**Mohsin Iban Hossain**

**AIUB, Compiler Final-Term Notes**

# COMPILER DESIGN

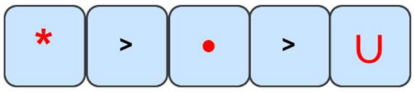
# Table of Contents

# REGULAR EXPRESSION

# Regular Expression

≫Regular expressions are used to represent certain sets of strings in an **algebraic** fashion, moving beyond simple English representations of strings.

- **Structure**:
  Built using alphabets and **regular operations**:
  - * (star/**Closure**) – Represents zero or more occurrences of a string.
  - • (Concatenation) – joining two patterns
  - ∪ (Union) – either/or (can also be written as |)
- **Precedence Order**:




## ≫Examples:

1. **(0∪1)0***
   - Means: Starts with 0 or 1, followed by any number of 0s
   - Language: `{w | w starts with 0 or 1, followed by 0s}`
2. **(0∪1)***
   - Means: Any combination of 0s and 1s
   - Language: `{all binary strings}`
3. **(0∪1)1**
   - **Means**: Either 0 or 1, followed by 1
   - Language: `{w | w ends with 1, and starts with 0 or 1}`

# Designing Regular Expressions

🧠 Example1: Write the Regular Expression of the following languages, where ∑= {a,b},
"Language Accepting **Strings of Length Exactly Two**"

| **Step 1:** Understand the Alphabet | **Step 2:** Understand the Language Requirement |
|---|---|
| **We are working with:**<br><br>Σ = {a, b}<br>This means only the characters `a` and `b` can appear in our strings. | ≫The language accepts only strings of **length exactly two**.<br><br>This means strings like:<br>• `aa`<br>• `ab`<br>• `ba`<br>• `bb`<br><br>Only these four strings are valid.<br>Any string with **length ≠ 2** is not allowed. |
| **Step 3:** List All Valid Strings | **Step 4:** Write the Regular Expression |
| ≫As we saw above, valid strings of length 2 over {a, b} are:<br>• **aa**<br>• **ab**<br>• **ba**<br>• **bb** | ∴ R = (aa ∪ ab ∪ ba ∪ bb) Or<br>∴ R = (aa \| ab \| ba \| bb) |

🧠 Example2: Write the Regular Expression of the following languages, where ∑= {a,b},
"which accepts **all strings of length at least two**."

| **Step 1:** Understand the Alphabet | **Step 2:** Understand the Language Requirement |
|---|---|
| **We are working with:**<br><br>Σ = {a, b}<br>This means only the characters `a` and `b` can appear in our strings. | ≫The language accepts **all strings of length at least two**<br>This means:<br>• The string must be **two or more characters long**.<br>• This includes all strings of length 2, 3, 4, and so on. |
| **Step 3:** List All Valid Strings | **Step 4:** Write the Regular Expression |
| ≫ For strings of length at least two:<br>• **Valid examples: aa, ab, ba, bb, aaa, aab, abb, baa, bab, bba, bbb, etc.**<br><br>• **Invalid examples** (strings of length less than two): `a`, `b` | ∴ R = (a \| b)(a \| b)(a \| b)* |

| Explanation: |
| --- |
| • `(a | b)`: The first character can be either `a` or `b`.<br>• `(a | b)`: The second character can also be either `a` or `b`.<br>• `(a | b)*`: After the second character, any number (including zero) of `a` or `b` characters can follow. This makes the string longer than two characters if needed. |

Example3: Write the Regular Expression of the following languages, where ∑= {a,b}, "which accepts **Strings of Length At Most Two**."

| **Step 1:** Understand the Alphabet | **Step 2:** Understand the Language Requirement |
| --- | --- |
| **We are working with:**<br>Σ = {a, b}<br>This means only the characters `a` and `b` can appear in our strings. | ≫The language accepts **all strings of length at Most two.**<br>This means:<br>• Length 0 (the **empty string**, denoted by **ε**)<br>• Length 1 (like `a`, `b`)<br>• Length 2 (like `aa`, `ab`, `ba`, `bb`) |
| **Step 3:** List All Valid Strings | **Step 4:** Write the Regular Expression |
| ≫ For strings of length at **Most** two:<br>• **ε**<br>• **a**<br>• **b**<br>• **aa**<br>• **ab**<br>• **ba**<br>• **bb** | ∴ R = ε \| a \| b \| (a \| b) (a \| b) |

# Practice Problem

Problem1: Write the Regular Expression of the following languages, where ∑= {a,b}, "Language Accepting **Strings of Length Exactly Two**"

**➔ Here,**

**»Σ = {a,b},**

**» L(M) = {aa, ab, ba, bb}**

$$\therefore R = (a \cup b)\,(a \cup b)$$

Or

$$\therefore R = (a \mid b)\,(a \mid b)$$

Problem2: Write the Regular Expression of the following languages, where ∑= {a,b}, "which accepts **all strings of length at least two**."

**➔ Here,**

**»Σ = {a,b},**

**» L(M) = {aa, ab, ba, bb, aaa, aab, abab…, bbaa..,}**

$$\therefore R = (a \mid b)\,(a \mid b)\,(a \mid b)^*$$

Problem3: Write the Regular Expression of the following languages, where ∑= {a,b}, "which accepts **all strings of length at Most two**."

**➔ Here,**

**»Σ = {a,b},**

**» L(M) = {ε, a, b, aa, bb, ab, ba}**

$$\therefore R = \varepsilon \mid a \mid b \mid (a \mid b)\,(a \mid b)$$

Problem4: Write the Regular Expression of the following languages, where ∑= {a,b}, "L={w| each a in w is followed by at least two b}"

➔ **Here,**

»**Σ = {a,b},**

» **L(M) = {abb, b…abb, abbb…, abbabb…, abbbabb…}**

$$\therefore R = b^* \, abb \, b^* \, (abb \, b^*)^*$$

Let's check whether the string "**abbabb**" matches the regular expression for the language:

1. **Breakdown of** abbabb**:**
   - The string can be grouped as: a b b a b b
     (Each a is followed by exactly two bs, which satisfies the condition.)

2. **Matching with** b* a b b b* (a b b b*)***:**
   - b* → Matches 0 leading bs (empty).
   - a b b b* → Matches a b b (the first b* matches 0 trailing bs here).
   - (a b b b*)* → Matches the remaining a b b
     (again, b* matches 0 trailing bs).
   - **Result:** The entire string abbabb is matched.

Problem5: Write the Regular Expression of the following languages, where ∑= {a,b}, "L={w| w contains odd number of b}"

➔ **Here,**

»**Σ = {a,b},**

» **L(M) = {b, a…b, ba…, bababa…,ba…ba…ba…}**

$$\therefore R = a^*b \, (a^*b \, a^*b \, a^*)^*$$

Problem5: Write the Regular Expression of the following languages, where ∑= {a,b},
"L={w| w contains ***abbab*** substring}"

➔ **Here,**

»**Σ = {a,b},**

» **L(M) = {abbab, a..abbab, b..abbab, abbaba..., abbabb...}**

$$\therefore R = (a \cup b)^* \text{ abbab } (a \cup b)^*$$

Problem6: Write the Regular Expression of the following languages, where ∑= {m,n}

i. "{w | each m in w is followed by at least three n}"
ii. "{w | w has odd length and starts with m}"
iii. "{w | w contains 'mnnmmn' substring}"

**(i)**

➔ **Here,**

»**Σ = {m, n},**

» **L(M) = {mnnn, n...mnnn, mnnnn..., mnnmnn... }**

$$\therefore R = n^* \text{ mnnn } n^* (\text{mnnn } n^*)^*$$

**(ii)**

➔ **Here,**

»**Σ = {m, n},**

» **L(M) = {m, mmn, mnn,mnnnm,... }**

$$\therefore R = m((m \cup n)(m \cup n))^*$$

**(iii)**

➔ **Here,**

»**Σ = {m, n},**

» **L(M) = { mnnmmn, mmn…mnnmmn, mnnmmn…mnnnm,… }**

$$\therefore R = (m|n)^*\ mnnmmn\ (m|n)^*$$

Problem7: Give the **Description** of the following Regular Expressions.

Consider ∑ = {x,y}

i. "(x∪y)* xyyxxy"
ii. "x* y x* y x* y y*"
iii. "(x ∪ y). ((x ∪ y ). (x ∪ y))*"

**(i)**

➔**(x∪y)* xyyxxy:**

- Language: {w | w is a string over ∑ = {x, y} that ends with "xyyxxy"}.

**(ii)**

➔**x* y x* y x* y y*:**

- Language: {w | w is a string over ∑ = {x, y} with at least three 'y's, and the third 'y' is followed by zero or more 'y's}.

**(iii)**

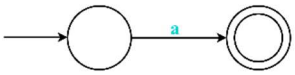
➔**(x ∪ y). ((x ∪ y ). (x ∪ y))***

- Language: {w | w string starts with x or y, string has odd number of length}

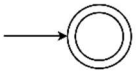
# CONVERSION OF RE TO NFA

# Formal procedure of Regular Language

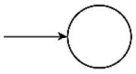
1. R = a ; a ∈ ∑



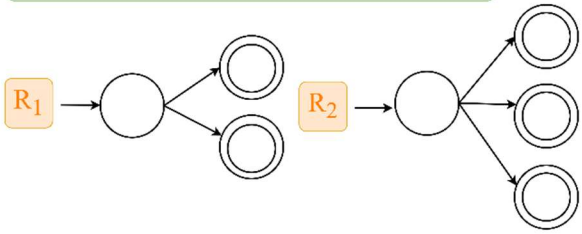

2. R = ∈ ; initial state is the final state

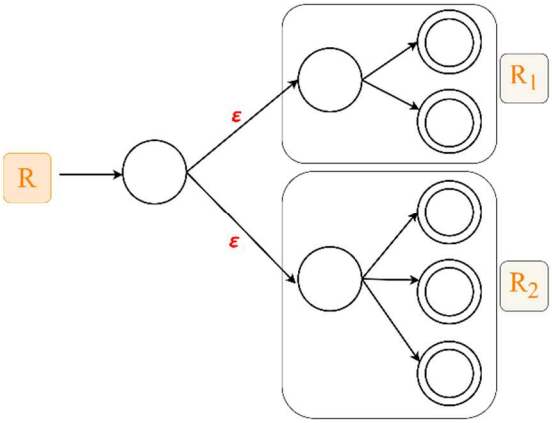


3. R = $\phi$

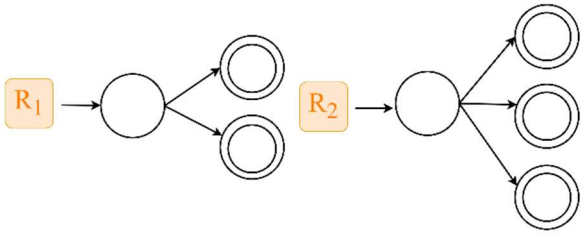


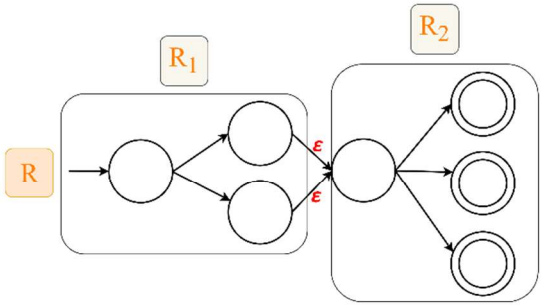
4. R = $R_1$ U $R_2$ **or** R= $R_1$|$R_2$ **or** R=$R_1$ or $R_2$



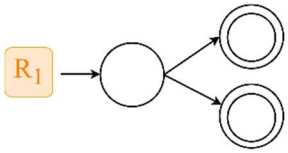



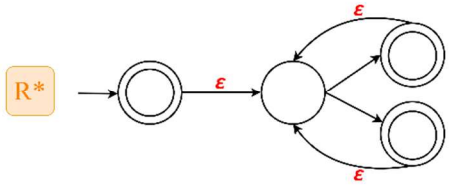

5. R = $R_1 \circ R_2$ **or** R= $R_1R_2$ **or** R=$R_1$ Followed by $R_2$



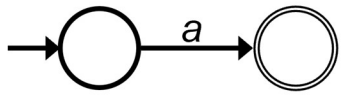
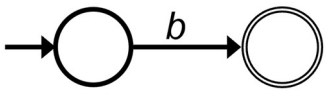
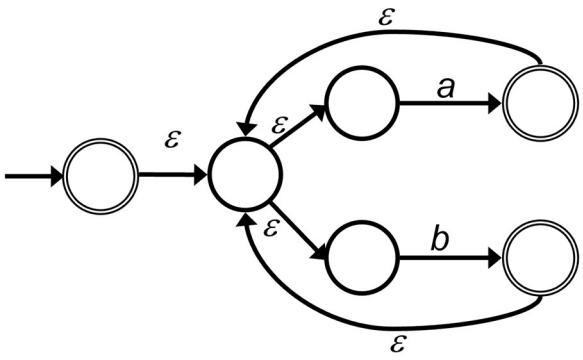
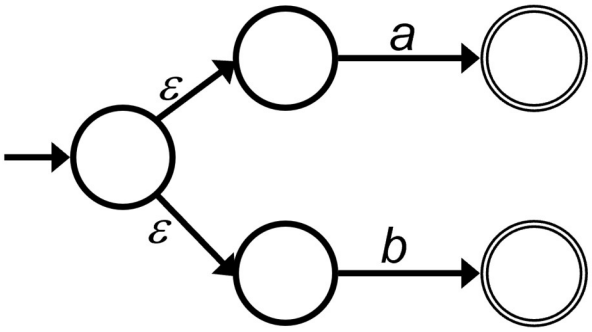
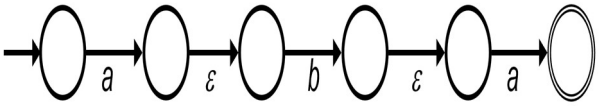
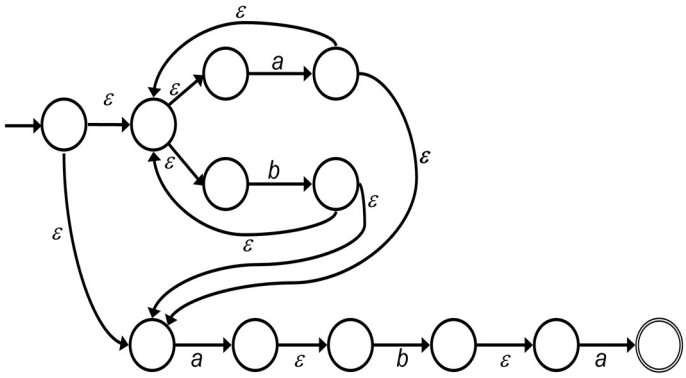

6. R* = $R_1$*



# Practice Problem

🧠 Problem1: Convert the following Regular Expression into equivalent NFA using formal procedure. "***(a∪b)*aba***".

<table>
<tr><td>Step 1: Parse the RE into Subexpressions</td><td colspan="2">Step 2: Build NFAs for Basic Symbols</td></tr>
<tr><td rowspan="2">≫ The RE "(a|b)*aba" can be decomposed as:<br>1. (a|b)*<br>2. a<br>3. b<br>4. a<br><br>We'll build NFAs for each component and then concatenate them.</td><td>NFA for a</td><td>NFA for b</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>Step 4: NFA for (a|b)* (Star)</td><td colspan="2">Step 3: NFA for (a|b) (Union)</td></tr>
<tr><td> 
</td><td colspan="2"> 
</td></tr>
<tr><td>Step 5: NFA for "aba" (Concatenation)</td><td colspan="2">Step 6: Combine (a|b)* and aba (Concatenation)</td></tr>
<tr><td> 
</td><td colspan="2"> 
</td></tr>
</table>

🧠 Problem2: Convert the following Regular Expression into equivalent NFA using formal procedure. "**ab*a U a*b*a**".

**Step 1:** Parse the RE into Subexpressions

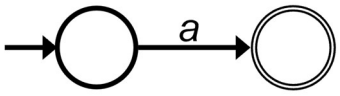
» ***The RE*** "ab*a U a*b*a" **can be decomposed as:**

1. ab*a
2. a*b*a
3. a
4. b
5. a*
6. b*

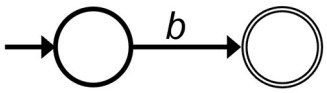
We'll build NFAs for each component and then union them.

**Step 2:** Build NFAs for Basic Symbols

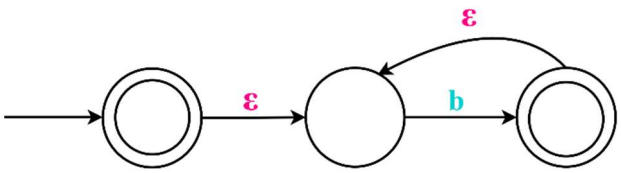
NFA for a



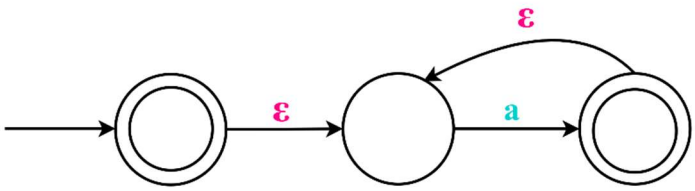

NFA for b



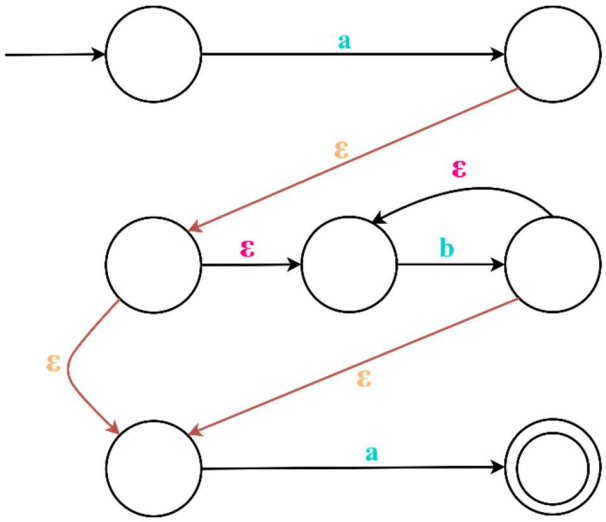

**Step 4:** NFA for b* (Star)



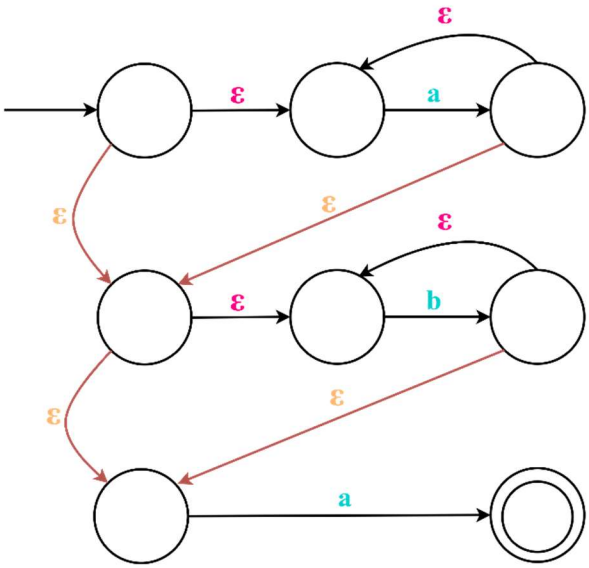

**Step 3:** NFA for a* (Star)

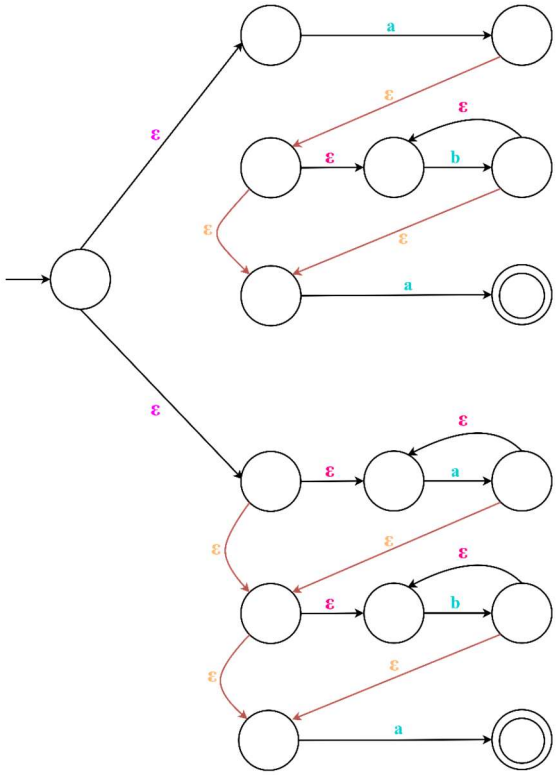



**Step 5:** NFA for ab*a (Concatenation)



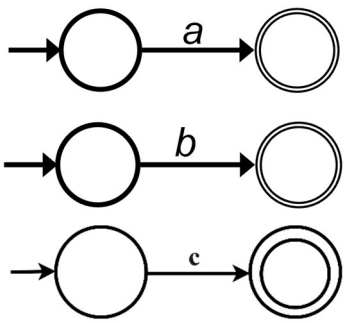

**Step 6:** NFA for a*b*a (Concatenation)

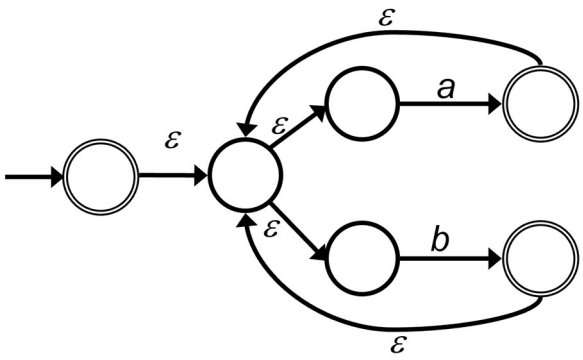
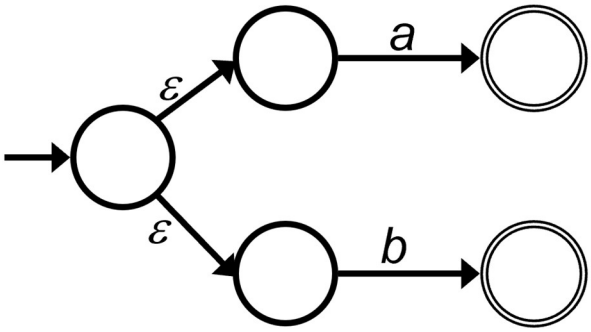
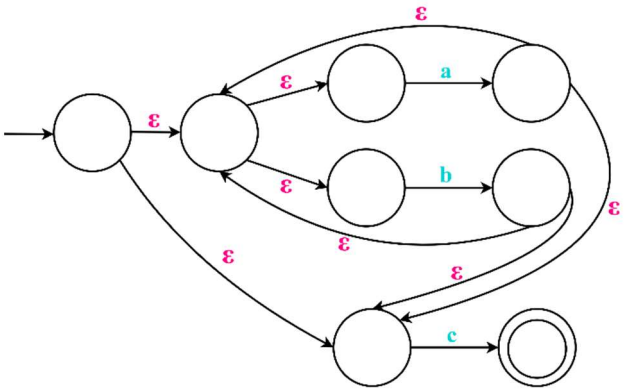
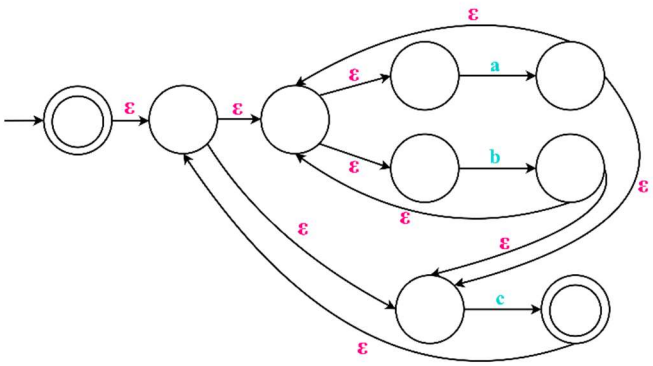


| **Step 7:** Final NFA for ab*a U a*b*a (Union) |
|---|
|  |

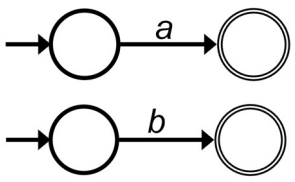
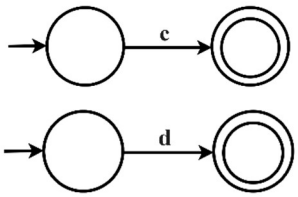


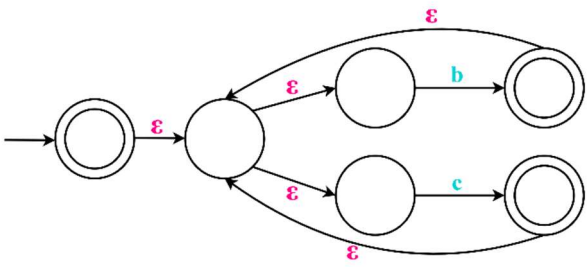
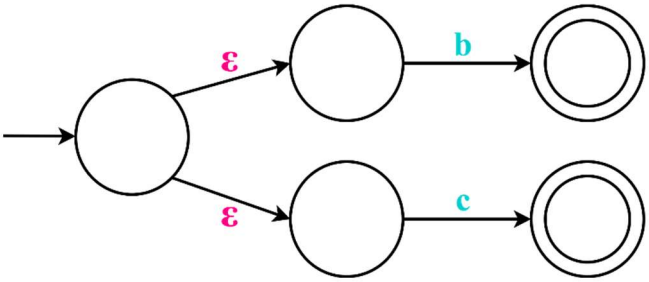
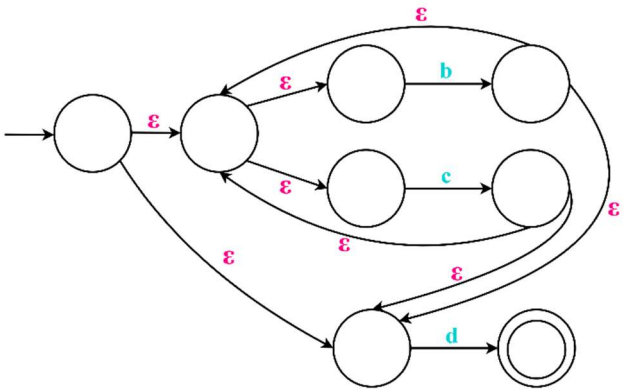
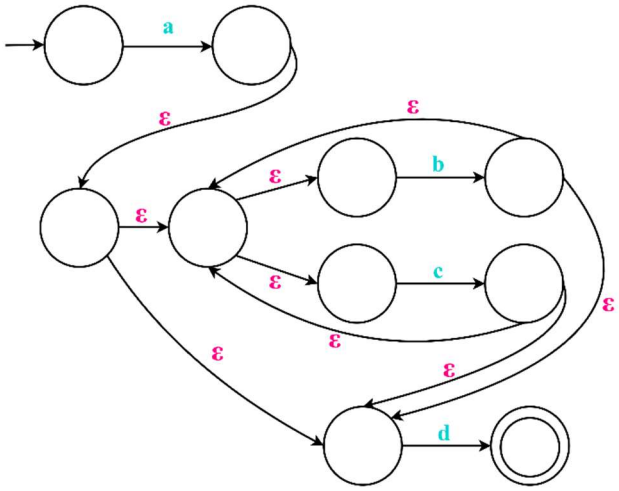
🧠 Problem3: Convert the following Regular Expression into equivalent NFA using formal procedure. "**((a | b)*c)***".

| **Step 1:** Parse the RE into Subexpressions | **Step 2:** Build NFAs for Basic Symbols |
|---|---|
| ≫ ***The* RE** "((a\|b)*c)*" **can be decomposed as:**<br>1. (a\|b)*<br>2. a<br>3. b<br>4. c<br><br>We'll build NFAs for each component and then add them. | NFA for a<br>NFA for b<br>NFA for c<br>  |

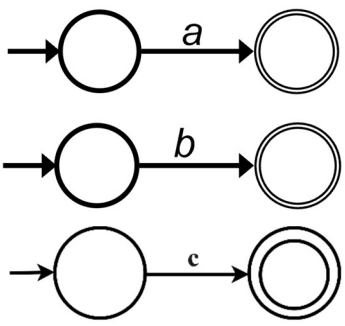
| **Step 4:** NFA for **(a\|b)*** (Star) | **Step 3:** NFA for **(a\|b)** (Union) |
|---|---|
|  |  |
| **Step 5:** NFA for "**(a\|b)*c**" (Concatenation) | **Step 6:** Combine "**((a \| b)*c)***". (Star) |
|  |  |

🧠 Problem4: Convert the following Regular Expression into equivalent NFA using formal procedure. "**a (b | c)*d**"

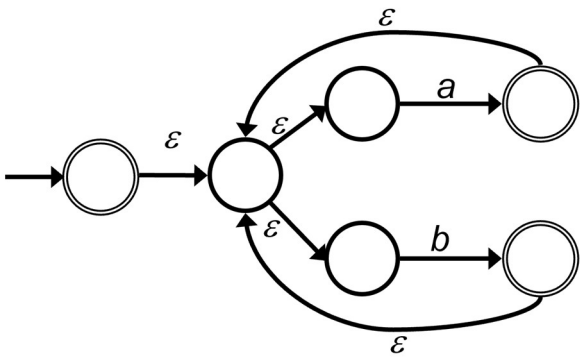
| **Step 1:** Parse the RE into Subexpressions | **Step 2:** Build NFAs for Basic Symbols | |
|---|---|---|
| | NFA for a<br>NFA for b | NFA for c<br>NFA for d |
| ≫ ***The* RE "a (b \| c)*d" can be decomposed as:**<br>1. (b\|c)<br>2. a<br>3. b<br>4. c<br>5. d<br><br>We'll build NFAs for each component and then concatenate them. |  |  |

| **Step 4:** NFA for **(b\|c)*** (Star) | **Step 3:** NFA for **(b\|c)** (Union) |
|---|---|
|  |  |
| **Step 5:** NFA for "**(b\|c)*d**" (Concatenation) | **Step 6:** Combine "**a(b\|c)*d**" (Concatenation) |
|  |  |

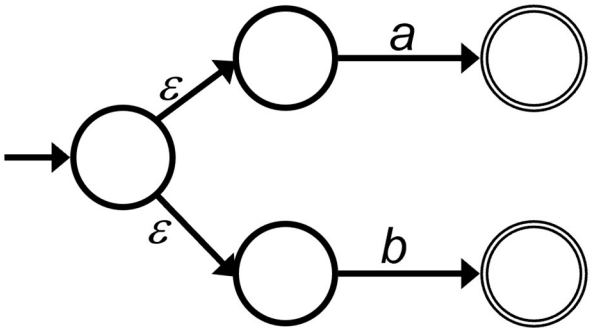
🧠 Problem5: Convert the following Regular Expression into equivalent NFA using formal procedure. "**(a | b)*abc**".

| **Step 1:** Parse the RE into Subexpressions | **Step 2:** Build NFAs for Basic Symbols |
|---|---|
| ≫ ***The* RE** "(a\|b)*abc" **can be decomposed as:**<br>1. (a\|b)*<br>2. a<br>3. b<br>4. c<br><br>We'll build NFAs for each component and then concatenate them. | NFA for a<br>NFA for b<br>NFA for c<br>  |

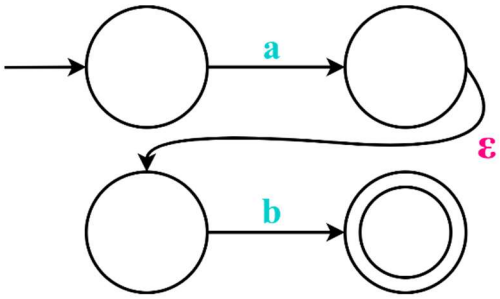
**Step 4:** NFA for **(a|b)*** (Star)



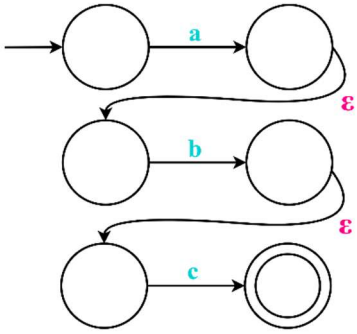

**Step 3:** NFA for **(a|b)** (Union)



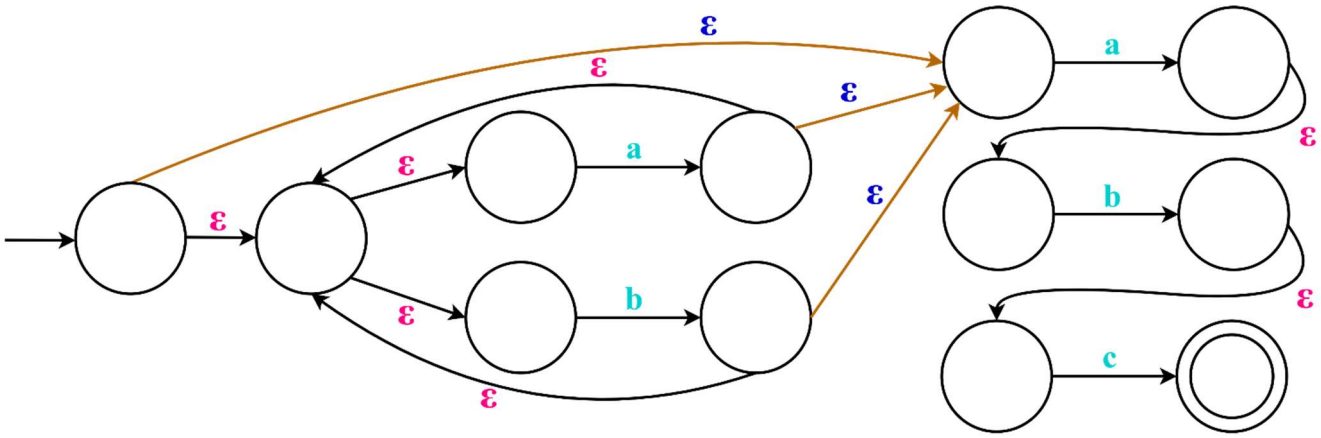

**Step 5:** NFA for "**ab**" (Concatenation)



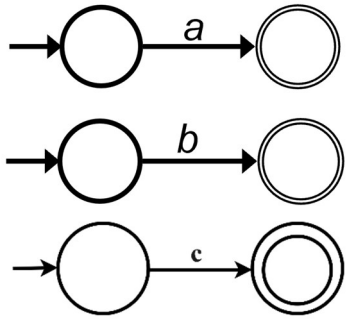
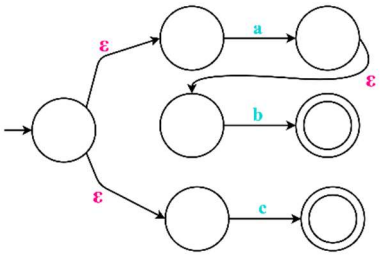
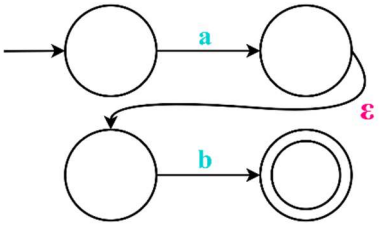
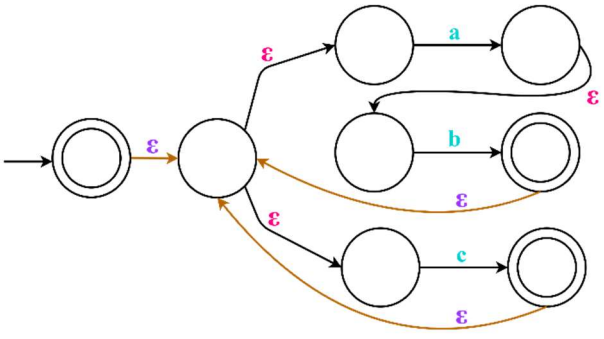
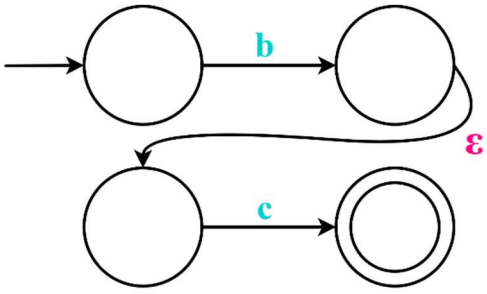

**Step 6:** Combine "**abc**". (Concatenation)



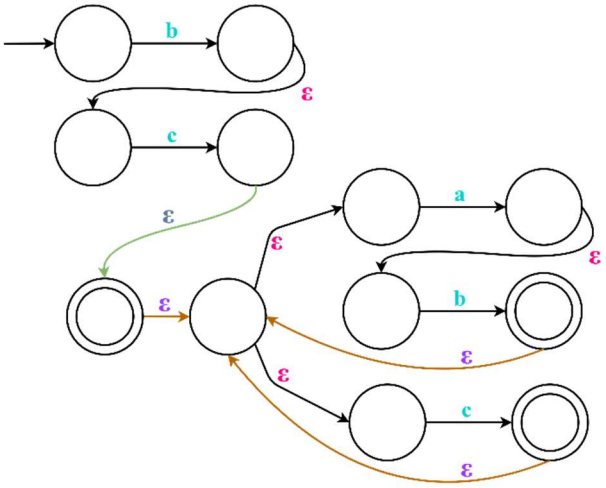

**Step 7:** Combine "**(a | b)*abc**". (Concatenation)



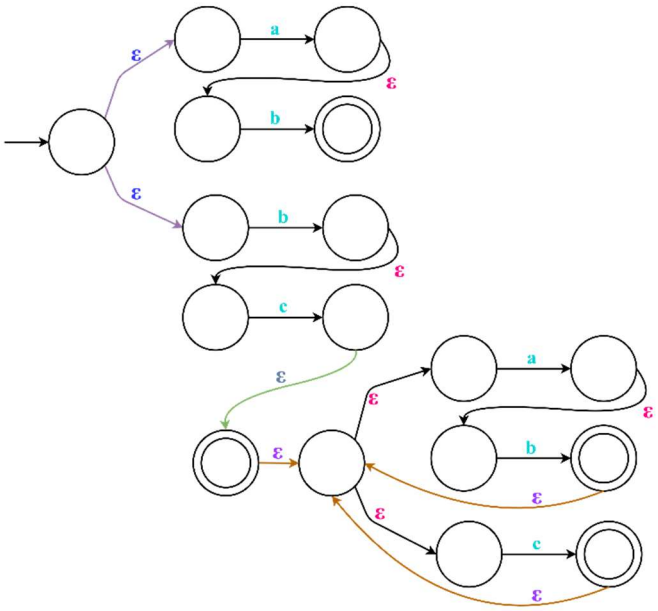
🧠 Problem6: Convert the following Regular Expression into equivalent NFA using formal procedure. "**(abUbc(abUc)*)***".

<table>
<tr><td>Step 1: Parse the RE into Subexpressions</td><td>Step 2: Build NFAs for Basic Symbols</td></tr>
<tr><td rowspan="2">≫ The RE "(abUbc(abUc)*)*" can be decomposed as:<br>1. (ab|c)*<br>2. ab<br>3. bc<br>4. a<br>5. b<br>6. c<br><br>We'll build NFAs for each component and then add them.</td><td>NFA for a<br>NFA for b<br>NFA for c</td></tr>
<tr><td> 
</td></tr>
<tr><td>Step 4: NFA for (ab|c) (Union)</td><td>Step 3: NFA for (ab) (Concatenation)</td></tr>
<tr><td> 
</td><td> 
</td></tr>
<tr><td>Step 5: NFA for "(ab|c)*" (Star)</td><td>Step 6: Combine "bc". (Concatenation)</td></tr>
<tr><td> 
</td><td> 
</td></tr>
</table>

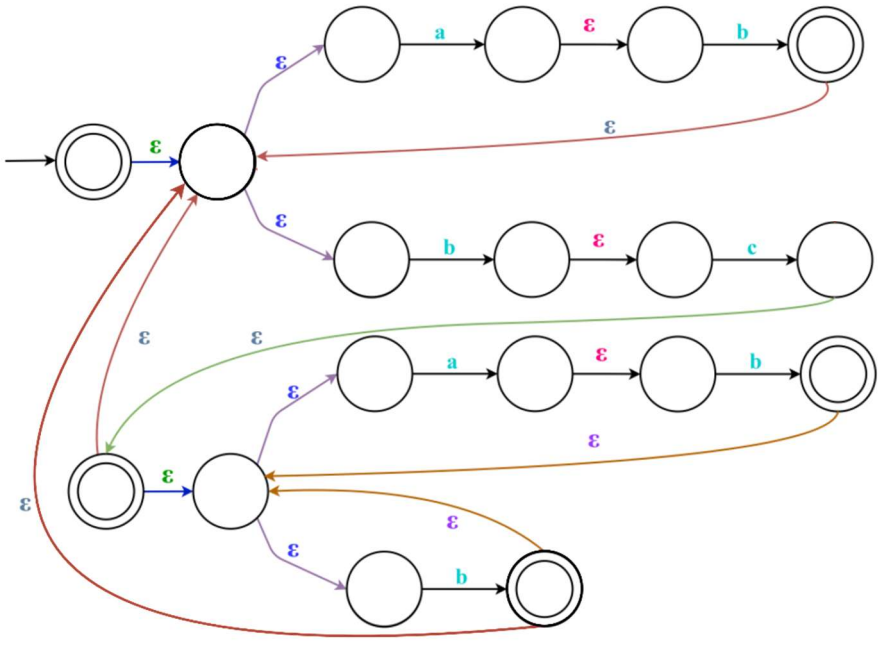
**Step 7:** Combine "**bc(abUc)***". (Concatenation)




**Step 8:** Combine "**abUbc(abUc)***". (union)




**Step 9:** Combine "**(abUbc(abUc)*)***". (Star)



# CONVERSION OF NFA TO DFA

# Conversion from NFA with ε to DFA

➔Non-deterministic finite automata (NFA) is a finite automata where for some cases when a **specific input is given to the current state**, the machine **goes to multiple states** or **more than 1 states**. It can **contain ε move**.

It can be represented as **M = {Q, ∑, δ, $q_0$, F}.**

**Were,**

1. **Q: finite set of states**
2. **∑: finite set of the input symbol**
3. **$q_0$: initial state**
4. **F: final state**
5. **δ: Transition function**

**NFA with ∈ move:** If any FA **contains ε transaction or move**, the finite automata is called NFA with ∈ move.

**ε-closure:** ε-closure for a given **state A means a set of states** which can be **reached from the state A with only ε(null) move including the state A itself.**

## Steps for converting NFA with ε to DFA

**Step 1:** We will take the ε-closure for the starting state of NFA as a starting state of DFA.

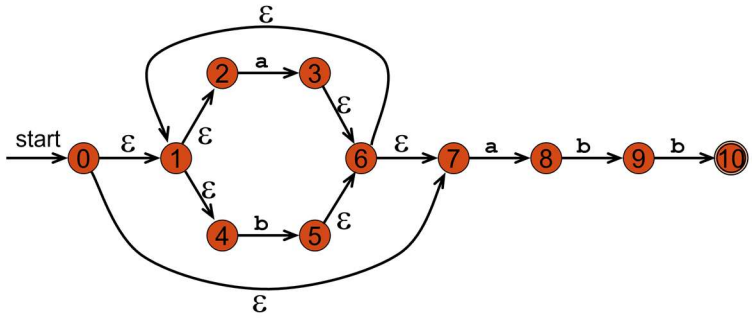
**Step 2:** Find the states for each input symbol that can be traversed from the present. That means the union of transition value and their closures for each state of NFA present in the current state of DFA.

**Step 3:** If we found a new state, take it as current state and repeat step 2.

**Step 4:** Repeat Step 2 and Step 3 until there is no new state present in the transition table of DFA.

**Step 5:** Mark the states of DFA as a final state which contains the final state of NFA.

**Problem1:** Convert the following NFA with ε into equivalent DFA using Subset Construction Method



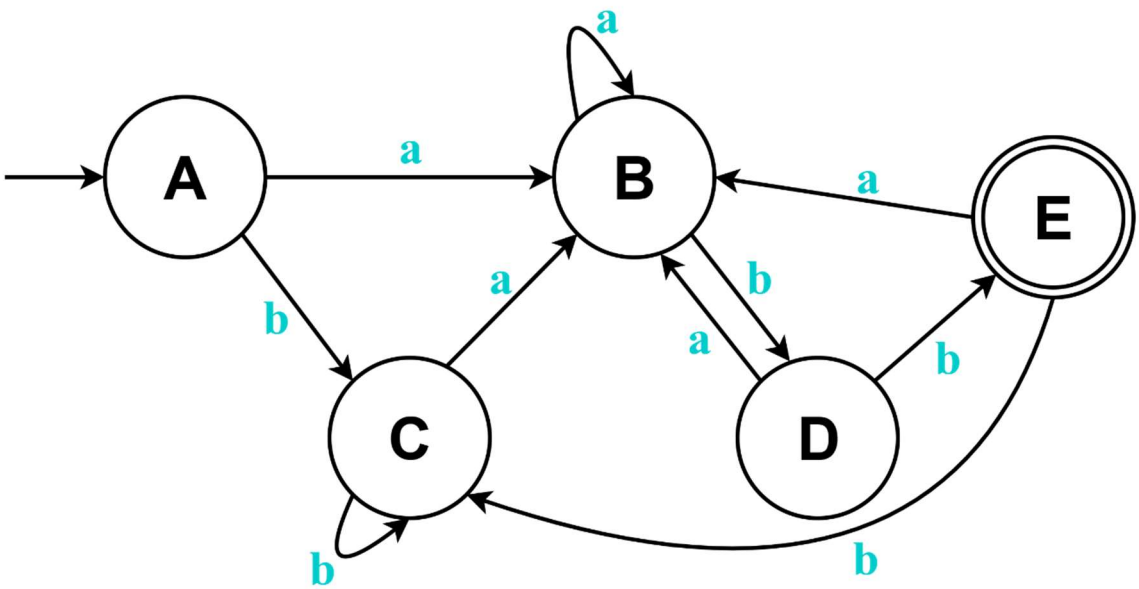

**ε-Closure Table for DFA States**

| DFA State | ε -closure of | ε -closure outcome states |
|---|---|---|
| A | ε-Closure of ({0}) | 0,1,2,4,7 |
| B | ε-Closure of ({3,8}) | 3,6,7,1,2,4,8 |
| C | ε-Closure of ({5}) | 5,6,7,1,2,4 |
| D | ε-Closure of ({5,9}) | 5,6,7,1,2,4,9 |
| E | ε-Closure of ({5,10}) | 5,6,7,1,2,4,10 |

**DFA Transition Table Constructed from NFA**

| NFA States | DFA State | a | b |
|---|---|---|---|
| 0,1,2,4,7 | → A | B | C |
| 3,6,7,1,2,4,8 | B | B | D |
| 5,6,7,1,2,4 | C | B | C |
| 5,6,7,1,2,4,9 | D | B | E |
| 5,6,7,1,2,4,10 | E | B | C |

Now, Construct the DFA from transition table:



# "An Alternative Approach to Solving It"

**Step1:** Write the transition table of **NFA**

| $\delta$ | a | b |
|---|---|---|
| 0 | $\phi$ | $\phi$ |
| 1 | $\phi$ | $\phi$ |
| 2 | 3 | $\phi$ |
| 3 | $\phi$ | $\phi$ |
| 4 | $\phi$ | 5 |
| 5 | $\phi$ | $\phi$ |
| 6 | $\phi$ | $\phi$ |
| 7 | 8 | $\phi$ |
| 8 | $\phi$ | 9 |
| 9 | $\phi$ | 10 |
| 10 | $\phi$ | $\phi$ |

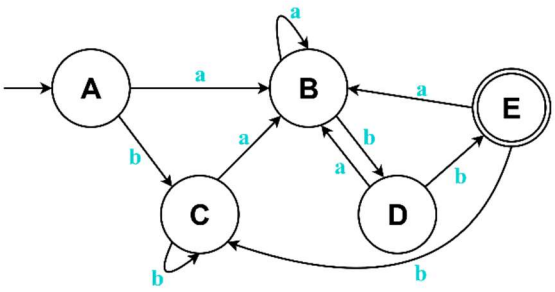
**Step2:** Find the **Epsilon (ε) Closure** of all the state

- ε-Closure of ({0})={0,1,2,4,7}
- ε-Closure of ({1})={1,2,4}
- ε-Closure of ({2})={2}
- ε-Closure of ({3})={3,6,7,1,2,4}
- ε-Closure of ({4})={4}
- ε-Closure of ({5})={5,6,7,1,2,4}
- ε-Closure of ({6})={6,7,1,2,4}
- ε-Closure of ({7})={7}
- ε-Closure of ({8})={8}
- ε-Closure of ({9})={9}
- ε-Closure of ({10})={10}

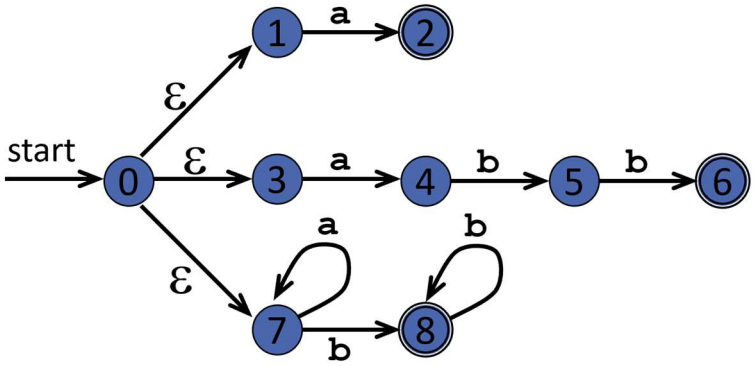
**Step3**: Convert this NFA transition table to its equivalent **DFA** transition table.

| ε -closure of | NFA | DFA | a | b |
|---|---|---|---|---|
| ε-Closure of ({0}) | 0,1,2,4,7 | → A | B | C |
| ε-Closure of ({3,8}) | 3,6,7,1,2,4,8 | B | B | D |
| ε-Closure of ({5}) | 5,6,7,1,2,4 | C | B | C |
| ε-Closure of ({5,9}) | 5,6,7,1,2,4,9 | D | B | E |
| ε-Closure of ({5,10}) | 5,6,7,1,2,4,10 | E | B | C |

**Step4:** Now, Construct the DFA from transition table.



**Problem2:** Convert the following NFA with ε into equivalent DFA using Subset Construction Method




**Step1:** Write the transition table of **NFA**

| δ | a | b |
|---|---|---|
| 0 | $\phi$ | $\phi$ |
| 1 | 2 | $\phi$ |
| 2 | $\phi$ | $\phi$ |
| 3 | 4 | $\phi$ |
| 4 | $\phi$ | 5 |
| 5 | $\phi$ | 6 |
| 6 | $\phi$ | $\phi$ |
| 7 | 7 | 8 |
| 8 | $\phi$ | 8 |

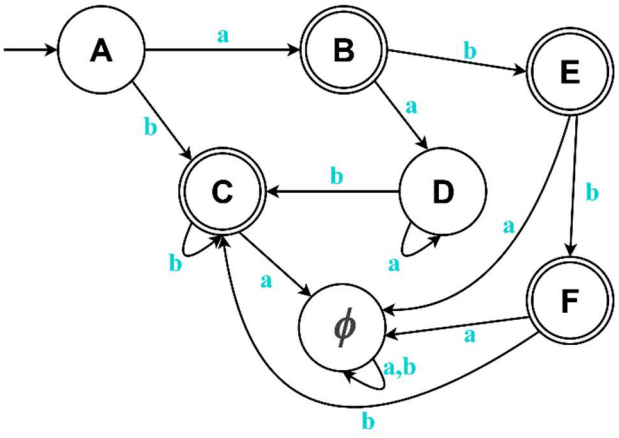
**Step2:** Find the **Epsilon (ε) Closure** of all the state

- ε-Closure of ({0})={0,1,3,7}
- ε-Closure of ({1})={1}
- ε-Closure of ({2})={2}
- ε-Closure of ({3})={3}
- ε-Closure of ({4})={4}
- ε-Closure of ({5})={5}
- ε-Closure of ({6})={6}
- ε-Closure of ({7})={7}
- ε-Closure of ({8})={8}

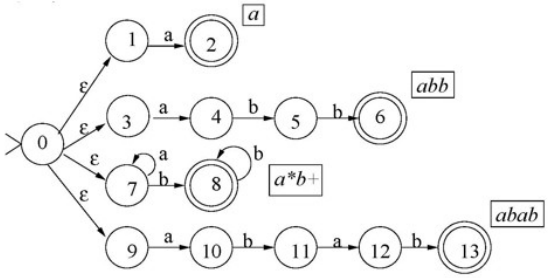
**Step3**: Convert this NFA transition table to its equivalent **DFA** transition table.

| ε -closure of | NFA | DFA | a | b |
|---|---|---|---|---|
| ε-Closure of ({0}) | 0,1,3,7 | A | B | C |
| ε-Closure of ({2,4,7}) | 2,4,7 | B | D | E |
| ε-Closure of ({8}) | 8 | C | $\phi$ | C |
| ε-Closure of ({7}) | 7 | D | D | C |
| ε-Closure of ({5,8}) | 5,8 | E | $\phi$ | F |
| ε-Closure of ({6,8}) | 6,8 | F | $\phi$ | C |
| | | $\phi$ | $\phi$ | $\phi$ |

**Step4:** Now, Construct the DFA from transition table.



**Problem2:** Convert the following NFA with ε into equivalent DFA using Subset Construction Method




**Step1:** Write the transition table of **NFA**

| δ | a | b |
|---|---|---|
| 0 | ϕ | ϕ |
| 1 | 2 | ϕ |
| 2 | ϕ | ϕ |
| 3 | 4 | ϕ |
| 4 | ϕ | 5 |
| 5 | ϕ | 6 |
| 6 | ϕ | ϕ |
| 7 | 7 | 8 |
| 8 | ϕ | 8 |
| 9 | 10 | ϕ |
| 10 | ϕ | 11 |
| 11 | 12 | ϕ |
| 12 | ϕ | 13 |
| 13 | ϕ | ϕ |

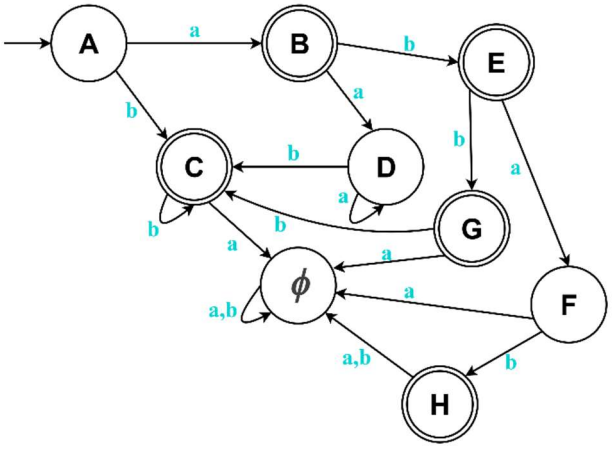
**Step2:** Find the **Epsilon (ε) Closure** of all the state

- ε-Closure of ({0})={0,1,3,7,9}
- ε-Closure of ({1})={1}
- ε-Closure of ({2})={2}
- ε-Closure of ({3})={3}
- ε-Closure of ({4})={4}
- ε-Closure of ({5})={5}
- ε-Closure of ({6})={6}
- ε-Closure of ({7})={7}
- ε-Closure of ({8})={8}
- ε-Closure of ({9})={9}
- ε-Closure of ({10})={10}
- ε-Closure of ({11})={11}
- ε-Closure of ({12})={12}
- ε-Closure of ({13})={13}

**Step3**: Convert this NFA transition table to its equivalent **DFA** transition table.

| ε -closure of | NFA | DFA | a | b |
|---|---|---|---|---|
| ε-Closure of ({0}) | 0,1,3,7,9 | A | B | C |
| ε-Closure of ({2,4,7,10}) | 2,4,7,10 | B | D | E |
| ε-Closure of ({8}) | 8 | C | ϕ | C |
| ε-Closure of ({7}) | 7 | D | D | C |
| ε-Closure of ({5,8,11}) | 5,8,11 | E | F | G |
| ε-Closure of ({12}) | 12 | F | ϕ | H |
| ε-Closure of ({6,8}) | 6,8 | G | ϕ | C |
| ε-Closure of ({13}) | 13 | H | ϕ | ϕ |

**Step4:** Now, Construct the DFA from transition table.

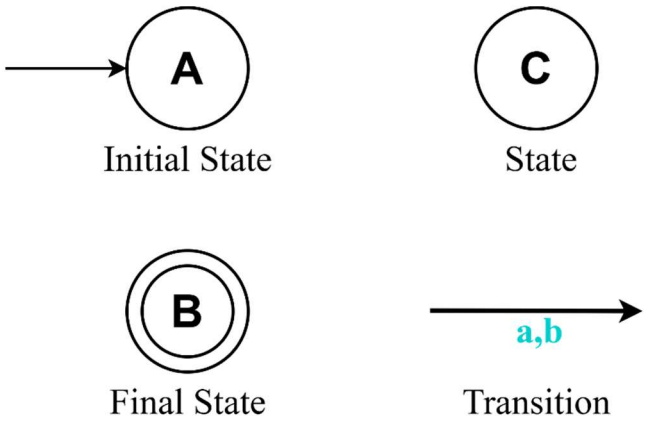


# DETERMINISTIC FINITE AUTOMATON (DFA)

# Finite Representation (FR) / Finite Automaton (FA)

- **Definition:** If a language can be recognized with a finite number of states, it's called Finite Representation (FR) or Finite Automaton (FA).
- **Key Point:** FA uses a finite number of states to represent and accept or reject strings from a language.

## ➔ Finite Representation:

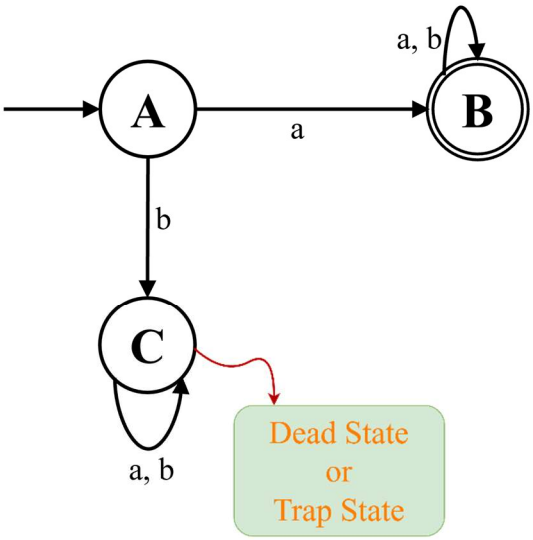



### Example 1:

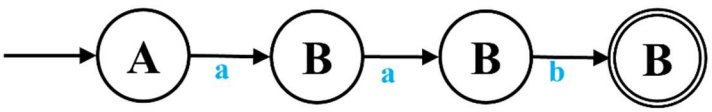
➔∑ = {a, b}

➔L1 = Set of all strings starting with 'a'

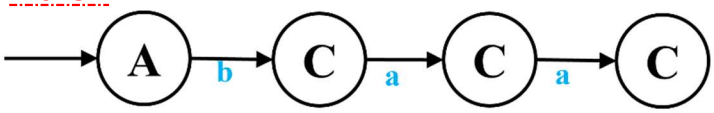
➔L1 = {a, aa, ab, abb, aba, ...}




**ACCEPT:**




**REJECT:**

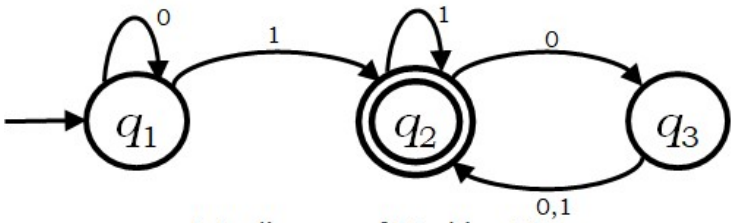


# Formal Definition - DFA

➔ A DFA is a **5-tuple**: **(Q, Σ, δ, $q_0$, F)**

1. **Q** → Finite set of states.
2. **Σ** → Finite input alphabet.
3. **δ: Q × Σ → Q** → Transition function (maps state & input to next state).
4. **$q_0$ ∈ Q** → Start state.
5. **F ⊆ Q** → Set of accept (final) states.

Example 1:




*state diagram* of Machine $M_1$

➔Here,

➢ $M_1 = (Q, \Sigma, d, q_0, F)$,

➢ Where:

$Q = \{q1, q2, q3\}$,

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

$F = \{q2\}$,

$q0 = \{q1\}$,

***δ is describe as:***

$\delta(q_1,0) = q_1, \quad \delta(q_1,1) = q_2,$

$\delta(q_2,0) = q_3, \quad \delta(q_2,1) = q_2,$

$\delta(q_3,0) = q_2, \quad \delta(q_3,1) = q_2.$

Transition Function

or

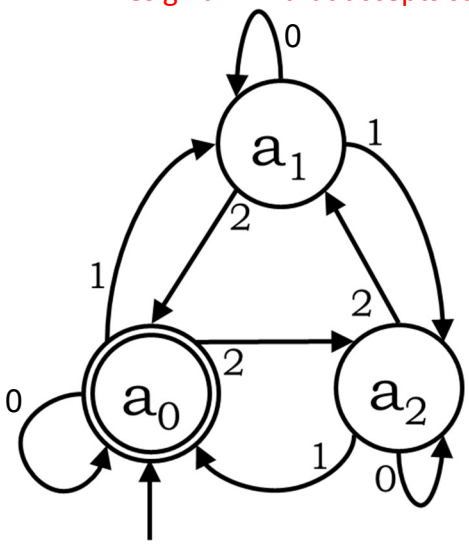
| δ | 0 | 1 |
|---|---|---|
| $q_1$ | $q_1$ | $q_2$ |
| $q_2$ | $q_3$ | $q_2$ |
| $q_3$ | $q_2$ | $q_2$ |

Transition Table

# Designing DFA

## Example 1:

➔ Design a DFA that accepts strings where the sum of all symbols (0,1,2) is a multiple of 3.




### ➔ DFA for Sum of Symbols ≡ 0,1,2 (mod 3):

- **Alphabet**: Σ = {0,1,2}
- **Language**: Strings where the sum of symbols is a **multiple of 3**.
- **States**: Q = {$a_0$, $a_1$, $a_2$}, where **$a_i$ represents S mod 3 = i**.
- **Start State**: $q_0 = a_0$
- **Final State**: F = {$a_0$}
- **Transition Table ($\delta_1$):**

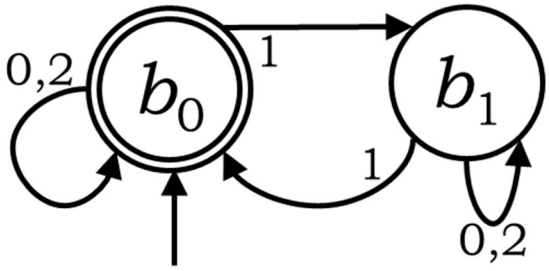
| δ | 0 | 1 | 2 |
|---|---|---|---|
| $a_0$ | $a_0$ | $a_1$ | $a_2$ |
| $a_1$ | $a_1$ | $a_2$ | $a_0$ |
| $a_2$ | $a_2$ | $a_0$ | $a_1$ |

Input example: 01120101

**Input symbol: Accepted**

## Example 2:

➔ Design a DFA that accepts strings where the sum of all symbols (0,1,2) is an even number.




### DFA for Sum of Symbols ≡ 0,1,2 (mod 2) (Even Sum):

- **Alphabet**: Σ = {0,1,2}
- **Language**: Strings where the sum of symbols is an **even number**.
- **States**: Q = {$b_0$, $b_1$}, where **$b_i$ represents S mod 2 = i**.
- **Start State**: $q_0 = b_0$
- **Final State**: F = {$b_0$}
- **Transition Table ($\delta_2$):**

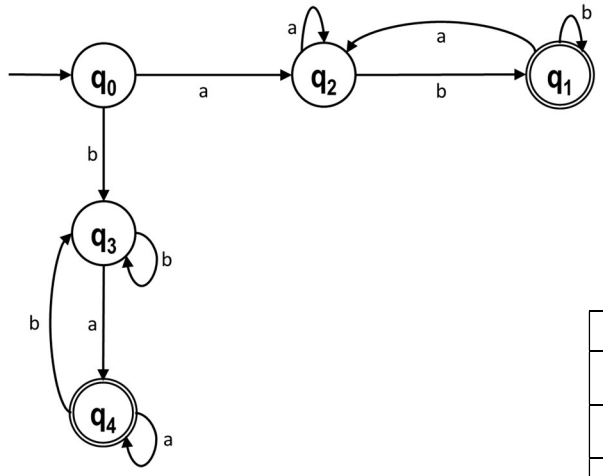
| δ | 0 | 1 | 2 |
|---|---|---|---|
| $b_0$ | $b_0$ | $b_1$ | $b_0$ |
| $b_1$ | $b_1$ | $b_0$ | $b_1$ |

Input example: 01120101

**Input symbol: Accepted**

## Example 3:

➔ Identify the Components of the DFA




### Components of the DFA:

- **Alphabet**: Σ = {a, b}
- **States**: Q = {$q_0$, $q_1$, $q_2$, $q_3$, $q_4$},
- **Start State**: $q_0$ = $q_0$
- **Final State**: F = {$q_1$, $q_4$}
- **Transition Table (δ):**

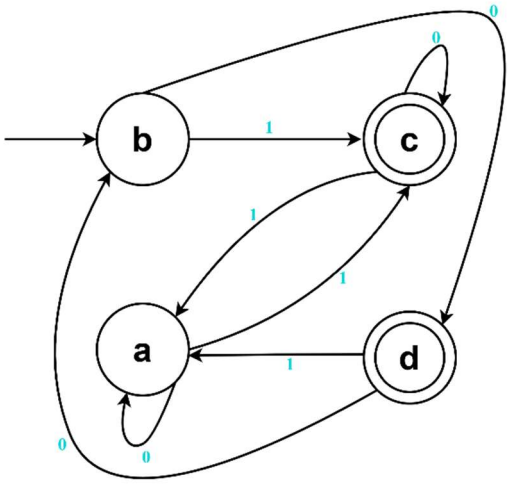
| δ | a | b |
|---|---|---|
| $q_0$ | $q_2$ | $q_3$ |
| $q_1$ | $q_2$ | $q_1$ |
| $q_2$ | $q_2$ | $q_1$ |
| $q_3$ | $q_4$ | $q_3$ |
| $q_4$ | $q_4$ | $q_3$ |

## Example 4:

➔ Design a DFA that Q = {a,b,c,d}, Σ = {0, 1}, $q_0$ = b, F = {c,d} ,

| δ | 0 | 1 |
|---|---|---|
| a | a | c |
| b | d | c |
| c | c | a |
| d | b | a |

## ➔ Solution:



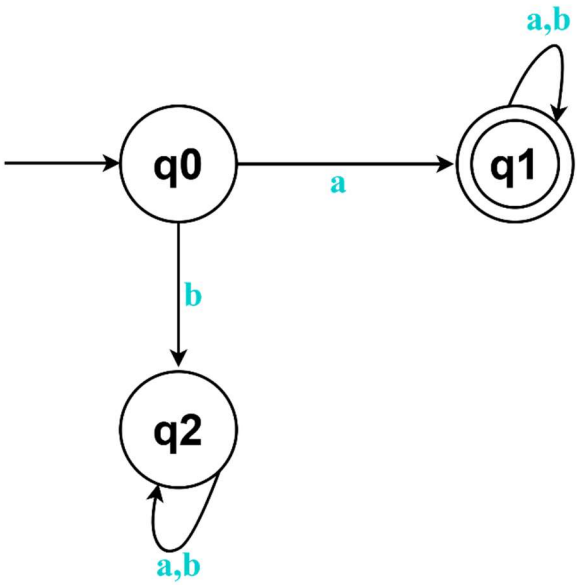
## Example 5:

➔ Given the state diagram of the machine ( $\{q_0, q_1, q_2\}$, {a, b}, δ, $q_0$, $\{q_1\}$ ) Where δ **is given the following table**

| δ | a | b |
|---|---|---|
| $q_0$ | $q_1$ | $q_2$ |
| $q_1$ | $q_1$ | $q_1$ |
| $q_2$ | $q_2$ | $q_2$ |

➔Design The DFA

## ➔ Solution:

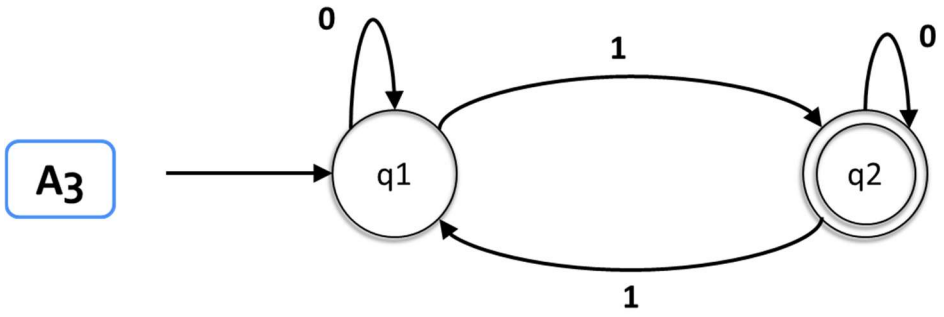



## Example 6:

➔ **A3 = {w : w is a binary string containing an odd number of *1*s}. ∑ = {0,1}**

➔ **So, $w_1 = 1$, $w_2 = 10$, $w_3 = 10101$, $w_4 = 10101$…**

**Now, L = {1, 100, 10101, 11010, 11100…}**



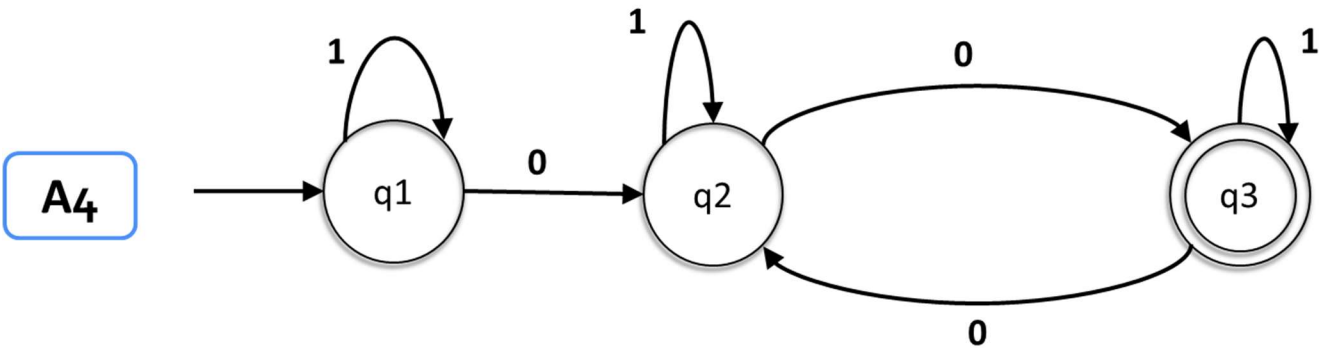

**Accepted**

## Example 7:

➔ **A4 = {w : w is a binary string containing an even number of *0*s}. ∑ = {0,1}**

➔ **So, $w_1$ = 00, $w_2$ = 001, $w_3$ = 0101, $w_4$ = 0101010…**

Now, L = {00, 100, 1010, 11010, 11100…}

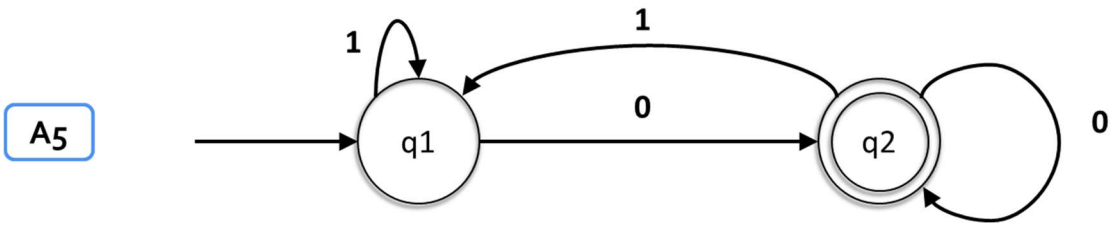



## Example 8:

➔ **A5 = {w| w ends with a *0*}. ∑ = {0,1}**

➔ **So, $w_1$ = 010, $w_2$ = 000, $w_3$ = 10, $w_4$ = 0101010…**

Now, L = {0, 10, 100, 11010, 11100…}

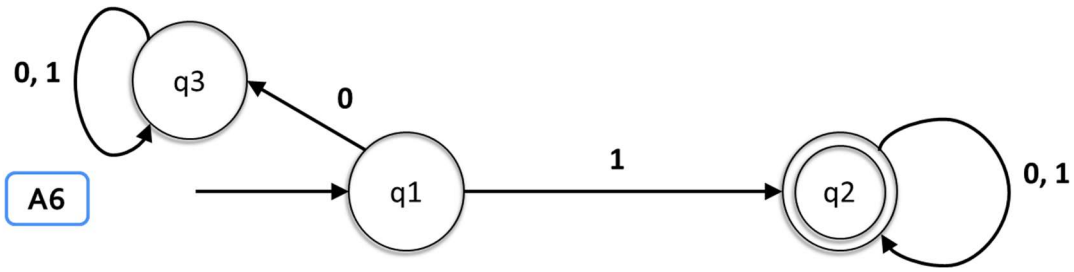



## Example 9:

➔ **A6 = {w| w begins with a *1*}. ∑ = {0,1}**

➔ **So, $w_1$ = 100, $w_3$ = 10, $w_5$ = 101010…**

Now, L = {1, 11, 101, 1001, 11100…}

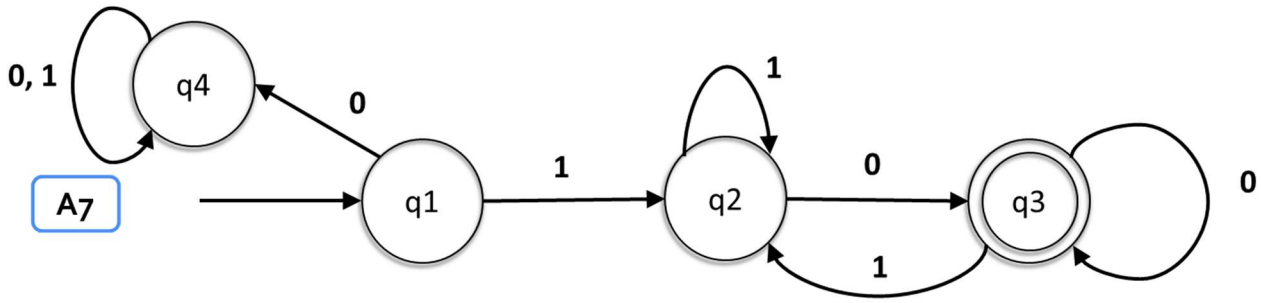


## Example 10:

➔ **A7 ={w| w begins with a *1* and ends with a *0*}. ∑ = {0,1}**

➔ **So, $w_1$ = 100, $w_2$ = 10, $w_3$ = 101010…**

**Now, L = {1, 100, 1010, 11010, 11100…}**

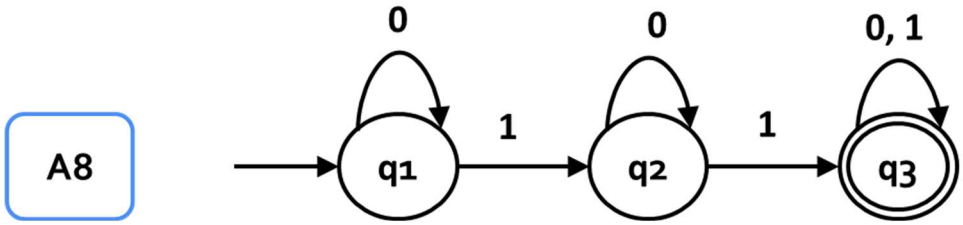



## Example 11:

➔ **A8 ={w| w has at least two *1*s}. ∑ = {0,1}**

➔ **So, $w_1$ = 1010, $w_2$ = 101, $w_4$ = 01010, $w_5$ = 101010…**

**Now, L = {11, 1100, 1010, 11010, 11100…}**

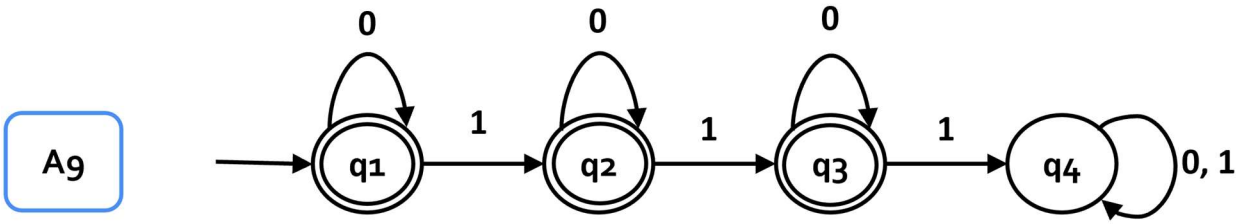



## Example 12:

➔ **A9 ={w| w has at most two *1*s}. ∑ = {0,1}**

➔ **So, $w_1$ = 1010, $w_2$ = 001, $w_3$ = 000, $w_4$ = 10100…**

**Now, L = {11, 1100, 1010, 11010, 11100…}**

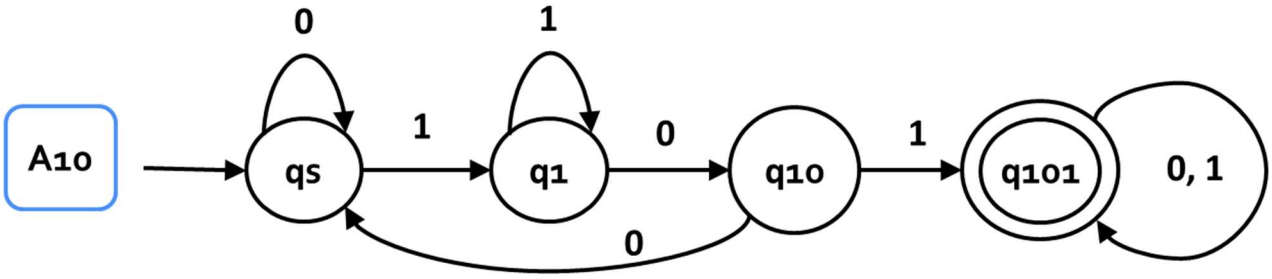


## Example 13:

➔ **A10 ={w| w has substring 101}. ∑ = {0,1}**

➔ **So, $w_1$ = 1010, $w_2$ = 001, $w_3$= 10100…**

**Now, L = {101, 1101, 10101, 11010, 111010…}**

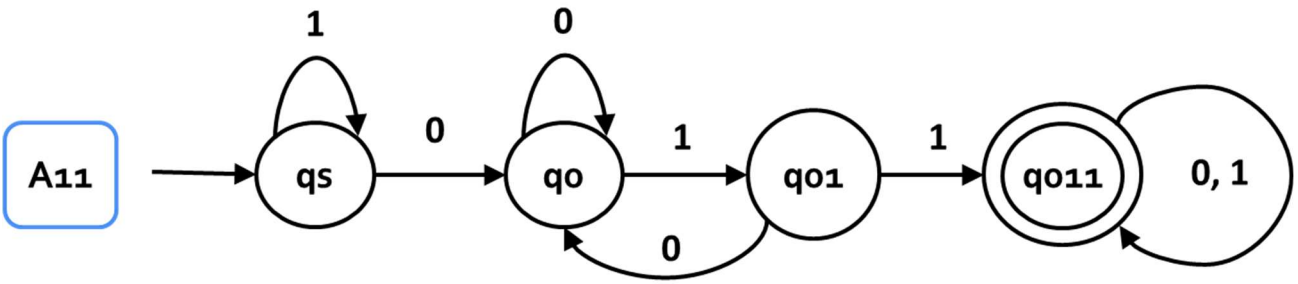



## Example 14:

➔ **A11 ={w| w has substring 011}. ∑ = {0,1}**

➔ **So, $w_1$ = 10110, $w_2$ = 000110, $w_3$ = 101100…**

**Now, L = {011, 11011, 10101011, 110110, 1011010…}**




➔ What happens for the language, **A11 ={w| w does not have substring 011}**?

➔ For **$A_{11}$ = {w | w has substring "011"}**, the DFA **accepts** strings containing "011".

➔ For **$A'_{11}$ = {w | w does not have substring "011"}**, the DFA **rejects** any string with "011" and stays in states ensuring "011" never appears.

➔ A **dead state** is used for rejection.

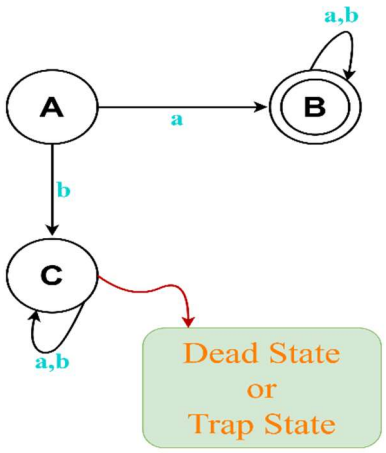
## Problem 1:

≫ Construct a DFA which accepts set of all string over the ∑ = {a,b}, where each string start with an **"a"**

**Or L(M) = {w| w Start with an "a"}**

➔ **Is Given,**

**∑ = {a,b},**
**L(M) ={ a, aa, aba, aabba,….}**



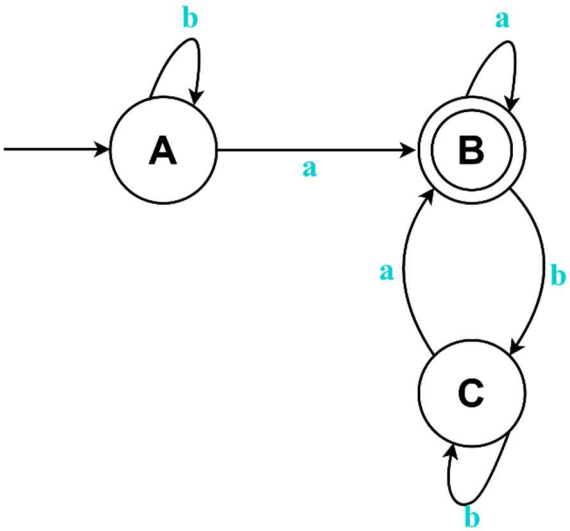

## Problem 2:

≫ Construct a DFA which accepts set of all string over the ∑ = {a,b}, where each string end with an **"a"**

**Or L(M) = {w| w End with an "a"}**

➔ **Is Given,**

**∑ = {a,b},**
**L(M) ={a, aa, aba, aabba, abababa, bba, ….}**

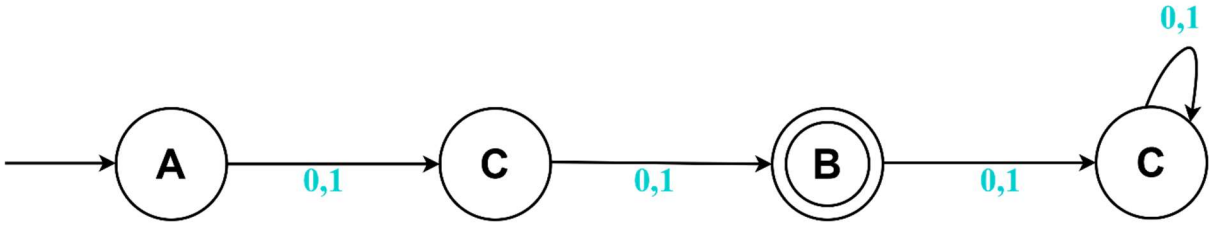


## Problem 3:

≫ Construct a DFA that accepts set of all string over the ∑ = {0,1}, of length 2.

➔ **Is Given,**

**∑ = {0,1},**
**L(M) ={00,01,10,11}**




## Problem 4:

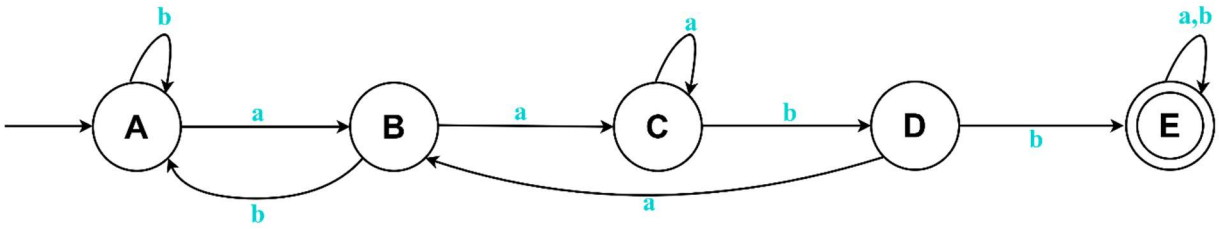
≫ Construct a DFA that accepts any string over the ∑ = {a,b}, that **does not contain** the string "aabb" in it.

➔ **Is Given,**

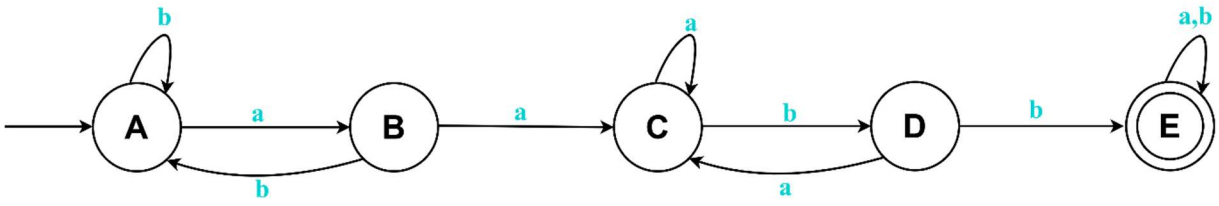
**∑ = {a,b},**

➔Let, Construct a DFA that accepts any string over the ∑ = {a,b}, that **contains** the string "aabb" in it.

**L(M) = {aabb, baabb, babaabb, babaaabb, aabaabb, aababaabb,…}**



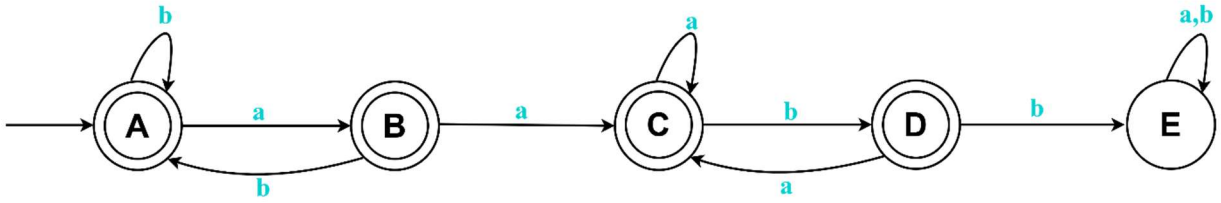

**OR**




➔ Okay, This is not our desired output since it does contain the string "aabb."

➔ So, To have our desired output, we just need to:

- Flip the states.
- Make the final states non-final state.
- Make the non-final states final state.




## Problem 5:

≫ Design DFA for the following Language. Where ∑ = {a,b} and give the formal definition of your machine. (Any-ONE)

i. L= {w | ends with 'abbab' substring}

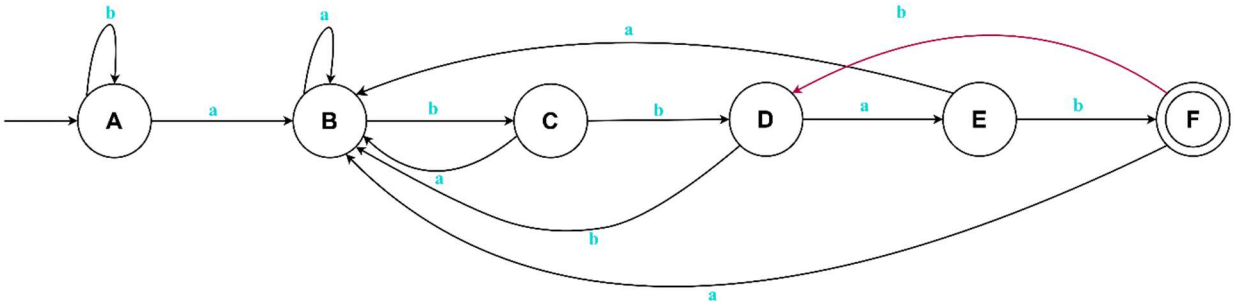
ii. L= { w| w contains exactly two a's and odd number of b } Mode1#

**(i)**

➔ **Is Given,**

**∑ = {a,b},**

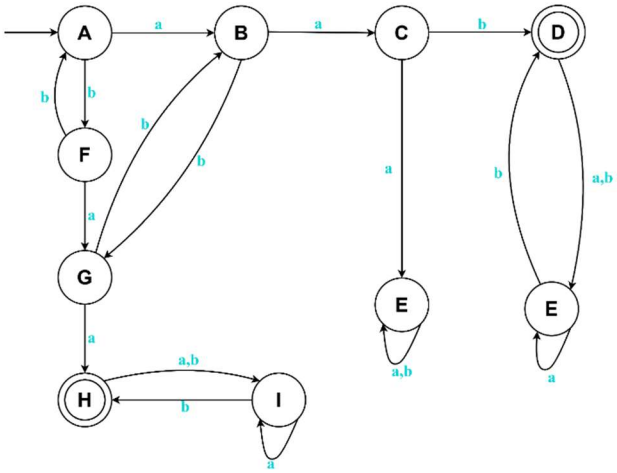
**L(M1) ={abbab, babbab, aabbab, ababbab, abbbabbab, abbabbab, abbababbab,…….}**



**(ii)**

➔ **Is Given,**

$\sum$ = {a,b},
**L(M1) ={baa, aab, baabb, babab, bbababb, bbababb, abbab,…}**




## Problem 6:

≫ Design DFA for the following Language. Consider Σ = {x,y},

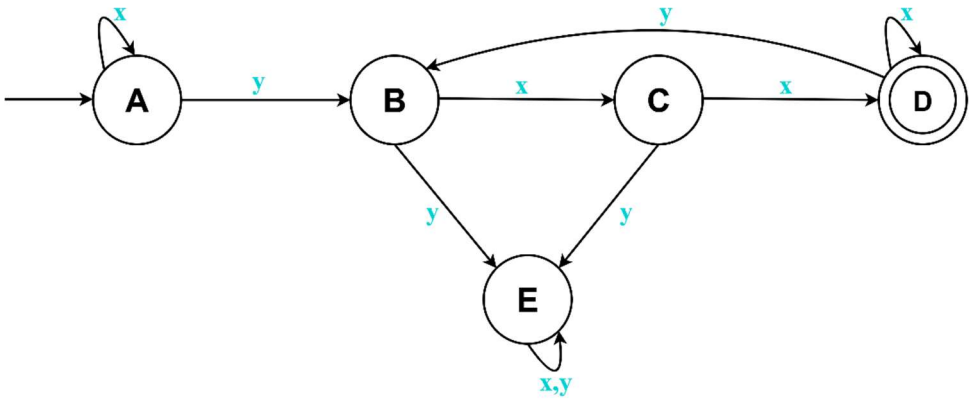
**i.** {w| **each 'y'** in w is followed by at **least two 'x'** } and simulate your machine using the string: **"xyxxyxx"** and comment whether this string is **accepted/ rejected** by your machine.

**ii.** {w | w **starts and ends** with **same symbol**} and simulate your machine using the string: **"yyxyxxy"** and comment whether this string is **accepted/ rejected** by your machine.

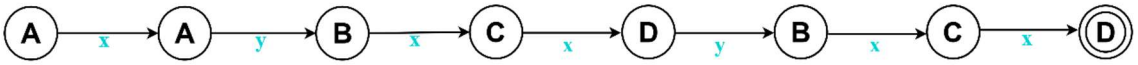
**(i)**

➔ **Is Given,**

$\sum$ = {x,y},
**L(M1) ={yxx, yxxyxx, yxxxx,…….}**



Eg: xyxxyxx

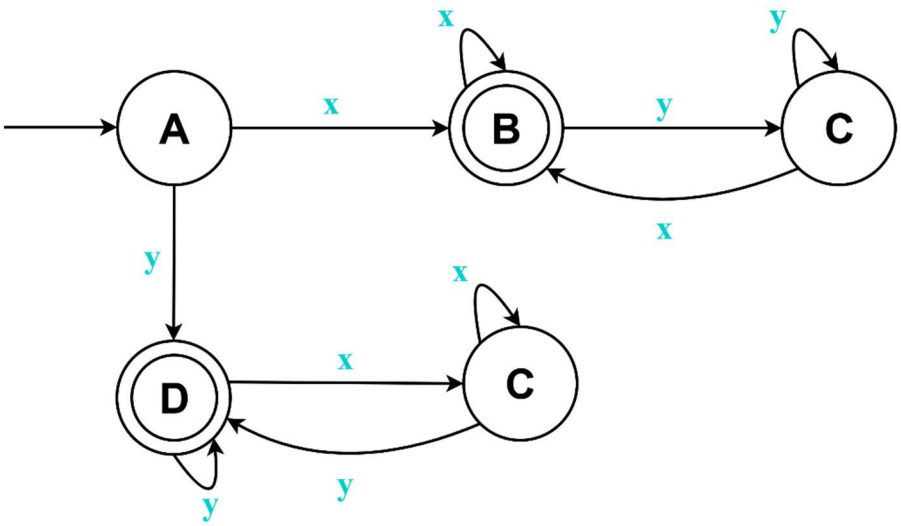



*Accepted*

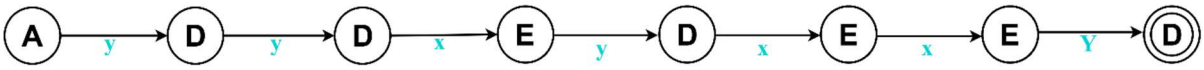
## (ii)

➔ **Is Given,**

**∑ = {x,y},**
**L(M1) ={yxxy, yxxyxxy, xyxyxx,…….}**




Eg: yyxyxxy




*Accepted*


FIRST &
FOLLOW

# FIRST

- The **FIRST set** of a grammar symbol (terminal or non-terminal) is the set of terminals that can appear **first** in some string derived from that symbol.
- If a non-terminal can derive **ε (epsilon)**, then ε is also included in its FIRST set.

## Rules for FIRST Set

| **Rule1: Terminal → itself** | **Rule2: ε → ε** |
|---|---|
| If the symbol is a **terminal**, its FIRST is the terminal itself. | If the production is **epsilon (ε)**, then FIRST is just `{ ε }`. |
| `FIRST(a) = { a }`<br>`FIRST(+) = { + }` | `A → ε`<br>`FIRST(A) = { ε }` |

| **Rule3: Non-terminal → check first symbol** |
|---|
| If a non-terminal has a production, look at the **first symbol** on the right-hand side:<br>➢ If it's a terminal → put it in FIRST.<br>➢ If it's a non-terminal → take its FIRST (but **exclude ε**).<br>➢ If ε is in that FIRST, then **look at the next symbol**.<br>➢ If all symbols can produce ε, then **include ε**. |

| **Grammar:** | **FIRST:** |
|---|---|
| `A → a B \| b`<br>`B → c \| ε` | `FIRST(A) = { a, b }`<br>`FIRST(B) = { c, ε }` |

**Example 2:**

| **Grammar:** | **FIRST:** |
|---|---|
| `E → T E'`<br>`E' → + T E' \| ε`<br>`T → id` | `FIRST(E) = { id }`<br>`FIRST(E') = { +, ε }`<br>`FIRST(T) = { id }` |

## Problem 1:

≫ Calculate the first functions for the given grammar-

S → aBDh
B → cC
C → bC | ∈
D → EF
E → g | ∈
F → f | ∈

**Solutions:**

| Grammar: | FIRST: |
|---|---|
| S → aBDh<br>B → cC<br>C → bC \| ∈<br>D → EF<br>E → g \| ∈<br>F → f \| ∈ | FIRST(S) = { a }<br>FIRST(B) = { c }<br>FIRST(C) = { b, ε }<br>FIRST(D) = { g, f, ε }<br>FIRST(E) = { g, ε }<br>FIRST(F) = { f, ε } |

## Problem 2:

≫ Calculate the first functions for the given grammar-

S → A
A → aB | dA
B → b
C → g

**Solutions:**

| Grammar: | FIRST: |
|---|---|
| S → A<br>A → aB \| d<br>B → b<br>C → g | FIRST(S) = { a, d }<br>FIRST(A) = { a, d }<br>FIRST(B) = { b }<br>FIRST(C) = { g } |

## Problem 3:

≫ Calculate the first functions for the given grammar-

S → (L) | a
L → SL'
L' → ,SL' | ∈

**Solutions:**

| Grammar: | FIRST: |
|---|---|
| S → (L) \| a<br>L → SL'<br>L' → ,SL' \| ∈ | FIRST(S) = { (, a }<br>FIRST(L) = { (, a }<br>FIRST(L') = { ,, ∈} |

## Problem 4:

≫ Calculate the first functions for the given grammar-

S → AaAb | BbBa

A → ∈

B → ∈

**Solutions:**

| Grammar: | FIRST: |
|---|---|
| S → AaAb \| BbBa | FIRST(S) = { a, b } |
| A → ∈ | FIRST(A) = {∈} |
| B → ∈ | FIRST(B) = {∈} |

## Problem 5:

≫ Calculate the first functions for the given grammar-

E → F + T | T

T → E x F | F

F → (E) | id

**Solutions:**

| Grammar: | FIRST: |
|---|---|
| E → F + T \| T | FIRST(E) = {(, id} |
| T → E x F \| F | FIRST(T) = {(, id} |
| F → (E) \| id | FIRST(F) = {(, id} |

## Problem 6:

≫ Calculate the first functions for the given grammar-

S → ACB | CbB | Ba

A → da | BC

B → g | ∈

C → h | ∈

**Solutions:**

| Grammar: | FIRST: |
|---|---|
| S → ACB \| CbB \| Ba | FIRST(S) = { d, g, h, ∈, b, a} |
| A → da \| BC | FIRST(A) = { d, g, h, ∈ } |
| B → g \| ∈ | FIRST(B) = { g, ∈ } |
| C → h \| ∈ | FIRST(C) = { h, ∈ } |

# FOLLOW

- The **FOLLOW(A)** of a non-terminal A is the set of terminals that can appear **immediately to the right of A** in some derivation.
- Think of it as: *"what symbols can come after A in the grammar?"*

## Rules for FIRST Set

<table>
<tr><td>Rule1: Start Symbol Rule</td><td>Rule2: A → αBβ</td></tr>
<tr><td>For the start symbol s of the grammar:</td><td>If a non-terminal B is followed by something (β), then</td></tr>
<tr><td rowspan="2">$ ∈ FOLLOW(S)</td><td>FIRST(β) − { ε } ⊆ FOLLOW(B)</td></tr>
<tr><td>A → B c<br>Here c comes after B, so FOLLOW(B)={c}.</td></tr>
<tr><td colspan="2">Rule3: A → αB or A → αBβ where ε ∈ FIRST(β)</td></tr>
<tr><td colspan="2">➢ If B is at the end of production, or what follows can be ε, then:<br>FOLLOW(A) ⊆ FOLLOW(B)</td></tr>
<tr><td>Grammar:</td><td>FOLLOW:</td></tr>
<tr><td>A → a B | b<br>B → c | Ab</td><td>FOLLOW(A) = { $, b }<br>FOLLOW(B) = { $, b }</td></tr>
<tr><td colspan="2">Rule4: A → αBC where C is a non-terminal.</td></tr>
<tr><td colspan="2">➢ If If B is followed by C →<br>FIRST(C) − { ε } ⊆ FOLLOW(B)</td></tr>
<tr><td>Grammar:</td><td>FIRST:</td></tr>
<tr><td>A → B C<br>B → b B<br>C → c | ε</td><td>FIRST(A) = { b }<br>FIRST(B) = { b }<br>FIRST(C) = { c, ε }</td></tr>
<tr><td colspan="2">FOLLOW:</td></tr>
<tr><td colspan="2">FOLLOW(A) = { $ }<br>FOLLOW(B) = { c, $ }<br>FOLLOW(C) = { $ }</td></tr>
</table>

## Problem 1:

≫ Calculate the first and follow functions for the given grammar-

S → aBDh
B → cC
C → bC | ∈
D → EF
E → g | ∈
F → f | ∈

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = { a } | FOLLOW(S) = { $ } |
| FIRST(B) = { c } | FOLLOW(B) = { g, f, h } |
| FIRST(C) = { b, ε } | FOLLOW(C) = { g, f, h } |
| FIRST(D) = { g, f, ε } | FOLLOW(D) = { h } |
| FIRST(E) = { g, ε } | FOLLOW(E) = { f, h } |
| FIRST(F) = { f, ε } | FOLLOW(F) = { h } |

## Problem 2:

≫ Calculate the first and follow functions for the given grammar-

S → A
A → aBb | dA
B → bCc
C → g

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = { a, d } | FOLLOW(S) = { $ } |
| FIRST(A) = { a, d } | FOLLOW(A) = { $ } |
| FIRST(B) = { b } | FOLLOW(B) = { b } |
| FIRST(C) = { g } | FOLLOW(C) = { c } |

## Problem 3:

≫ Calculate the first and follow functions for the given grammar-

S → (L) | a

L → SL'

L' → ,SL' | ∈

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = { (, a } | FOLLOW (S) = { $, , , ) } |
| FIRST(L) = { (, a } | FOLLOW (L) = { ) } |
| FIRST(L') = { ,, ∈} | FOLLOW (L') = { ) } |

## Problem 4:

≫ Calculate the first and follow functions for the given grammar-

S → AaAb | BbBa

A → ∈

B → ∈

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = { a, b } | FOLLOW (S) = { $ } |
| FIRST(A) = {∈} | FOLLOW (A) = { a, b } |
| FIRST(B) = {∈} | FOLLOW (B) = { b, a } |

## Problem 5:

≫ Calculate the first and follow functions for the given grammar-

E → F + T | T

T → E x F | F

F → (E) | id

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(E) = {(, id } | FOLLOW (E) = { $, x, ) } |
| FIRST(T) = {(, id } | FOLLOW (T) = { $, x, ) } |
| FIRST(F) = {(, id } | FOLLOW (F) = { $, x, ) } |

## Problem 6:

≫ Calculate the first and follow functions for the given grammar-

E → TE′
E′→ +TE′ | ∈
T → FT′
T′→ *FT′ | ∈
F → (E) | id

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(E) = {(, id } | FOLLOW(E) = {$, )} |
| FIRST(E′) = {+, ∈ } | FOLLOW(E′) = {$, )} |
| FIRST(T) = {(, id } | FOLLOW(T) = {+, $, )} |
| FIRST(T′) = {*, ∈ } | FOLLOW(T′) = {+, $, )} |
| FIRST(F) = {(, id } | FOLLOW(F) = {*, +, $, ) } |

## Problem 7:

≫ Calculate the first and follow functions for the given grammar-

S → ACB | CbB | Ba
A → da | BC
B → g | ∈
C → h | ∈

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = { d, g, h, ∈ , b, a} | FOLLOW(S) = { $ } |
| FIRST(A) = { d, g, ∈ } | FOLLOW(A) = { h, g, $ } |
| FIRST(B) = { g, ∈ } | FOLLOW(B) = { $, a, h, g} |
| FIRST(C) = { h, ∈ } | FOLLOW(C) = { g, b, h, $ } |

## Problem 8:

≫ Calculate the first and follow functions for the given grammar-

S → ABCDE

A → a | ∈

B → b | ∈

C → c

D → d | ∈

E → e | ∈

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = { a, b, c} | FOLLOW(S) = { $ } |
| FIRST(A) = { a, ∈ } | FOLLOW(A) = { b, c } |
| FIRST(B) = { b, ∈ } | FOLLOW(B) = { c } |
| FIRST(C) = { c } | FOLLOW(C) = { d, e, $ } |
| FIRST(D) = { d, ∈ } | FOLLOW(D) = { e, $ } |
| FIRST(E) = { e, ∈ } | FOLLOW(E) = { $ } |

## Problem 9:

≫ Calculate the first and follow functions for the given grammar-

S → ABC | DE

A → id | (S) | ∈

B → +B | *B | ∈

C → DA | ∈

D → num(A)| num | num(E)

E → S | ∈

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = { id, (, +, *, num, ∈ } | FOLLOW(S) = { $, ) } |
| FIRST(A) = { id, (, ∈ } | FOLLOW(A) = { +, *, num, $, )} |
| FIRST(B) = { +, *, ∈ } | FOLLOW(B) = { num, $, ) } |
| FIRST(C) = { num, ∈ } | FOLLOW(C) = { $, ) } |
| FIRST(D) = { num } | FOLLOW(D) = { id, (, +, *, num, $, ) } |
| FIRST(E) = { id, (, +, *, num, ∈ } | FOLLOW(E) = { $, )} |

# Left Recursion in Grammar

⇨ A grammar is **left recursive** if there exists a non-terminal **A** such that:

$$A \Rightarrow A\ \alpha$$

A can derive a string starting with itself. This causes **infinite recursion** in **top-down parsers (like recursive descent or LL(1))**.

## Types of Left Recursion

| **Rule1:** (a) **Immediate Left Recursion** | |
|---|---|
| ➢ When a production refers to itself directly: | |
| **Example Grammar:** | **Here:** |
| A → A α \| β | **α = a sequence of grammar symbols (must not be ε)**<br>**β = some alternative that does not begin with A** |
| **Elimination Rule:** | |
| **Before Elimination:** | **After Elimination:** |
| A → A α \| β | A → β A'<br>A' → α A' \| ε |

**Example 1: Immediate Left Recursion**

| **Grammar:** | **After Elimination:** |
|---|---|
| E → E + T \| T | E → T E'<br>E'→ + T E' \| ε |

**Rule2:** (b) **Indirect Left Recursion**

- When recursion happens through another non-terminal:

| **Example Grammar:** | **Here:** |
|---|---|
| A → B α<br>B → A β | This means A ⇒ B α ⇒ A β α → left recursion indirectly. |

**Elimination Rule:**

- First, reorder productions so non-terminals appear in a proper sequence.
- Substitute productions to make recursion direct.
- Then remove using the **immediate left recursion rule**.

| **Before Elimination:** Gramma | **Step 1:** Substitute B in A ⟶ B a |
|---|---|
| A → B a \| b<br>B → A c \| d | A → (A c \| d) a \| b<br>A → A c a \| d a \| b |

**Step 2:** Now we see immediate left recursion **A → A c a.**
**Fix it like before:**

A → d a A' | b A'
A' → c a A' | ε

## Example 1: Indirect Left Recursion

| **Grammar:** | **Step 1:** Substitute B in A ⟶ B a |
|---|---|
| A → B a \| c<br>B → C b \| d<br>C → A c \| e | A → C b a \| c<br>B → C b \| d<br>C → A c \| e |
| **Step2:** Substitute C in A ⟶ C b a | **After Elimination:** |
| A → A c b a \| c<br>B → C b \| d<br>C → A c \| e | A → c A'<br>A' → c b a A' \| ε<br>B → C b \| d<br>C → A c \| e |

# Parsing Table Construction (LL(1))

- An **LL(1) parsing table** is a table used in **predictive parsing** (top-down).
- "LL(1)" means:
  - **L**: Left-to-right scanning of input
  - **L**: Leftmost derivation
  - **1**: One token lookahead
- The table helps the parser decide **which production rule to apply** based only on:
- the **current non-terminal (row)**, and
- the **next input symbol (column)**.

## Requirements

- Grammar must be **free of left recursion**.
- Grammar must be **left factored**.
- Otherwise, conflicts appear (two entries in the same cell).

## Construction Rules

⇨ For each production $A \rightarrow \alpha$

| **Rule1:** If $\varepsilon \notin FIRST(\alpha)$: | **Rule2:** If $\varepsilon \in FIRST(\alpha)$: |
|---|---|
| For each terminal $a \in FIRST(\alpha)$, add: | For each terminal $b \in FOLLOW(A)$, add: |
| $M[A, a] = A \rightarrow \alpha$ | $M[A, b] = A \rightarrow \alpha$ |
| **Rule3:** If $\$ \in FOLLOW(A)$, also add: | |
| $M[A, \$] = A \rightarrow \alpha$ | |

## Problem 1:

≫ Calculate the first and follow functions for the given grammar- also construct parsing table.

E → T E'
E'→ + T E' | ∈
T → F T'
T'→ * F T' | ∈
F → ( E ) | id

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(E) = {(, id } | FOLLOW(E) = {$, )} |
| FIRST(E') = {+, ∈ } | FOLLOW(E') = {$, )} |
| FIRST(T) = {(, id } | FOLLOW(T) = {+, $, )} |
| FIRST(T') = {*, ∈ } | FOLLOW(T') = {+, $, )} |
| FIRST(F) = {(, id } | FOLLOW(F) = {*, +, $, ) } |

Parsing Table.

| FIRST | FOLLOW | Nonterminal | + | * | ( | ) | id | $ |
|---|---|---|---|---|---|---|---|---|
| {(, id } | {$, )} | E | | | E → T E' | | E → T E' | |
| {+, ∈ } | {$, )} | E' | E'→ + T E' | | | E'→∈ | | E'→∈ |
| {(, id } | {+, $, )} | T | | | T → F T' | | T → F T' | |
| {*, ∈ } | {+, $, )} | T' | T'→∈ | T'→ * F T' | | T'→∈ | | T'→∈ |
| {(, id } | {*, +, $, ) } | F | | | F → ( E ) | | F → id | |

## Problem 2:

≫ Calculate the first and follow functions for the given grammar- also construct parsing table.

S → A
A → aB | Ad
B → bBC | f
C → g

**Solutions:**

**Eliminate the left recursive for the given grammar**

| Before Elimination: | After Elimination: |
|---|---|
| S → A<br>A → aB \| A d<br>B → bBC \| f<br>C → g | S → A<br>A → aBA'<br>A' → dA' \| ∈<br>B → bBC \| f<br>C → g |

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = {a}<br>FIRST(A) = {a}<br>FIRST(A') = {d, ∈ }<br>FIRST(B) = {b,f}<br>FIRST(C) = {g} | FOLLOW(S) = {$}<br>FOLLOW(A) = {$}<br>FOLLOW(A') = {$}<br>FOLLOW(B) = { d, $, g }<br>FOLLOW(C) = { d, $, g } |

Parsing Table.

| FIRST | FOLLOW | Nonterminal | a | b | d | f | g | $ |
|---|---|---|---|---|---|---|---|---|
| {a} | {$} | S | S → A | | | | | |
| {a} | {$} | A | A → aBA' | | | | | |
| {d, ∈ } | {$} | A' | | | A' → dA' | | | A' → ∈ |
| {b,f} | { d, $, g } | B | | B → bBC | | B → f | | |
| {g} | { d, $, g } | C | | | | | C → g | |

## Problem 3:

≫ Calculate the first and follow functions for the given grammar- also construct parsing table.

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +E \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow T \mid \epsilon \\
F &\rightarrow PF' \\
F' &\rightarrow *F' \mid \epsilon \\
P &\rightarrow (E) \mid \mathrm{a} \mid \mathrm{b} \mid \mathrm{ep}
\end{aligned}
$$

Solutions:

| FIRST: | FOLLOW: |
|---|---|
| FIRST(E) = {(, a, b, ep}<br>FIRST(E') = {+, ∈ }<br>FIRST(T) = {(, a, b, ep}<br>FIRST(T') = {(, a, b, ep, ∈}<br>FIRST(F) = {(, a, b, ep}<br>FIRST(F') = {*, ∈ }<br>FIRST(P) = {(, a, b, ep} | FOLLOW(E) = {$, )}<br>FOLLOW(E') = {$, )}<br>FOLLOW(T) = {+, $, )}<br>FOLLOW(T') = {+, $, )}<br>FOLLOW(F) = {(, a, b, ep, +, $, )}<br>FOLLOW(F') = {(, a, b, ep, +, $, )}<br>FOLLOW(P) = {*, (, a, b, ep, +, $, )} |

**Parsing Table.**

| FIRST | FOLLOW | Non Terminal | ( | ) | a | b | ep | + | * | $ |
|---|---|---|---|---|---|---|---|---|---|---|
| {(, a, b, ep} | {$, )} | **E** | E → T E' | | E → T E' | E → T E' | E → T E' | | | |
| {+, ∈ } | {$, )} | **E'** | | E' → ∈ | | | | E' → + E | | E' → ∈ |
| {(, a, b, ep} | {+, $, )} | **T** | T → F T' | | T → F T' | T → F T' | T → F T' | | | |
| {(, a, b, ep, ∈} | {+, $, )} | **T'** | T' → T | T' → ∈ | T' → T | T' → T | T' → T | T' → ∈ | | T' → ∈ |
| {(, a, b, ep} | {(, a, b, ep, +, $, )} | **F** | F → PF' | | F → PF' | F → PF' | F → PF' | | | |
| {*, ∈ } | {(, a, b, ep, +, $, )} | **F'** | F' → ∈ | F' → ∈ | F' → ∈ | F' → ∈ | F' → ∈ | F' → ∈ | F' → *F' | F' → ∈ |
| {(, a, b, ep} | {*, (, a, b, ep, +, $, )} | **P** | P → (E) | | P → a | P → b | P → ep | | | |

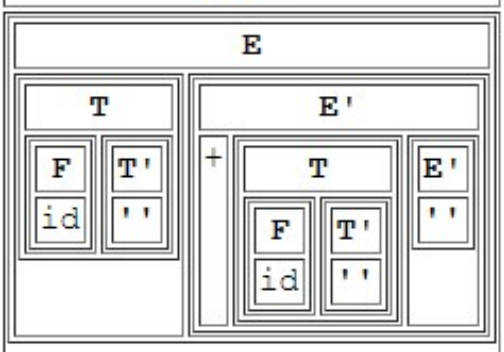
# Stack Movement a Predictive Parser (Example 1)

Given input String: id + id

**Trace**

| Stack | Input | Rule |
|---|---|---|
| $ E | id + id $ | |
| $ E' T | id + id $ | E -> T E' |
| $ E' T' F | id + id $ | T -> F T' |
| $ E' T' id | id + id $ | F -> id |
| $ E' T' | + id $ | |
| $ E' | + id $ | T' -> '' |
| $ E' T + | + id $ | E' -> + T E' |
| $ E' T | id $ | |
| $ E' T' F | id $ | T -> F T' |
| $ E' T' id | id $ | F -> id |
| $ E' T' | $ | |
| $ E' | $ | T' -> '' |
| $ | $ | E' -> '' |

**Tree**



## Problem 4:

≫ Check the following context free grammar whether it is LL(1) or not

$$S \rightarrow A \mid B$$
$$A \rightarrow A\ a \mid B\ b$$
$$B \rightarrow b \mid \varepsilon$$

**Solutions:**

We know a grammar to be **LL(1)**:

1. If it must **not have left recursion**.
2. FIRST sets of productions must be **disjoint**.
3. If a production has ε, **FIRST (non-terminal) ∩ FOLLOW (non-terminal) = ∅**.

| **Rule1:** Check left recursion | **Rule2:** **Check FIRST sets of multiple productions** |
|---|---|
| $A \rightarrow A\ a$ | FIRST(S) = {b, b, ε }<br>FIRST(A) = {b} (ignoring recursion for a moment)<br>FIRST(B) = {b, ε} |
| **direct left recursion** | The FIRST sets **intersect** |

| **Rule3:** If a production has ε, **FIRST (non-terminal) ∩ FOLLOW (non-terminal) = ∅** | |
|---|---|
| **FOLLOW: B → ε →** ε-production | **So,** This grammar **violates all LL(1) rules**: |
| FOLLOW(B) = {$, b} | 1. Contains **left recursion**<br>2. Has **intersecting FIRST sets**<br>3. Has **ε-production conflicting with FOLLOW** |
| **FIRST(B) ∩ FOLLOW(B) ≠ ∅** | So, grammar is not LL(1) |

## Problem 5:

≫ Check the following context free grammar whether it is LL(1) or not

$$X \rightarrow aZbXY \mid c$$
$$Y \rightarrow dX \mid \varepsilon$$
$$Z \rightarrow e$$

**Solutions:**

We know a grammar to be **LL(1)**:

1. If it must **not have left recursion**.
2. FIRST sets of productions must be **disjoint**.
3. If a production has ε, **FIRST (non-terminal) ∩ FOLLOW (non-terminal) = ∅**.

| **Rule1:** Check left recursion | **Rule2:** Check FIRST sets of multiple productions |
|---|---|
| No left recursion. | FIRST(X) = {a, c }<br>FIRST(Y) = {d, ε }<br>FIRST(Z) = {e}<br>FIRST sets of its alternative productions are pairwise disjoint. |

**Rule3:** If a production has ε, **FIRST (non-terminal) ∩ FOLLOW (non-terminal) = ∅**

| **FOLLOW:** Y → ε → ε-production | **So,** This grammar not **justify all LL(1) rules**: |
|---|---|
| FOLLOW(Y) = {$,d} | 1. **No left recursion.**<br>2. NO **intersecting FIRST sets**<br>3. **ε-production conflicting with FOLLOW** |
| **FIRST(A) ∩ FOLLOW(A) = {d) ≠ ∅** | So, grammar is not LL(1) |

**OR**

X → aZbXY | c
Y → dX | ε
Z → e

**Solutions:**

| **FIRST:** | **FOLLOW:** |
|---|---|
| FIRST(X) = {a, c} | FOLLOW(X) = {$, d} |
| FIRST(Y) = {d, ε } | FOLLOW(Y) = {$, d} |
| FIRST(Z) = {e} | FOLLOW(Z) = {b} |

**Parsing Table.**

| FIRST | FOLLOW | Nonterminal | a | b | c | d | e | $ |
|---|---|---|---|---|---|---|---|---|
| {a, c} | {$, d} | X | X → aZbXY | X → c | | | | |
| {d, ε } | {$, d} | Y | | | | Y → dX, ε | | Y → ε |
| {e} | {b} | Z | | | | | Z → e | |

Since a cell has more than one entry that means there is a parsing conflict, so the given grammar is not LL1

# Practice Problem

Problem1: Draw the NFA for the following regular expression "`(0|1)*11*(1|0)*1`".

**Step 1:** Parse the RE into Subexpressions

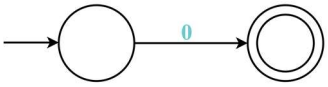
≫ ***The RE "(0|1)*11*(1|0)*1"can be decomposed as:***

1. (0|1)*
2. (1|0)*
3. 1*
4. 0
5. 1

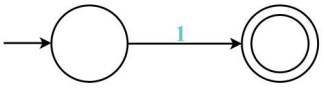
We'll build NFAs for each component and then add them.

**Step 2:** Build NFAs for Basic Symbols

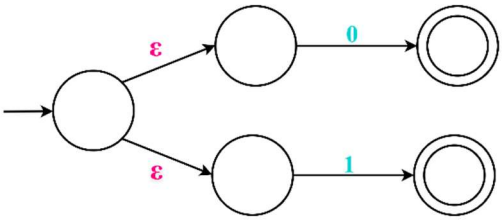
NFA for 0



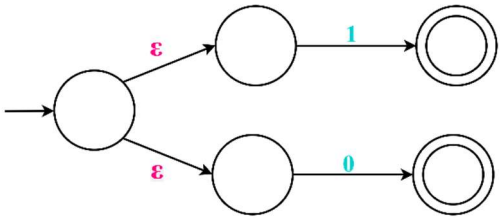

NFA for 1



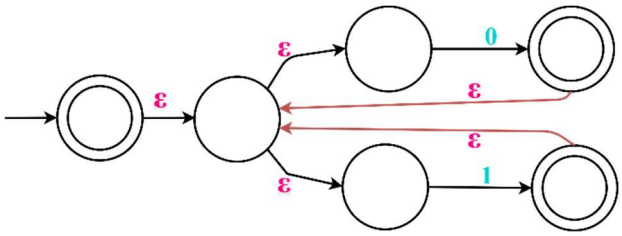

**Step 3:** NFA for **(0|1)** (Union)



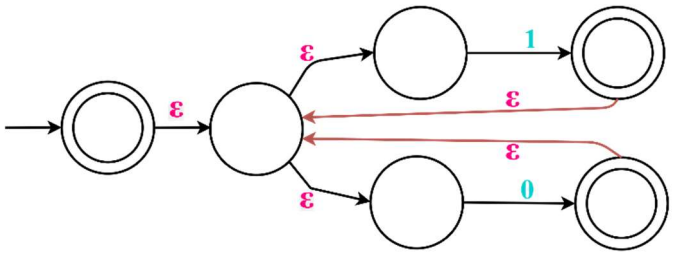

**Step 4:** NFA for **(1|0)** (Union)



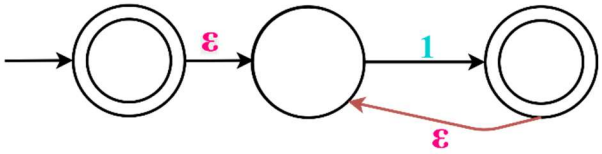

**Step 5:** NFA for "**(0|1)***" (Star)



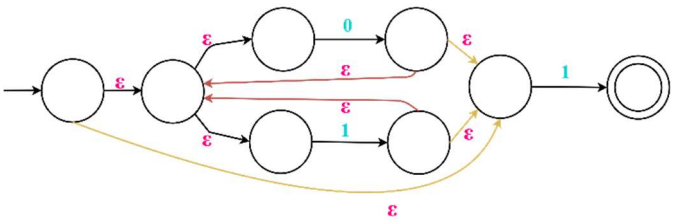

**Step 6:** NFA for "**(1|0)***" (Star)



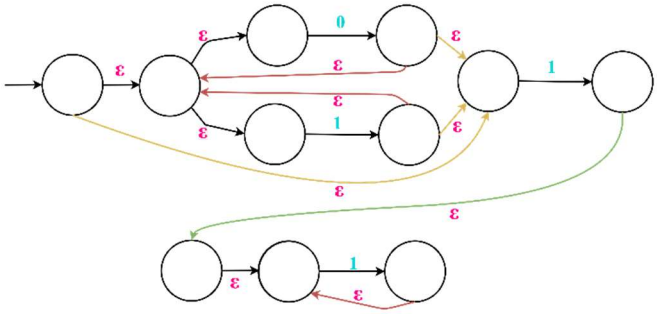

**Step 7:** NFA for "**1***" (Star)



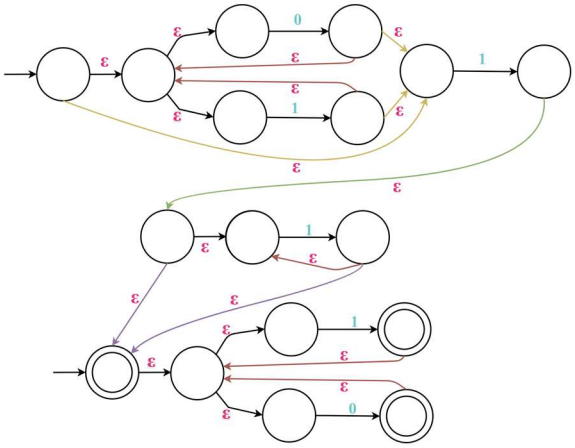

**Step 8:** Combine "**(0|1)*1**". (Concatenation)



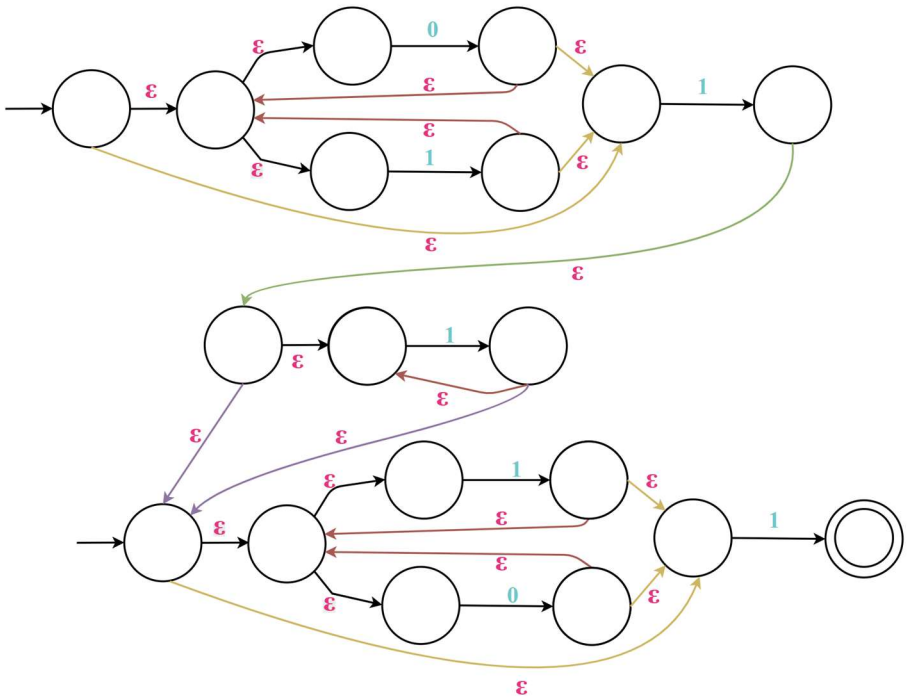
**Step 9:** Combine "**(0|1)*11***". (Concatenation)




**Step 10:** Combine "**(0|1)*11*(1|0)*)**"(Concatenation)




**Step 11:** Combine "**("(0|1)*11*(1|0)*1"** (Concatenation)



Problem2: **a.** Find the FIRST and FOLLOW function from the following grammar.

```
S → FTc
T → id = A*
A → FB
B → +FB | ε
F → int | a | b
```

**b.** Construct the parsing table for the context free grammar given in problem **2a** and show the moves made by the predictive parser for the string: **int id = a + b* c**

**(a)**

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = {int, a, b } | FOLLOW(S) = {$} |
| FIRST(T) = {id } | FOLLOW(T) = {c} |
| FIRST(A) = {int, a, b } | FOLLOW(A) = {*} |
| FIRST(B) = {+, ∈ } | FOLLOW(B) = {*} |
| FIRST(F) = {int, a, b } | FOLLOW(F) = {id, +, * } |

**(b)**

**Solutions:**

**Parsing Table.**

| FIRST | FOLLOW | Non Terminal | a | b | c | int | id | = | + | * | $ |
|---|---|---|---|---|---|---|---|---|---|---|---|
| {int, a, b } | {$} | S | S → FTc | S → FTc | | S → FTc | | | | | |
| {id} | {c} | T | | | | | T → id = A* | | | | |
| {int, a, b } | {*} | A | A → FB | A → FB | | A → FB | | | | | |
| {+, ∈ } | {*} | B | | | | | | | B → +FB | B → ε | |
| {int, a, b } | {id, +, * } | F | F → a | F → b | | F → int | | | | | |

## Stack Movement a Predictive Parser

Given input String: **int id = a + b* c**

| Trace | | |
|---|---|---|
| **Stack** | **Input** | **Rules** |
| $ S | int id = a + b* c $ | |
| $ c T F | int id = a + b* c $ | S → FTc |
| $ c T int | int id = a + b* c $ | F → int |
| $ c T | id = a + b* c $ | |
| $ c * A = id | id = a + b* c $ | T → id = A* |
| $ c * A = | = a + b* c $ | |
| $ c * A | a + b* c $ | |
| $ c * B F | a + b* c $ | A → FB |
| $ c * B a | a + b* c $ | F → a |
| $ c * B | + b* c $ | |
| $ c * B F + | + b* c $ | B → +FB |
| $ c * B F | b* c $ | |
| $ c * B b | b* c $ | F → b |
| $ c * B | * c $ | |
| $ c * | * c $ | B → ε |
| $ c | c $ | |
| $ | $ | |

Problem3: Find the FIRST and FOLLOW set from the following grammar.

```
E → TX
X → +E | ε
T → int Y |(E)
Y → *T | ε
```

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(E) = {int, ( } | FOLLOW(E) = {$,)} |
| FIRST(X) = {+, ∈ } | FOLLOW(X) = {$,)} |
| FIRST(T) = {int, ( } | FOLLOW(T) = {+, $,)} |
| FIRST(Y) = {*, ∈ } | FOLLOW(Y) = {+, $,)} |

🧠 Problem4: Find the FIRST and FOLLOW set from the following grammar.

```
S → x U y
U → M V
V → M V | ε
M → num * E
E → H W
W → / H W | ε
H → (E) | num | x |y
```

**Solutions:**

| FIRST: | FOLLOW: |
|---|---|
| FIRST(S) = {x} | FOLLOW(S) = {$} |
| FIRST(U) = {num} | FOLLOW(U) = {y} |
| FIRST(V) = {num, ∈ } | FOLLOW(V) = {y} |
| FIRST(M) = {num} | FOLLOW(M) = {num, y} |
| FIRST(E) = {(, num, x, y} | FOLLOW(E) = {num, y, )} |
| FIRST(H) = {/, ∈ } | FOLLOW(W) = {num, y, )} |
| FIRST(W) = {(, num, x, y} | FOLLOW(H) = {(, num, x, y} |

## Problem 5:

≫ In the context of compiler design, explain the significance of estimating the FIRST and FOLLOW sets for a given Context-Free Grammar (CFG)

➢ **Significance of FIRST and FOLLOW sets in Compiler Design:**

- **FIRST set**: Shows which terminals can appear first from a symbol/production; used to decide which production rule to apply.
- **FOLLOW set**: Shows which terminals can appear immediately after a non-terminal; important when ε-productions are present.
- Together, they are used to **construct LL(1) parsing tables**, detect conflicts, and ensure unambiguous predictive parsing.