CSE 3204: Formal Language, Automata and Computability

❑ A theoretical branch of computer science

- study of ***abstract machines*** and the *computation problems* that can be solved using these machines.
- The abstract machine is called the automata.
- a team consists of biologists, psychologists, mathematicians, engineers and few computer scientists

Goals:
- to model the human thought process, in a machine
- the need to formally understand what can (and cannot) be ***computed***.

References

- **"Introduction to Automata Theory, Languages, and Computation"** by John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman.
- **"Theory of Computer Science: Automata, Languages and Computation"** by K.L.P. Mishra and N. Chandrasekaran.

Notable early contributions on Computability

❑ **Turing Machine** - *Alan Turing*

- Provided a simple yet powerful model to define what it means for a function to be computable.
- Established the theoretical underpinnings for modern computing and algorithms.

❑ **Lambda Calculus** - *Alonzo Church*

- Introduced a mathematical framework for expressing computation via functions.
- Influenced the development of programming languages like Lisp, Haskell, and Scala.
- Played a crucial role in exploring the theoretical limits of what can be computed.

Notable early contributions on Computability (cont.)

Church-Turing Thesis

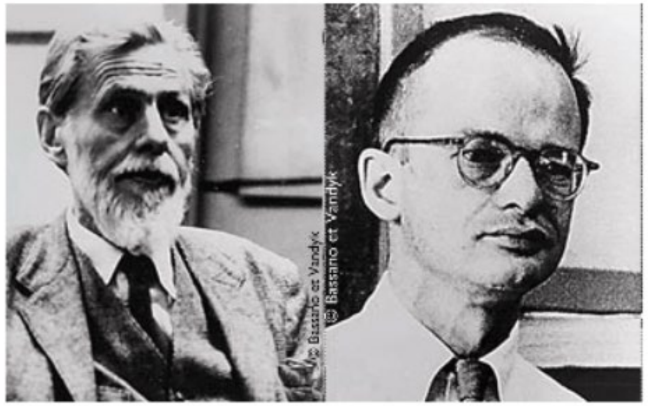
- Hypothesizes that any function ***effectively calculable by an algorithm*** can be computed by a Turing Machine or expressed in Lambda Calculus.
- **Unifies** various models of computation (recursive functions, Lambda Calculus, Turing Machines).
- Widely accepted despite ***not being formally provable***.
- Underpins much of modern computational theory, capturing the essence of computation.

Uncomputable Problems *Undecidable Problem (e.g. Halting Problem)*

- Alan Turing's introduction of the Halting Problem was a pivotal moment in theoretical CS.
- By **proving** that the **Halting Problem is undecidable** and, consequently, uncomputable
    - demonstrated that there are clear limits to what can be achieved through algorithms and computation.
    - illustrated that some problems cannot be solved by any algorithmic means, no matter how powerful our computational models become.

Notable early contributions on Automata

- Two neurophysiologists, Warren McCulloch and Walter Pitts, were the first to present a description of ***finite automata*** in 1943.
- Stephen Kleene introduced ***regular expressions*** and the concept of regular sets, which are crucial in the theory of finite automata.
  - His work on these topics laid the groundwork for text processing and compiler construction.
- Later, G.H. Mealy and E.F. Moore, generalized the theory to design much more powerful machines in 1955.



Warren McCulloch and Walter Pitts

Applications and objectives

❑ Basis of many Applications

- Compilers and interpreters
- Text editors and processors
- Search engines
- System verification components

❑ Study the limits of computations

- What kinds of problems can be solved with a computer?
- What kinds of problems can be solved efficiently?

Important Terminologies

1. Alphabets
2. Strings
3. Languages
4. Problems

- An ***alphabet*** is a finite set of symbols.
  - Usually, we use Σ to represent an alphabet.

    Examples:
    - Σ = {0, 1}, the set of binary digits.
    - Σ = {a, b, ..., z}, the set of all lower-case letters.
    - Σ = {(,)}, the set of open and close parentheses.

- A string is a finite sequence of symbols from an alphabet.

  Examples:
  - 0011 and 11 are strings from Σ = {0, 1}.
  - abacy and cca are strings from Σ = {a, b, ..., z}.
  - (()()) and () are strings from Σ = {(,)}.

Basics of Strings

- A string is a finite sequence of symbols from an alphabet.
  - 0011 and 11 are strings from Σ = {0, 1}.
  - abacy and cca are strings from Σ = {a, b, ..., z}.
  - (()()) and () are strings from Σ = {(,)}.

- **Empty** string: ϵ
- **Length** of string: $|0010| = 4$, $|aa| = 2$, $|\epsilon| = 0$
- **Prefix** of string: $\underline{\text{aa}}\text{abc}$, $\underline{\text{aaab}}\text{c}$, $\underline{\text{aaabc}}$
- **Proper prefix** of string: $\underline{\text{aa}}\text{abc}$, $\underline{\text{aaab}}\text{c}$
- **Suffix** of string: $\text{aaa}\underline{\text{bc}}$, $\text{a}\underline{\text{aabc}}$, $\underline{\text{aaabc}}$
- **Proper suffix** of string: $\text{aaa}\underline{\text{bc}}$, $\text{a}\underline{\text{aabc}}$
- **Substring** of string: $\text{a}\underline{\text{aa}}\text{bc}$, $\underline{\text{aaab}}\text{c}$, $\text{aa}\underline{\text{a}}\text{bc}$

A proper prefix and proper suffix of a string is not equal to the string itself and non- empty.

Basics of Strings

- **Concatenation**: $\omega = abd$, $\alpha = ce$, $\omega\alpha = abdce$
- **Exponentiation**: $\omega = abd$, $\omega^3 = abdabdabd$, $\omega^0 = \epsilon$
- **Reversal**: $\omega = abd$, $\omega^R = dba$
- Σ^k = set of all k-length strings formed by symbols in Σ
    - e.g., $\Sigma = \{a, b\}$, $\Sigma^2 = \{ab, ba, aa, bb\}$, $\Sigma^0 = \{\epsilon\}$

Languages and Problems

In automata theory, a **problem** is to decide whether a given string is a member of some particular **language**.

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

$$L = \{0^n 1^n \mid n \geq 1\}$$

$$\text{is } S_1 = 0011 \in L? \quad \text{YES}$$

$$\text{is } S_2 = 00011 \in L? \quad \text{NO}$$

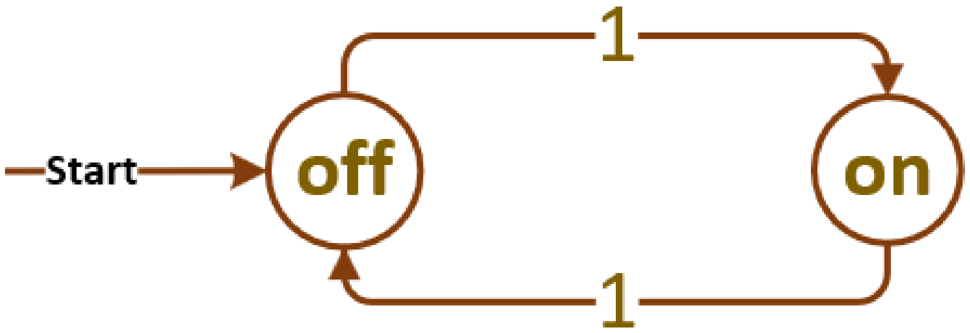
1. Alphabets
2. Strings
3. **Languages**
4. **Problems**

This formulation is general enough to capture the difficulty levels of all computing problems.

Finite Automata or Finite State Machine

We will study **3 types** of Finite Automata

- Deterministic Finite Automata (DFA)
- Non-deterministic Finite Automata (NFA)
- Finite Automata with ε-transitions (ε-NFA)



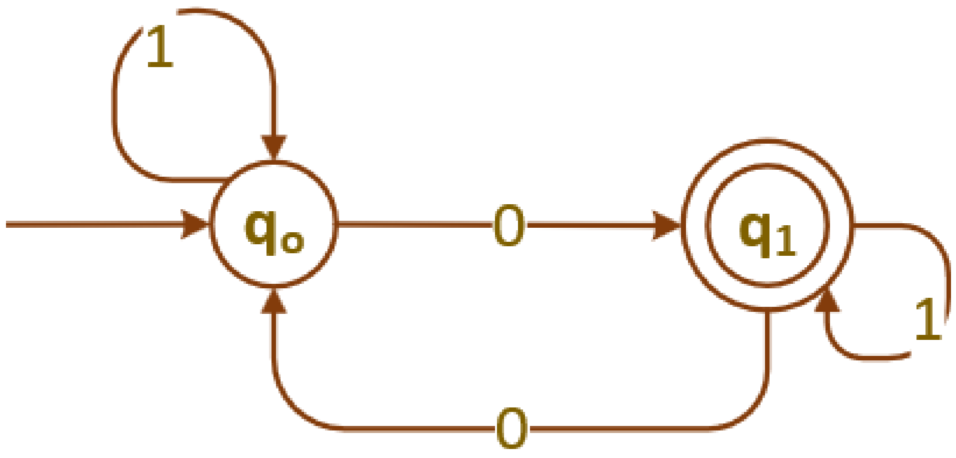
- ❑ There are some states and transitions (edges) between the states.
- ❑ An edge label defines the move from one state to another.

Deterministic Finite Automata

A DFA is a 5-tuple $<Q, \Sigma, \delta, q_0, F>$:

- Q is a finite set of states
- Σ is a finite input alphabet
- δ is the transition function mapping Q×Σ to Q
- $q_0 \in Q$ is the initial state (only one)
- F ∈ Q is a set of final states (zero or more)

Note that there is one transition only for each input symbol from each state



Transition Diagram

Present State	Transition Function – δ	
	Input: *0*	Input: *1*
→ q_0	q_1	q_0
q_1	q_0	q_1

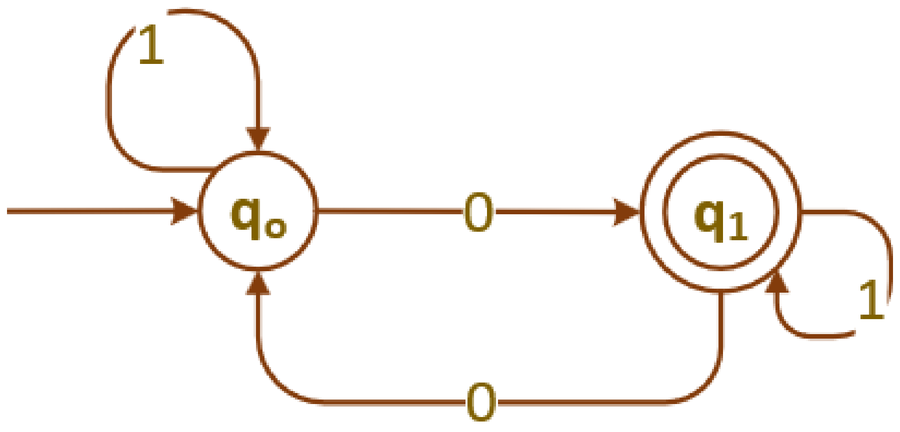
Transition Table

Deterministic Finite Automata

A DFA is a 5-tuple $<Q, \Sigma, \delta, q_o, F>$:

- Q is a finite set of states
- Σ is a finite input alphabet
- δ is the transition function mapping Q×Σ to Q
- $q_0 \in Q$ is the initial state (only one)
- F ∈ Q is a set of final states (zero or more)

Note that there is one transition only for each input symbol from each state



Transition Diagram

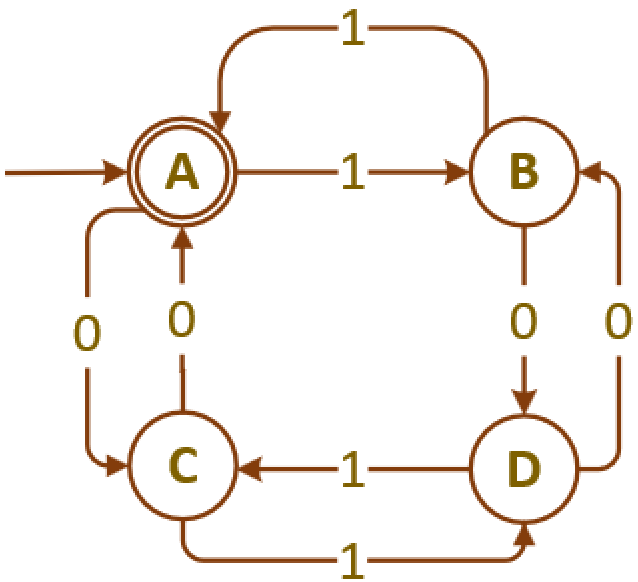
	Transition Function – δ	
Present State	Input: *0*	Input: *1*
→q_0	q_1	q_0
q_1	q_0	q_1

Transition Table

A DFA for Σ={0,1} that accepts strings with odd number of 0's and any number of 1's.

Constructions of DFA

L={w: w has both an even number of 0's and an even number of 1's}

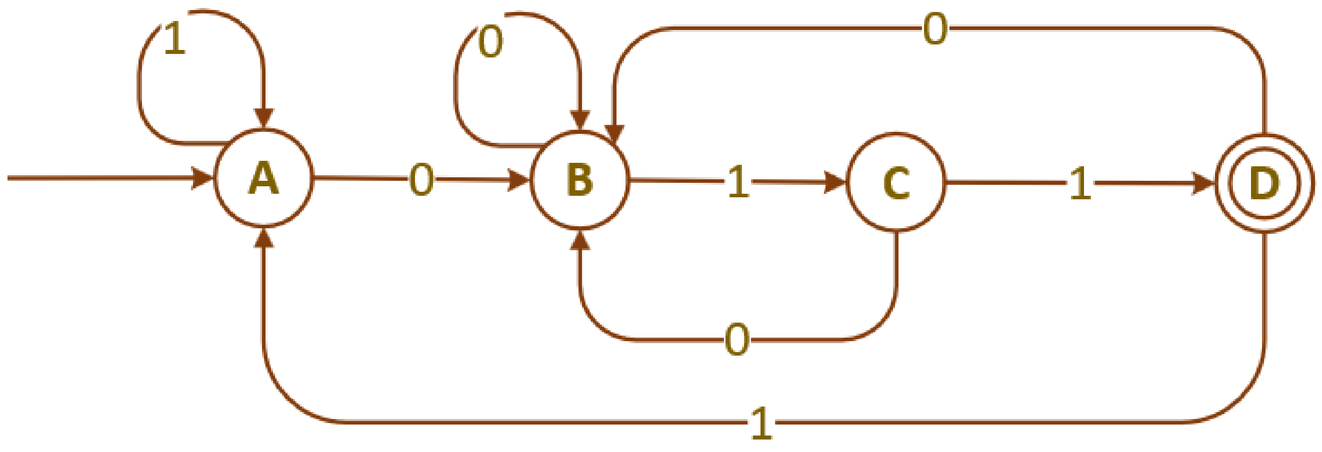


Constructions of DFA

- Design a DFA that accepts strings of 0's and 1's ending with 011.

Constructions of DFA

- Design a DFA that accepts strings of 0's and 1's ending with 011.

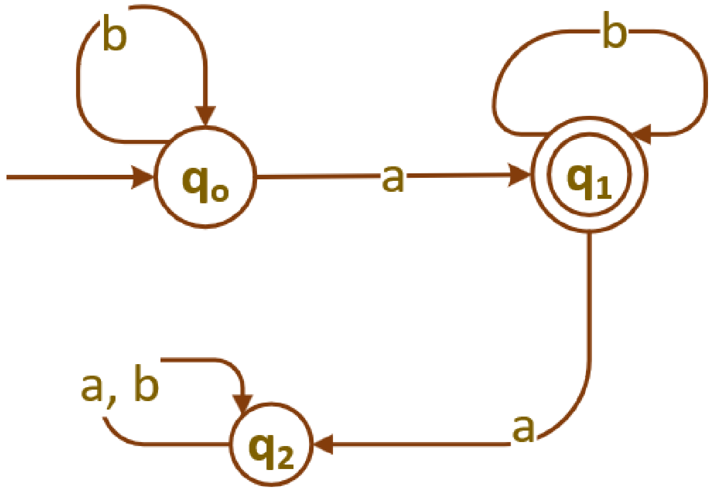


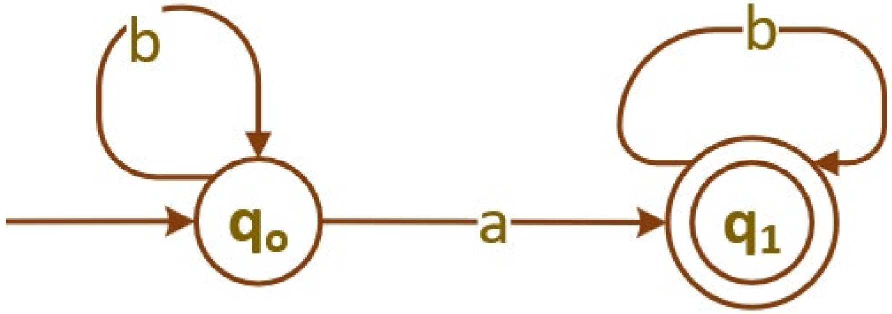
Constructions of DFA

- Design a DFA that accepts strings of a's and b’s having exactly one a.

Constructions of DFA

- Design a DFA that accepts strings of a's and b's having exactly one a.





❑ **A dead state** in a DFA is generally defined as a non-accepting state from which no sequence of inputs can lead to an accepting state. Two school of thoughts -

1. This state may have outgoing transitions that either loop back to itself or lead to other non-accepting states, but they do not lead to any accepting state.
2. this state must not have any outgoing transition apart from self-loop.

Constructions of DFA

Give a DFA for $\Sigma = \{0, 1\}$; that accepts any string with 001 as a substring.

Constructions of DFA

Construct a DFA for **$\Sigma = \{0, 1\}$;** that accepts any string with 001 as a **subsequence**.

Constructions of DFA

L={w: w has both an even number of 0’s and an even number of 1’s}

Practice DFA Constructions

- Design a DFA for each of the following

  - To accept strings of a’s and b’s **that contains substring aba**
  - To accept strings of a’s and b’s **that start with baba**
  - To accept strings of a’s and b’s **that end with abba**
  - To accept strings of a’s and b’s **that contains exactly two b’s**

Extended transition function for DFA

- The DFA defines a language
  - the set of all strings that results in a sequence of state transitions from the initial state to a final state

- Extended Transition Function
  - denoted by $\hat{\delta}$
  - $\hat{\delta}$ takes a state q, a string ω and returns a state p
    - p is the states where the automaton reaches when starting in a state q and processing the sequence of inputs ω

Extended transition function for DFA

Basis

$\hat{\delta}(q, \varepsilon) = q$

Induction

Suppose, $\omega = xa$

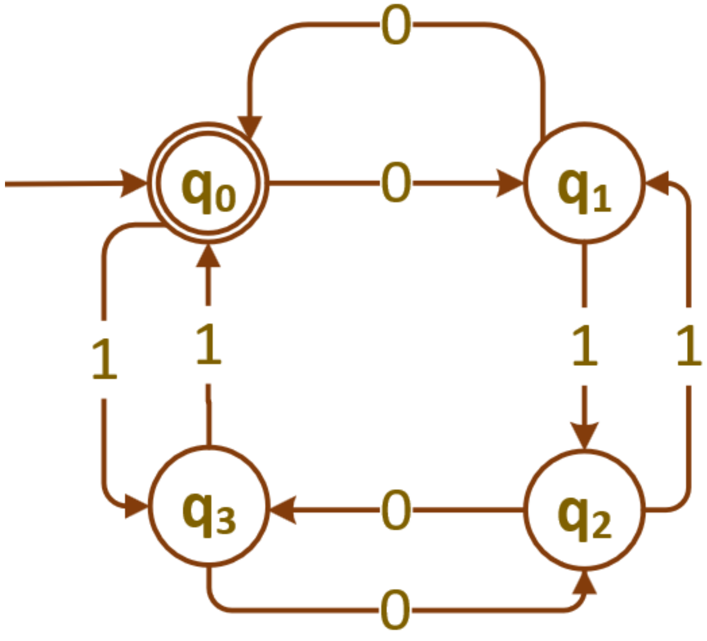
i.e.; a is the last symbol of ω and x is the string consisting of all but the last symbol

$$\text{Then, } \hat{\delta}(q,\omega) = \delta(\hat{\delta}(q,x), a)$$

- To compute $\hat{\delta}(q,\omega)$, first compute $\hat{\delta}(q,x)$, the state that the automaton is in after processing all but the last symbol of ω.
- Suppose this state is p i.e.; $\hat{\delta}(q,x) = p$.
- Then $\hat{\delta}(q,\omega)$ is what we get by making a transition from state p on input a - the last symbol of ω.

Extended transition function for DFA

The check involves computing $\hat{\delta}(q_0,\omega)$ for each prefix ω of 110101, starting at ϵ and going in increasing size.



Extended transition function for DFA

The check involves computing $\hat{\delta}(q_0,\omega)$ for each prefix ω of 110101, starting at ϵ and going in increasing size.

- $\hat{\delta}(q_0, \epsilon) = q_0$
- $\hat{\delta}(q_0, 1) = \delta(\hat{\delta}(q_0, \epsilon), 1) = \delta(q_0, 1) = q_3$
- $\hat{\delta}(q_0, 11) = \delta(\hat{\delta}(q_0, 1), 1) = \delta(q_3, 1) = q_0$
- $\hat{\delta}(q_0, 110) = \delta(\hat{\delta}(q_0, 11), 0) = \delta(q_0, 0) = q_1$
- $\hat{\delta}(q_0, 1101) = \delta(\hat{\delta}(q_0, 110), 1) = \delta(q_1, 1) = q_2$
- $\hat{\delta}(q_0, 11010) = \delta(\hat{\delta}(q_0, 1101), 0) = \delta(q_2, 0) = q_3$
- $\hat{\delta}(q_0, 110101) = \delta(\hat{\delta}(q_0, 11010), 1) = \delta(q_3, 1) = q_0$

Non-deterministic Finite Automata (NFA)

A NFA is a 5-tuple $<Q, \Sigma, \delta, q_0, F>$:

- Q is a finite set of states
- Σ is a finite input alphabet
- δ is the transition function mapping $Q \times \Sigma$ to a subset of Q
- $q_0 \in Q$ is the initial state (only one)
- $F \in Q$ is a set of final states (zero or more)

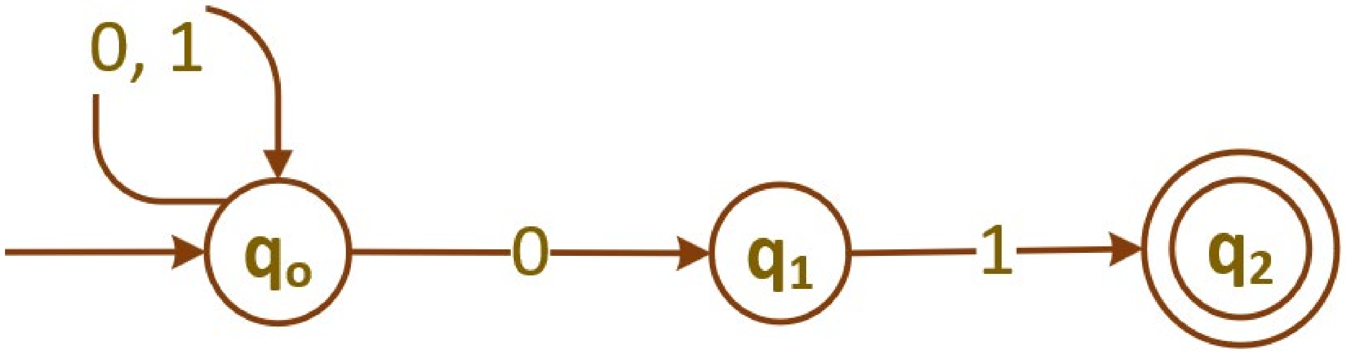
The difference between DFA and NFA is in type of value δ returns

Non-deterministic Finite Automata (NFA)

Design a NFA that accepts all binary strings that end with 01

Non-deterministic Finite Automata (NFA)

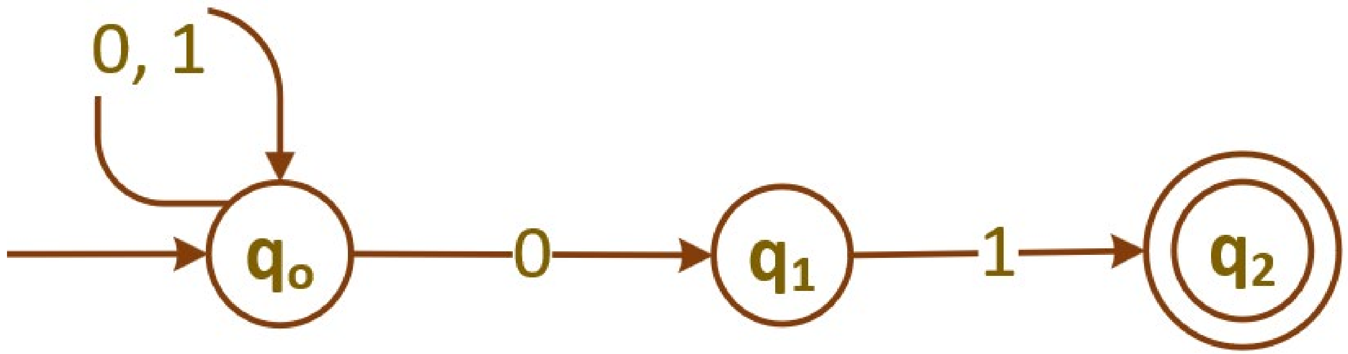
Design a NFA that accepts all binary strings that end with 01



$A = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_2\})$
where the transition function δ is given by:

	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
$*q_2$	$\emptyset$	$\emptyset$

Extended Transition Function for NFA



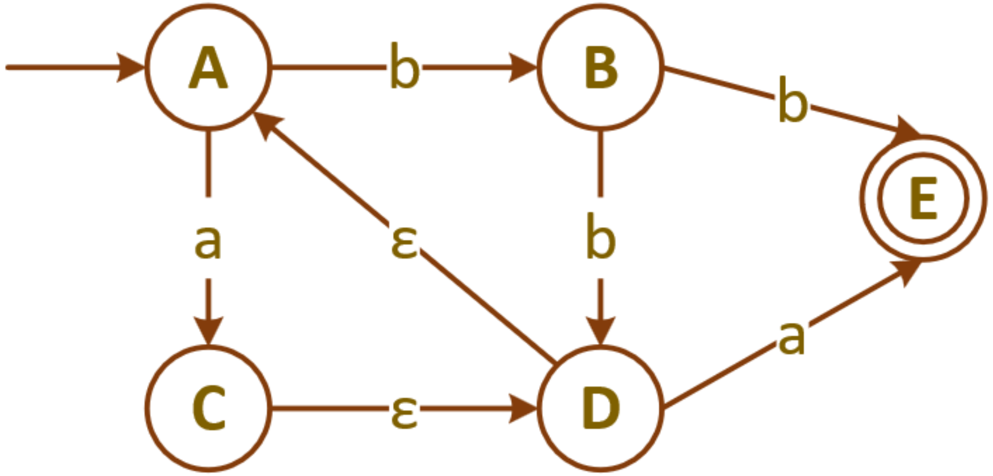
Processing $\omega = 00101$

- $\hat{\delta}(q_0, \epsilon) = \{q_0\}$
- $\hat{\delta}(q_0, 0) = \delta(q_0, 0) = \{q_0, q_1\}$
- $\hat{\delta}(q_0, 00) = \delta(q_0, 0) \cup \delta(q_1, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- $\hat{\delta}(q_0, 001) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
- $\hat{\delta}(q_0, 0010) = \delta(q_0, 0) \cup \delta(q_2, 0) = \{q_0, q_1\} \cup \emptyset = \{q_0, q_1\}$
- $\hat{\delta}(q_0, 00101) = \delta(q_0, 1) \cup \delta(q_1, 1) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$

Accepted

The concept of ϵ-closure

ϵ-closure(T) = T + all NFA states reachable from any state in T using only ϵ-transitions.



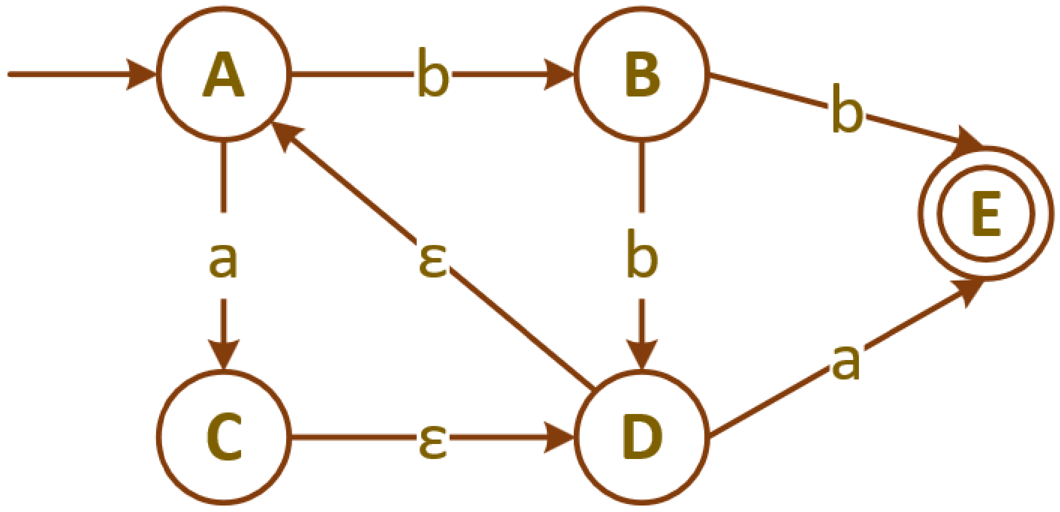
1. ϵ-closure(D)
2. ϵ-closure(A)
3. ϵ-closure(A, B, E)
4. ϵ-closure(C, E)

5. ϵ-closure(δ(A, a))
6. ϵ-closure(δ(A, b))

7. ϵ-closure(δ(B, a))
8. ϵ-closure(δ(B, b))

The concept of ϵ-closure

ϵ-closure(T) = T + all NFA states reachable from any state in T using only ϵ-transitions.

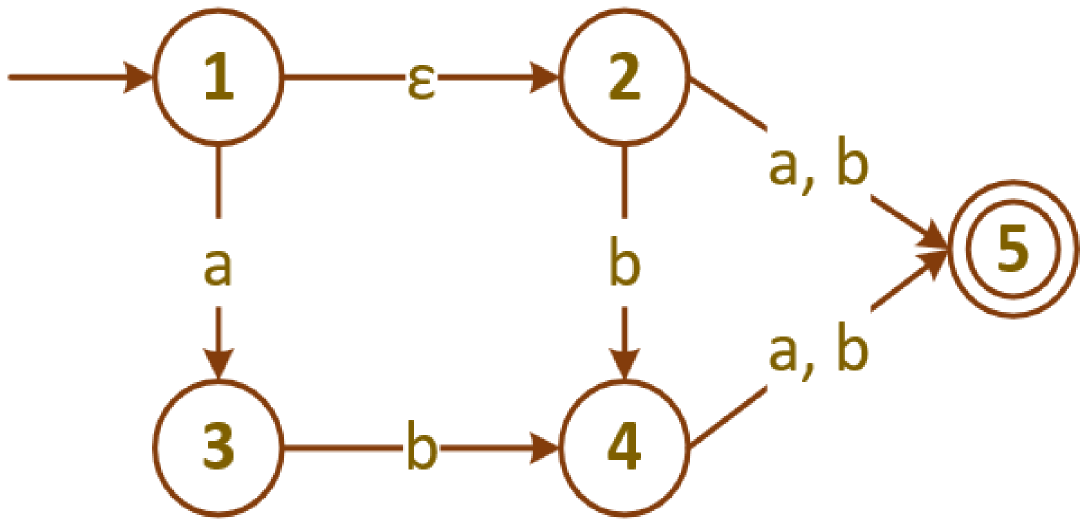


1. ϵ-closure(D) = {D, A}
2. ϵ-closure(A) = {A}
3. ϵ-closure(A, B, E) = {A, B, E}
4. ϵ-closure(C, E) = {A, C, D, E}

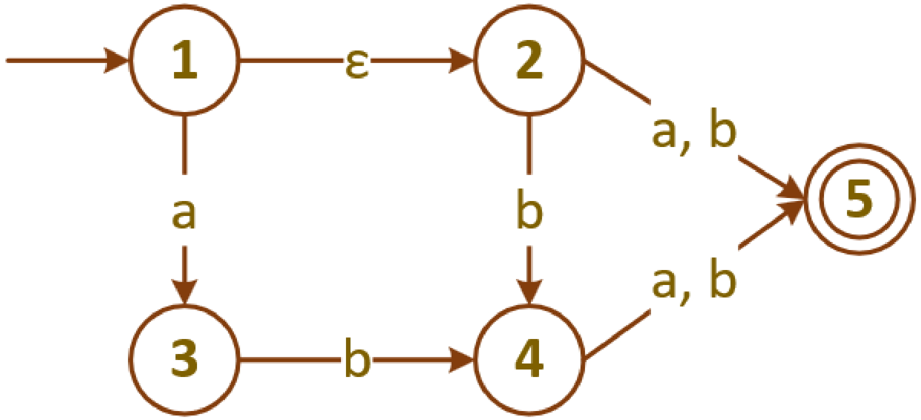
5. ϵ-closure(δ(A, a)) = {C, D, A}
6. ϵ-closure(δ(A, b)) = {B}

7. ϵ-closure(δ(B, a)) = { }
8. ϵ-closure(δ(B, b)) = {E, D, A}

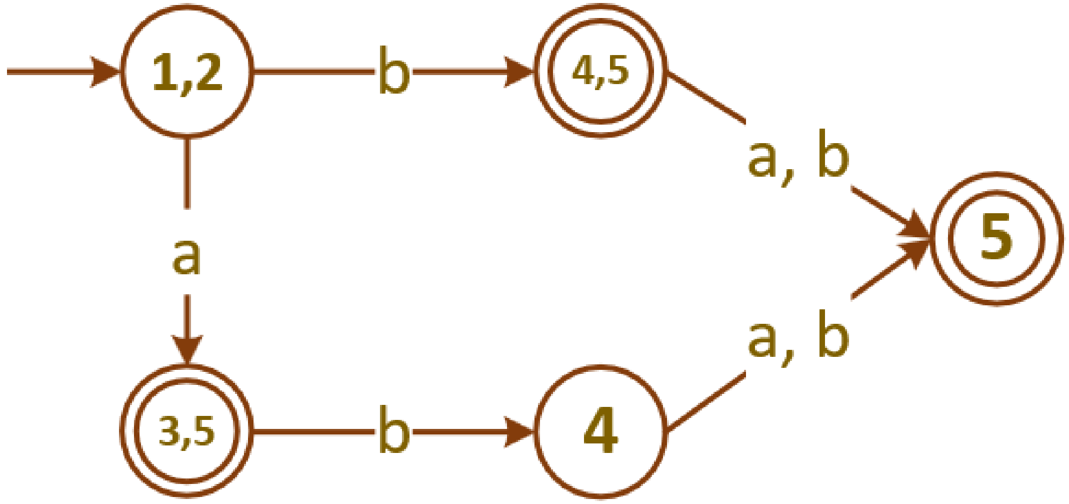
Subset Construction Algorithm: From ε-NFA or NFA to DFA



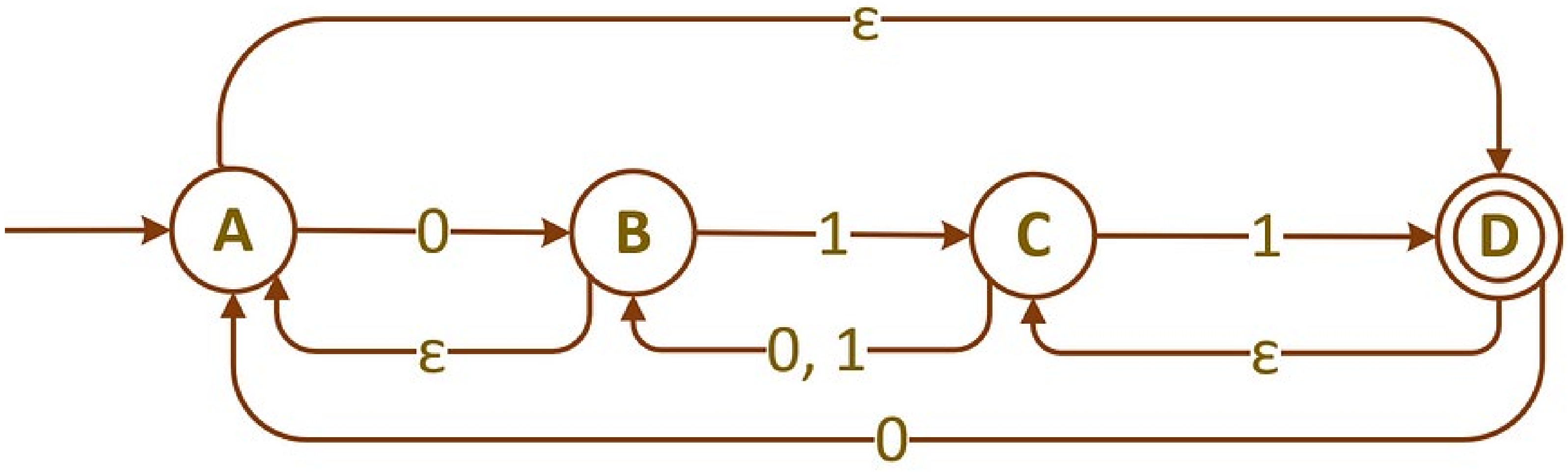
Subset Construction Algorithm: From ε-NFA or NFA to DFA



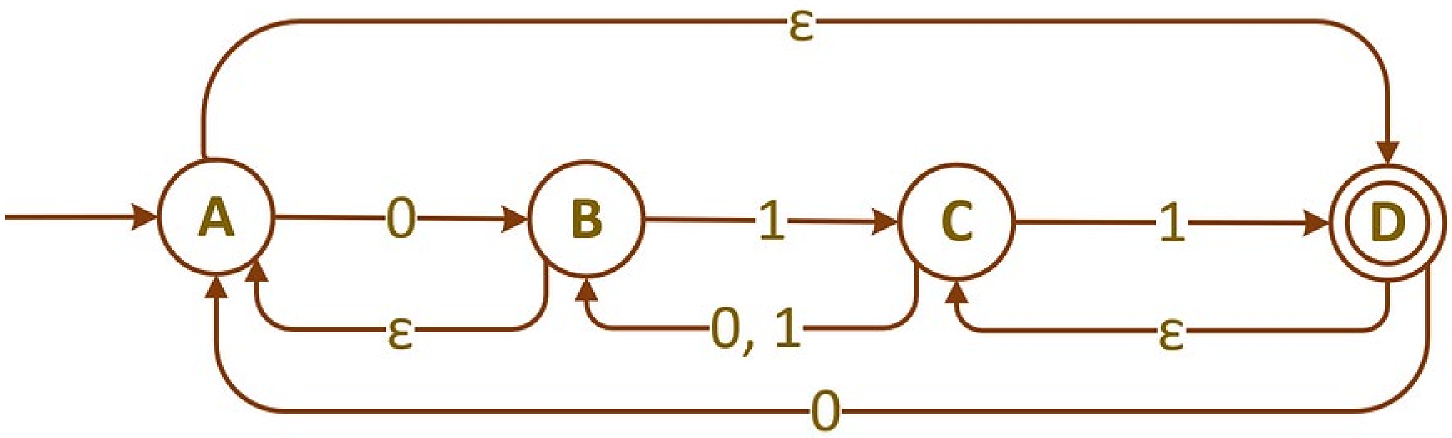
States	a	b
→ {1,2}	{3, 5}	{4, 5}
*{3, 5}	-	{4}
*{4, 5}	{5}	{5}
{4}	{5}	{5}
*{5}	-	-



Subset Construction Algorithm: From ε-NFA or NFA to DFA

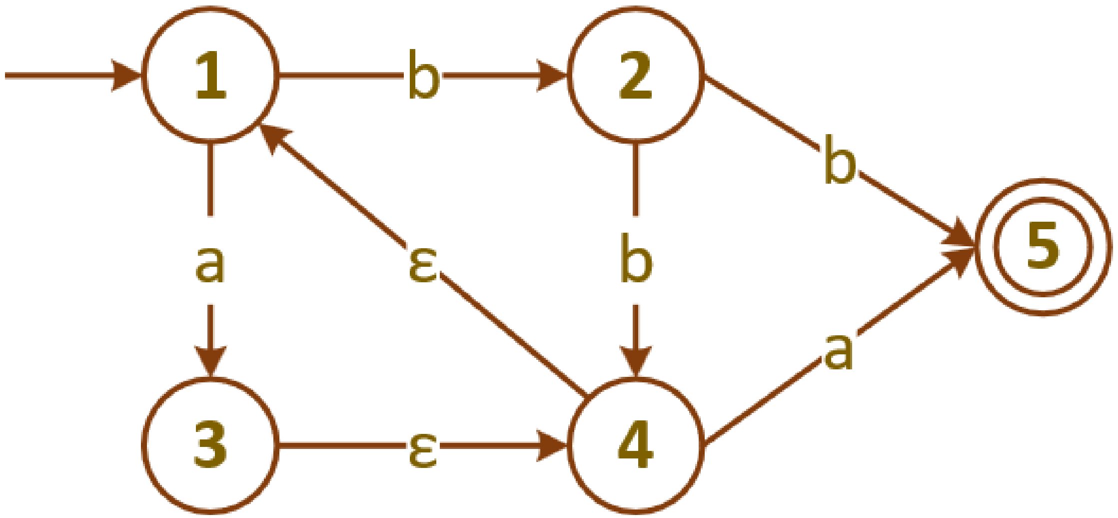


Subset Construction Algorithm: From ε-NFA or NFA to DFA

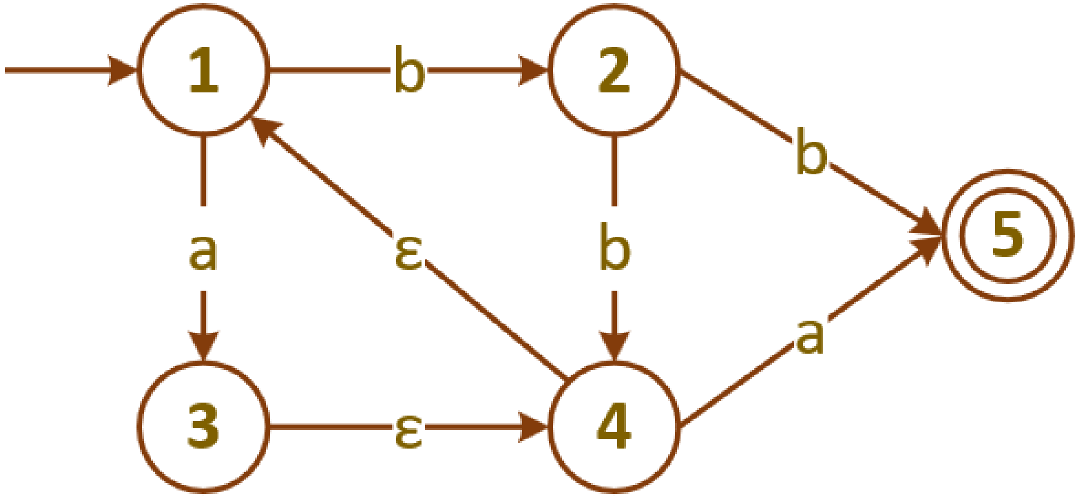


States	0	1
→*{A, D, C}	{A, B, D, C}	{B, A, D, C}
*{A, B, C, D}	{A, B, C, D}	{A, B, C, D}

Subset Construction Algorithm: From ε-NFA or NFA to DFA



Subset Construction Algorithm: From ε-NFA or NFA to DFA



States	a	b
➔ {1}	{1, 3, 4}	{2}
{2}	-	{1, 4, 5}
{1, 3, 4}	{1, 3, 4, 5}	{2}
*{1, 4, 5}	{1, 3, 4, 5}	{2}
*{1, 3, 4, 5}	{1, 3, 4, 5}	{2}

Subset Construction Algorithm: From ε-NFA or NFA to DFA

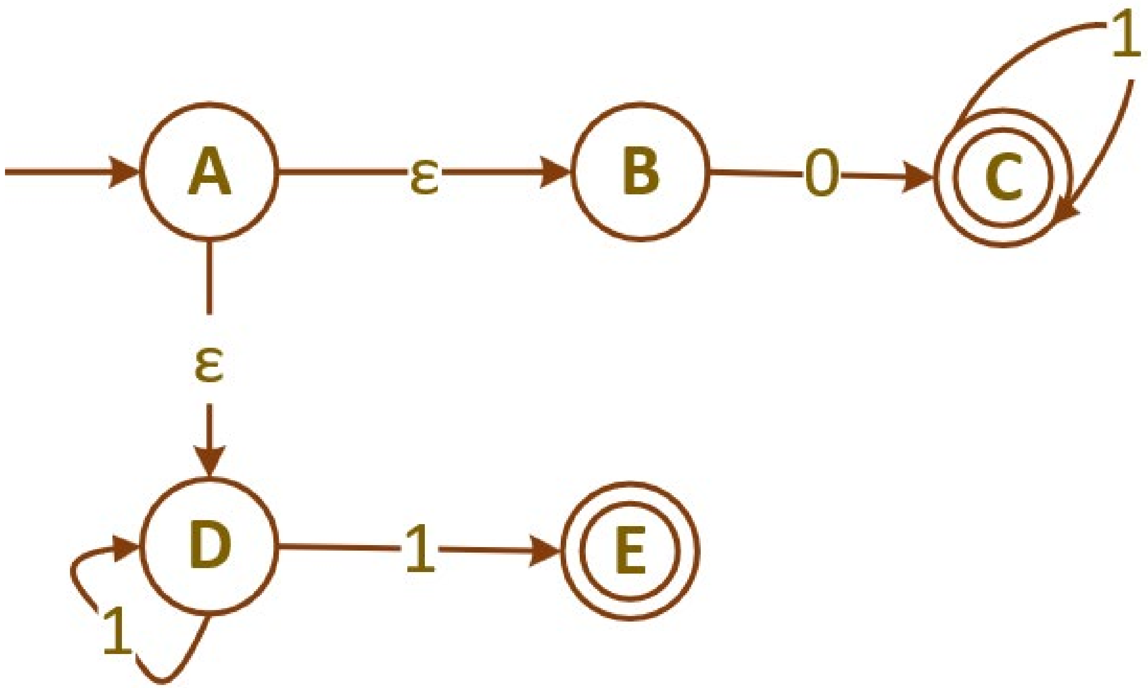


Table Filling Method: DFA Minimization and Equivalence Testing

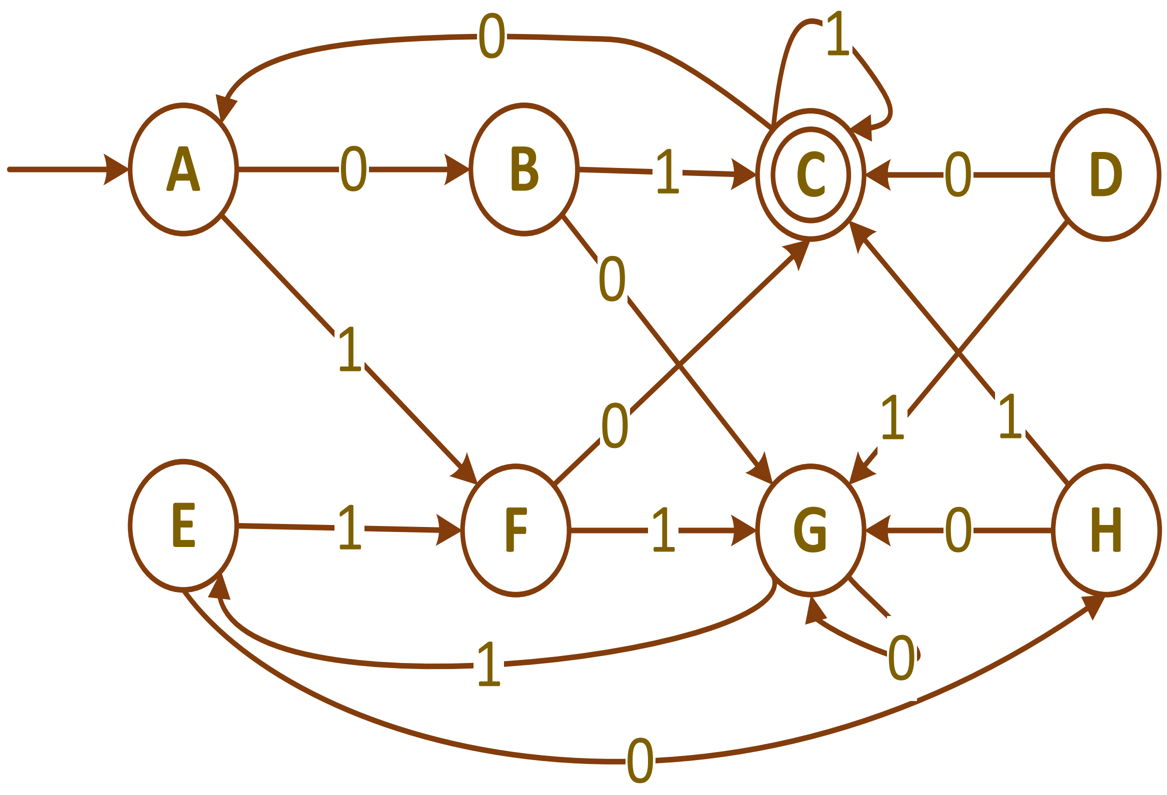
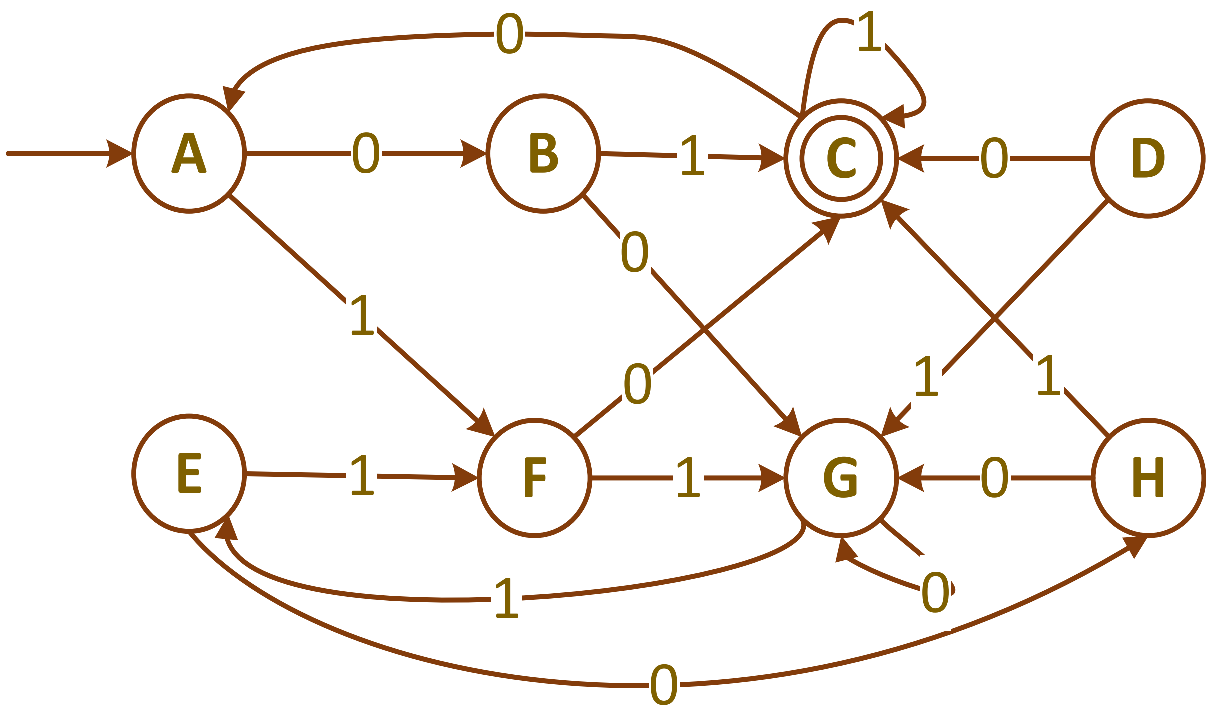
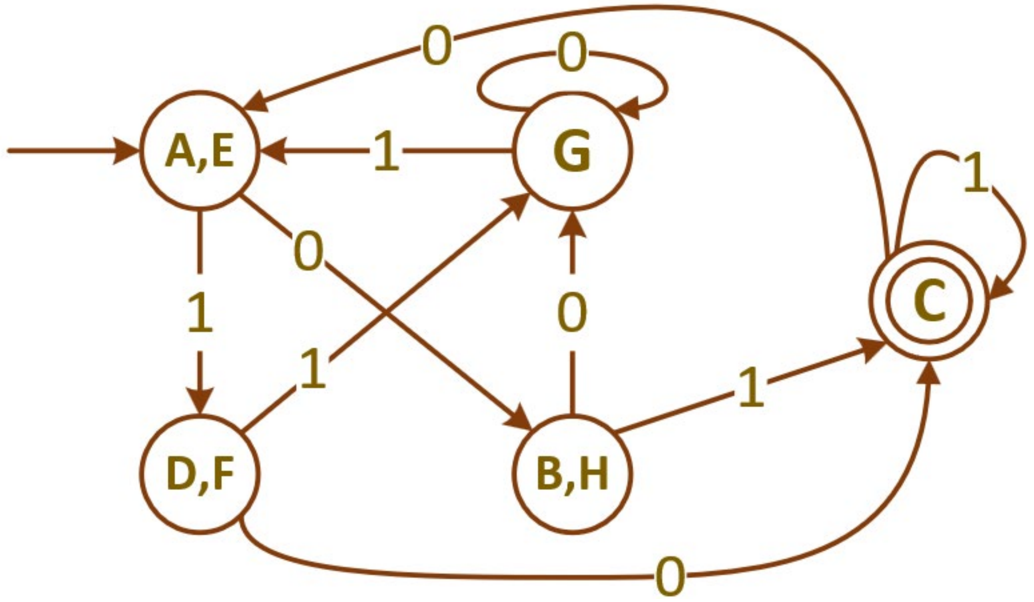


Table Filling Method: DFA Minimization

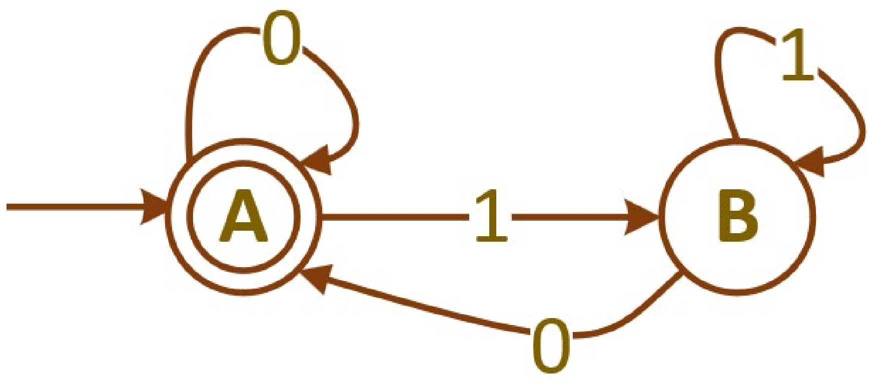


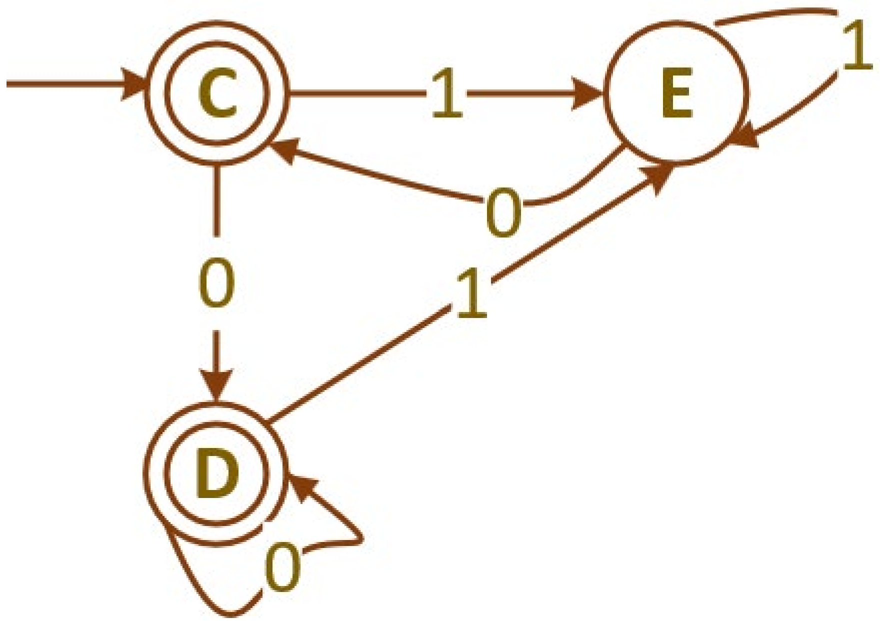
Original DFA



Minimized DFA

Table Filling Method: Equivalence Testing





Finite Automata with Outputs: Mealy and Moore Machines

A Moore machine is a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, where

- Q is a finite set of states
- Σ is the input alphabet
- Δ is the output alphabet
- δ is the transition function **$\delta: Q \times \Sigma \rightarrow Q$**
- λ is the output function **$\lambda: Q \rightarrow \Delta$**
- $q_0 \in Q$ is the initial state.

A Mealy machine is a six-tuple $(Q, \Sigma, \Delta, \delta, \lambda, q_0)$, same as Moore machine except λ

Output Transition Function (λ): In the Mealy machine, the output is determined both by its current state and the input it's currently reading. This function, represented as $\lambda: \Sigma \times Q \rightarrow \Delta$, maps each combination of a state and an input symbol to a corresponding output from the output alphabet.

Moore Machine

Present State	Transition Function – δ			
	Next state for Input: *x*	Next state for Input: *y*	Next state for Input: *z*	Output λ
→A	B	A	E	*7*
B	B	C	D	*8*
C	E	E	C	*7*
D	A	D	C	*9*
E	D	B	A	*8*

Input String: *y* *y* *x* *z* *z* *y*

$$A \xrightarrow{y} A \xrightarrow{y} A \xrightarrow{x} B \xrightarrow{z} D \xrightarrow{z} C \xrightarrow{y} E$$

Output String: *7* *7* *7* *8* *9* *7* *8*

Mealy Machine

	Next State					
	Input: *a*		Input: *b*		Input: c	
Present State	State	Output	State	Output	State	Output
→A	B	0	E	1	D	0
B	B	1	D	b	D	b
C	E	b	C	0	B	0
D	A	b	F	1	C	0
E	D	1	A	1	B	b
F	E	0	C	0	F	1

Input String		*b*	*b*	*c*	*a*	*c*	*b*
	A →	E →	A →	D →	A →	D →	F
Output String		1	1	0	b	0	1

Transforming a Mealy Machine into a Moore Machine

	Next State			
	Input: 0		Input: 1	
Present State	State	Output	State	Output
→q_1	q_3	*0*	q_2	*0*
q_2	q_1	*1*	q_4	*0*
q_3	q_2	*1*	q_1	*1*
q_4	q_4	*1*	q_3	*0*

	Next State			
	Input: 0		Input: 1	
Present State	State	Output	State	Output
→q_1	q_3	*0*	q_{20}	*0*
q_{20}	q_1	*1*	q_{40}	*0*
q_{21}	q_1	*1*	q_{40}	*0*
q_3	q_{21}	*1*	q_1	*1*
q_{40}	q_{41}	*1*	q_3	*0*
q_{41}	q_{41}	*1*	q_3	*0*

	Transition Function – δ		
Present State	Next state for Input: *0*	Next state for Input: 1	Output λ
→q_1	q_3	q_{20}	*1*
q_{20}	q_1	q_{40}	*0*
q_{21}	q_1	q_{40}	*1*
q_3	q_{21}	q_1	*0*
q_{40}	q_{41}	q_3	*0*
q_{41}	q_{41}	q_3	*1*

	Transition Function – δ		
Present State	Next state for Input: *0*	Next state for Input: 1	Output λ
→q_{dummy}	q_3	q_{20}	---
q_1	q_3	q_{20}	*1*
q_{20}	q_1	q_{40}	*0*
q_{21}	q_1	q_{40}	*1*
q_3	q_{21}	q_1	*0*
q_{40}	q_{41}	q_3	*0*
q_{41}	q_{41}	q_3	*1*

Transforming a Moore Machine into a Mealy Machine

	Transition Function – δ		
Present State	Next state for Input: *0*	Next state for Input: 1	Output λ
→q_0	q_3	q_1	*0*
q_1	q_1	q_2	*1*
q_2	q_2	q_3	*0*
q_3	q_3	q_0	*0*

Transforming a Moore Machine into a Mealy Machine

	Transition Function – δ		
Present State	**Next state for Input: *0***	**Next state for Input: 1**	**Output λ**
→q_0	q_3	q_1	*0*
q_1	q_1	q_2	*1*
q_2	q_2	q_3	*0*
q_3	q_3	q_0	*0*

	Next State			
	Input: 0		Input: 1	
Present State	**State**	**Output**	**State**	**Output**
→q_0	q_3	*0*	q_1	*1*
q_1	q_1	*1*	q_2	*0*
q_2	q_2	*0*	q_3	*0*
q_3	q_3	*0*	q_0	*0*

Transforming a Moore Machine into a Mealy Machine

	Transition Function – δ			
Present State	Next state for Input: *x*	Next state for Input: *y*	Next state for Input: *z*	Output λ
→A	B	A	E	*7*
B	B	C	D	*8*
C	E	E	C	*7*
D	A	D	C	*9*
E	D	B	A	*8*

Transforming a Moore Machine into a Mealy Machine

	Transition Function – δ			
Present State	Next state for Input: *x*	Next state for Input: *y*	Next state for Input: *z*	Output λ
→A	B	A	E	*7*
B	B	C	D	*8*
C	E	E	C	*7*
D	A	D	C	*9*
E	D	B	A	*8*

	Next State					
	Input: *x*		Input: *y*		Input: *z*	
Present State	State	Output	State	Output	State	Output
→A	B	*8*	A	*7*	E	*8*
B	B	*8*	C	*7*	D	*9*
C	E	*8*	E	*8*	C	*7*
D	A	*7*	D	*9*	C	*7*
E	D	*9*	B	*8*	A	*7*

Transforming a Moore Machine into a Mealy Machine

	Transition Function – δ		
Present State	Next state for Input: *0*	Next state for Input: 1	Output λ
→q_1	q_1	q_2	*0*
q_2	q_1	q_3	*0*
q_3	q_1	q_3	*1*

	Next State			
	Input: 0		Input: 1	
Present State	State	Output	State	Output
→q_1	q_1	*0*	q_2	*0*
q_2	q_1	*0*	q_3	*1*
q_3	q_1	*0*	q_3	*1*

	Next State			
	Input: 0		Input: 1	
Present State	State	Output	State	Output
→q_1	q_1	*0*	q_2	*0*
q_3	q_1	*0*	q_3	*1*

Regular Language, Regular Grammar

Regular Expressions (RE)

- The regular expressions are useful for representing certain sets of strings in an algebraic fashion.
- These describe the languages accepted by finite state automata.

We give a formal recursive definition of regular expressions over Σ as follows:

1. Any **terminal symbol a** (an element of Σ), **∈** and **∅** are regular expressions.
2. The **union** of two regular expressions R_1 and R_2, written as $\mathbf{R_1 + R_2}$, is also a regular expression.
3. The **concatenation** of two regular expressions R_1 and R_2, written as $\mathbf{R_1.R_2}$ **(or** $\mathbf{R_1R_2}$**)**, is also a RE.
4. The **iteration (or closure)** of a regular expression R, written as **R***, is also a RE.
5. If R is a regular expression, then **(R)** is also a regular expression.
6. The REs over Σ are those obtained **recursively** by the application of the ***rules 1–5*** once or several times.

Regular Expression Examples

- L_1 = the set of all strings of 0's and 1's ending in 00.
- L_2 = the set of all strings of 0's and 1's beginning with 0 and ending with 1.
- L_3 = $\{\epsilon$, 11, 1111, 111111, ...$\}$.

Regular Expression Examples (cont.)

- L_1 = the set of all strings of 0's and 1's ending in 00.
  - (0+1)*00
- L_2 = the set of all strings of 0's and 1's beginning with 0 and ending with 1.
  - 0(0+1)*1
- L_3 = {ϵ, 11, 1111, 111111, ...}.
  - (11)*

Identities of Regular Expressions

- Two regular expressions P and Q are equivalent (we write P = Q) if P and Q represent the same set of strings.

$$I_1 : \emptyset + R = R$$

$$I_2 : \emptyset R = R\emptyset = \emptyset$$

$$I_3 : \epsilon R = R\epsilon = R$$

$$I_4 : \epsilon^* = \epsilon \text{ and } \emptyset^* = \epsilon$$

$$I_5 : R + R = R$$

$$I_6 : R^* R^* = R^*$$

$$I_7 : RR^* = R^* R$$

Identities of Regular Expressions (cont.)

$$I_8 : (R^*)^* = R^*$$

$$I_9 : \epsilon + RR^* = R^* = \epsilon + R^*R$$

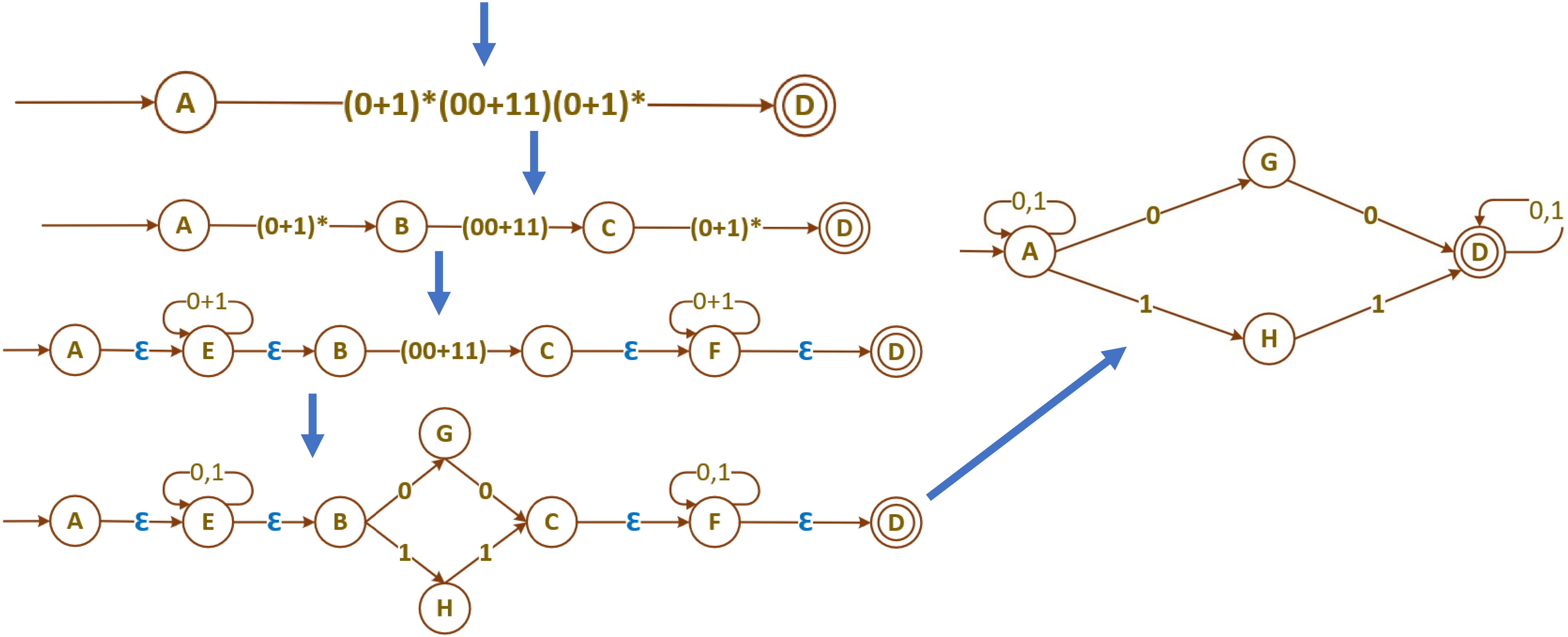
$$I_{10} : (PQ)^*P = P(QP)^*$$

$$I_{11} : (P + Q)^* = (P^*Q^*)^* = (P^* + Q^*)^*$$

$$I_{12} : (P + Q)R = PR + QR \text{ and } R(P + Q) = RP + RQ$$

Construct Finite Automaton equivalent to the Regular Expression

$$(0+1)^*(00+11)(0+1)^*$$



Arden's Theorem

Let P and Q be two regular expressions over Σ. If P does not contain ϵ, then the following equation in R, namely

$$R = Q + RP \cdots\cdots\cdot \quad \textbf{(1)}$$

has a unique solution (i.e. one and only one solution) given by $R = QP^*$.

Proof:

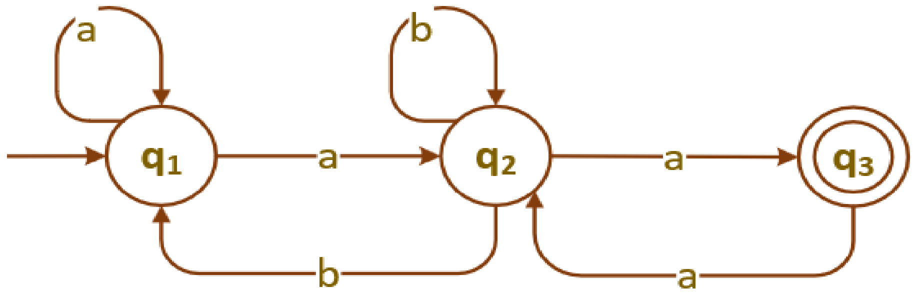
$$Q + (QP^*)P = Q(\epsilon + P^*P) = QP^* \, by \, I_9$$

Hence Equation 1 is satisfied when $R = QP^*$. This means $R = QP^*$ is a solution of Equation 1.

To prove uniqueness, consider Equation 1. Here, replacing R by $Q + RP$ on the R.H.S., we get the equation

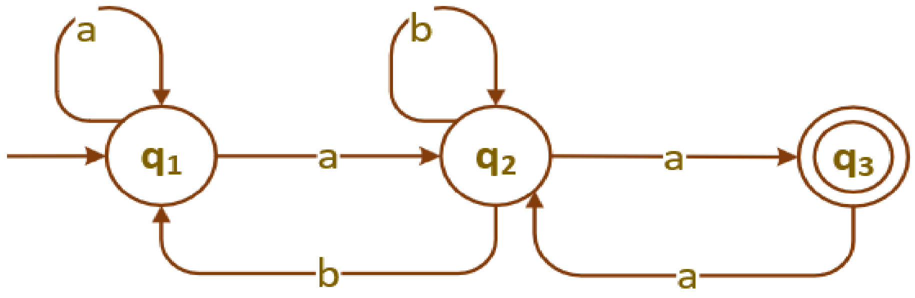
$$
\begin{aligned}
Q + RP &= Q + (Q + RP)P \\
&= Q + QP + RPP \\
&= Q + QP + RP^2 \\
&\vdots \\
&= Q + QP + QP^2 + \cdots + QP^i + RP^{i+1} \\
&= Q(\epsilon + P + P^2 + \cdots + P^i) + RP^{i+1} \\
&= Q(\epsilon + P + P^2 + \cdots + P^i) + QP^*P^{i+1} \\
&= Q(\epsilon + P + P^2 + \cdots + P^i + P^*P^{i+1}) = QP^*
\end{aligned}
$$

Application of Arden’s Theorem: Convert FA to RE



$$
\begin{aligned}
q_1 &= q_1 a + q_2 b + \epsilon \\
q_2 &= q_1 a + q_2 b + q_3 a \\
q_3 &= q_2 a
\end{aligned}
$$

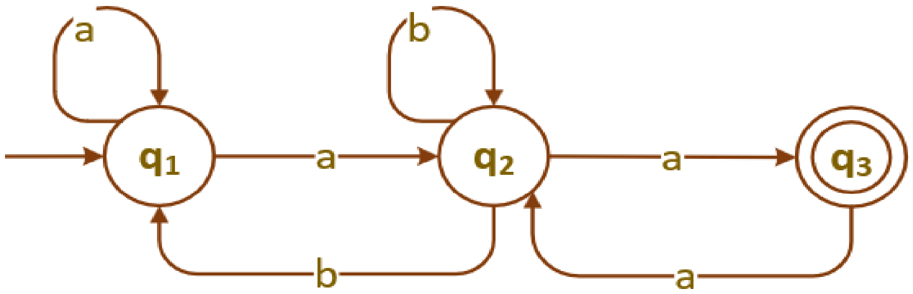
Application of Arden's Theorem: Convert FA to RE



$$
\begin{aligned}
q_1 &= q_1 a + q_2 b + \epsilon \\
q_2 &= q_1 a + q_2 b + q_3 a \\
q_3 &= q_2 a
\end{aligned}
$$

$$
\begin{aligned}
q_2 &= q_1 a + q_2 b + q_2 aa \\
&= q_1 a + q_2 (b + aa) \\
&= q_1 a (b + aa)^*
\end{aligned}
$$

Application of Arden’s Theorem: Convert FA to RE



$$
\begin{aligned}
q_1 &= q_1 a + q_2 b + \epsilon \\
q_2 &= q_1 a + q_2 b + q_3 a \\
q_3 &= q_2 a
\end{aligned}
$$

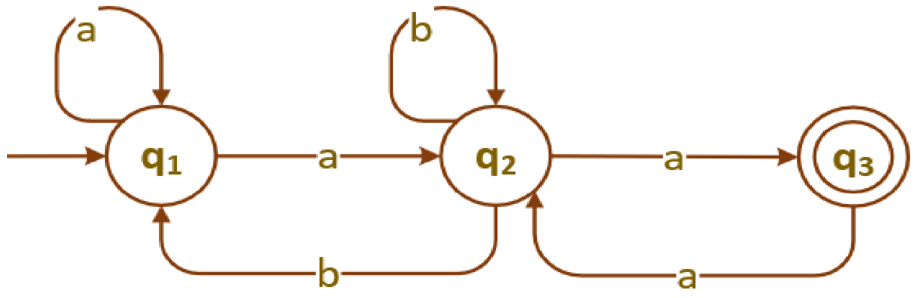
$$
\begin{aligned}
q_2 &= q_1 a + q_2 b + q_2 aa \\
&= q_1 a + q_2 (b + aa) \\
&= q_1 a (b + aa)^*
\end{aligned}
$$

$$
\begin{aligned}
q_1 &= q_1 a + q_1 a (b + aa)^* b + \epsilon \\
&= q_1 (a + a(b + aa)^* b) + \epsilon
\end{aligned}
$$

$$q_1 = \epsilon (a + a(b + aa)^* b)^*$$

$$q_1 = (a + a(b + aa)^* b)^*$$

Application of Arden's Theorem: Convert FA to RE



$$\boxed{\begin{aligned} q_1 &= q_1 a + q_2 b + \epsilon \\ q_2 &= q_1 a + q_2 b + q_3 a \\ q_3 &= q_2 a \end{aligned}}$$

$$\begin{aligned} q_2 &= q_1 a + q_2 b + q_2 aa \\ &= q_1 a + q_2 (b + aa) \\ &= q_1 a (b + aa)^* \end{aligned}$$

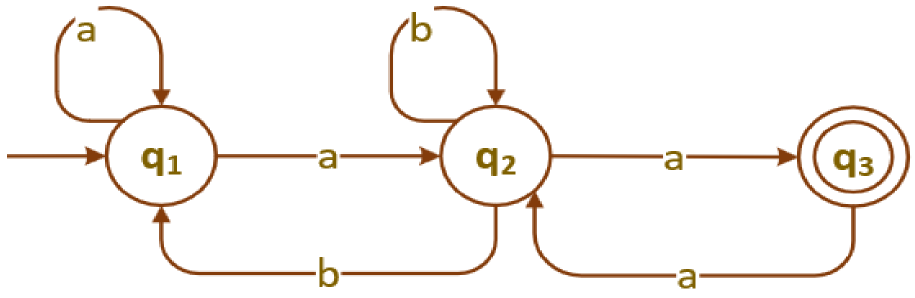
$$\boxed{q_2 = (a + a(b + aa)^* b)^* a (b + aa)^*}$$

$$\begin{aligned} q_1 &= q_1 a + q_1 a (b + aa)^* b + \epsilon \\ &= q_1 (a + a(b + aa)^* b) + \epsilon \end{aligned}$$

$$q_1 = \epsilon (a + a(b + aa)^* b)^*$$

$$q_1 = (a + a(b + aa)^* b)^*$$

Application of Arden's Theorem: Convert FA to RE



$$\boxed{\begin{aligned} q_1 &= q_1a + q_2b + \epsilon \\ q_2 &= q_1a + q_2b + q_3a \\ q_3 &= q_2a \end{aligned}}$$

$$\begin{aligned} q_2 &= q_1a + q_2b + q_2aa \\ &= q_1a + q_2(b + aa) \\ &= q_1a(b + aa)^* \end{aligned}$$

$$\begin{aligned} q_1 &= q_1a + q_1a(b + aa)^*b + \epsilon \\ &= q_1(a + a(b + aa)^*b) + \epsilon \end{aligned}$$

$$q_1 = \epsilon(a + a(b + aa)^*b)^*$$

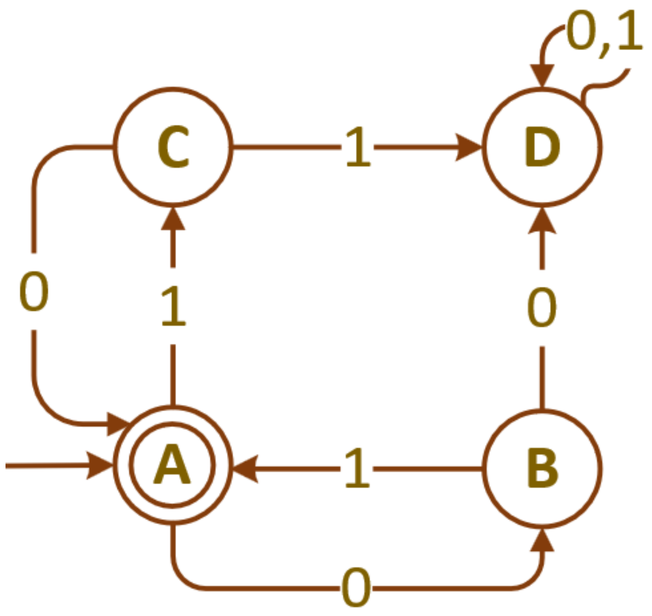
$$q_1 = (a + a(b + aa)^*b)^*$$

$$\boxed{q_2 = (a + a(b + aa)^*b)^*a(b + aa)^*}$$

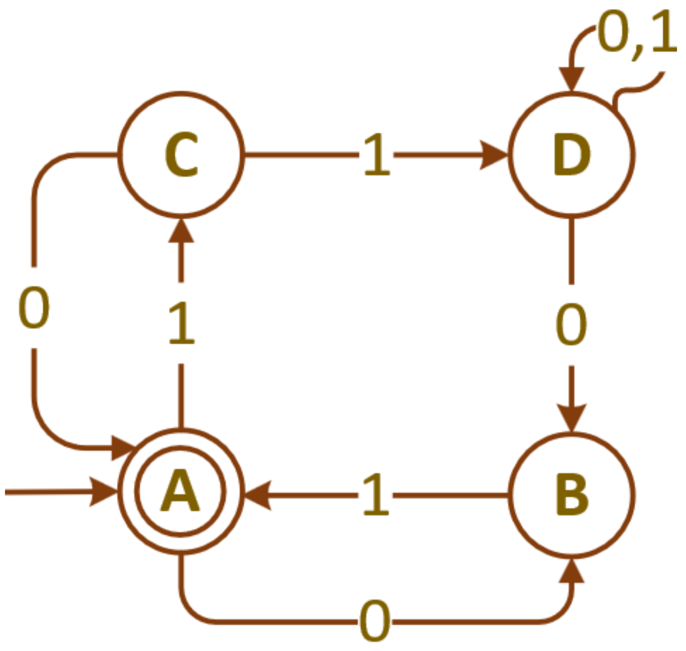
$$\boxed{q_3 = (a + a(b + aa)^*b)^*a(b + aa)^*a}$$

Since q_3 is a final state, the set of strings recognized by the graph is $(a + a(b + aa)^*b)^*a(b + aa)^*a$

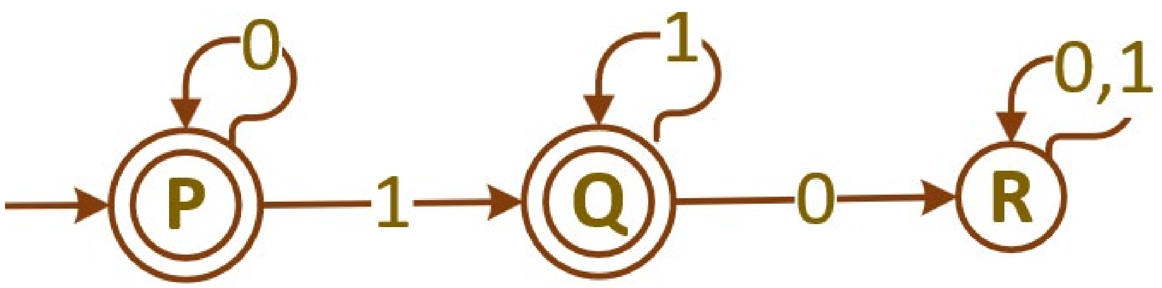
Application of Arden’s Theorem: Convert FA to RE



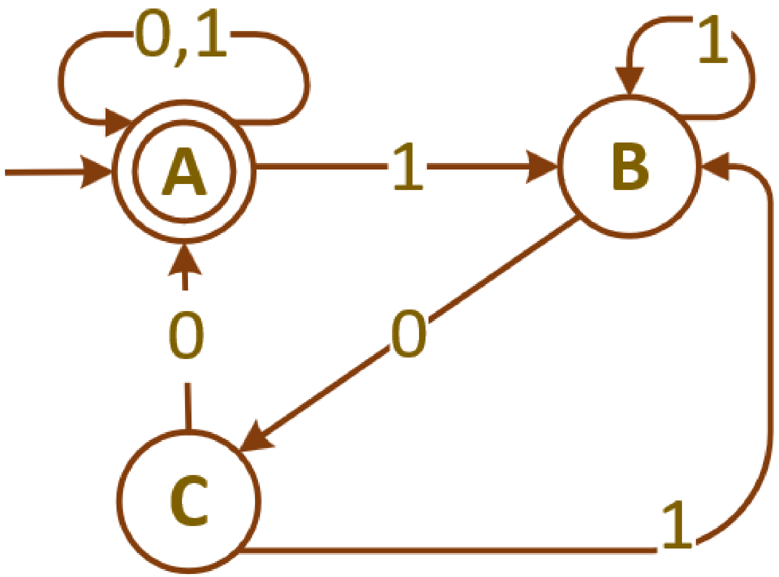
Application of Arden's Theorem: Convert FA to RE



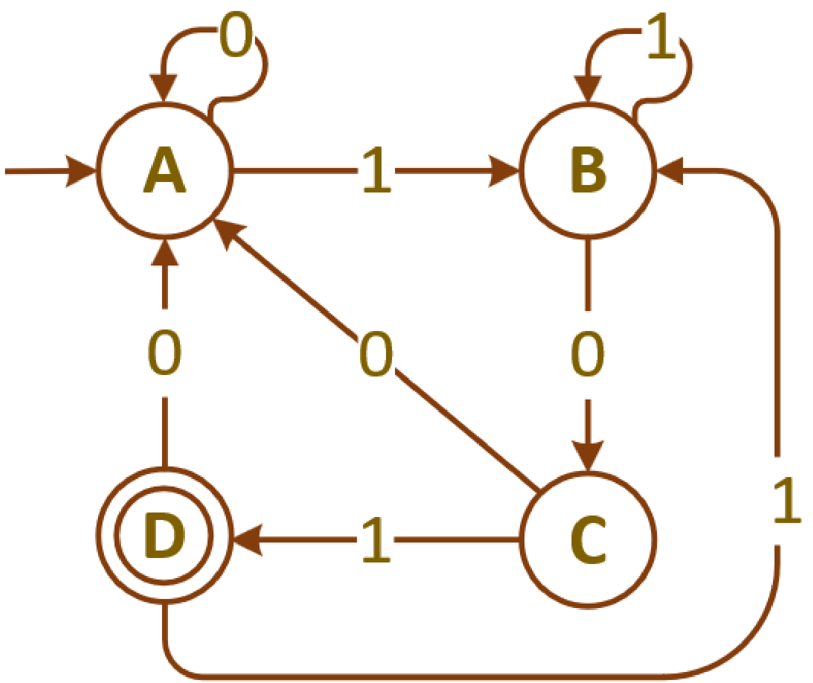
Application of Arden's Theorem: Convert FA to RE



Application of Arden's Theorem: Convert FA to RE



Application of Arden’s Theorem: Convert FA to RE



Pumping Lemma

- The Pumping Lemma is a fundamental concept in the theory of formal languages, particularly useful for demonstrating that certain languages are not regular.
  - However, the converse—that a language satisfying the conditions of the Pumping Lemma is necessarily regular—is not true.

- This lemma offers a systematic approach to demonstrate the non-regularity of languages by exploiting the inherent "repetitive" structure required of regular languages.
-
- Such a structural insight reveals why some languages, due to their complexity, cannot be captured by regular expressions or finite automata.

- If a language **L** is regular, then there exists an integer p (called the pumping length) such that any string s in L with length at least p can be divided into three parts, ***s=xyz***, satisfying the following conditions:

1. For each $i \geq 0$, the string xy^iz is in L.
2. $|y| > 0$ (i.e., y is not empty).
3. $|xy| \leq p$.

Pumping Lemma

- If a language **L** is regular, then there exists an integer p (called the pumping length) such that any string s in L with length at least p can be divided into three parts, ***s=xyz***, satisfying the following conditions:

1. For each $i \geq 0$, the string xy^iz is in L.
2. $|y| > 0$ (i.e., y is not empty).
3. $|xy| \leq p$.

- The lemma essentially says that every long enough string in a regular language can be "pumped" or repeated in a certain segment (denoted as y) without leaving the language.
- When processing a long string, a finite automaton must enter at least one state more than once (due to the pigeonhole principle), creating a loop that can be repeated.

Pumping Lemma

- If a language **L** is regular, then there exists an integer p (called the pumping length) such that any string s in L with length at least p can be divided into three parts, ***s=xyz***, satisfying the following conditions:

  1. For each $i \geq 0$, the string xy^iz is in L.
  2. $|y| > 0$ (i.e., y is not empty).
  3. $|xy| \leq p$.

1. Assume that L is regular.
2. By the lemma, there exists a pumping length p.
3. Construct or find a string s in L with length at least p, such that no matter how s is divided into xyz, with $|xy| \leq p$ and $|y| > 0$, at least one of the pumped strings xy^iz for some $i \geq 0$ is not in L.
4. This contradiction implies that L is not regular.

Pumping Lemma

- If a language **L** is regular, then there exists an integer p (called the pumping length) such that any string s in L with length at least p can be divided into three parts, ***s=xyz***, satisfying the following conditions:

1. For each $i \geq 0$, the string xy^iz is in L.
2. $|y| > 0$ (i.e., y is not empty).
3. $|xy| \leq p$.

Language: $L = \{0^n1^n \mid n \geq 1\}$

1. Assume that L is regular.
2. By the lemma, there exists a pumping length p.
3. Construct or find a string s in L with length at least p, such that no matter how s is divided into xyz, with $|xy| \leq p$ and $|y| > 0$, at least one of the pumped strings xy^iz for some $i \geq 0$ is not in L.
4. This contradiction implies that L is not regular.

Proof of Non-regularity using Pumping Lemma:

1. Assume L is regular.
2. Let p be the pumping length given by the lemma.
3. Choose $s = 0^p1^p$. This string is in L and has length $2p$, which is greater than p.
4. According to the lemma, s can be split into xyz, where $|xy| \leq p$ and $|y| > 0$. This ensures y consists only of 0s (since the first p characters of s are all 0s).
5. Pumping y (i.e., repeating y), the string $xy^2z = 0^{p+|y|}1^p$ should also be in L, but this string does not have equal numbers of 0s and 1s, contradicting the definition of L.
6. Hence, L is not regular.

Pumping Lemma

Assume that the language L of palindromes is regular. According to the Pumping Lemma, there exists a pumping length p.

According to the Pumping Lemma, s can be decomposed into three parts $s = xyz$ such that:

1. For each $i \geq 0$, the string $xy^i z$ is in L.
2. $|y| > 0$.

Given these constraints, y consists only of a's because $|xy| \leq p$ means y is part of the initial sequence of a's.

Pump y by setting $i = 0$ and $i > 1$:

- **Pumping Down** ($i = 0$): $xy^0 z = xz$ results in $s = a^{p-|y|} b^p a^p$, which is not a palindrome because the symmetry is broken.
- **Pumping Up** ($i > 1$): $xy^2 z = a^{p+|y|} b^p a^p$, which also isn't a palindrome because the numbers of a's before and after b's don't match.

Both resulting strings fail to be palindromes, contradicting the lemma's stipulation that $xy^i z$ must remain in L for all i.

Context Free Grammars and Languages

$$G = (V, T, P, S)$$

- V – a set of variables, e.g. {S, A, B, C, D, E}
- T – a set of terminals, e.g. {a, b, c}
- P – a set of productions rules

  In the form of $A \rightarrow \alpha$, where $A \in V$, $\alpha \in (V \cup T)^*$
- S is a special variable called the **start symbol**

A Context Free Grammar (CFG) example

$G = (V, T, P, S)$

- V – a set of variables, e.g. {S, A, B, C, D, E}
- T – a set of terminals, e.g. {a, b, c}
- P – a set of productions rules

  In the form of $A \rightarrow \alpha$, where $A \in V$, $\alpha \in (V \cup T)^*$
- S is a special variable called the **start symbol**

$$P \rightarrow \epsilon$$
$$P \rightarrow 0$$
$$P \rightarrow 1$$
$$P \rightarrow 0P0$$
$$P \rightarrow 1P1$$

OR

$$P \rightarrow \epsilon \mid 0 \mid 1 \mid 0P0 \mid 1P1$$

Two ways to represent a context free grammar: palindromes over 0's and 1's

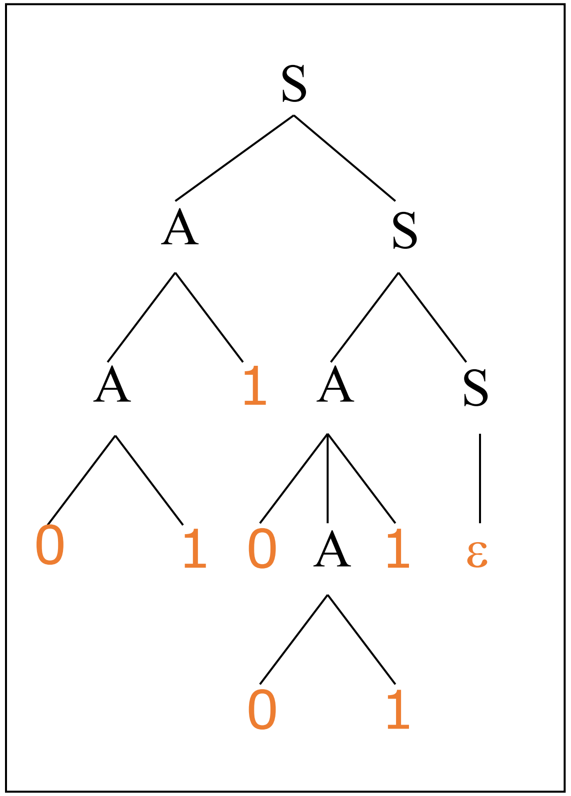
Parse tree and Derivation in CFGs

How a string ω is generated by a grammar G?

$S \rightarrow AS \mid \varepsilon$

$A \rightarrow A1 \mid 0A1 \mid 01$

$S \Rightarrow^* 0110011\ ?$



Parse Tree or Derivation Tree

$$
\begin{aligned}
S &\Rightarrow AS \\
&\Rightarrow A1S \\
&\Rightarrow 011S \\
&\Rightarrow 011AS \\
&\Rightarrow 0110A1S \\
&\Rightarrow 0110011S \\
&\Rightarrow 0110011\varepsilon \\
&\Rightarrow 0110011
\end{aligned}
$$

Leftmost Derivation

$$
\begin{aligned}
S &\Rightarrow AS \\
&\Rightarrow AAS \\
&\Rightarrow AA\,\varepsilon \\
&\Rightarrow A0A1\varepsilon \\
&\Rightarrow A0011\varepsilon \\
&\Rightarrow A10011\varepsilon \\
&\Rightarrow 0110011\varepsilon \\
&\Rightarrow 0110011
\end{aligned}
$$

Rightmost Derivation

Parse tree and Derivation in CFGs

$P \rightarrow Bb \mid aSa \mid a$

$A \rightarrow a \mid aSa$

$B \rightarrow aBaC \mid b$

$C \rightarrow aSa$

$P \Rightarrow^* aabaaaaba\ ?$

$A \rightarrow Bac \mid bTC \mid ba$

$T \rightarrow aB \mid TB \mid \varepsilon$

$B \rightarrow aTB \mid bBC \mid b$

$C \rightarrow TBc \mid aBC \mid ac$

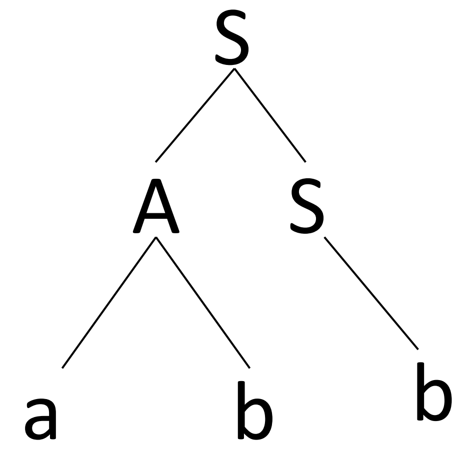
$A \Rightarrow^* babbcac?$

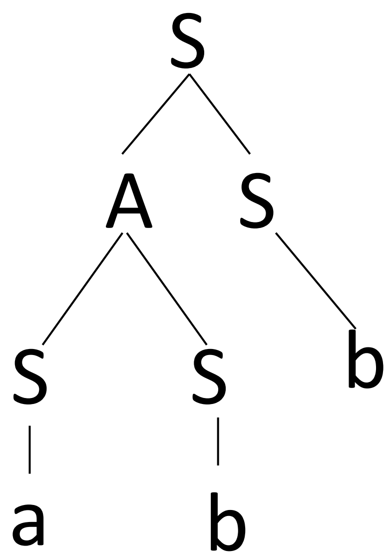
Ambiguity in CFGs

- Each parse tree has one unique leftmost derivation and one unique rightmost derivation.
- A grammar is considered ambiguous if there exists at least one string that can be generated by the grammar in more than one way (i.e., has more than one distinct parse tree).
    - This implies that the string has more than one leftmost derivation or more than one rightmost derivation.
    - Ambiguity in grammars is a critical issue because it can lead to confusion in parsing and interpreting the strings generated by the grammar, especially in compiler design and natural language processing.

Consider the following grammar G:

$$S \rightarrow AS \mid a \mid b$$
$$A \rightarrow SS \mid ab$$





Ambiguity in CFGs (cont.)

- Each parse tree has one unique leftmost derivation and one unique rightmost derivation.
- A grammar is considered ambiguous if there exists at least one string that can be generated by the grammar in more than one way (i.e., has more than one distinct parse tree).

Consider the following grammar:

$$E \rightarrow E + E \mid E * E \mid (E) \mid x \mid y \mid z$$

Is the grammar ambiguous?

Ambiguity in CFGs (cont.)

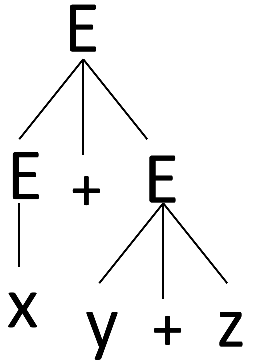
- Each parse tree has one unique leftmost derivation and one unique rightmost derivation.
- A grammar is considered ambiguous if there exists at least one string that can be generated by the grammar in more than one way (i.e., has more than one distinct parse tree).

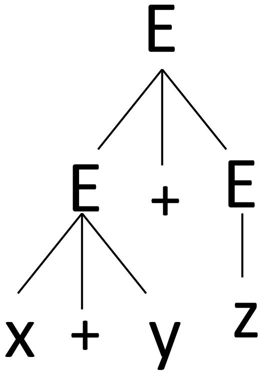
Consider the following grammar:

$$E \rightarrow E + E \mid E * E \mid (E) \mid x \mid y \mid z$$

Is the grammar ambiguous?

$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow x + E \\ &\Rightarrow x + E + E \\ &\Rightarrow x + y + E \\ &\Rightarrow x + y + z \end{aligned}$$





$$\begin{aligned} E &\Rightarrow E + E \\ &\Rightarrow E + E + E \\ &\Rightarrow x + E + E \\ &\Rightarrow x + y + E \\ &\Rightarrow x + y + z \end{aligned}$$

Construction of Reduced Grammar

Simplification of Context Free Grammars (CFGs)

- ❑ Remove useless symbols
  - Generating Variables
  - Reachable symbols (variables and terminals)
- ❑ Remove ε-productions, e.g. $A \rightarrow \varepsilon$
- ❑ Remove unit-productions, e.g. $A \rightarrow B$

Construction of Reduced Grammar

Remove non-generating variables and keep productive variables

$S \to AB$
$A \to a$
$B \to b \mid D$
$E \to c$

- The set $W_1 = \{A, B, E\}$ includes the symbols that have productions with a terminal string on the right-hand side (RHS): $A \to a$; $B \to b$; $E \to c$
- $W_2 = \{S, A, B, E\}$, S is included in list as it can generate through A **and** B
- $W_3 = \{S, A, B, E\} = W_2$ //the stopping condition has been met

Now remove all non-generating variables and construct the revised grammar:

$S \to AB$
$A \to a$
$B \to b$
$E \to c$

Construction of Reduced Grammar

Orifinal CFG:- $S \rightarrow AB$; $A \rightarrow a$; $B \rightarrow b \mid D$; $E \rightarrow c$

Remove non-reachable symbols

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b$

$E \rightarrow c$

- $W_1 = \{S\}$ // start with start symbol **S**
- $W_2 = \{S, A, B\}$ // from $S \rightarrow AB$, we can reach A and B
- $W_2 = \{S, A, B\} \cup \{a, b\}$ //from A and B, terminal a and b can be reached
- $W_3 = \{S, A, B\} \cup \{a, b\} = W_3$ //the stopping condition has been met

Now remove all non-reachable symbols and construct the revised grammar:

$$\boxed{\begin{array}{l} S \rightarrow AB \\ A \rightarrow a \\ B \rightarrow b \end{array}}$$

- **For this example, 40% of the original CFG has been removed without compromising on the solution quality.**

Construction of Reduced Grammar

Remove useless symbols

$S \rightarrow aAa$

$A \rightarrow Sb \mid bCC \mid DaA$

$C \rightarrow abb \mid DD$

$E \rightarrow aC$

$D \rightarrow aDA$

Construction of Reduced Grammar

Remove useless symbols

$S \rightarrow aAa$

$A \rightarrow Sb \mid bCC \mid DaA$

$C \rightarrow abb \mid DD$

$E \rightarrow aC$

$D \rightarrow aDA$

Reduced grammar

$S \rightarrow aAa$

$A \rightarrow Sb \mid bCC$

$C \rightarrow abb$

Construction of Reduced Grammar

Remove useless symbols

$$S \rightarrow AB \mid CA$$
$$B \rightarrow BC \mid AB$$
$$A \rightarrow a$$
$$C \rightarrow aB \mid b$$

Construction of Reduced Grammar

Remove useless symbols

$$S \rightarrow AB \mid CA$$
$$B \rightarrow BC \mid AB$$
$$A \rightarrow a$$
$$C \rightarrow aB \mid b$$

Reduced grammar

$$S \rightarrow CA$$
$$A \rightarrow a$$
$$C \rightarrow b$$

Construction of Reduced Grammar

Remove useless symbols

$A \rightarrow AB \mid CA$

$B \rightarrow DC \mid AB \mid ab$

$C \rightarrow a$

$D \rightarrow aB \mid b$

Reduced grammar

None of the productions are usefull !!!

Construction of Reduced Grammar

Remove Null Productions

$S \rightarrow aS \mid AB$

$A \rightarrow \epsilon$

$B \rightarrow \epsilon$

$D \rightarrow b$

Construction of Nullable variables

$W_1 = \{A, B\}$

$W_2 = \{S, A, B\}$

$W_3 = \{S, A, B\} = W_2$ //the stopping condition has been met

Final CFG after removing null productions

$S \rightarrow aS \mid a \mid AB \mid A \mid B$
$D \rightarrow b$

Construction of Reduced Grammar

Remove Unit Productions

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow C \mid b$

$C \rightarrow D$

$D \rightarrow E$

$E \rightarrow ab \mid aBa$

$W(S) = \{S\}$
$W(A) = \{A\}$
$W(B) = \{B, C, D, E\}$
$W(C) = \{C, D, E\}$
$W(D) = \{D, E\}$
$W(E) = \{E\}$

Final CFG after removing unit productions

$S \rightarrow AB$

$A \rightarrow a$

$B \rightarrow b \mid ab \mid aBa$

$C \rightarrow ab \mid aBa$

$D \rightarrow ab \mid aBa$

$E \rightarrow ab \mid aBa$

Construction of Reduced Grammar

Remove Unit Productions

$S \rightarrow aS \mid a \mid AB \mid A \mid B$
$D \rightarrow b$

$W(S) = \{S, A, B\}$
$W(A) = \{A\}$
$W(B) = \{B\}$
$W(D) = \{D\}$

Final CFG after removing unit productions

$S \rightarrow aS \mid a \mid AB$
$D \rightarrow b$

What happens if we remove useless symbols as well?

Chomsky Normal Form - CNF

Chomsky Normal Form (CNF) is a way of organizing and simplifying the production rules of a context-free grammar (CFG) to assist in various computational processes.

A CFG is in Chomsky Normal Form if all of its production rules satisfy one of the following conditions:

- A non-terminal produces exactly two non-terminals, **$A \rightarrow BC$,** where A, B, and C are non-terminal symbols.
- A non-terminal produces exactly one terminal symbol **$A \rightarrow 0$**, where 0 is a terminal symbol.
- Optionally, a rule that allows the start symbol to produce the empty string, **$A \rightarrow \varepsilon$**, if necessary for deriving ε from the grammar.

Chomsky Normal Form - CNF

Converting a general CFG to Chomsky Normal Form involves several steps:

1. Remove ε-productions, Except for the start symbol
2. Remove unit-productions, e.g. $A \rightarrow B$
3. Remove useless symbols (optional)
   I. Generating Variables
   II. Reachable symbols (variables and terminals)
4. Finally convert the resultant CFG to CNF

Chomsky Normal Form - CNF

Reduce the following grammar to its equivalent CNF:

$S \rightarrow aAD$

$A \rightarrow aB \mid bAB$

$B \rightarrow b$

$D \rightarrow d$

Since there is no null or unit production in the original grammar, we can directly start converting it to CNF.

$S \rightarrow aAD$ ⟶ $S \rightarrow PAD$, $P \rightarrow a$ ⟶ $S \rightarrow PQ$, $P \rightarrow a$, $Q \rightarrow AD$

$A \rightarrow aB$ ⟶ $A \rightarrow PB$

$A \rightarrow bAB$, $B \rightarrow b$, $D \rightarrow d$ ⟶ $A \rightarrow BAB$, $B \rightarrow b$, $D \rightarrow d$ ⟶ $A \rightarrow BR$, $R \rightarrow AB$, $B \rightarrow b$, $D \rightarrow d$

CNF:

$S \rightarrow PQ$

$P \rightarrow a$

$Q \rightarrow AD$

$A \rightarrow PB$

$A \rightarrow BR$

$R \rightarrow AB$

$B \rightarrow b$

$D \rightarrow d$

Chomsky Normal Form – CNF (cont.)

Reduce the following grammar to its equivalent CNF:

$S \rightarrow aAbB$

$A \rightarrow aA \mid a$

$B \rightarrow bB \mid b$

$D \rightarrow b$

Since there is no null or unit production in the original grammar, we can directly start converting it to CNF.

$S \rightarrow aAbD \Rightarrow$ $S \rightarrow PADB$, $P \rightarrow a$, $D \rightarrow b$ $\Rightarrow$ $S \rightarrow MN$, $M \rightarrow PA$, $N \rightarrow DB$, $P \rightarrow a$, $D \rightarrow b$

$A \rightarrow aA$, $A \rightarrow a$ $\Rightarrow$ $A \rightarrow PA$, $A \rightarrow a$

$B \rightarrow bB$, $B \rightarrow b$ $\Rightarrow$ $B \rightarrow DB$, $B \rightarrow b$

CNF:

$S \rightarrow MN$

$M \rightarrow PA$

$N \rightarrow DB$

$P \rightarrow a$

$A \rightarrow PA \mid a$

$B \rightarrow DB$

$B \rightarrow b$

$D \rightarrow d$

The CYK algorithm – a dynamic programming approach for parsing

- The CYK algorithm (Cocke-Younger-Kasami algorithm) is a prominent parsing algorithm for context-free grammars, particularly useful when the grammar is presented in CNF.
- The CYK algorithm is used to decide whether a given string belongs to the language generated by a grammar.
- The algorithm uses **dynamic programming** to build a table (often a triangular array) that represents possible substrings of the input string and their corresponding derivations according to the grammar rules.

CNF grammar **G**

- S → AB | BC
- A → BA | a
- B → CC | b
- C → AB | a

w is baaba

Question Is baaba in L(G)?

Checking Membership: After filling the table, the string w is in the language of the grammar if the starting symbol (usually S) appears in $P[1, n]$ (the cell representing the whole string).

The CYK algorithm – a dynamic programming approach for parsing

CNF grammar **G**

- $S \rightarrow AB \mid BC$
- $A \rightarrow BA \mid a$
- $B \rightarrow CC \mid b$
- $C \rightarrow AB \mid a$

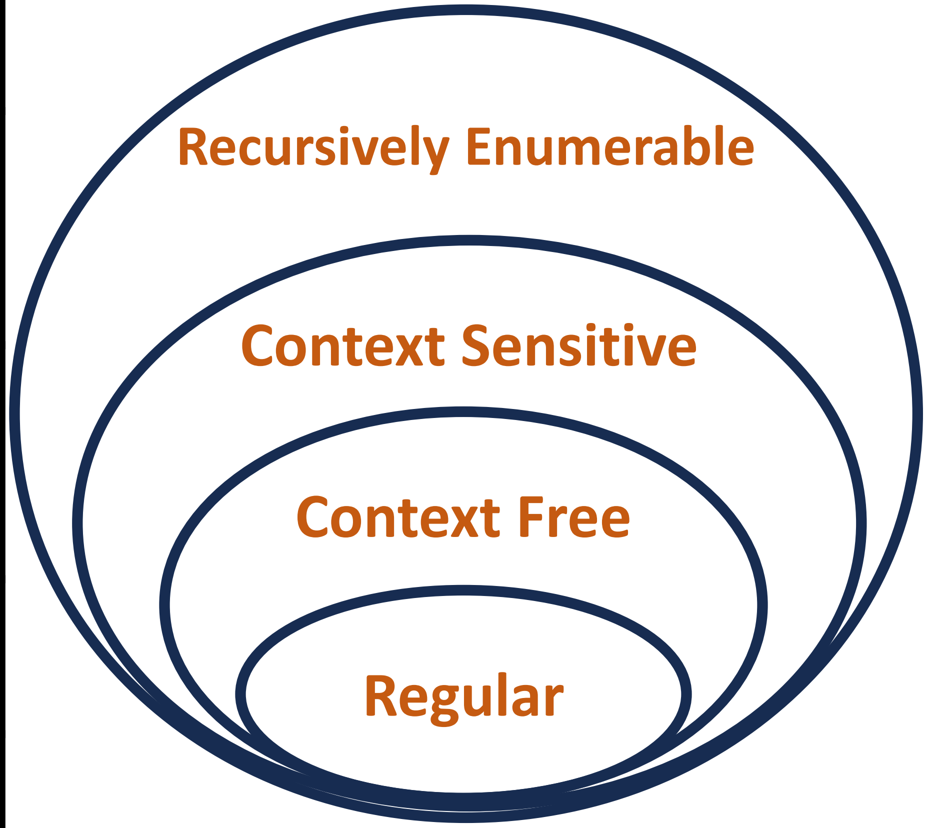
{S, A, C}				
Ø	{S, A, C}			
Ø	{B}	{B}		
{S, A}	{B}	{S, C}	{S, A}	
{B}	{A, C}	{A, C}	{B}	{A, C}
b	a	a	b	a

w is baaba

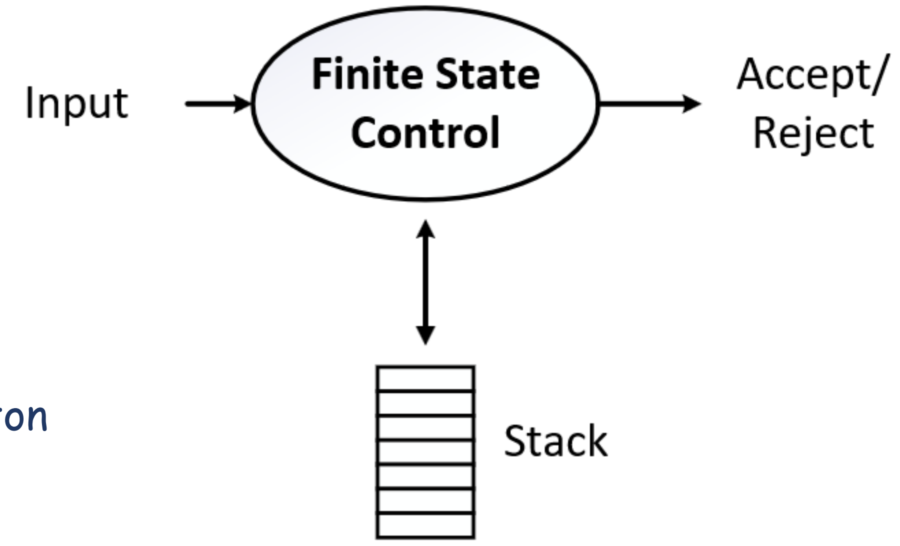
Question Is baaba in L(G)?

Chomsky hierarchy: Chomsky classification of grammars

Grammar Type	Grammar Accepted	Language Accepted	Automation
Type 0	*Unrestricted Grammar*	*Recursively Enumerable Language*	*Turing Machine*
Type 1	*Context Sensitive Grammar*	*Context Sensitive language*	*Linear-bounded automaton*
Type 2	*Context Free Grammar*	*Context Free language*	*Push Down Automata*
Type 3	*Regular Grammar*	*Regular Language*	*Finite Automata*



Push Down Automata



A PDA is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$

- Q is a finite set of states
- Σ is a finite input symbols
- Γ is a finite stack alphabet
- δ is the transition function that governs the behaviour of the automaton

  Formally, δ takes as argument a triple $\delta(q, a, X)$,

  i. q is a state in Q

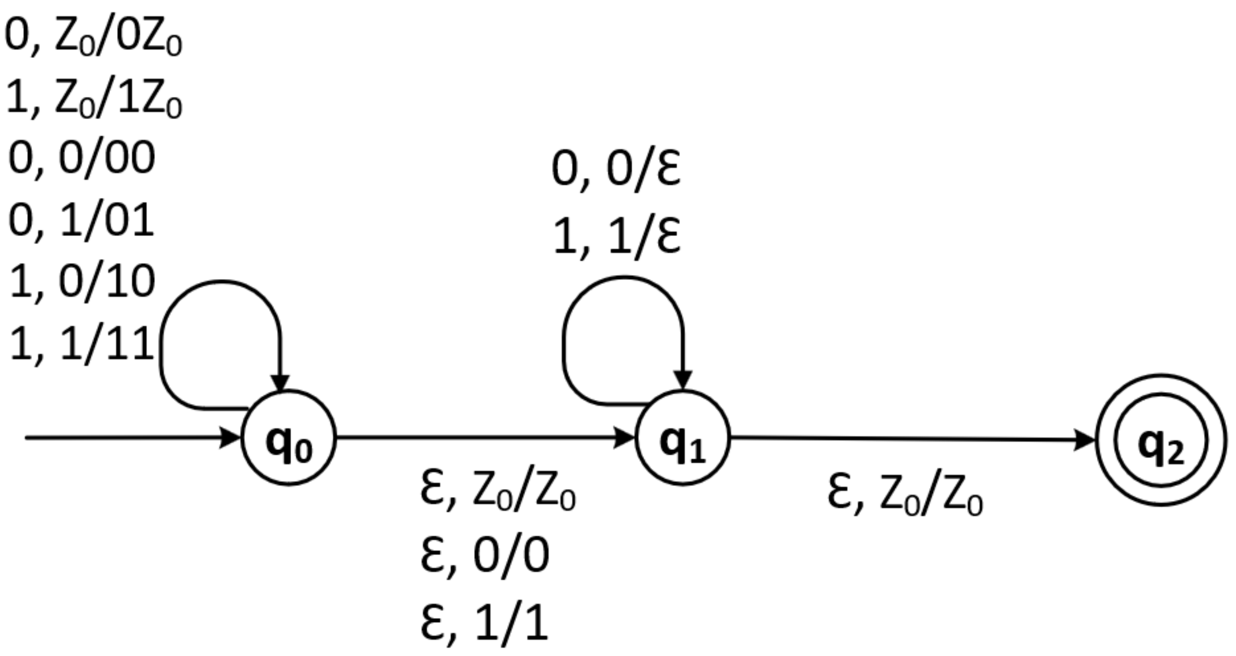
  ii. a is either an input symbol in Σ or $a = \epsilon$, the empty string, which is assumed not to be an input symbol.

  iii. X is a stack symbol, that is, a member of Γ.

  - The output of δ is a finite set of pairs (p, γ), where p is the new state, and γ is the string of stack symbols that replaces X at the top of the stack.
  - For instance, if $\gamma = \epsilon$, then the stack is popped; if $\gamma = X$, then the stack is unchanged, and if $\gamma = YZ$, then X is replaced by Z, and Y is pushed onto the stack.

- $q_0 \in Q$ is the initial state (only one)
- $Z_0 \in \Gamma$ is the initial stack symbol, the PDA's stack consists of one instance of this symbol and nothing else.
- $F \in Q$ is a set of final states (zero or more)

A Push Down Automata (PDA)



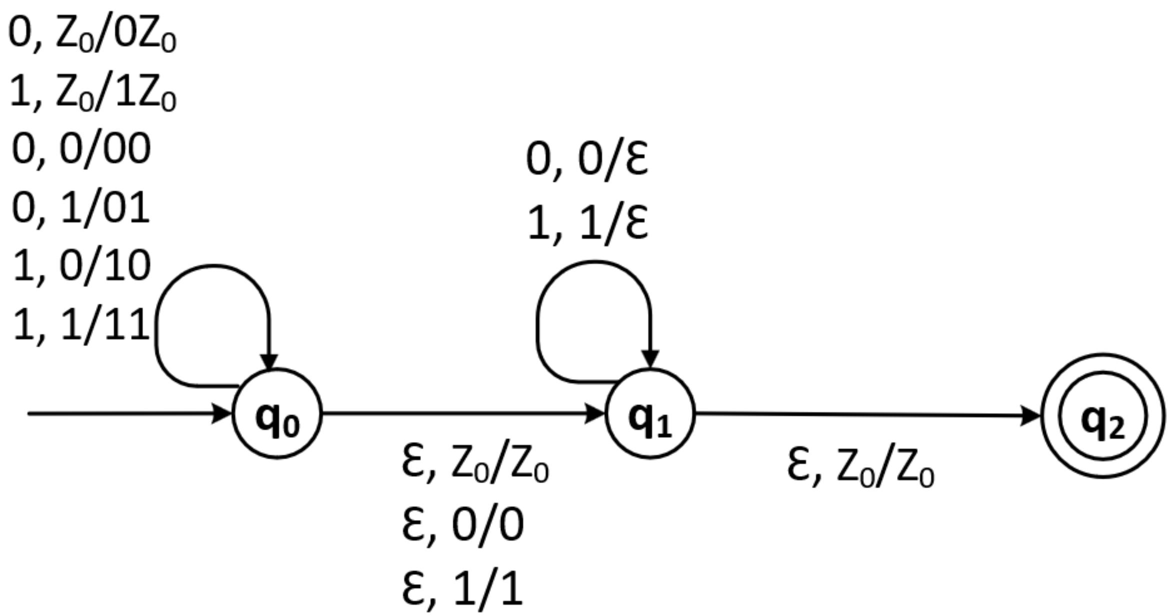
$L = \{ww^R \mid w \text{ is in } (0+1)^*\}$

$P = (\{q_0, q_1, q_2\}, \{0, 1\}, \{0, 1, Z_0\}, \delta, q_0, Z_0, \{q_2\})$

1. $\delta(q_0, 0, Z_0) = \{(q_0, 0Z_0)\}$ and $\delta(q_0, 1, Z_0) = \{(q_0, 1Z_0)\}$. One of these rules applies initially; when we are in state q_0 and we see the start symbol Z_0 at the top of the stack. We read the first input, and push it onto the stack, leaving Z_0 below to mark the bottom.

2. $\delta(q_0, 0, 0) = \{(q_0, 00)\}$, $\delta(q_0, 1, 0) = \{(q_0, 10)\}$, $\delta(q_0, 0, 1) = \{(q_0, 01)\}$, and $\delta(q_0, 1, 1) = \{(q_0, 11)\}$. These four, similar rules allow us to stay in state q_0 and read inputs, pushing each onto the top of the stack and leaving the previous top stack symbol alone.

A Push Down Automata (PDA)



$$L = \{ww^R \mid w \text{ is in } (0+1)^*\}$$

$$P = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\})$$

3. $\delta(q_0, \epsilon, Z_0) = \{(q_1, Z_0)\}$, $\delta(q_0, \epsilon, 0) = \{(q_1, 0)\}$, and $\delta(q_0, \epsilon, 1) = \{(q_1, 1)\}$. These three rules allow P to go from state q_0 to state q_1 spontaneously (on ϵ input), leaving intact whatever symbol is at the top of the stack.

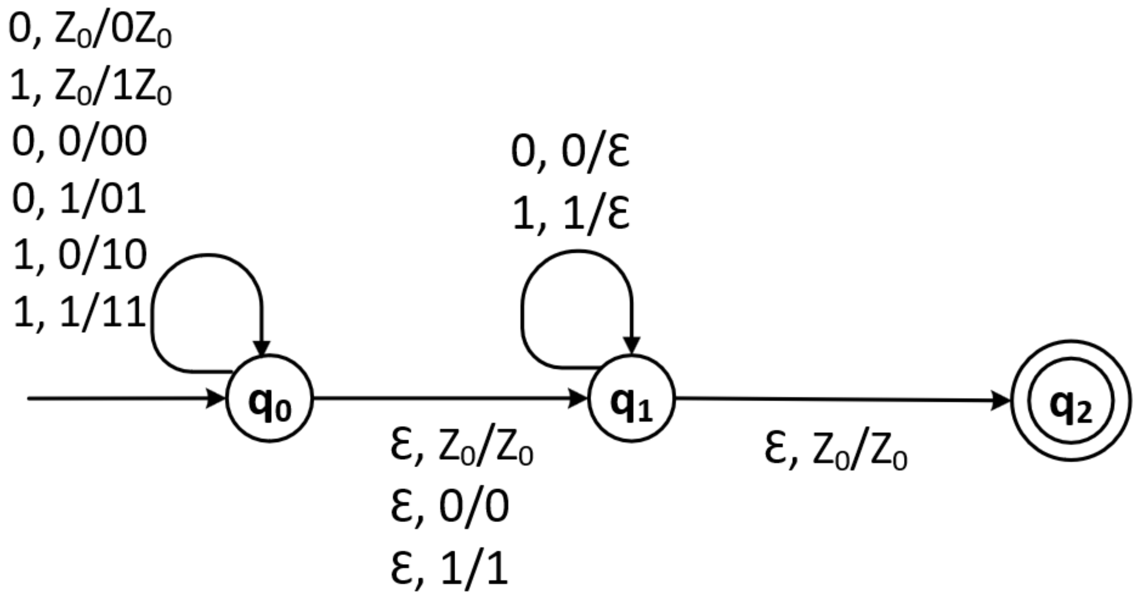
4. $\delta(q_1, 0, 0) = \{(q_1, \epsilon)\}$, and $\delta(q_1, 1, 1) = \{(q_1, \epsilon)\}$. Now, in state q_1 we can match input symbols against the top symbols on the stack, and pop when the symbols match.

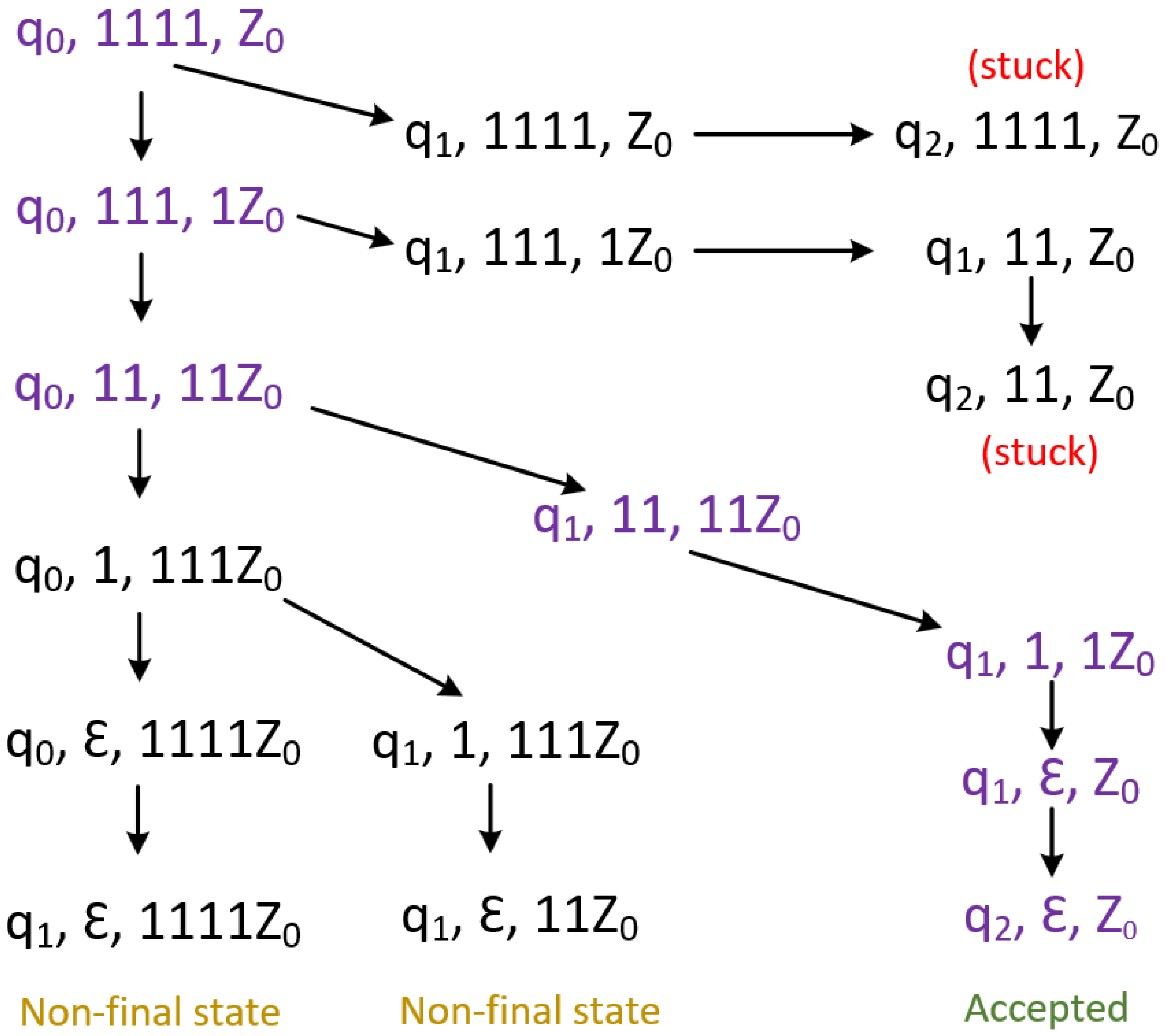
5. $\delta(q_1, \epsilon, Z_0) = \{(q_2, Z_0)\}$. Finally, if we expose the bottom-of-stack marker Z_0 and we are in state q_1, then we have found an input of the form ww^R. We go to state q_2 and accept.

The derivation in a PDA

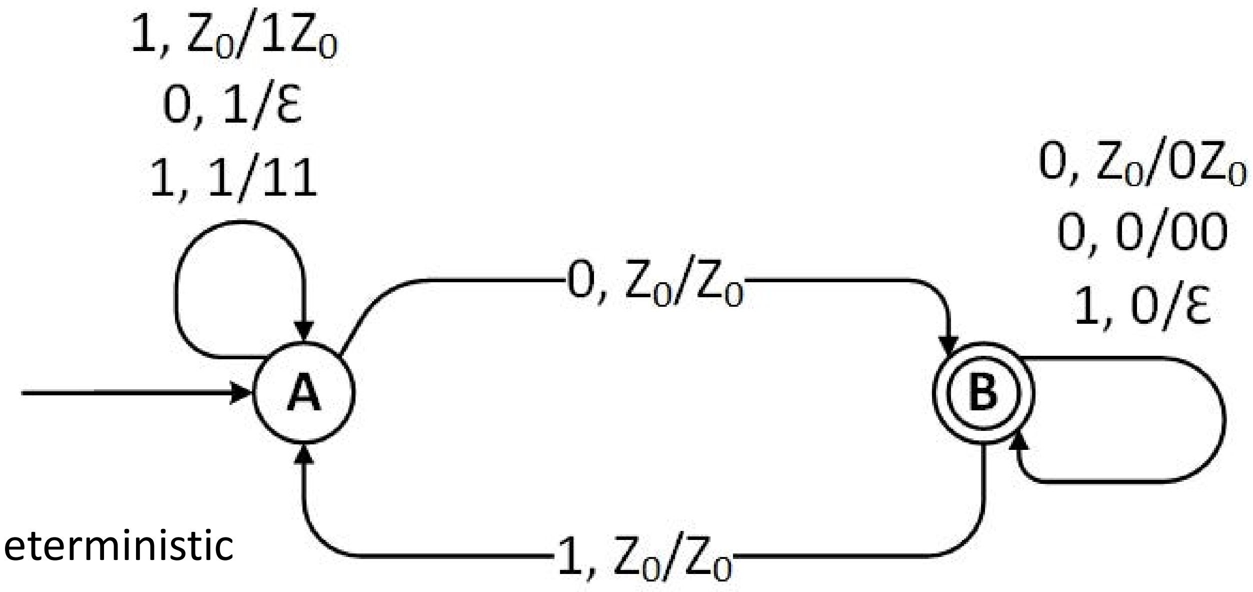
$$L = \{ww^R \mid w \text{ is in } (0+1)^*\}$$

$$P = (\{q_0, q_1, q_2\}, \{0,1\}, \{0,1,Z_0\}, \delta, q_0, Z_0, \{q_2\})$$





Deterministic PDA



A Language **L** that accepts strings with more 0's than 1's.

$$L = \{x \in \{0, 1\}^* \mid n_0\ (x) > n_1\ (x)\}$$

- Deterministic Push Down Automata (DPDA) follows deterministic rules for its transitions.
- a DPDA ensures that for any combination of current state, input symbol, and stack symbol, there is at most one valid transition.

To be a DPDA, the following two conditions must be met:

1. For any state $q \in Q$, any input symbol $s \in \Sigma \cup \varepsilon$, and any stack symbol $t \in \Gamma$, the set $\delta\ (q, s, t)$ has at most one element.

2. For any state $q \in Q$ and any stack symbol $t \in \Gamma$, if $\delta\ (q, \varepsilon, t)$ is not empty, then $\delta\ (q, s, t) = \emptyset$ for each $s \in \Sigma$.

PDA Accepted by Final State

Let P = (Q, Σ, Γ, δ, q_0, Z_0, F) be a PDA. Then $L(P)$ the language accepted by P final state, is

$$L(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\vdash}_P (q, \epsilon, \alpha)\}$$

- for some state q in F and any stack string α.
- That is, starting in the initial ID with w waiting on the input, P consumes w from the input and enters an accepting state.
- The contents of the stack at that time is irrelevant.

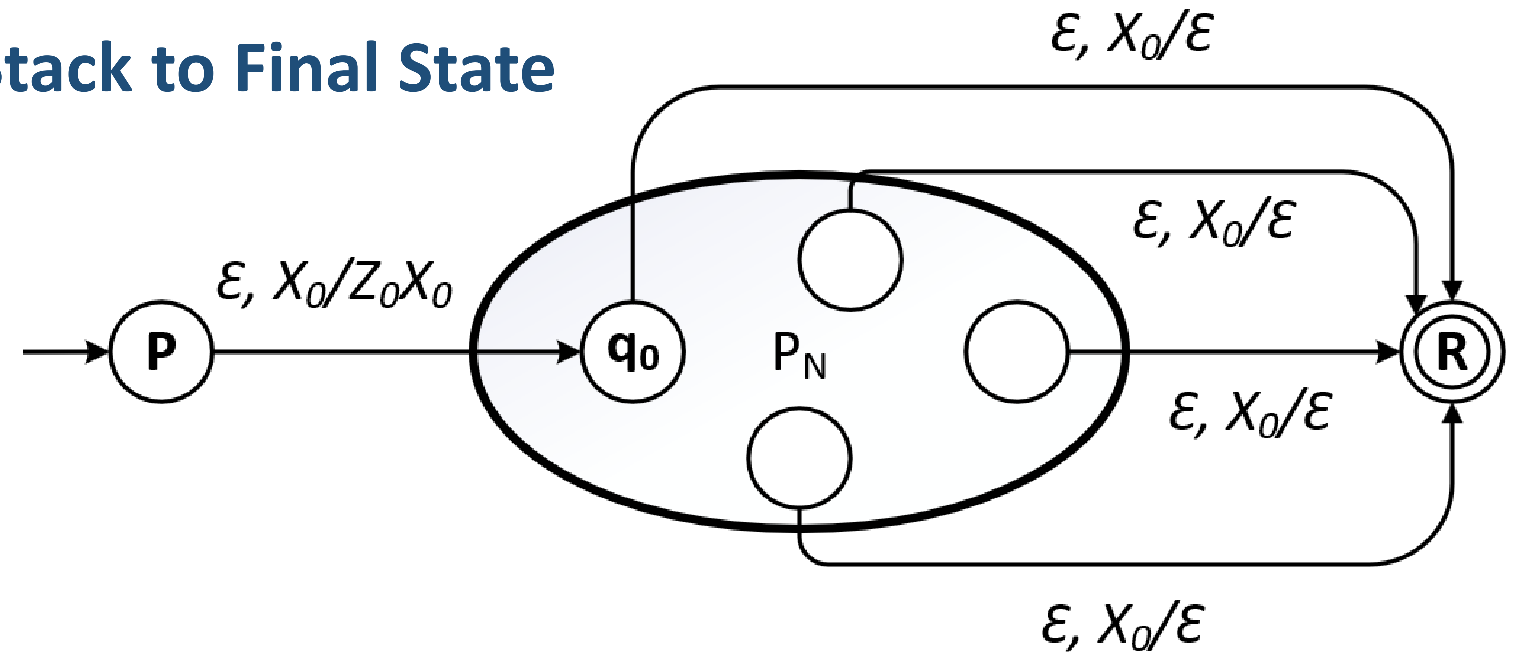
PDA Accepted by Empty Stack

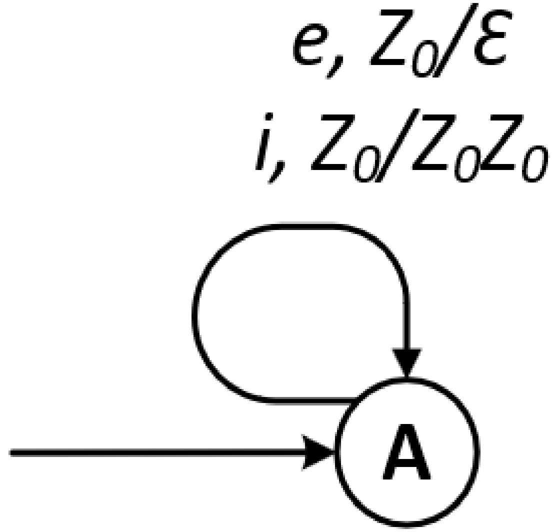
Let P = (Q, Σ, Γ, δ, q_0, Z_0, F) be a PDA. We define

$$N(P) = \{w \mid (q_0, w, Z_0) \overset{*}{\vdash}_P (q, \epsilon, \epsilon)\}$$

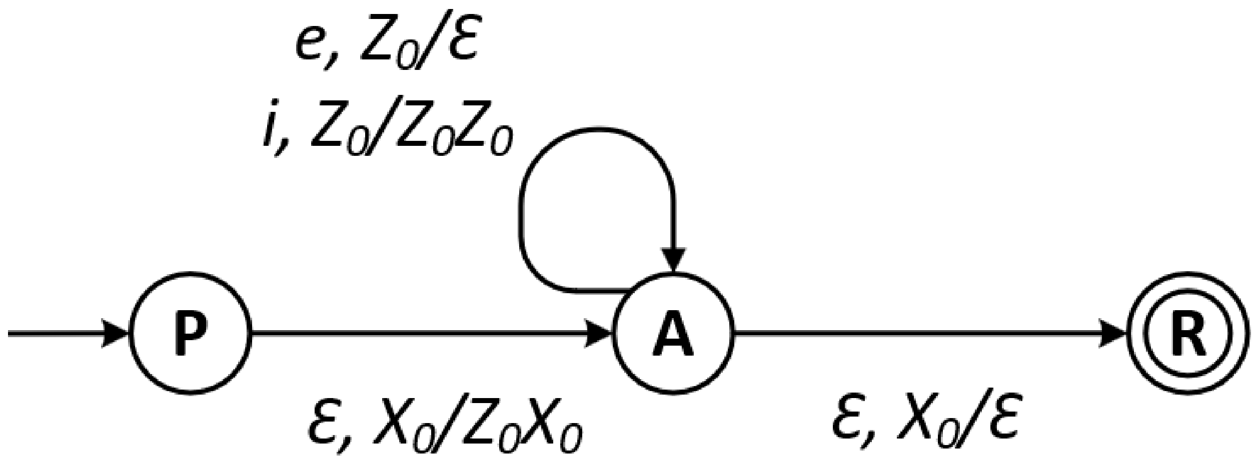
- for any state q. That is, N(P) is the set of inputs w that P can consume and at the same time empty its stack.

PDA: Conversion from Empty Stack to Final State



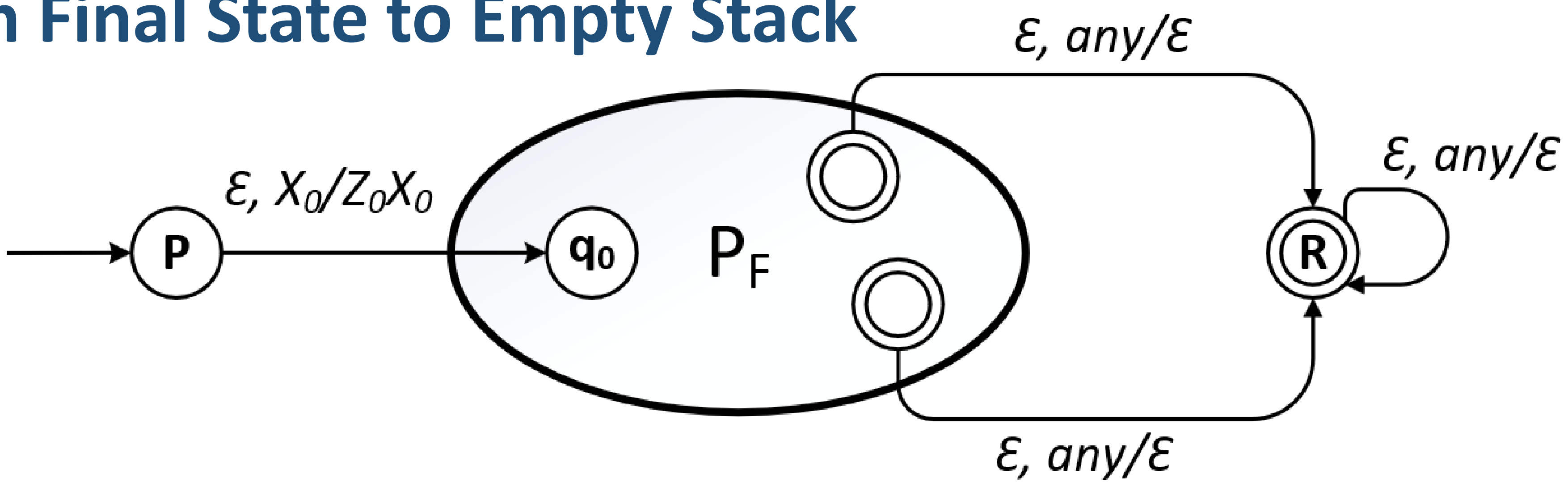


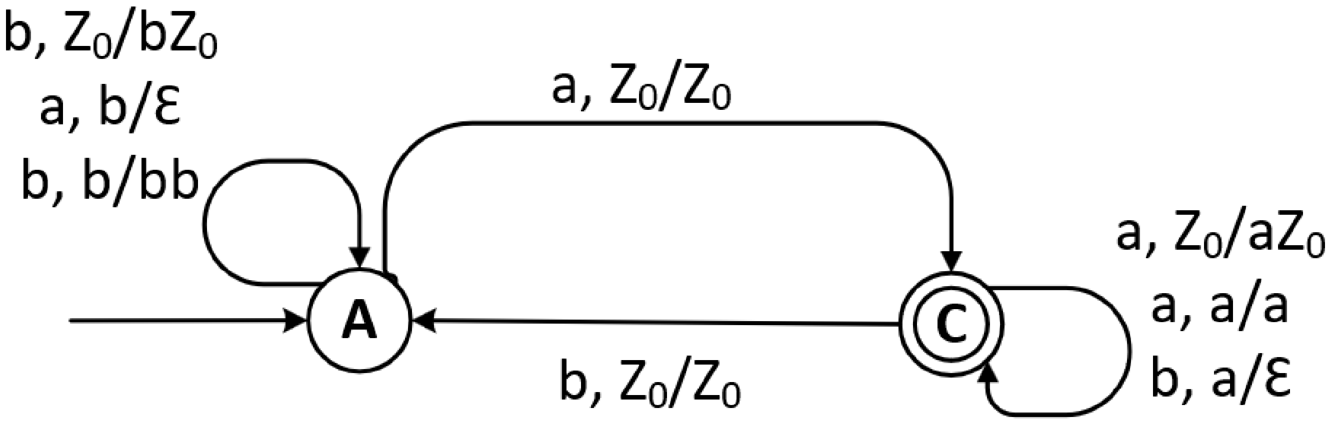
A PDA that accepts the if/else errors by empty stack.



Construction of a PDA accepting by final state from the sample PDA accepted by empty stack.

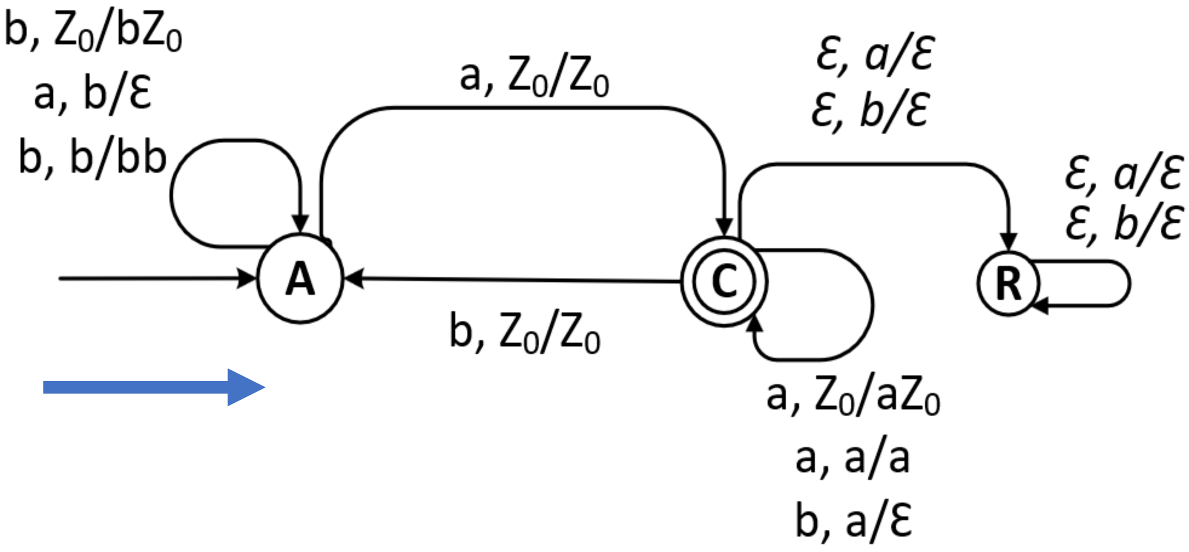
PDA: Conversion from Final State to Empty Stack





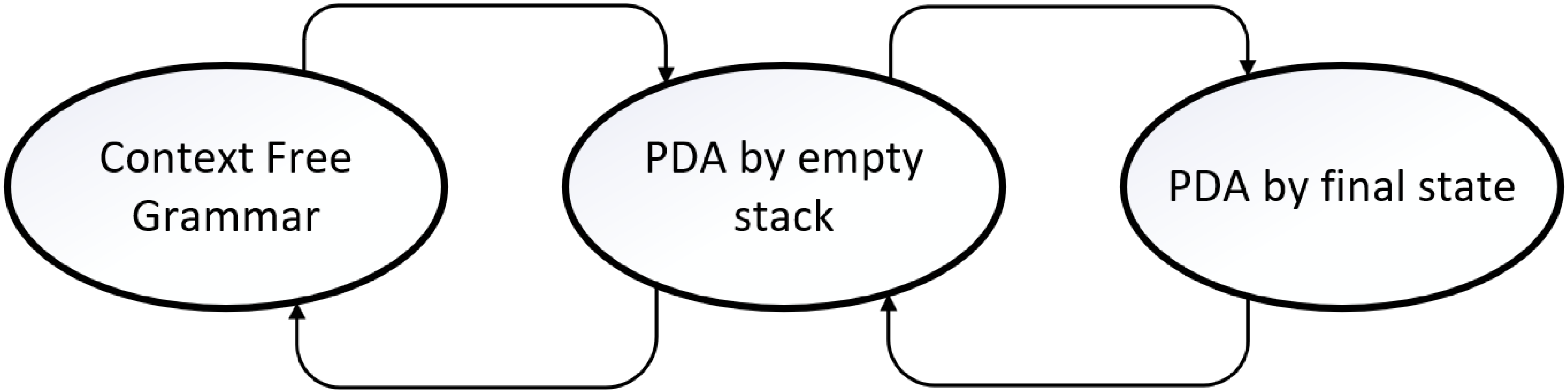
A sample PDA that is accepted by Final State

MMK@CSEDU



Construction of PDA accepting by empty state from the sample PDA accepted by final state.

Summary of Transformation



Turing Machine and Undecidability

- The Turing machine, conceptualized by Alan Turing in the 1930s, is a theoretical computational device that models the behaviour of a general-purpose computer.
- Turing machines are used to study the limits and capabilities of computation systems, and to prove that certain problems are undecidable.
- An undecidable problem is a problem that cannot be solved by a Turing machine.
    - One example of an undecidable problem is the halting problem – **proposed by Turing in 1936 and proved that** the halting problem is undecidable.
    - The halting problem asks whether there exists a program that can determine, for any given program and input, whether that program will eventually halt or continue running indefinitely.

Turing Machine and Undecidability (cont.)

- The existence of undecidable problems has important implications for the theory of computation.
- It means that there are some problems that cannot be solved by computers, no matter how powerful they are.
- This has led to the development of new techniques for solving problems that are not undecidable, such as **approximation algorithms** and **heuristics**.

Nondeterministic Polynomial (NP)

- In computational complexity theory, NP refers to the class of decision problems for which a "yes" instance can be verified in polynomial time.
- In other words, if someone claims to have a solution to an NP problem, it can be verified efficiently. However, finding a solution itself may not be computationally efficient.

Nondeterministic Polynomial (NP)

- The relationship between NP and undecidable problems lies in the concept of the "P versus NP problem.“
- This problem asks whether every problem for which a solution can be verified in polynomial time (NP) can also be solved in polynomial time (P).

In other words, is NP equal to P?

- If P = NP, it would mean that any problem with an efficient verification algorithm also has an efficient solution algorithm.
- However, if P ≠ NP, it implies that there are problems for which no efficient solution algorithm exists, even though a solution can be verified efficiently.
- This would indicate a fundamental gap between the ability to verify solutions and the ability to find solutions.

Relationship between Undecidable and NP

- Undecidable problems and NP are connected in the sense that undecidable problems generally fall outside the jurisdiction of NP.

- Undecidable problems, such as the halting problem, are beyond the scope of computation, regardless of whether efficient verification algorithms exist.

- NP problems, on the other hand, are in the jurisdiction of computation, but finding efficient solutions remains an open question for many of them.

Turing Machine

A Turing Machine can be defined by 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

- Q is a finite set of states
- Σ is a finite input symbols
- Γ is a complete set of tape symbols, Σ is always a subset of Γ.
- δ is the transition function that governs the behaviour of the automaton

  Formally, the arguments of $\delta(q, X)$ are a state q and symbol X.

  i. p is the next state, in Q
  ii. Y is the symbol, in Γ, written in the cell being scanned, replacing whatever symbol was there.
  iii. D is direction, either L or R, standing for left or right, respectively, and telling us the direction in which the head moves.
- $q_0 \in Q$ The start state, a member of Q, in which the finite control is found initially.
- $B \in \Gamma$ is the blank symbol, and is in Γ but not in Σ.
- $F \in Q$ is a set of final states (zero or more)

Turing Machine $(Q, \Sigma, \Gamma, \delta, q_0, B, F)$

State	Symbol				
	0	**1**	**X**	**Y**	**B**
q_0	(q_1, X, R)	--	--	(q_3, Y, R)	--
q_1	$(q_1, 0, R)$	(q_2, Y, L)	--	(q_1, Y, R)	--
q_2	$(q_2, 0, L)$	--	(q_0, X, R)	(q_2, Y, L)	--
q_3	--	--	--	(q_3, Y, R)	(q_4, B, R)
q_4	--	--	--	--	--

$q_0 0011 \vdash X\, q_1 011 \vdash X0\, q_1 11 \vdash X\, q_2 0Y1 \vdash q_2 X0Y1 \vdash$

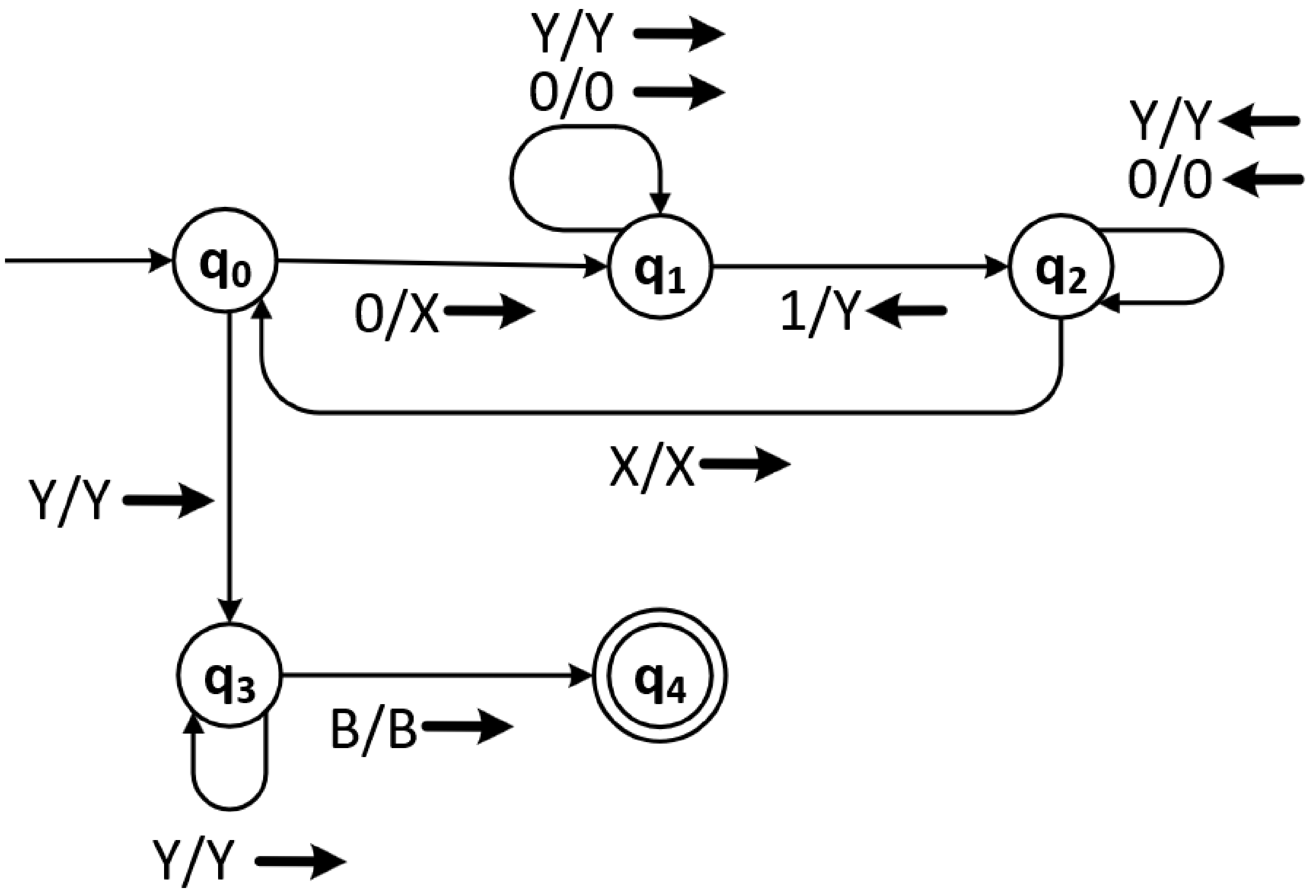
$X q_0 0Y1 \vdash XX q_1 Y1 \vdash XXY q_1 1 \vdash XX q_2 YY \vdash X\, q_2 XYY \vdash$

$XX q_0 YY \vdash XXY q_3 Y \vdash XXYY q_3 B \vdash XXYY q_4 B$

Constrcut a Turing machine that accepts the language $\{0^n 1^n \mid n \geq 1\}$

Turing Machine

$TM = (Q, \Sigma, \Gamma, \delta, q_0, B, F)$



State	Symbol				
	0	1	X	Y	B
q_0	(q_1, X, R)	--	--	(q_3, Y, R)	--
q_1	$(q_1, 0, R)$	(q_2, Y, L)	--	(q_1, Y, R)	--
q_2	$(q_2, 0, L)$	--	(q_0, X, R)	(q_2, Y, L)	--
q_3	--	--	--	(q_3, Y, R)	(q_4, B, R)
q_4	--	--	--	--	--

Constrcut a Turing machine that accepts the language $\{0^n1^n \mid n \geq 1\}$