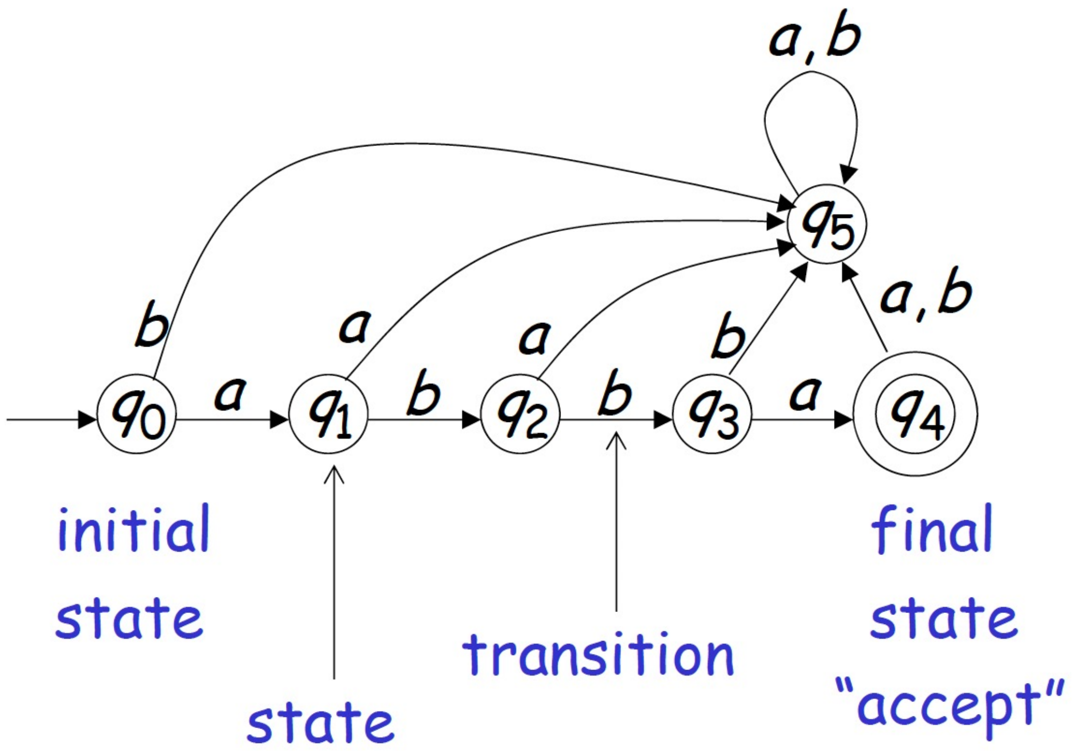


Finite Automata

- Input a string, output “accept” or “reject”



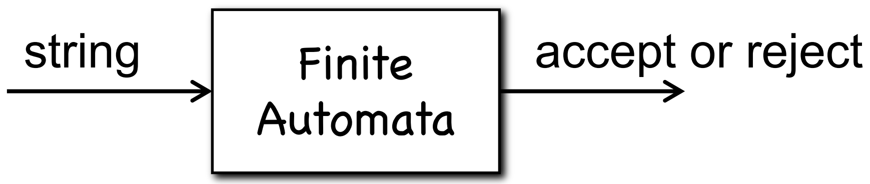
- **Example**: finite automata for *abba*



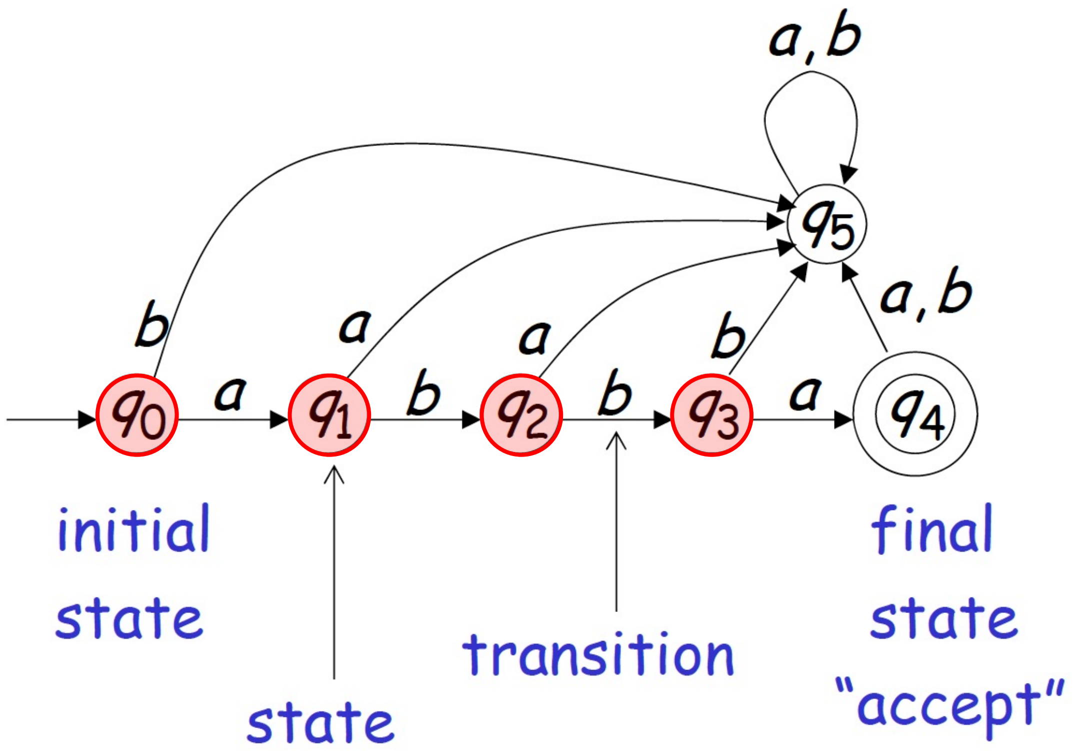
- *What if we input “abb”?*

Finite Automata

- Input a string, output “accept” or “reject”

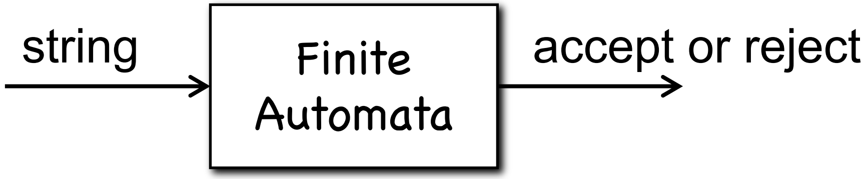


- **Example**: finite automata for *abba*
- *What if we input “abb”?*

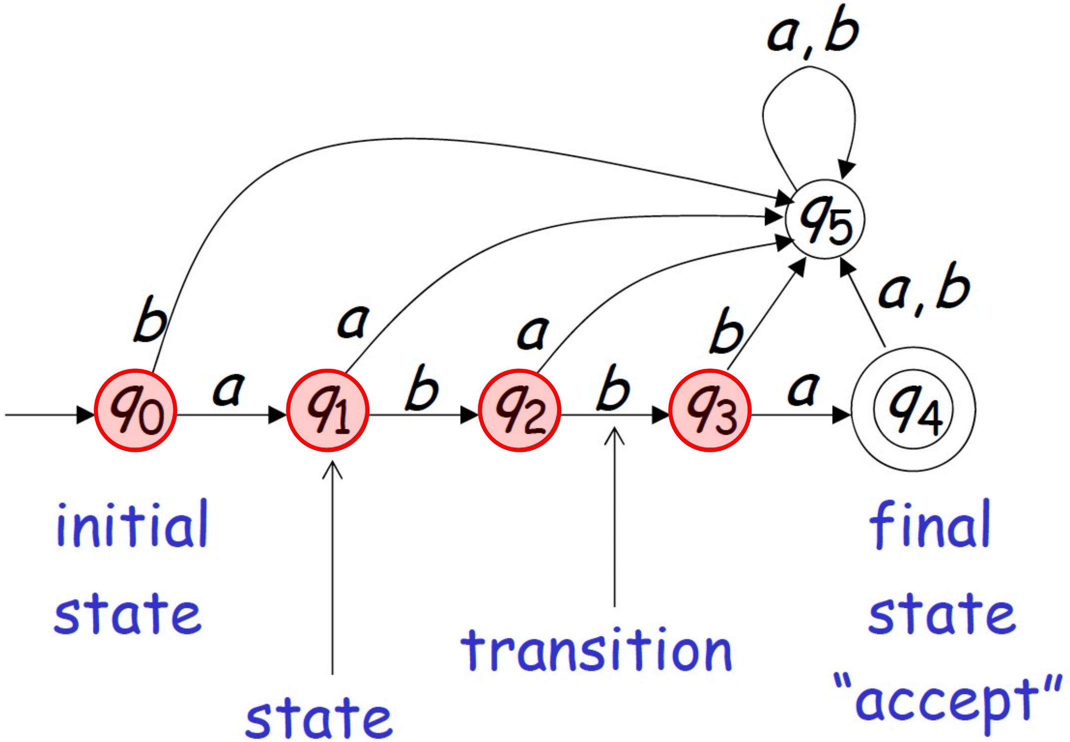


Finite Automata

- Input a string, output “accept” or “reject”



- **Example**: finite automata for *abba*
- *What if we input “abb”?*
- **Deterministic** Finite Automata!



Deterministic Finite Automata

- A DFA is a five-tuple: $(Q, \Sigma, \delta, q_0, F)$
  - Q: A finite set of states

Deterministic Finite Automata

- A DFA is a five-tuple: $(Q, \Sigma, \delta, q_0, F)$
  - Q: A finite set of states
  - Σ: A finite set of input characters, i.e., an alphabet

Deterministic Finite Automata

- A DFA is a five-tuple: $(Q, \Sigma, \delta, q_0, F)$
  - Q: A finite set of states
  - Σ: A finite set of input characters, i.e., an alphabet
  - $\delta: Q \times \Sigma \longmapsto$ Q: Transition function, e.g., $\delta(q, a) = q'$

Deterministic Finite Automata

- A DFA is a five-tuple: $(Q, \Sigma, \delta, q_0, F)$
  - Q: A finite set of states
  - Σ: A finite set of input characters, i.e., an alphabet
  - $\delta: Q \times \Sigma \longmapsto$ Q: Transition function, e.g., $\delta(q, a) = q'$
  - $q_0 \in Q$: The start state

Deterministic Finite Automata

- A DFA is a five-tuple: $(Q, \Sigma, \delta, q_0, F)$
  - Q: A finite set of states
  - Σ: A finite set of input characters, i.e., an alphabet
  - $\delta: Q \times \Sigma \longmapsto$ Q: Transition function, e.g., $\delta(q, a) = q'$
  - $q_0 \in Q$: The start state
  - $F \subseteq Q$: A finite subset of final states

Deterministic Finite Automata

- A DFA is a five-tuple: $(Q, \Sigma, \delta, q_0, F)$
  - Q: A finite set of states
  - Σ: A finite set of input characters, i.e., an alphabet
  - $\delta: Q \times \Sigma \longmapsto$ Q: Transition function, e.g., $\delta(q, a) = q'$
  - $q_0 \in Q$: The start state
  - $F \subseteq Q$: A finite subset of final states

- An extension of transition: $\delta^*: Q \times \Sigma^* \longmapsto$ Q
  - $\delta^*(q, abba) = q'; \quad \delta^*(q, \epsilon) = q;$

Defining a Language by DFA

- Take a DFA $M = (Q, \Sigma, \delta, q_0, F)$, language can be accepted by the DFA is written as $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$.

Defining a Language by DFA

- Take a DFA $M = (Q, \Sigma, \delta, q_0, F)$, language can be accepted by the DFA is written as $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$.
- **Common application**: checking if a string $w \in L(M)$.

Defining a Language by DFA

- Take a DFA $M = (Q, \Sigma, \delta, q_0, F)$, language can be accepted by the DFA is written as $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$.
- **Common application**: checking if a string $w \in L(M)$.

```
void check(w, M) {
    q = q0;
    while (true) {
        c = read(w); if (c == None) { print(q ∊ F ? “accept” : “reject”); }
    
    
    
    
    
    
    }}
```

Defining a Language by DFA

- Take a DFA $M = (Q, \Sigma, \delta, q_0, F)$, language can be accepted by the DFA is written as $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$.
- **Common application**: checking if a string $w \in L(M)$.

```
void check(w, M) {
    q = q₀;
    while (true) {
        c = read(w); if (c == None) { print(q ∊ F ? “accept” : “reject”); }
        switch(q) {
        case q₀:
        case q₁:
        case q₂:
        case ……
        }
    }}
```

Defining a Language by DFA

- Take a DFA $M = (Q, \Sigma, \delta, q_0, F)$, language can be accepted by the DFA is written as $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$.
- **Common application**: checking if a string $w \in L(M)$.

```
void check(w, M) {
    q = q0;
    while (true) {
        c = read(w); if (c == None) { print(q ∊ F ? "accept" : "reject"); }
        switch(q) {
        case q0: if (c == 'a') { q = q1; } break; // δ(q0,a)=q1; δ(q0,!a)=q0
        case q1:
        case q2:
        case ……
        }
    }}
```

Defining a Language by DFA

- Take a DFA $M = (Q, \Sigma, \delta, q_0, F)$, language can be accepted by the DFA is written as $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$
- **Common application**: checking if a string $w \in L(M)$

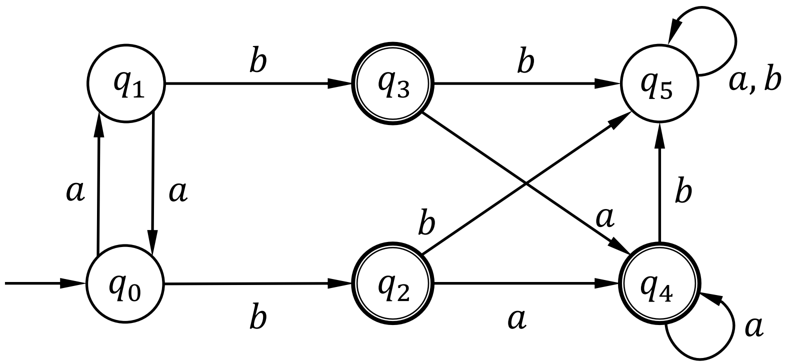
```
void check(w, M) {
    q = q0;
    while (true) {
        c = read(w); if (c == None) { print(q ∈ F ? "accept" : "reject"); }
        switch(q) {
        case q0: if (c == 'a') { q = q1; } break; // δ(q0,a)=q1; δ(q0,!a)=q0
        case q1: … break;
        case q2: … break;
        case ……
        }
    }}
```

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

DFA Minimization

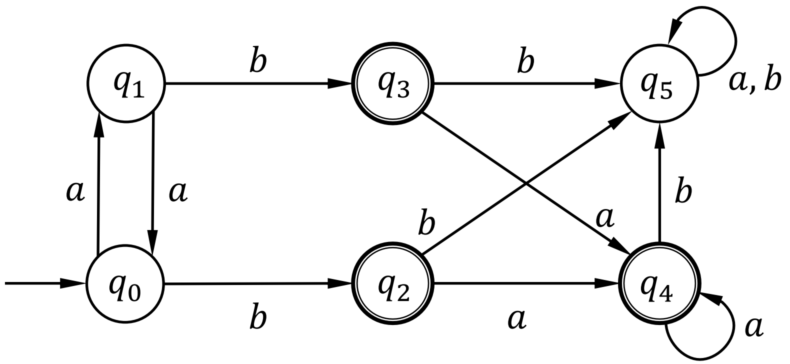
- Minimizing DFA can improve the efficiency of computation



	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

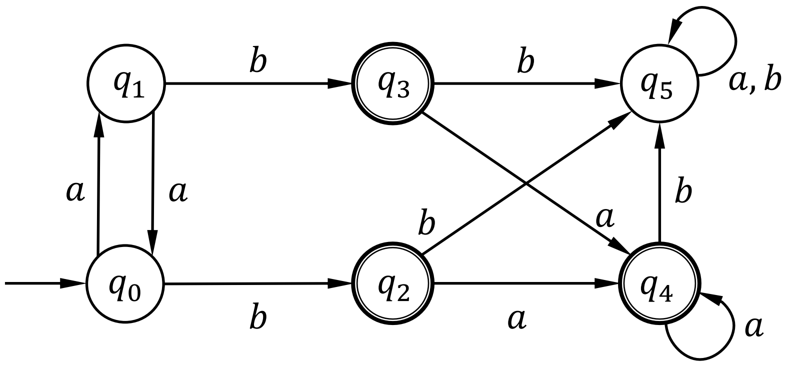


	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2		
Step 3		

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

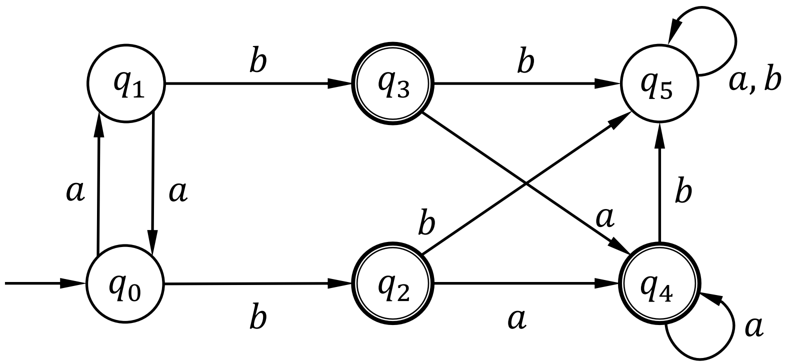


	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{ q_0, q_1$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set
Step 3		

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

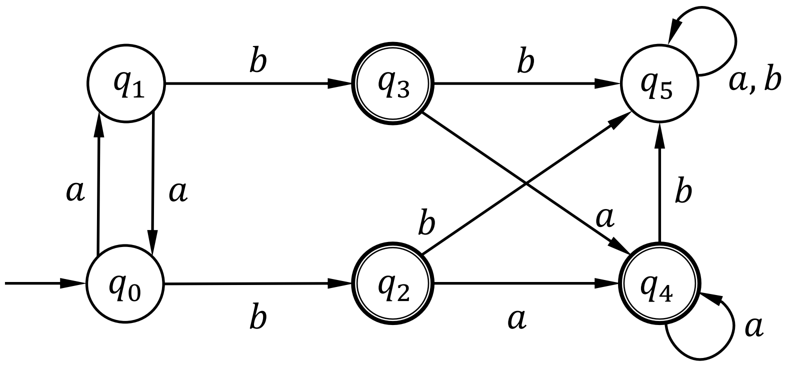


	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{ q_0, q_1 \}, \{ q_5 \}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3		

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

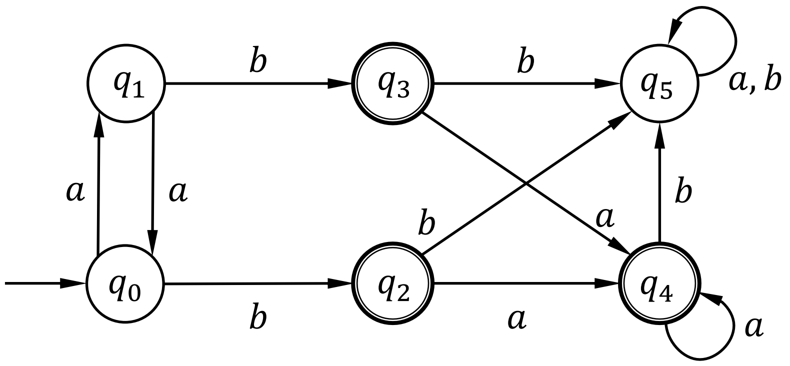


	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{ q_0, q_1, q_5 \}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{ q_0, q_1 \}, \{ q_5 \}, \{q_2, q_3, q_4\}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3		

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

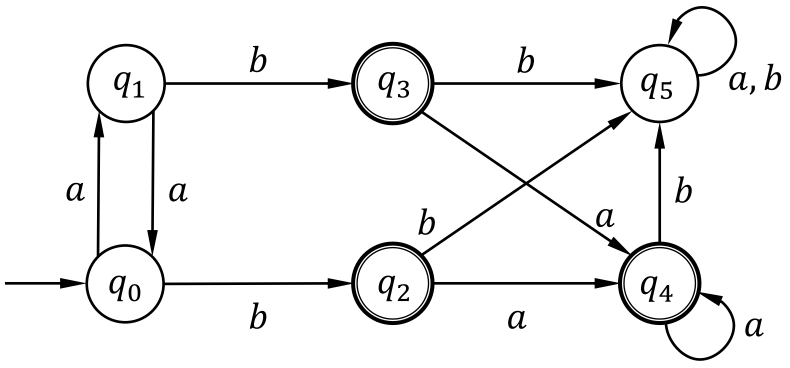


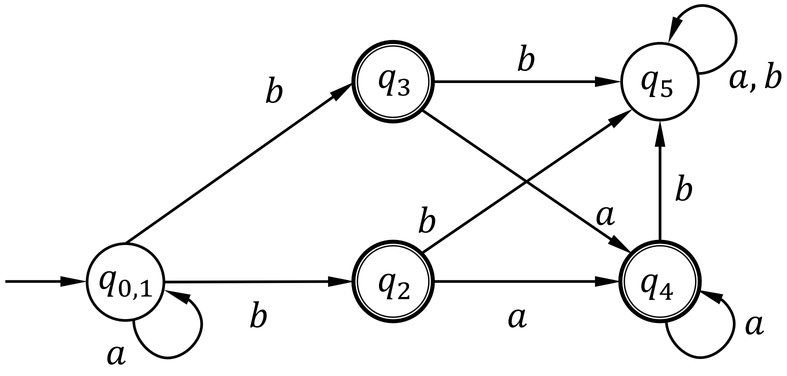
	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_4	q_5
q_3	q_4	q_5
q_4	q_4	q_5
q_5	q_5	q_5

Step 1	$\{q_0, q_1, q_5\}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{q_0, q_1\}, \{q_5\}, \{q_2, q_3, q_4\}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3	$\{q_0, q_1\}, \{q_5\}, \{q_2, q_3, q_4\}$	The result does not change and the algorithm completes

DFA Minimization

- Minimizing DFA can improve the efficiency of computation

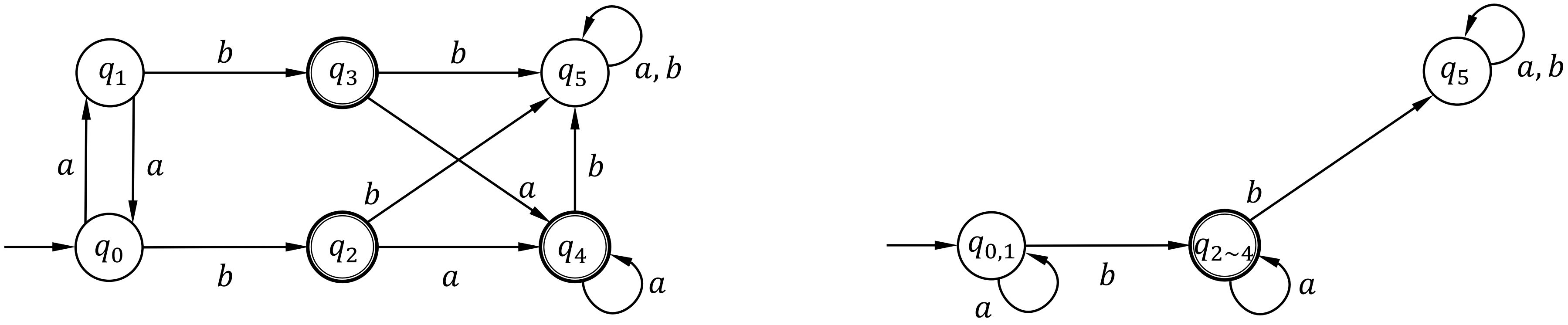




Step 1	$\{q_0, q_1, q_5\}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{q_0, q_1\}, \{q_5\}, \{q_2, q_3, q_4\}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3	$\{q_0, q_1\}, \{q_5\}, \{q_2, q_3, q_4\}$	The result does not change and the algorithm completes

DFA Minimization

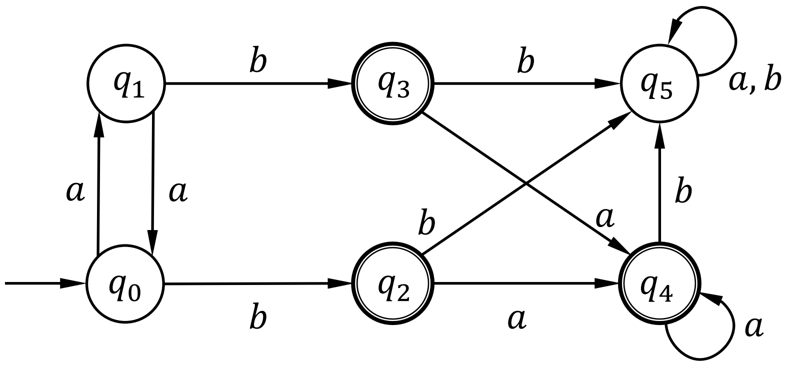
- Minimizing DFA can improve the efficiency of computation

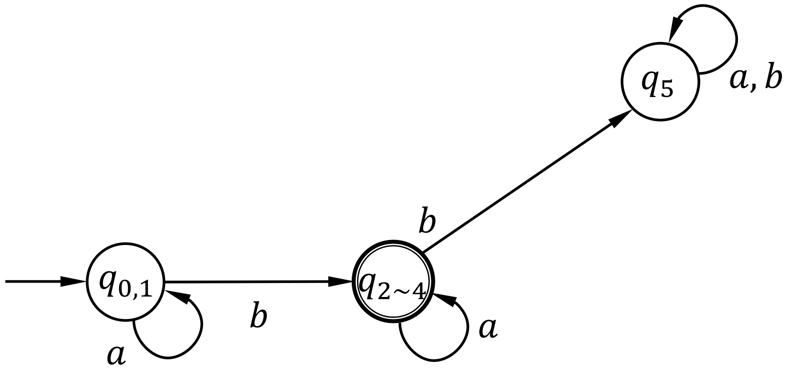


Step 1	$\{q_0, q_1, q_5\}, \{q_2, q_3, q_4\}$	Distinguish final and non-final states
Step 2	$\{q_0, q_1\}, \{q_5\}, \{q_2, q_3, q_4\}$	$\delta(q_0, a/b)$ and $\delta(q_1, a/b)$ are in the same set, but $\delta(q_5, a/b)$ not
Step 3	$\{q_0, q_1\}, \{q_5\}, \{q_2, q_3, q_4\}$	The result does not change and the algorithm completes

DFA Minimization

- Minimizing DFA can improve the efficiency of computation





- **We remove all unreachable states before the above steps**

DFA Minimization

- Best Average Complexity: $O(n \log \log n)$!

ELSEVIER

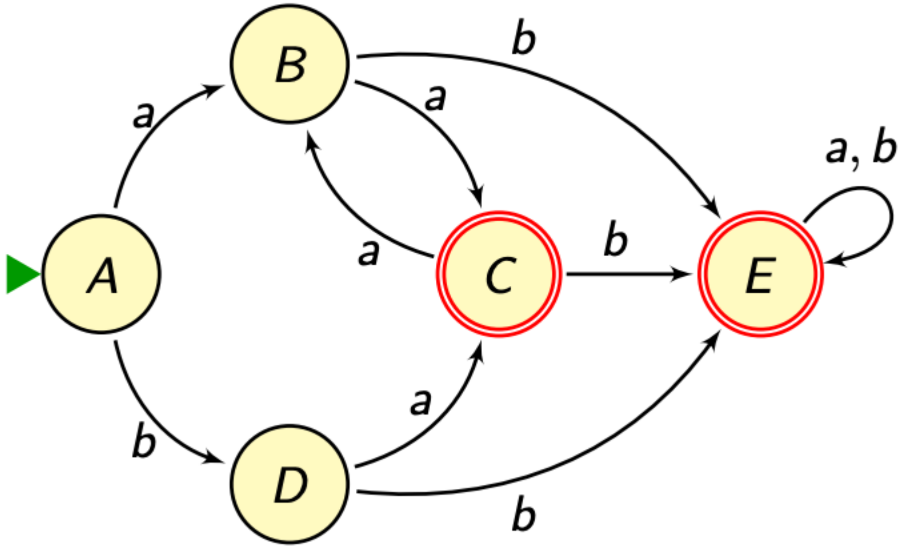
Theoretical Computer Science

Volume 417, 3 February 2012, Pages 50-65

Average complexity of Moore's and Hopcroft's algorithms

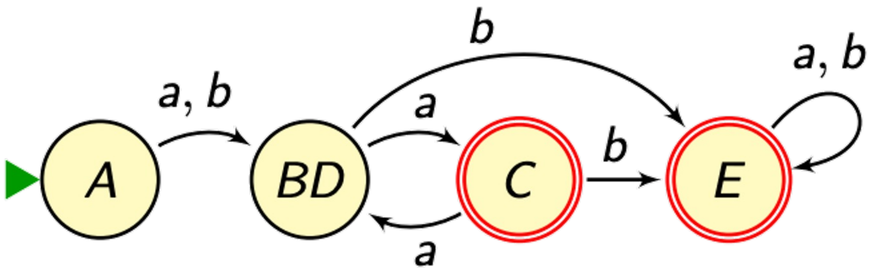
Julien David

DFA Minimization



Have a Try!!

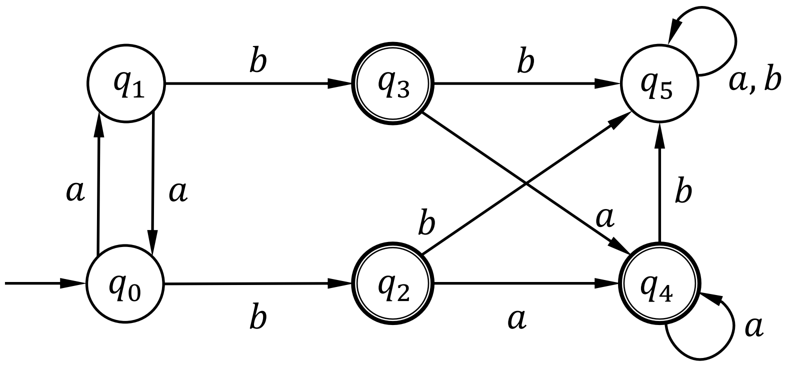
DFA Minimization

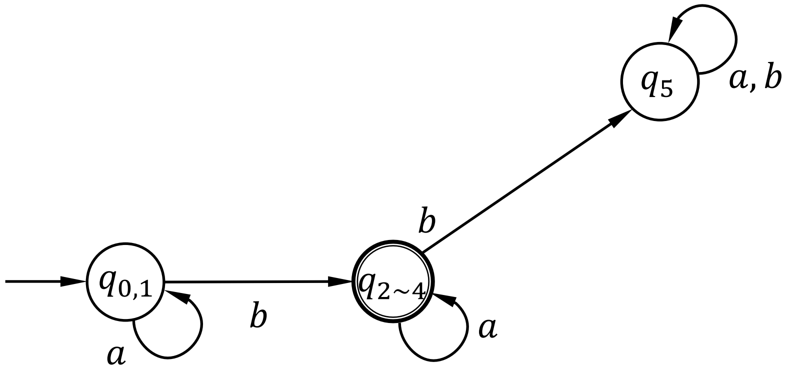


Solution

DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$





DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$
  - $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$

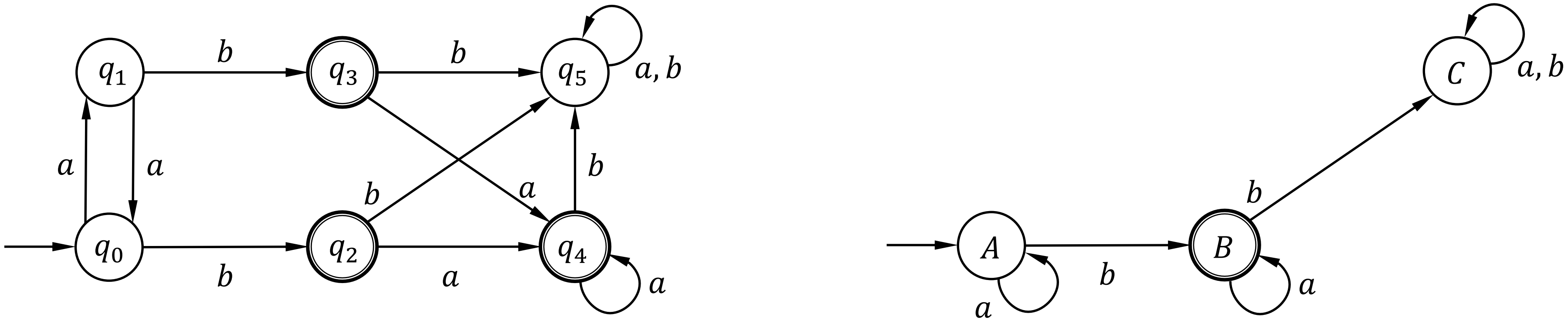
DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$
  - $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$

- Given an input string, let M_1 and M_2 evaluate it at the same time, M_1 reaches a final state if and only if M_2 reaches a final state

DFA Bi-Simulation

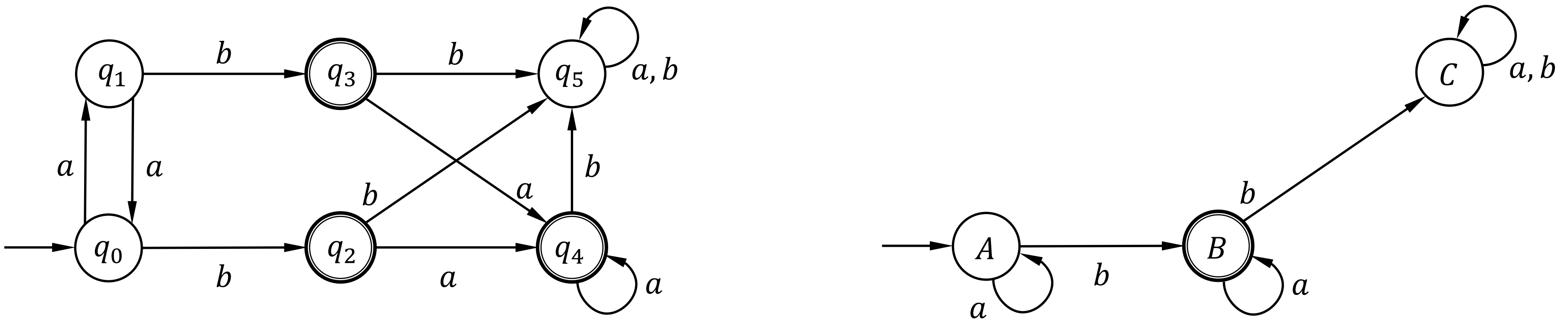
- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$



State Pairs	a	b	
$\{ q_0, A \}$			
			M_1 reaches final state if and only if M_2 reaches final state

DFA Bi-Simulation

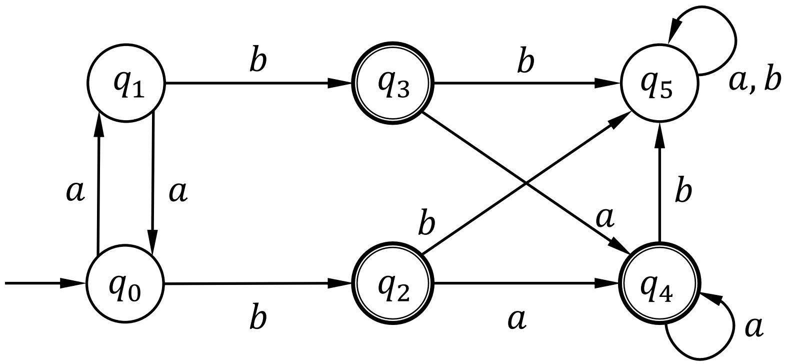
- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$

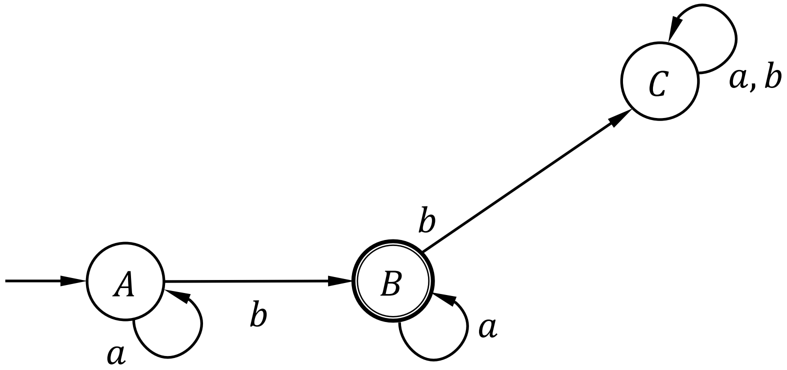


State Pairs	a	b	
$\{ q_0, A \}$	$\{ q_1, A \}$	$\{ q_2, B \}$	
			M_1 reaches final state if and only if M_2 reaches final state

DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$

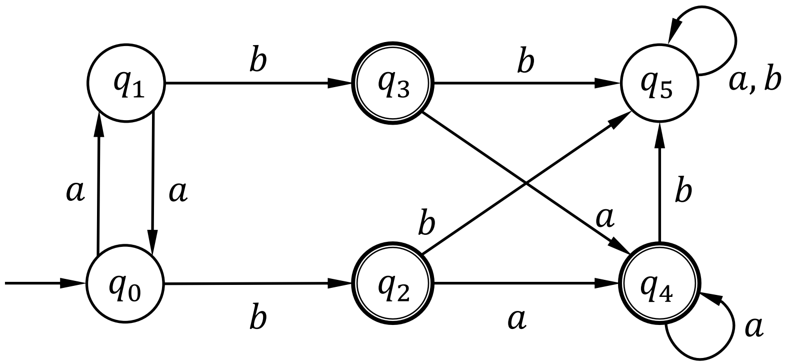


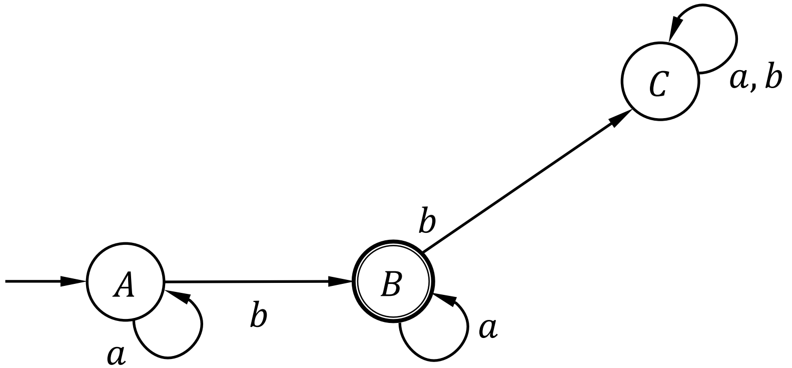


State Pairs	a	b	
$\{ q_0, A \}$	$\{ q_1, A \}$	$\{ q_2, B \}$	
$\{ q_1, A \}$			
$\{ q_2, B \}$			M_1 reaches final state if and only if M_2 reaches final state

DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$

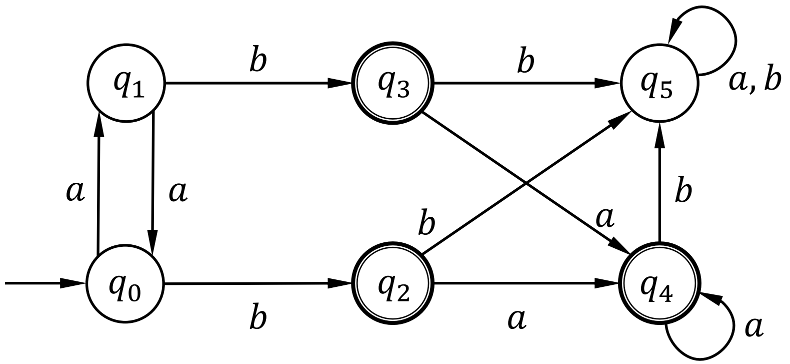


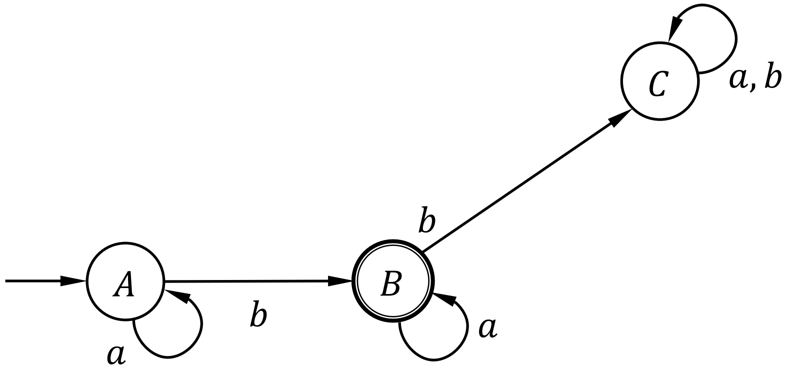


State Pairs	a	b	
$\{ q_0, A \}$	$\{ q_1, A \}$	$\{ q_2, B \}$	
$\{ q_1, A \}$	$\{ q_0, A \}$	$\{ q_3, B \}$	M_1 reaches final state if and only if M_2 reaches final state
$\{ q_2, B \}$			

DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$

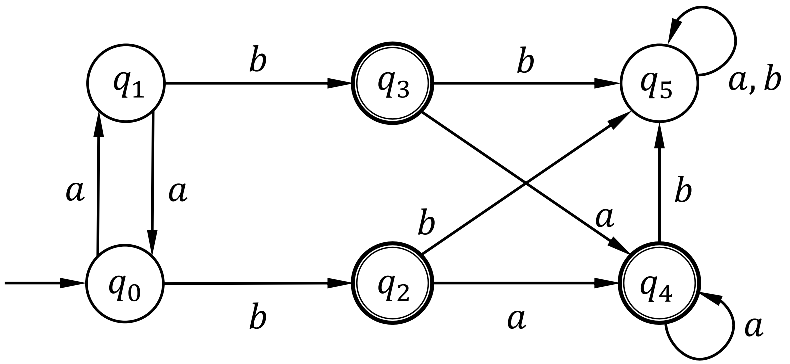


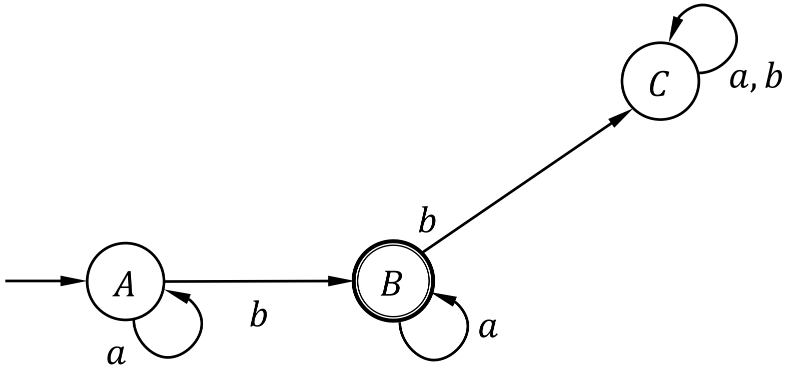


State Pairs	a	b	
$\{ q_0, A \}$	$\{ q_1, A \}$	$\{ q_2, B \}$	
$\{ q_1, A \}$	$\{ q_0, A \}$	$\{ q_3, B \}$	M_1 reaches final state if and only if M_2 reaches final state
$\{ q_2, B \}$			
$\{ q_3, B \}$			

DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$

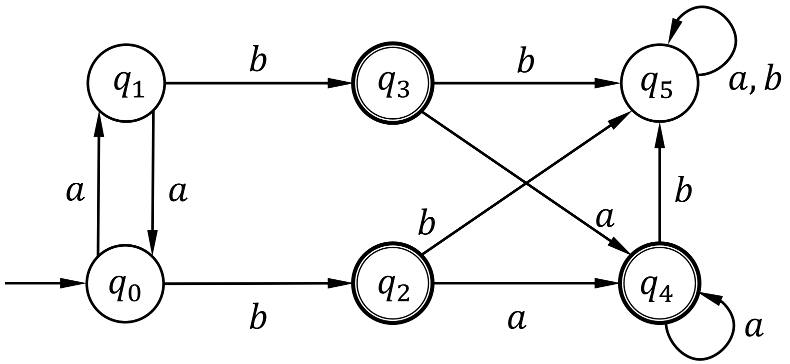


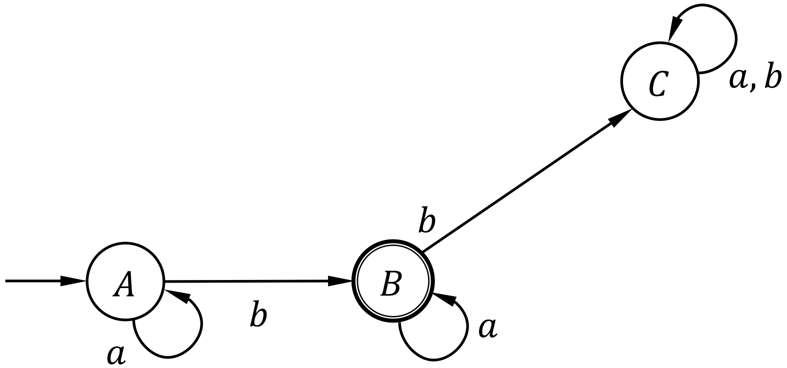


State Pairs	a	b	
$\{q_0, A\}$	$\{q_1, A\}$	$\{q_2, B\}$	
$\{q_1, A\}$	$\{q_0, A\}$	$\{q_3, B\}$	
$\{q_2, B\}$	$\{q_4, B\}$	$\{q_5, C\}$	M_1 reaches final state if and only if M_2 reaches final state
$\{q_3, B\}$			

DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$

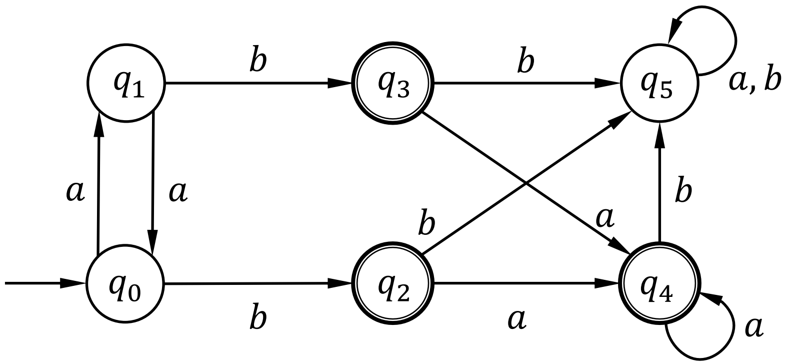


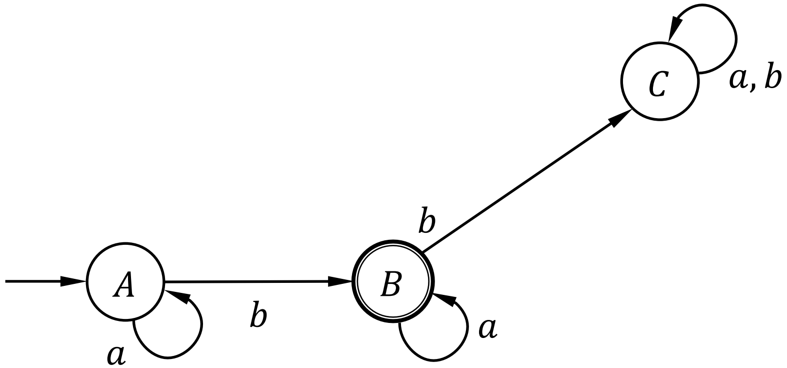


State Pairs	a	b	
$\{ q_0, A \}$	$\{ q_1, A \}$	$\{ q_2, B \}$	
$\{ q_1, A \}$	$\{ q_0, A \}$	$\{ q_3, B \}$	M_1 reaches final state if
$\{ q_2, B \}$	$\{ q_4, B \}$	$\{ q_5, C \}$	and only if
$\{ q_3, B \}$			M_2 reaches final state
$\{ q_4, B \}$			
$\{ q_5, C \}$			

DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$

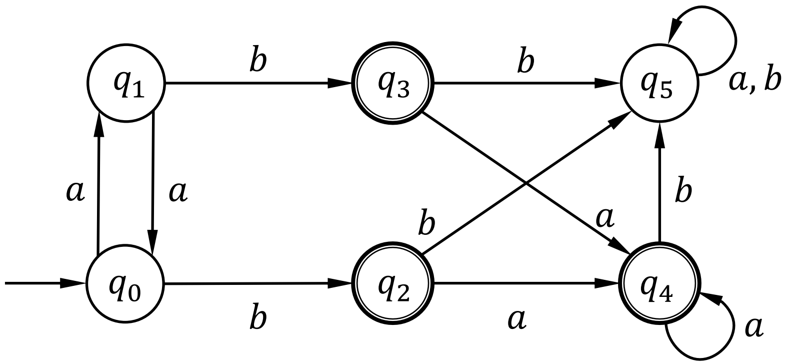


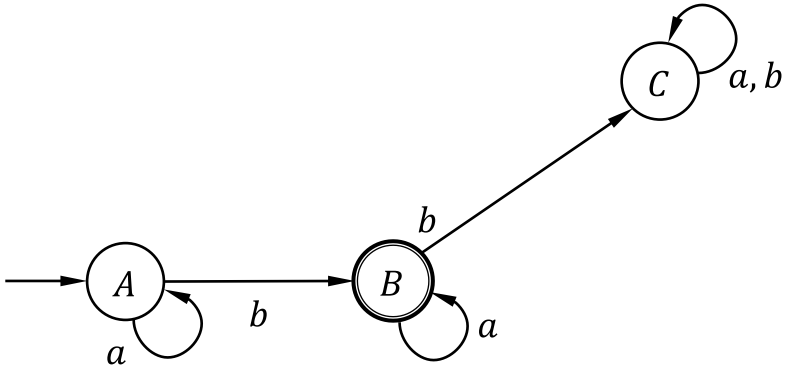


State Pairs	a	b	
$\{q_0, A\}$	$\{q_1, A\}$	$\{q_2, B\}$	
$\{q_1, A\}$	$\{q_0, A\}$	$\{q_3, B\}$	
$\{q_2, B\}$	$\{q_4, B\}$	$\{q_5, C\}$	M_1 reaches final state if and only if M_2 reaches final state
$\{q_3, B\}$	$\{q_4, B\}$	$\{q_5, C\}$	
$\{q_4, B\}$			
$\{q_5, C\}$			

DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$

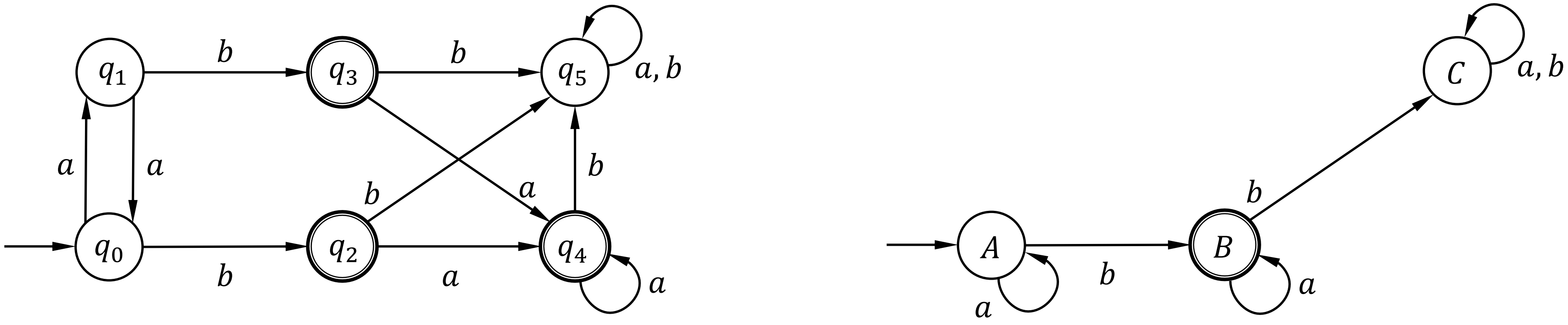




State Pairs	a	b	
$\{q_0, A\}$	$\{q_1, A\}$	$\{q_2, B\}$	
$\{q_1, A\}$	$\{q_0, A\}$	$\{q_3, B\}$	
$\{q_2, B\}$	$\{q_4, B\}$	$\{q_5, C\}$	M_1 reaches final state if and only if M_2 reaches final state
$\{q_3, B\}$	$\{q_4, B\}$	$\{q_5, C\}$	
$\{q_4, B\}$	$\{q_4, B\}$	$\{q_5, C\}$	
$\{q_5, C\}$			

DFA Bi-Simulation

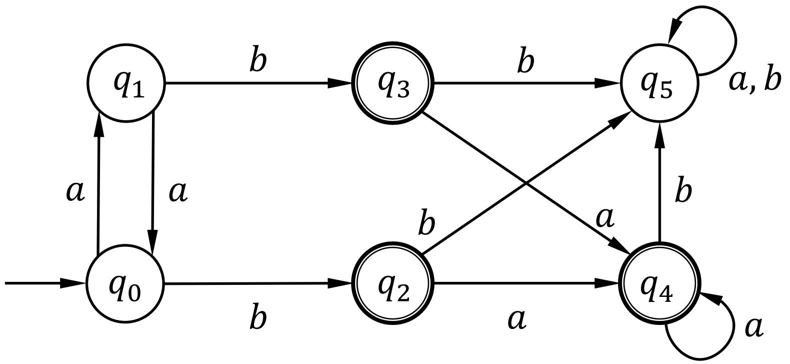
- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$

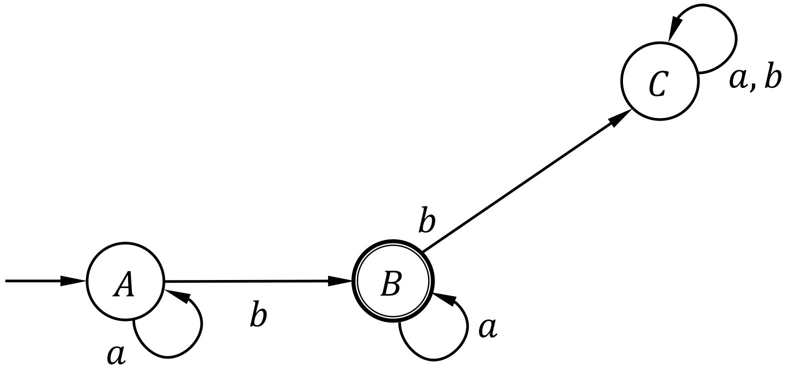


State Pairs	a	b	
$\{q_0, A\}$	$\{q_1, A\}$	$\{q_2, B\}$	
$\{q_1, A\}$	$\{q_0, A\}$	$\{q_3, B\}$	
$\{q_2, B\}$	$\{q_4, B\}$	$\{q_5, C\}$	M_1 reaches final state if
$\{q_3, B\}$	$\{q_4, B\}$	$\{q_5, C\}$	and only if
$\{q_4, B\}$	$\{q_4, B\}$	$\{q_5, C\}$	M_2 reaches final state
$\{q_5, C\}$	$\{q_5, C\}$	$\{q_5, C\}$	

DFA Bi-Simulation

- Checking the equivalence of the DFAs, i.e., $L(M_1) = L(M_2)$





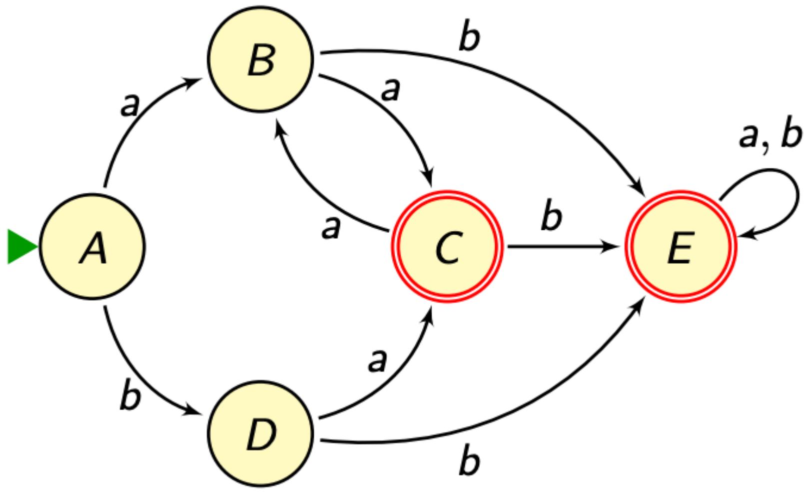
State Pairs	a	b
$\{q_0, A\}$	$\{q_1, A\}$	$\{q_2, B\}$
$\{q_1, A\}$	$\{q_0, A\}$	$\{q_3, B\}$
$\{q_2, B\}$	$\{q_4, B\}$	$\{q_5, C\}$
$\{q_3, B\}$	$\{q_4, B\}$	$\{q_5, C\}$
$\{q_4, B\}$	$\{q_4, B\}$	$\{q_5, C\}$
$\{q_5, C\}$	$\{q_5, C\}$	$\{q_5, C\}$

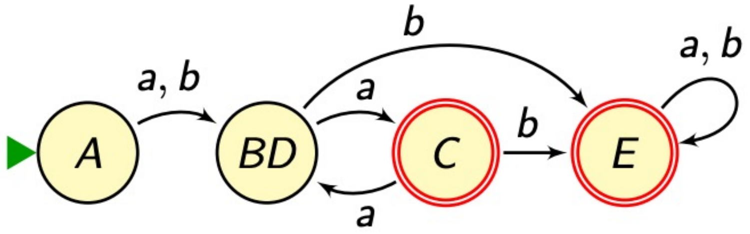
M_1 reaches final state if

and only if

M_2 reaches final state

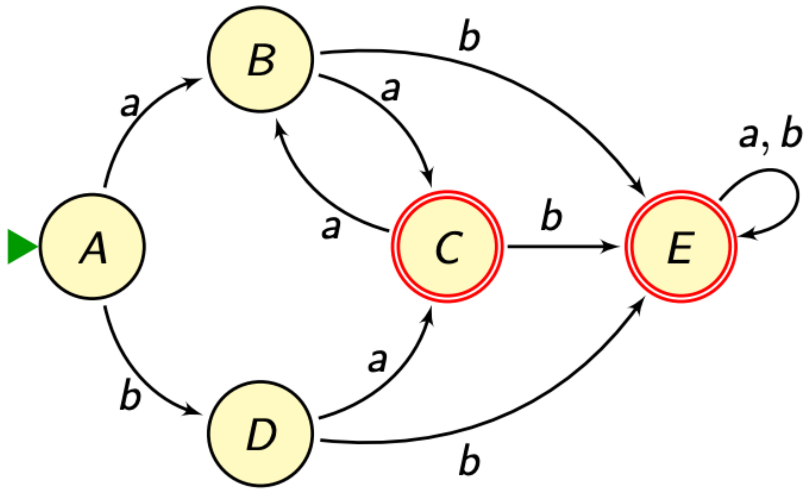
DFA Bi-Simulation

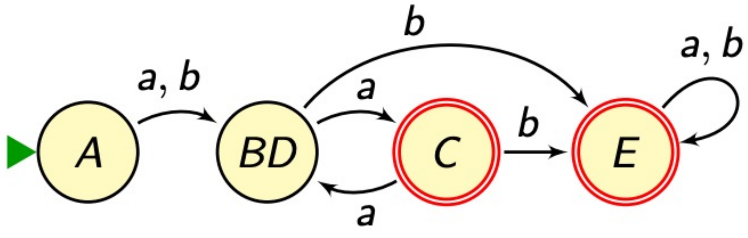




Have a Try!!

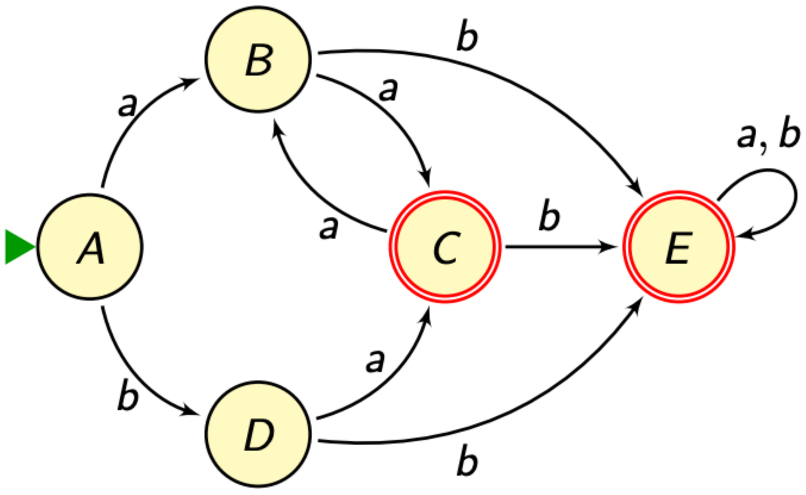
DFA Bi-Simulation

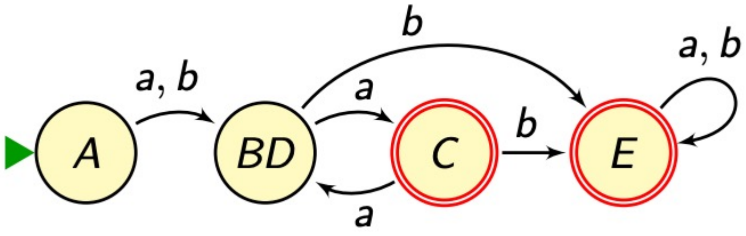




State Pairs	a	b	
(*A*, *A*’)	(*B*, *BD*’)	(*D*, *BD*’)	
			M_1 reaches final state if and only if M_2 reaches final state

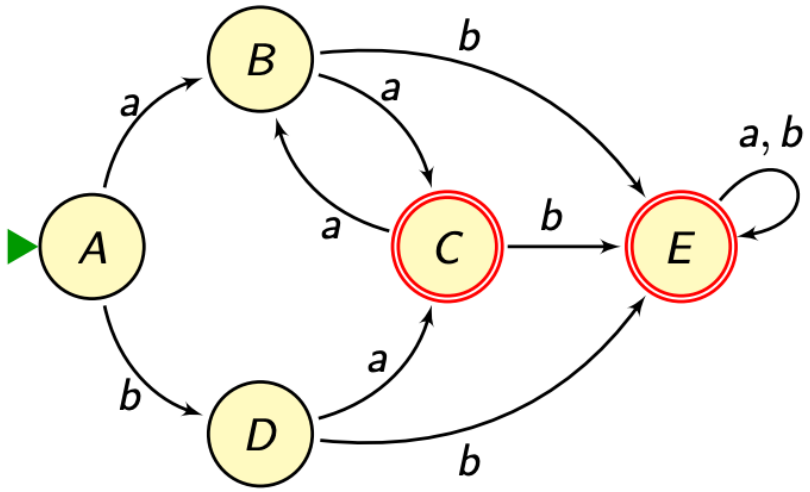
DFA Bi-Simulation

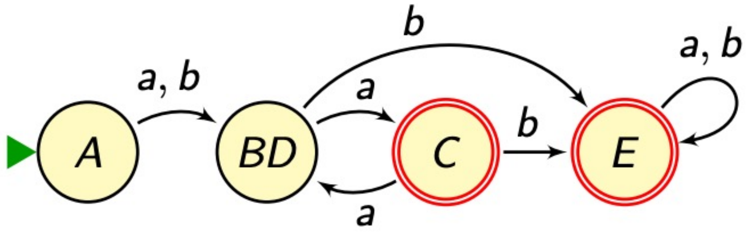




State Pairs	a	b	
(A, A')	(B, BD')	(D, BD')	
(B, BD')			M_1 reaches final state if and only if M_2 reaches final state
(D, BD')			

DFA Bi-Simulation

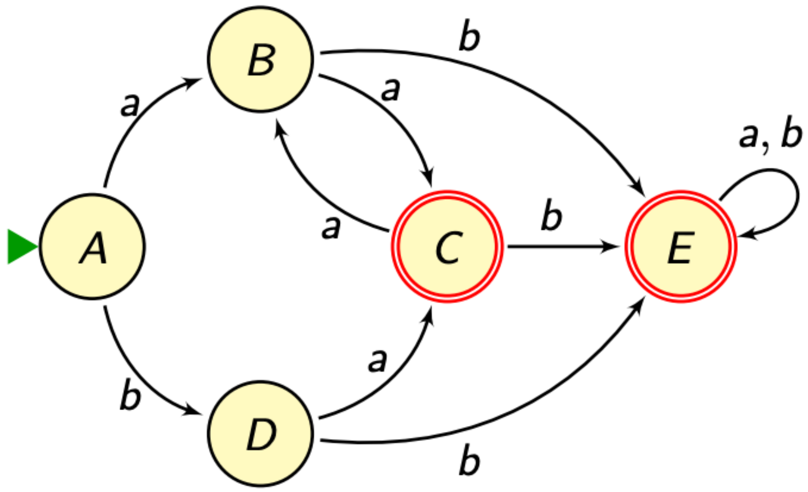


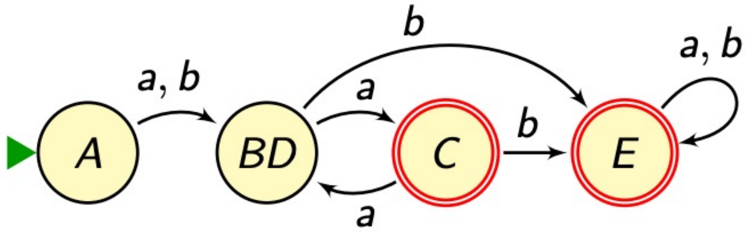


State Pairs	a	b	
(*A*, *A*')	(*B*, *BD*')	(*D*, *BD*')	
(*B*, *BD*')	(*C*, *C*')	(*E*, *E*')	
(*D*, *BD*')			

M_1 reaches final state if
and only if
M_2 reaches final state

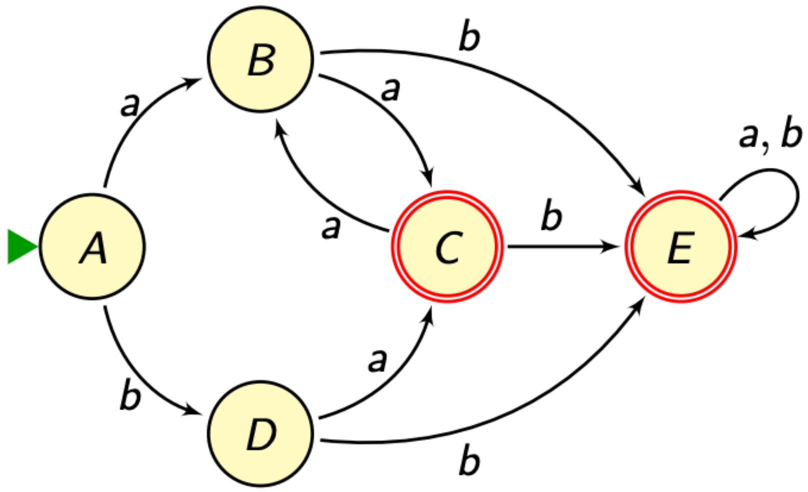
DFA Bi-Simulation

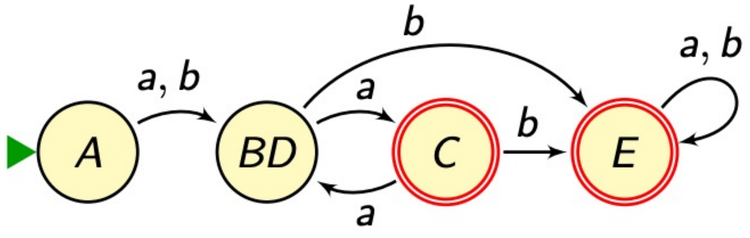




State Pairs	a	b	
(A, A')	(B, BD')	(D, BD')	
(B, BD')	(C, C')	(E, E')	
(D, BD')			M_1 reaches final state if and only if M_2 reaches final state
(C, C')			
(E, E')			

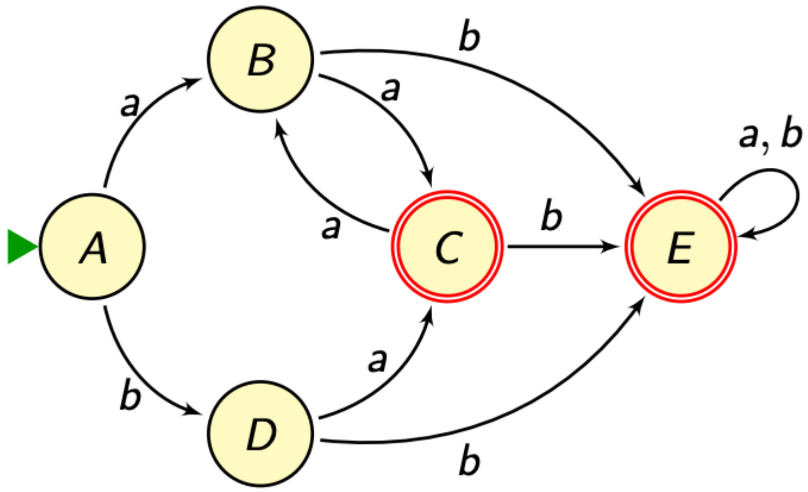
DFA Bi-Simulation

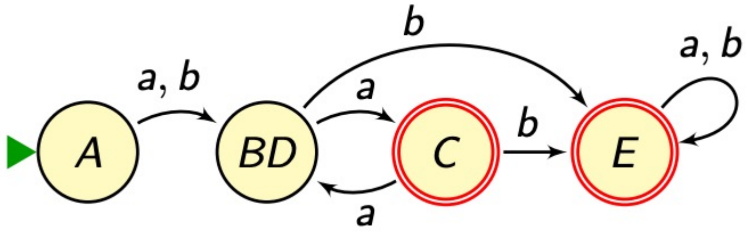




State Pairs	a	b	
(*A*, *A*’)	(*B*, *BD*’)	(*D*, *BD*’)	
(*B*, *BD*’)	(*C*, *C*’)	(*E*, *E*’)	M_1 reaches final state if
(*D*, *BD*’)	(*C*, *C*’)	(*E*, *E*’)	and only if
(*C*, *C*’)			M_2 reaches final state
(*E*, *E*’)			

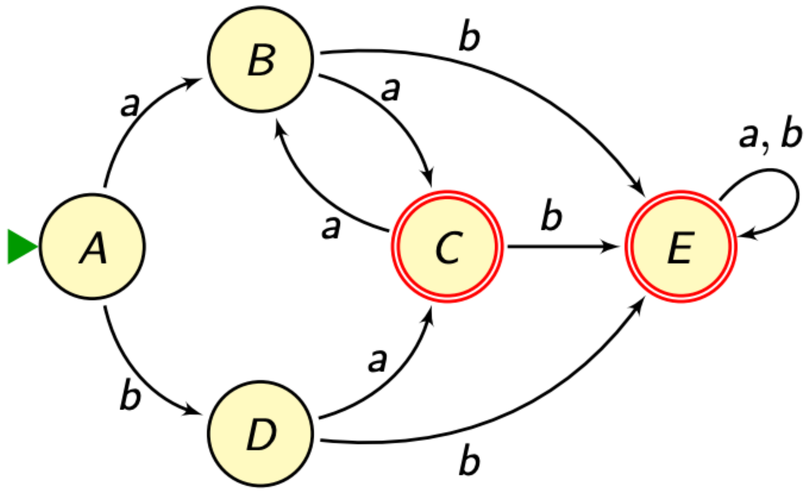
DFA Bi-Simulation

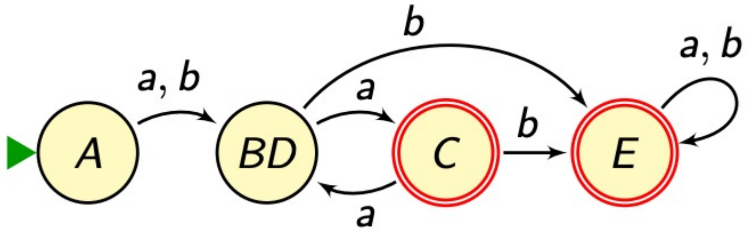




State Pairs	a	b	
(A, A')	(B, BD')	(D, BD')	
(B, BD')	(C, C')	(E, E')	M_1 reaches final state if
(D, BD')	(C, C')	(E, E')	and only if
(C, C')	(B, BD')	(E, E')	M_2 reaches final state
(E, E')			

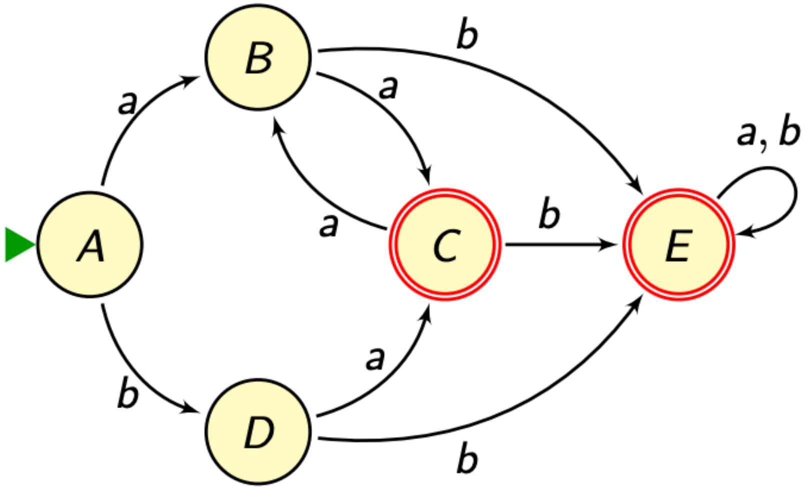
DFA Bi-Simulation

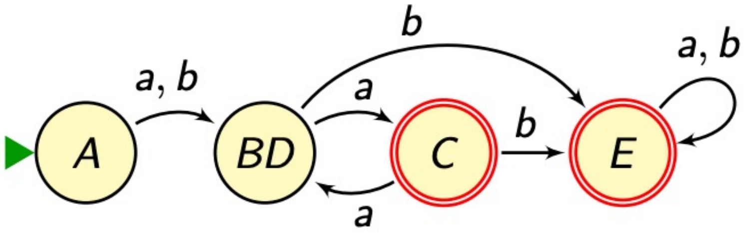




State Pairs	a	b	
(A, A')	(B, BD')	(D, BD')	
(B, BD')	(C, C')	(E, E')	M_1 reaches final state if and only if M_2 reaches final state
(D, BD')	(C, C')	(E, E')	
(C, C')	(B, BD')	(E, E')	
(E, E')	(E, E')	(E, E')	

DFA Bi-Simulation



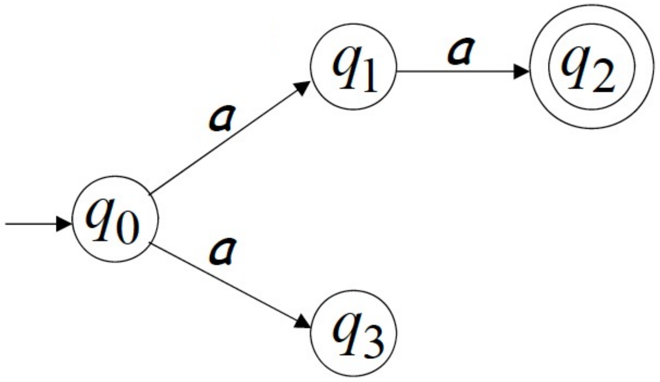


State Pairs	a	b	
(*A*, *A*’)	(*B*, *BD*’)	(*D*, *BD*’)	
(*B*, *BD*’)	(*C*, *C*’)	(*E*, *E*’)	M_1 reaches final state if
(*D*, *BD*’)	(*C*, *C*’)	(*E*, *E*’)	and only if
(*C*, *C*’)	(*B*, *BD*’)	(*E*, *E*’)	M_2 reaches final state
(*E*, *E*’)	(*E*, *E*’)	(*E*, *E*’)	

PART III:
Non-deterministic Finite Automata

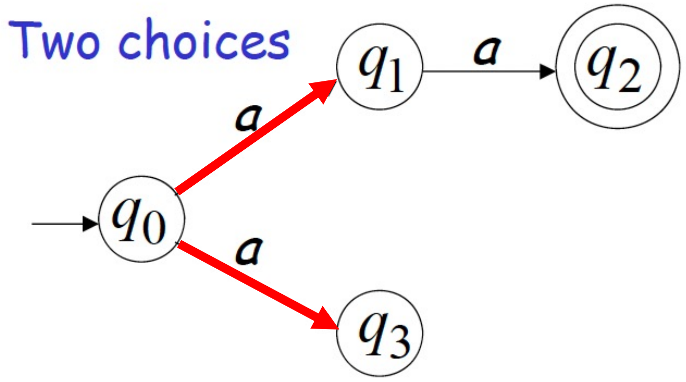
Non-deterministic Finite Automata

- There are multiple choices of state transition



Non-deterministic Finite Automata

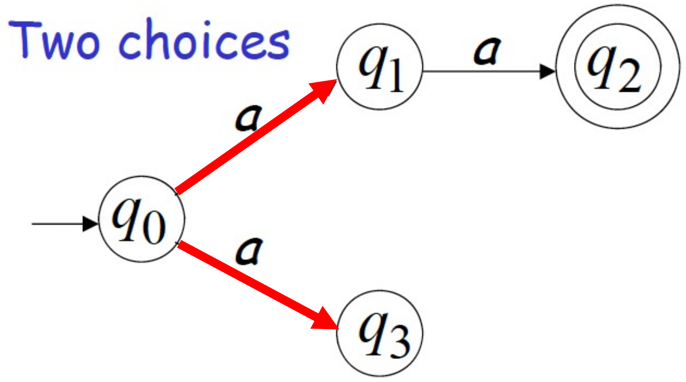
- There are multiple choices of state transition



- Given a string, *aa*, there are two choices for the first transition

Non-deterministic Finite Automata

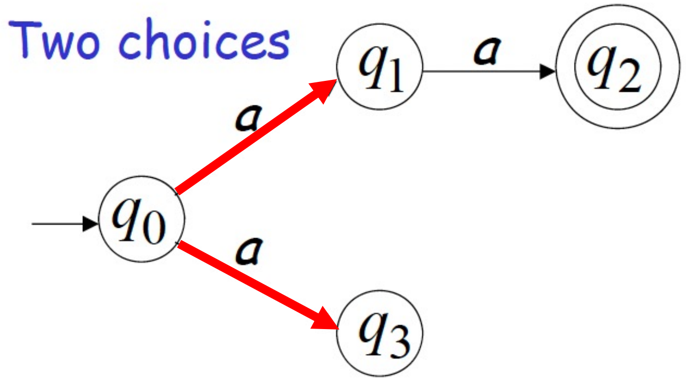
- There are multiple choices of state transition



- Given a string, *aa*, there are two choices for the first transition
- If one choice does not work, we need try others

Non-deterministic Finite Automata

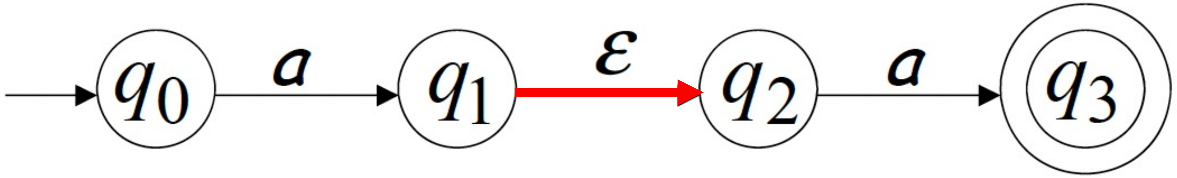
- There are multiple choices of state transition



- Given a string, *aa*, there are two choices for the first transition
- A string, e.g., *aa*, can be accepted by NFA as long as there exists one computation of the NFA accepts the string

ϵ-Transition

- Transition to the next states without reading any inputs



- NFA also allows ϵ-transitions!

Recap: Definition of DFA

- A DFA is a five-tuple: $(Q, \Sigma, \delta, q_0, F)$
  - Q: A finite set of states
  - Σ: A finite set of input characters, i.e., an alphabet
  - $\delta: Q \times \Sigma \longmapsto$ Q: Transition function, e.g., $\delta(q, a) = q'$
  - $q_0 \in Q$: The start state
  - $F \subseteq Q$: A finite subset of final states

Definition of NFA

- A ~~DFA~~NFA is a five-tuple: $(Q, \Sigma \cup \{\epsilon\}, \delta, q_0, F)$
  - Q: A finite set of states
  - Σ: A finite set of input characters, i.e., an alphabet
  - $\delta: Q \times (\Sigma \cup \{\epsilon\}) \mapsto \cancel{Q} 2^{Q}$: Transition function, e.g., $\delta(q, a) = \{q', q''\}$
  - $q_0 \in Q$: The start state
  - $F \subseteq Q$: A finite subset of final states

ϵ-Closure

- ϵ-*closure(q)* returns all states *q* can reach via ϵ-transitions, including *q* itself

ϵ-Closure

- *ϵ-closure(q)* returns all states *q* can reach via ϵ-transitions, including *q* itself

- An extension of transition: $\delta^*: Q \times \Sigma^* \longmapsto 2^Q$
  - $q' \in \delta^*(q, w)$, where $w \in \Sigma^*$, if and only if

ϵ-Closure

- ϵ-*closure(q)* returns all states *q* can reach via ϵ-transitions, including *q* itself

- An extension of transition: $\delta^*: Q \times \Sigma^* \longmapsto 2^Q$
  - $q' \in \delta^*(q, w)$, where $w \in \Sigma^*$, if and only if
    - (1) there's a walk from q to q'' with w
    - (2) $q' \in \epsilon-closure(q'')$

Defining a Language by NFA

- **Recap: language defined by DFA**
- Take a DFA $M = (Q, \Sigma, \delta, q_0, F)$, language can be accepted by the DFA is written as $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$

Defining a Language by NFA

- **Recap: language defined by DFA**
- Take a DFA $M = (Q, \Sigma, \delta, q_0, F)$, language can be accepted by the DFA is written as $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \subseteq F\}$

- Take a NFA $M = (Q, \Sigma \cup \{\epsilon\}, \delta, q_0, F)$, language can be accepted by the NFA is written as $L(M) = \{w \in \Sigma^*: \delta^*(q_0, w) \cap F \neq \emptyset\}$

NFA = DFA

- Every DFA is trivially an equivalent NFA
- Every NFA N can be converted to an equivalent DFA D

NFA = DFA

- Every DFA is trivially an equivalent NFA

- Every NFA N can be converted to an equivalent DFA D
  - Every string accepted by the NFA is accepted by the DFA
  - Every string rejected by the NFA is rejected by the DFA
  - i.e., $L(N) = L(D)$

From NFA to DFA

- Given an NFA $M = (Q, \Sigma \cup \{\epsilon\}, \delta, q_0, F)$
  - **Step 1:** initialize a DFA with a start state $\{\ q_0\ \}$, which is a set of NFA states

From NFA to DFA

- Given an NFA $M = (Q, \Sigma \cup \{\epsilon\}, \delta, q_0, F)$
  - **Step 1:** initialize a DFA with a start state $\{ q_0 \}$, which is a set of NFA states

DFA state is a subset of NFA states

From NFA to DFA

- Given an NFA $M = (Q, \Sigma \cup \{\epsilon\}, \delta, q_0, F)$
  - **Step 1:** initialize a DFA with a start state $\{\, q_0 \,\}$, which is a set of NFA states
  - **Step 2:** for each DFA state $\{\, q_i, q_j, \dots, q_m \,\}$, and char in the alphabet, $a \in \Sigma$

From NFA to DFA

- Given an NFA $M = (Q, \Sigma \cup \{\epsilon\}, \delta, q_0, F)$
  - **Step 1:** initialize a DFA with a start state $\{ q_0 \}$, which is a set of NFA states
  - **Step 2:** for each DFA state $\{ q_i, q_j, \ldots, q_m \}$, and char in the alphabet, $a \in \Sigma$

$$\text{let } S = \bigcup \begin{cases} \delta^*(q_i, a \in \Sigma) \\ \delta^*(q_j, a \in \Sigma) \\ \ldots \\ \delta^*(q_m, a \in \Sigma) \end{cases}$$

From NFA to DFA

- Given an NFA $M = (Q, \Sigma \cup \{\epsilon\}, \delta, q_0, F)$
  - **Step 1:** initialize a DFA with a start state $\{ q_0 \}$, which is a set of NFA states
  - **Step 2:** for each DFA state $\{ q_i, q_j, \ldots, q_m \}$, and char in the alphabet, $a \in \Sigma$

$$\text{let } S = \cup \begin{cases} \delta^*(q_i, a \in \Sigma) \\ \delta^*(q_j, a \in \Sigma) \\ \quad \ldots \\ \delta^*(q_m, a \in \Sigma) \end{cases}$$

  - **Step 3**: add transition $\delta(\{ q_i, q_j, \ldots, q_m \}, a) = S$ in the DFA

From NFA to DFA

- Given an NFA $M = (Q, \Sigma \cup \{\epsilon\}, \delta, q_0, F)$
  - **Step 1:** initialize a DFA with a start state $\{\, q_0 \,\}$, which is a set of NFA states
  - **Step 2:** for each DFA state $\{\, q_i, q_j, \ldots, q_m \,\}$, and char in the alphabet, $a \in \Sigma$

$$\text{let } S = \cup \begin{cases} \delta^*(q_i, a \in \Sigma) \\ \delta^*(q_j, a \in \Sigma) \\ \ldots \\ \delta^*(q_m, a \in \Sigma) \end{cases}$$

if any q_i is a final state, S is a final state of the DFA

  - **Step 3**: add transition $\delta(\{\, q_i, q_j, \ldots, q_m \,\}, a) = S$ in the DFA

From NFA to DFA

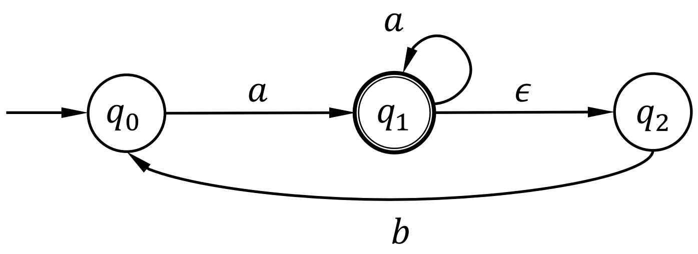
- Given an NFA $M = (Q, \Sigma \cup \{\epsilon\}, \delta, q_0, F)$
  - **Step 1:** initialize a DFA with a start state $\{\, q_0 \,\}$, which is a set of NFA states
  - **Step 2:** for each DFA state $\{\, q_i, q_j, \ldots, q_m \,\}$, and char in the alphabet, $a \in \Sigma$

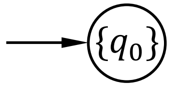
$$\text{let } S = \cup \begin{cases} \delta^*(q_i, a \in \Sigma) \\ \delta^*(q_j, a \in \Sigma) \\ \cdots \\ \delta^*(q_m, a \in \Sigma) \end{cases}$$

For any input string, the DFA and the NFA reach the final state at the same time, thus DFA=NFA

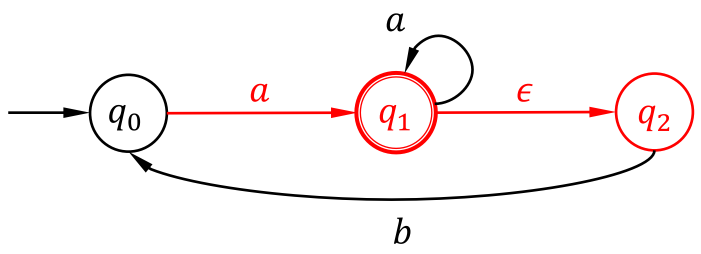
  - **Step 3**: add transition $\delta(\{\, q_i, q_j, \ldots, q_m \,\}, a) = S$ in the DFA

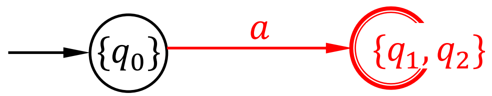
From NFA to DFA



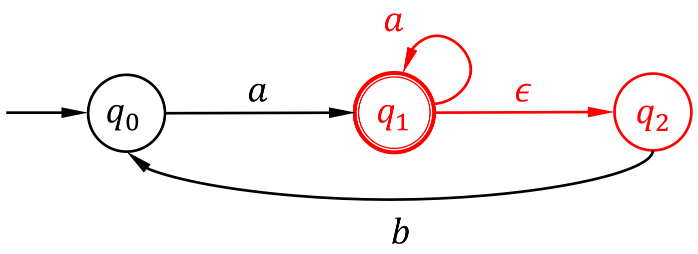


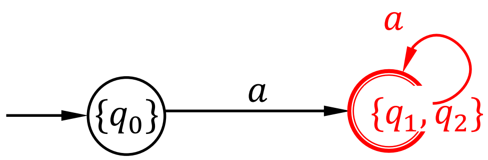
From NFA to DFA



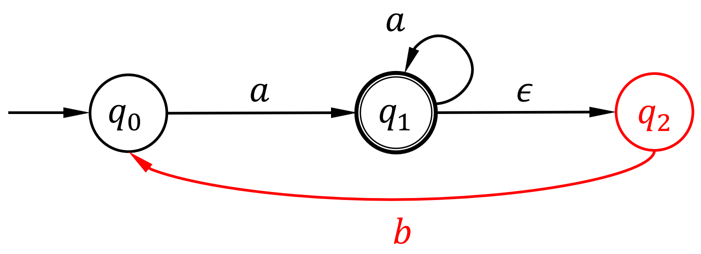


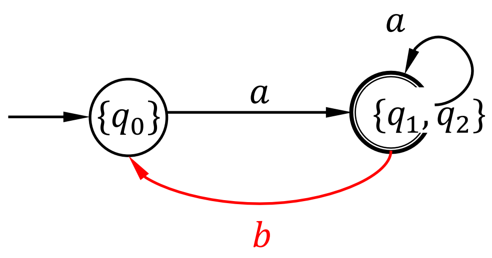
From NFA to DFA



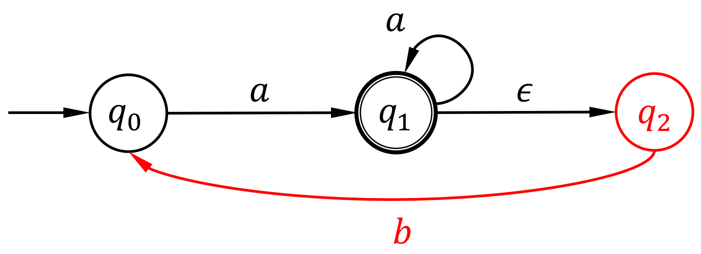


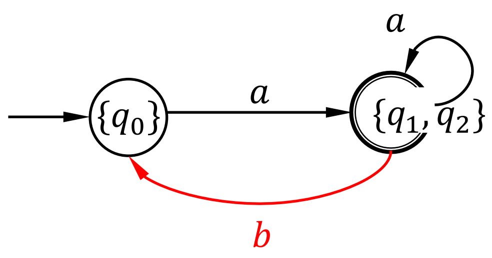
From NFA to DFA





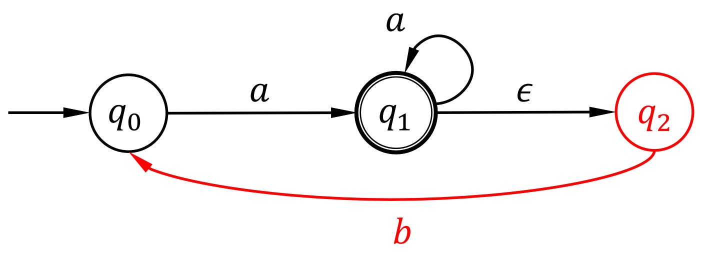
From NFA to DFA

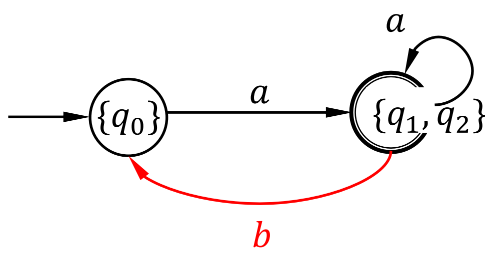




Because the DFA states consist of sets of NFA states, an *n*-state NFA may be converted to a DFA with at most 2^n states. For every *n*, there exist *n*-state NFAs such that every subset of states is reachable from the initial subset, so that the converted DFA has exactly 2^n states, giving $\Theta(2^n)$ worst-case time complexity.

From NFA to DFA

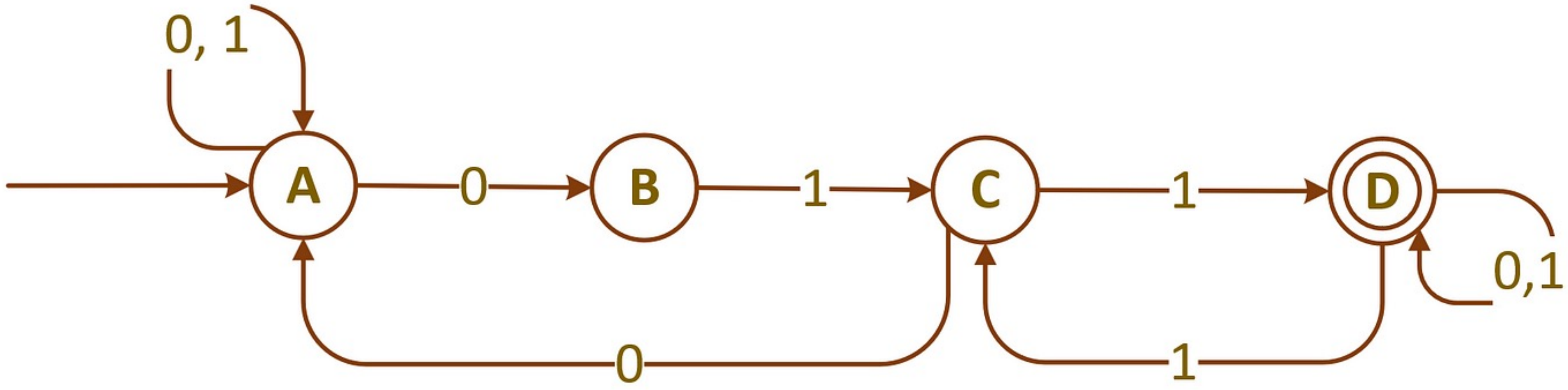




Because the DFA states consist of sets of NFA states, an *n*-state NFA may be converted to a DFA with at most 2^n states. For every *n*, there exist *n*-state NFAs such that every subset of states is reachable from the initial subset, so that the converted DFA has exactly 2^n states, giving $\Theta(2^n)$ worst-case time complexity.

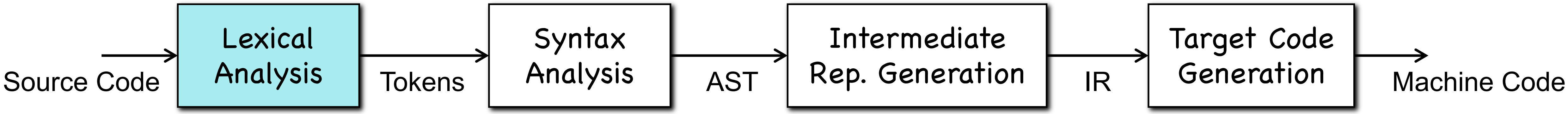
When converting an NFA to a DFA, there is no guarantee that we will have a smaller DFA.

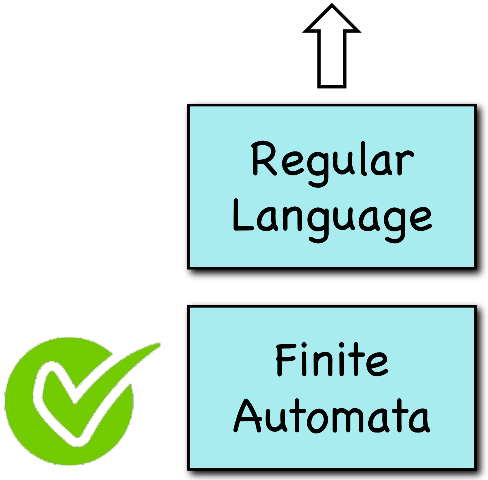
From NFA to DFA



Have a Try!!

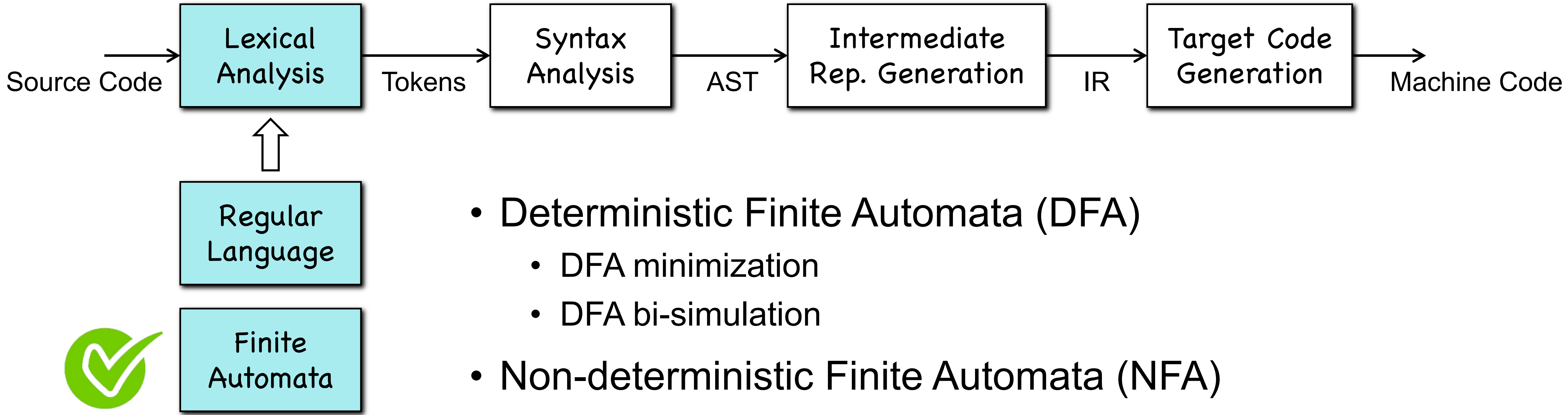
Summary





- Deterministic Finite Automata (DFA)
  - DFA minimization
  - DFA bi-simulation
- Non-deterministic Finite Automata (NFA)
  - NFA = DFA
  - NFA → DFA
- Languages defined by DFA/NFA

Summary



- Deterministic Finite Automata (DFA)
  - DFA minimization
  - DFA bi-simulation
- Non-deterministic Finite Automata (NFA)
  - NFA = DFA
  - NFA → DFA
- Languages defined by DFA/NFA → Regular Language

THANKS!