# CS 3313
# Foundations of Computing:

# Regular Expressions and Regular Languages

http://gw-cs3313-2021.github.io

© slides based on material from
Peter Linz book, Hopcroft, Narahari

# Next…Formal methods to define languages

- Can we provide formal methods to define a language
  - Instead of defining it as accepted by an automaton ?
- Grammars is one option
- For regular languages, we have a simpler formalism:

  Regular Expressions
- Applications of Regular Expressions:
  - Substring search
    - Define keywords in a programming language
  - Unix Commands – use an extended RE notation
  - Web search (Amazon’s module): integrating want ads
  - Lexical analysis – first job of compiler is to break a program into tokens
    - Substrings that together represent a unit

# RE's: Introduction

- *Regular expressions* describe languages by an algebra.
- They describe exactly the regular languages.
- If E is a regular expression, then L(E) is the language it defines.
- We'll describe RE's and their languages recursively.

# Recall Definitions and Notations:

- *Alphabet*: set of symbols, i.e. $\Sigma = \{a, b\}$
- *String*: finite sequence of symbols from $\Sigma$
  - Empty string: denoted $\lambda$ or $\epsilon$
- *Operations on strings*:*Concatenation, Reverse, ..*
- *Length of a string: number of symbols*
- $\Sigma^*$ = set of all strings formed by concatenating zero or more symbols in $\Sigma$
- $\Sigma^+$ = set of all non-empty strings formed by concatenating symbols in $\Sigma$, i.e., $\Sigma^+ = \Sigma^* - \{ \lambda \}$
- A *formal language* L is any subset of $\Sigma^*$

- Convention: we use *w,x,y* to denote strings and *a,b,c* to denote symbols from the alphabet

# Operations on Languages

- RE's use three operations: *union, concatenation,* and *Kleene star*.
- The union of languages is the usual thing, since languages are sets.
- Example: {01,111,10}$\cup${00, 01} = {01,111,10,00}.

# Concatenation

- The *concatenation* of languages L and M is denoted LM.
- It contains every string wx such that w is in L and x is in M.
- Example: {01,111,10}{00, 01} = {0100, 0101, 11100, 11101, 1000, 1001}.

# Kleene Star

- If L is a language, then L*, the *Kleene star* or just “star,” is the set of strings formed by concatenating zero or more strings from L, in any order.
- $L^* = \{\epsilon\} \cup L \cup LL \cup LLL \cup \ldots$
- Example: $\{0,10\}^* = \{\epsilon, 0, 10, 00, 010, 100, 1010, \ldots\}$

# Regular Expressions

- Regular Expressions provide a concise way to describe some languages
- Regular Expressions are defined recursively.  For any alphabet:
  - the empty set, the empty string, or any symbol from the alphabet are *primitive regular expressions*
  - the union (+), concatenation (·), and star closure (*) of regular expressions is also a regular expression
  - any string resulting from a finite number of these operations on primitive regular expressions is also a regular expression

# Languages Associated with Regular Expressions

- A regular expression (RE) **r** denotes a language **L(r)**
- Basis: Assuming that $r_1$ and $r_2$ are regular expressions:
  1. The regular expression $\varnothing$ denotes the empty set
  2. The regular expression $\lambda$ denotes the set $\{ \lambda \}$
  3. For any a in the alphabet, the regular expression **a** denotes the set $\{ a \}$
- Inductive step: if $r_1$ and $r_2$ are regular expressions, denoting languages $L(r_1)$ and $L(r_2)$ respectively, then
  1. $r_1 + r_2$ is a RE denoting the language $L(r_1) \cup L(r_2)$
  2. $r_1 \cdot r_2$ is a RE denoting the language $L(r_1). L(r_2)$
  3. $(r_1)$ is a RE denoting the language $L(r_1)$
  4. $r_1^*$ is a RE denoting the language $(L(r_1))^*$

# Determining the Language Denoted by a Regular Expression

- By combining regular expressions using the given rules, arbitrarily complex expressions can be constructed
- The concatenation symbol (·) is usually omitted
- In applying operations, we observe the following *precedence rules*:
  - star closure precedes concatenation
  - concatenation precedes union
- Parentheses are used to override the normal precedence of operators

# Examples: RE's

- L(**01**) = {01}.
- L(**01+0**) = {01, 0}.
- L(**0**(**1**+**0**)) = {01, 00}.
  - Note order of precedence of operators.
- L(**0***) = {ϵ, 0, 00, 000,… }.

# Sample Regular Expressions and Associated Languages

| Regular Expression | Language |
|---|---|
| (ab)* | { $(ab)^n$, $n \geq 0$ } |
| a + b | { a, b } |
| (a + b)* | { a, b }* (in other words, any string formed with a and b) |
| a(bb)* | { a, abb, abbbb, abbbbbb, … } |
| a*(a + b) | { a, aa, aaa, …, b, ab, aab, … } |
| (aa)*(bb)*b | { b, aab, aaaab, …, bbb, aabbb, … } |
| (0 + 1)*00(0 + 1)* | Binary strings containing at least one pair of consecutive zeros |

*Two regular expressions are equivalent if they denote the same language. Consider, for example, (a + b)* and (a*b*)**

# Practice....Exercises

- 1. Write a regular expression for the language L={w | w contains the substring 101 and w is a string over alphabet {0,1} }
- 2. Write a regular expression for the language L={ w | (a) w contains two consecutive 0’s or (b) w=xy and x contains substring 101 and y ends with two 2’s. }
- 3. Describe the language denoted by regular expression $((\mathbf{0}+\mathbf{10})^*(\epsilon+\mathbf{1}))$

# Practice….Exercises

- 1. Write a regular expression for the language L={ w | (a) w contains two consecutive 0's or (b) w=xy and x contains substring 101 and y ends with two 2's. }
- 2. Describe the set of valid email addresses using a regular expression:

  must contain only characters from alphabet, numbers,@

  must start with a letter

  has exactly one @

  no two consecutive appearances of .

  @ cannot come right after or before . (no @. etc.)

  no digits after @

  at least one . after @
- 3. Describe the language denoted by regular expression $((\mathbf{0}+\mathbf{10})^*(\epsilon+\mathbf{1}))$

# Algebraic Laws for RE's

- Union and concatenation behave sort of like addition and multiplication.
  - + is commutative and associative; concatenation is associative.
  - Concatenation distributes over +.
  - Exception: Concatenation is not commutative.

# Identities and Annihilators

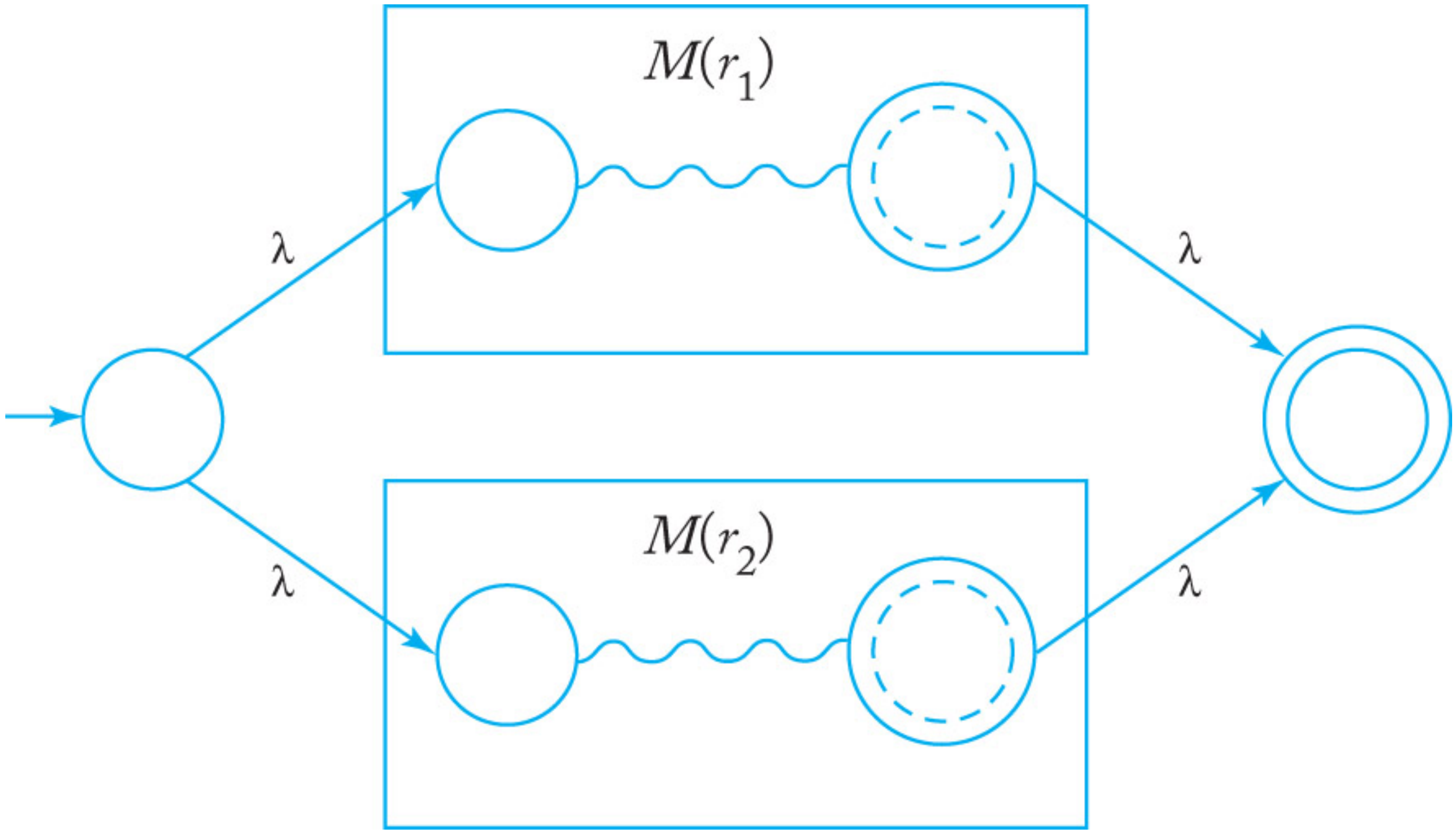
- ∅ is the identity for +.
  - R + ∅ = R.
- ϵ ( λ) is the identity for concatenation.
  - ϵR = Rϵ = R.
- ∅ is the annihilator for concatenation.
  - ∅R = R∅ = ∅.

# Regular Expressions and Regular Languages

- **Theorem**: For any regular expression ***r***, there is a nondeterministic finite automaton M that accepts the language denoted by ***r, i.e., L(M) = L(r)***
- Since nondeterministic and deterministic accepters are equivalent, regular expressions are associated precisely with regular languages
- A constructive proof provides a systematic procedure for constructing a nfa that accepts the language denoted by any regular expression

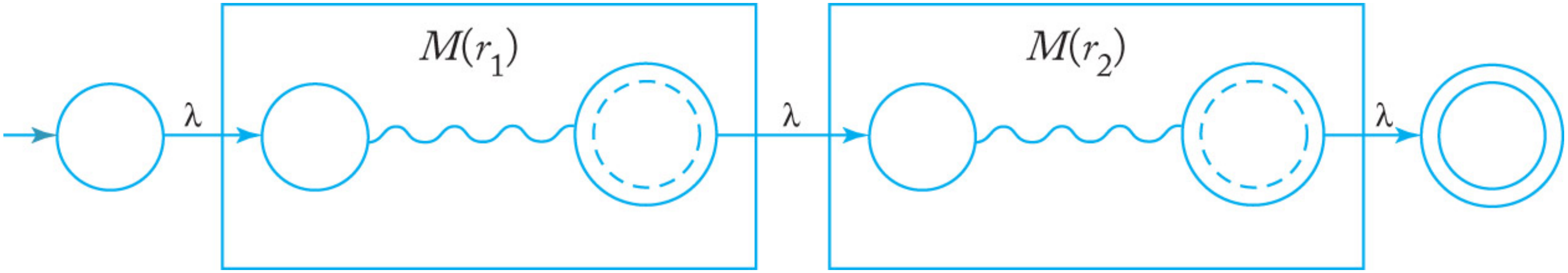
# Recall design of NFAs with E-moves

- What does this NFA accept, in terms of languages accepted by M1 and M2 ?
  - Notation: M1 is M(r1) and M2 is M(r2) ?



# Recall Design of NFAs with E-moves

- What does this NFA accept in terms of languages accepted by L(M1) and L(M2) ?



# Next: Equivalence of Regular Expressions and Finite Automata

- Constructive proof to show that a language is accepted by a DFA M if and only if it is represented by a Regular expression

- Given a RE r, **construct** a finite automaton that accepts L(r)
  - *Construct* = design algorithm that given RE as input will generate a finite automaton M

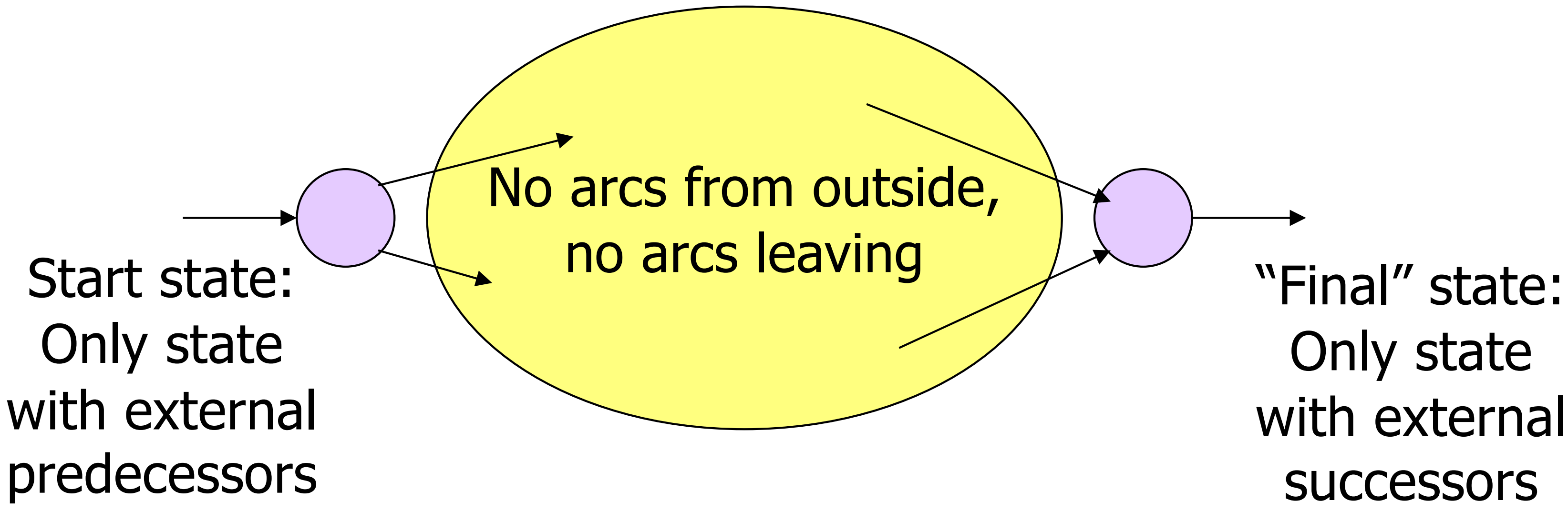
- Given a DFA M, construct a RE to represent L(M)

# Equivalence of RE's and Finite Automata

- We need to show that for every RE, there is a finite automaton that accepts the same language.
    - Pick the most powerful automaton type: the ϵ-NFA.
- And we need to show that for every finite automaton, there is a RE defining its language.
    - Pick the most restrictive type: the DFA.

# Converting a RE to an ϵ-NFA

- Proof is an induction on the number of operators (+, concatenation, *) in the RE.
- We always construct an automaton of a special form:



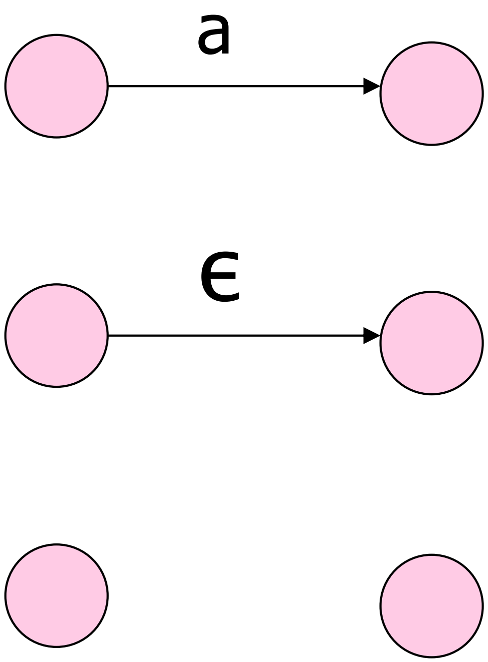
No arcs from outside, no arcs leaving

Start state: Only state with external predecessors

“Final” state: Only state with external successors

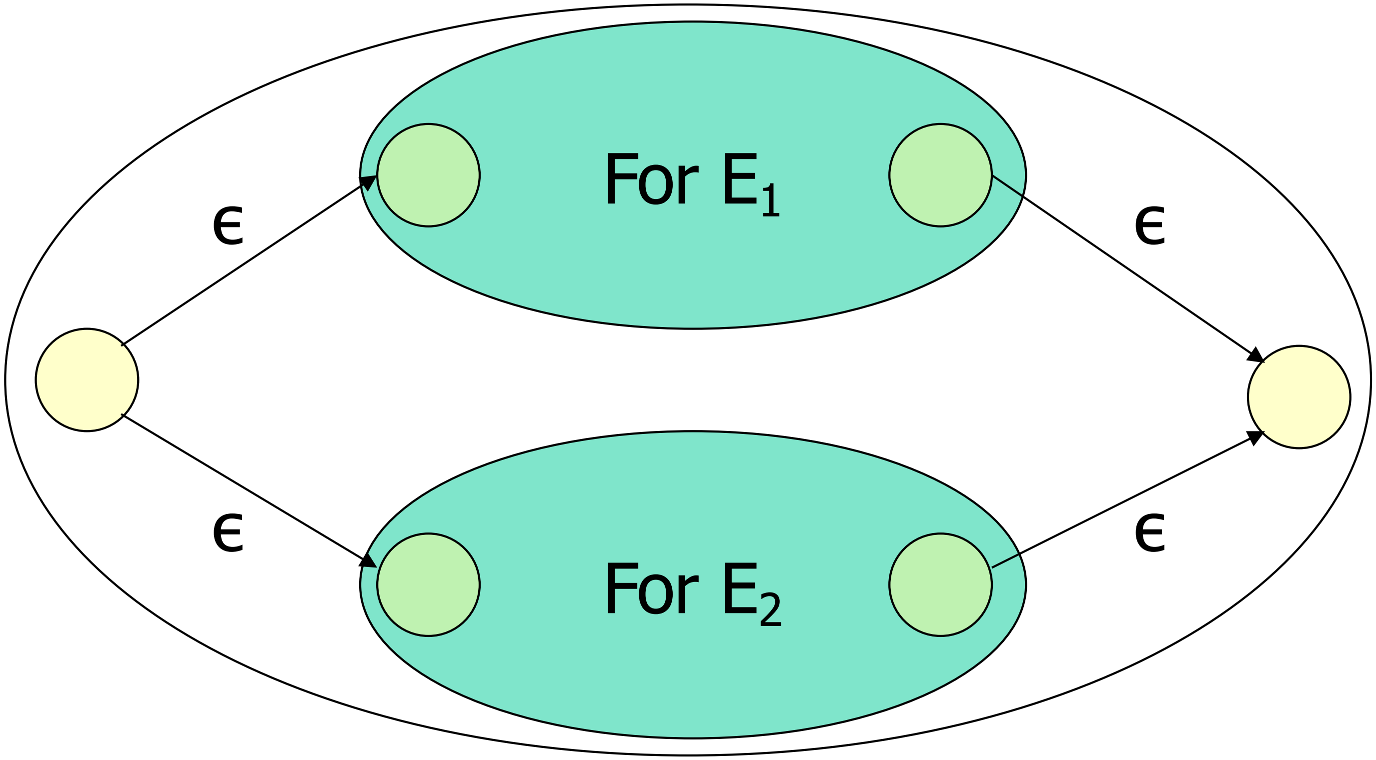
# RE to ϵ-NFA: Basis

- Symbol **a**:
- ϵ (or λ ):
- ∅:



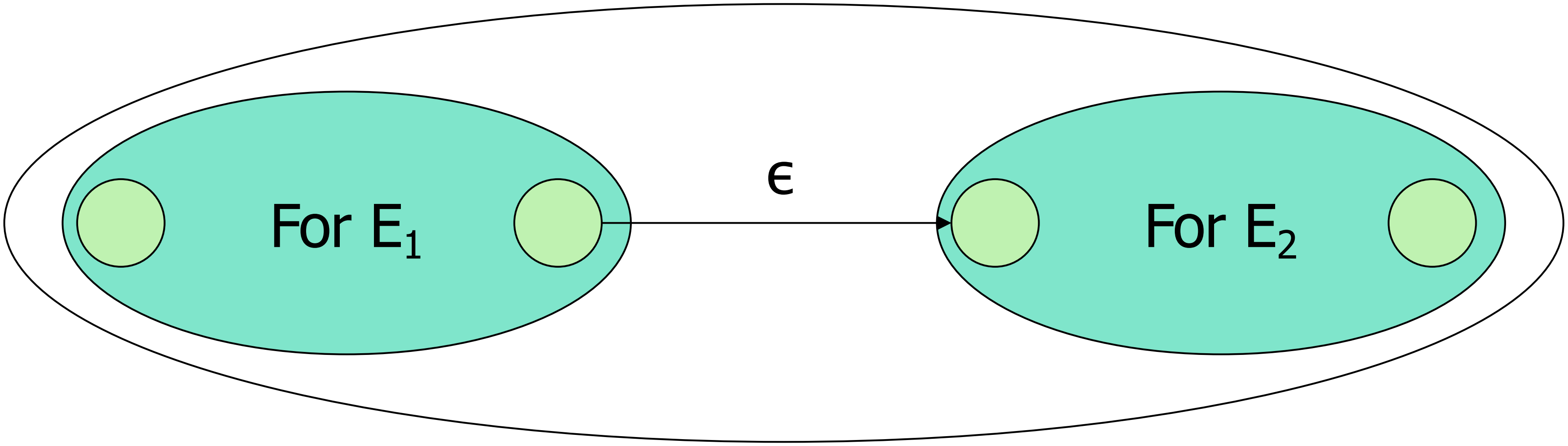
# Inductive Step

# RE to ϵ-NFA: Induction 1: + ( set Union)



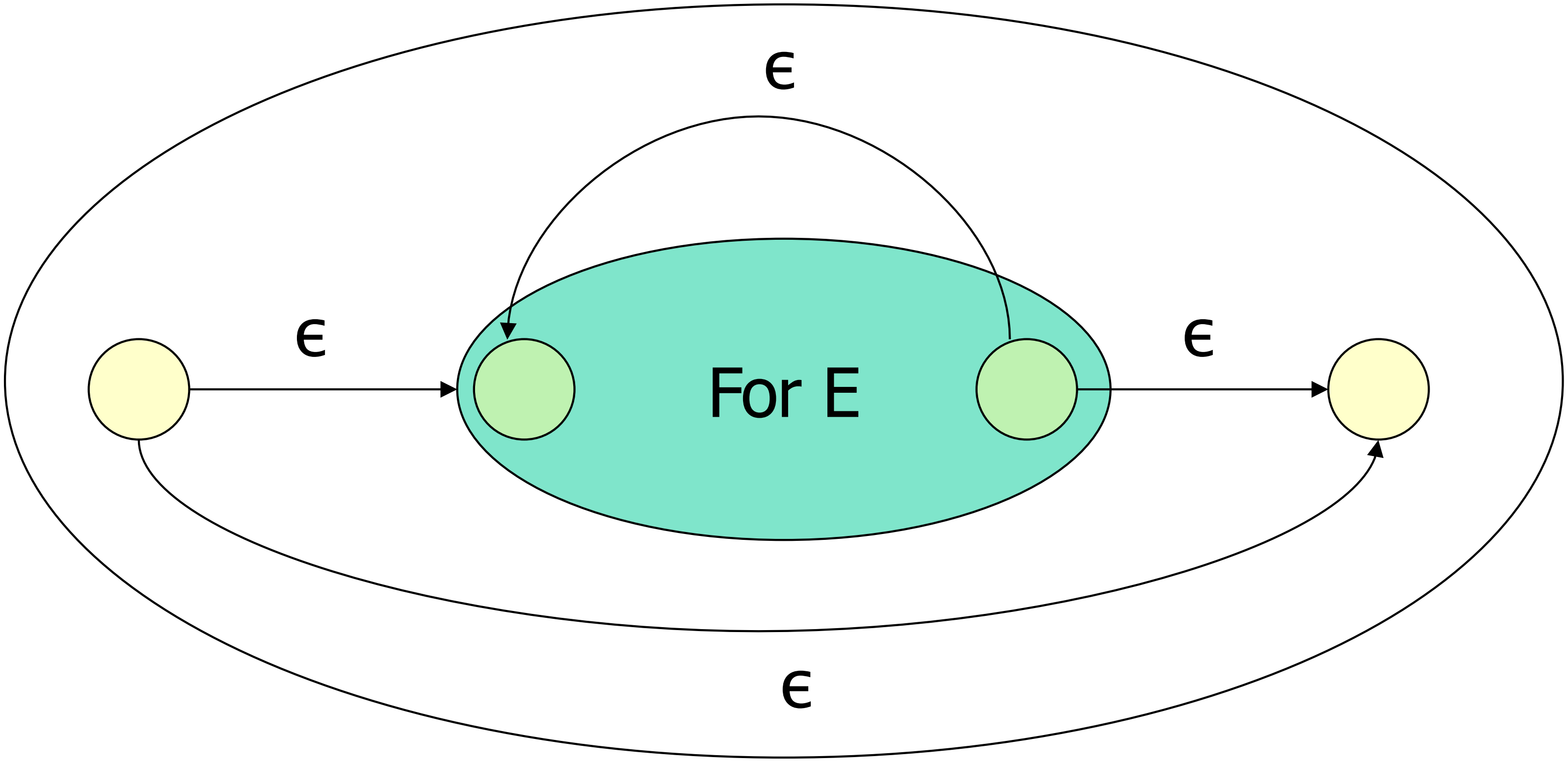
For $E_1 + E_2 = E_1 \cup E_2$

# RE to ϵ-NFA: Induction 2: Concatenation



For $E_1E_2$

# RE to ϵ-NFA: Induction 3: Closure



For E*

# Finite Automata to Regular Expressions

- Given a DFA M, construct a RE to represent L(M)
  - Constructive proof that can be implemented as an algorithm

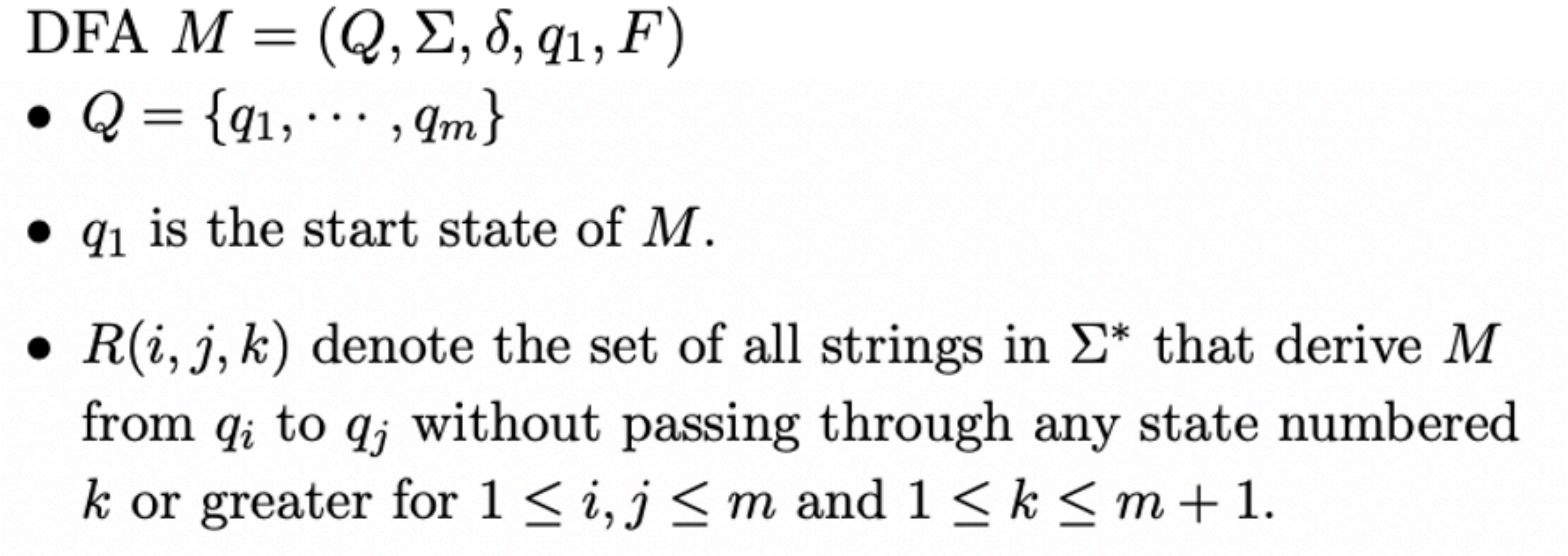
- What we present here is different from the textbook

- **key idea**: formulate the problem as a graph theoretic problem and develop dynamic programming solution
  - *Dynamic programming is a very important and often used technique to solve problems*

# DFA-to-RE

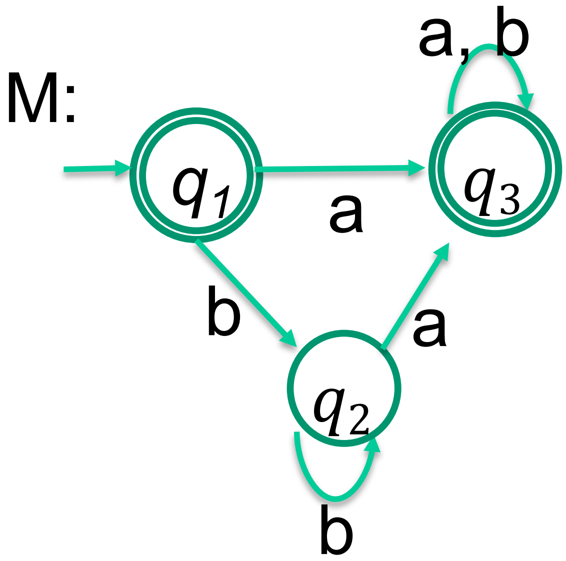
- A strange sort of induction.
- States of the DFA are named 1,2,…,n.
- Induction is on k, the maximum state number we are allowed to traverse along a path.

# Key Ideas for DFA-RE Algorithm



DFA $M = (Q, \Sigma, \delta, q_1, F)$

- $Q = \{q_1, \cdots, q_m\}$
- $q_1$ is the start state of $M$.
- $R(i, j, k)$ denote the set of all strings in $\Sigma^*$ that derive $M$ from $q_i$ to $q_j$ without passing through any state numbered $k$ or greater for $1 \leq i, j \leq m$ and $1 \leq k \leq m + 1$.



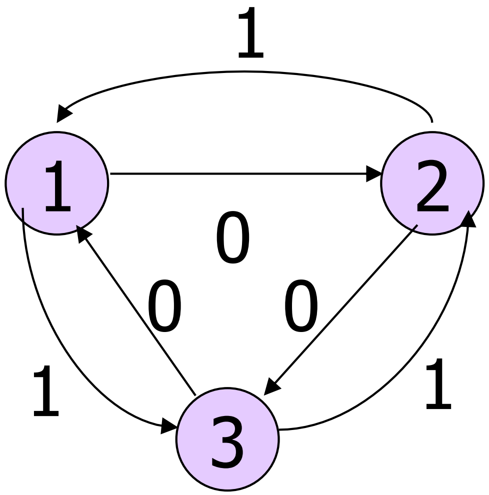
# Key Ideas for DFA-to-RE Algorithm

- DFA M= (Q, **Σ**, δ, $q_1$,F)
- N states: ($q_1$,$q_2$,…$q_n$)
- Start state: $q_1$
- Consider path from state $q_i$ to $q_j$ that pass through states number at most k -- call these k-paths
  - Denote the set of strings that take DFA from $q_i$ to $q_j$ going through states at most k as R(i,j,k)
  - Derive regular expression for this set of strings
  - When i=1 and $q_j$ is a final state, this represents the set of strings accepted by the DFA

# k-Paths

- A k-path is a path through the graph of the DFA that goes through no state numbered higher than k.
- Endpoints are not restricted; they can be any state.
- n-paths are unrestricted.
- RE is the union of RE's for the n-paths from the start state to each final state.

# Example: k-Paths



0-paths from 2 to 3:
RE for labels = **0**.

1-paths from 2 to 3:
RE for labels = **0**+**11**.

2-paths from 2 to 3:
RE for labels =
(**10**)***0**+**1**(**01**)***1**

3-paths from 2 to 3:
RE for labels = ??

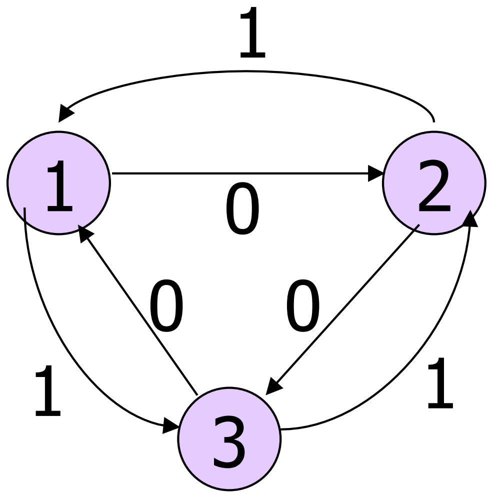
# DFA-to-RE

- Basis: k = 0; only arcs or a node by itself.
- Induction: construct RE's for paths allowed to pass through state k from paths allowed only up to k-1.

# k-Path Induction

- Let $R_{ij}^{k}$ be the regular expression for the set of labels of k-paths from state i to state j.
- **Basis**: k=0. $R_{ij}^{0}$ = sum of labels of arc from i to j.
  - ∅ if no such arc.
  - But add ϵ if i=j.

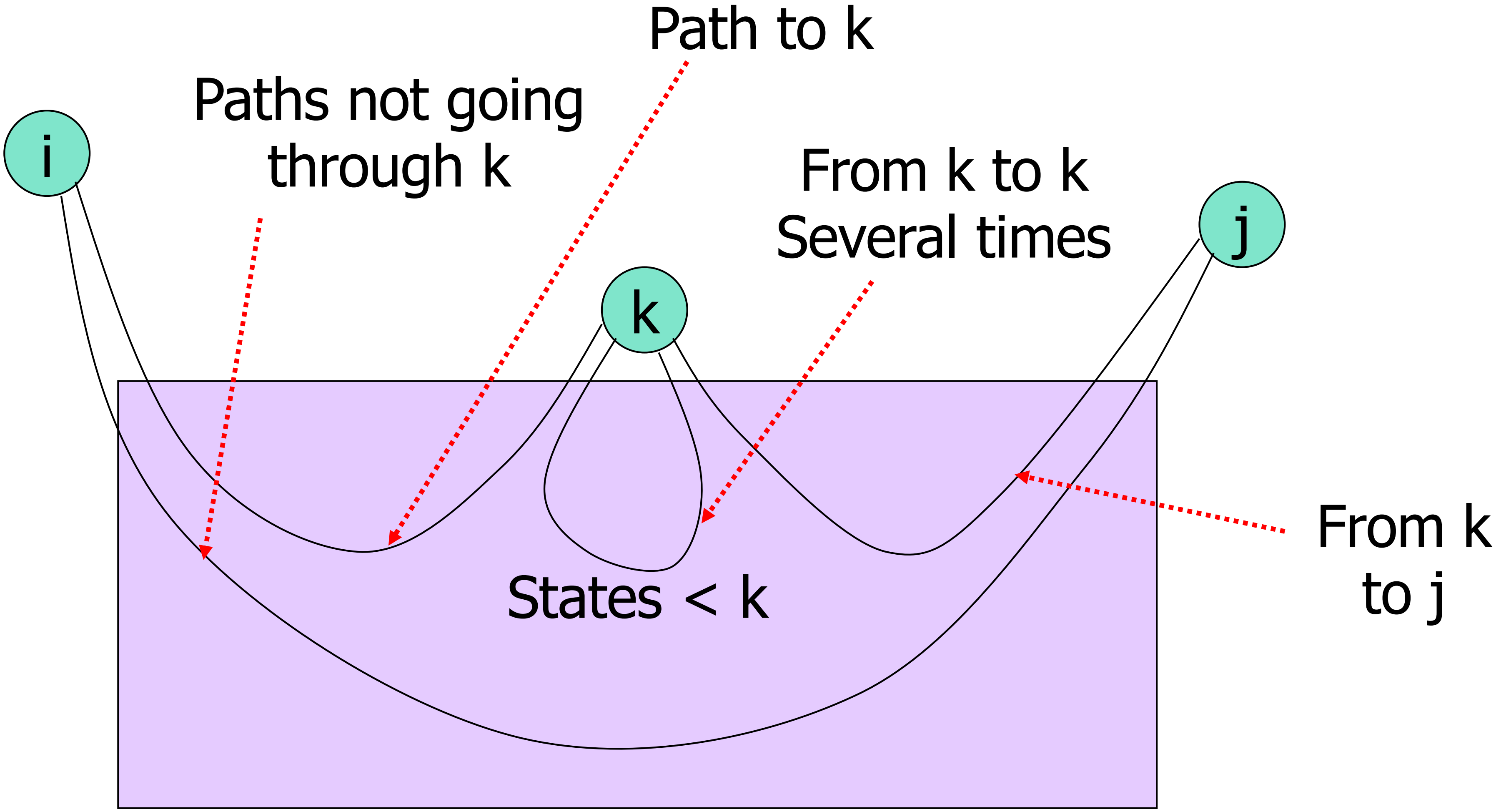
# Example: Basis



- $R_{12}^0 = \mathbf{0}$.
- $R_{11}^0 = \emptyset + \epsilon = \epsilon$.

Notice algebraic law:
∅ plus anything =
that thing.

# Illustration of Induction

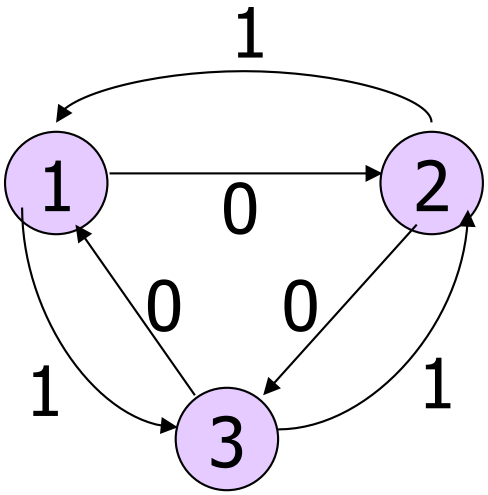


# k-Path Inductive Case

- A k-path from i to j either:
  1. Never goes through state k, or
  2. Goes through k one or more times.

$$R_{ij}^{k} = \underbrace{R_{ij}^{k-1}}_{\text{Doesn't go through k}} + \underbrace{R_{ik}^{k-1}}_{\text{Goes from i to k the first time}}(\underbrace{R_{kk}^{k-1}}_{\text{Zero or more times from k to k}})^{*}\ \underbrace{R_{kj}^{k-1}}_{\text{Then, from k to j going through at most k-1}}.$$

# Example: k=1



- $R_{12}^{1} = R_{12}^{0} + R_{11}^{0} ( R_{11}^{0})^{*} R_{12}^{0}$
  - $0 + \epsilon ( \epsilon)^{*} 0 = 0$
- $R_{22}^{1} = R_{22}^{0} + R_{21}^{0} ( R_{11}^{0})^{*} R_{12}^{0}$
  - $\epsilon + 1(\epsilon)^{*}0 = \epsilon + 10$
- $R_{23}^{1} = R_{23}^{0} + R_{21}^{0} ( R_{11}^{0})^{*} R_{13}^{0}$
  - $0 + 1 (\epsilon )^{*} 1 = (0+11)$

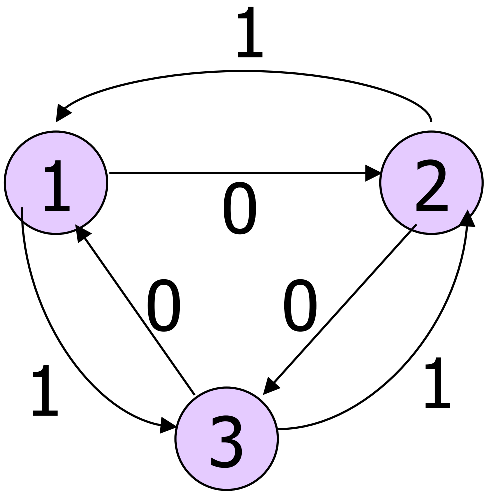
# Algorith:

- For each 1 <= i,j <= n, compute compute the table for R(i,j) for k=0,1,2…n where R(i,j) contains the regular expression for $R_{ij}^{k}$ (or to visualize as a table, R(i,j,k) )

k=0

| | **1** | **2** | **3** |
|---|---|---|---|
| 1 | e | 0 | 1 |
| 2 | 1 | e | 0 |
| 3 | 0 | 1 | e |

# Example: k=2



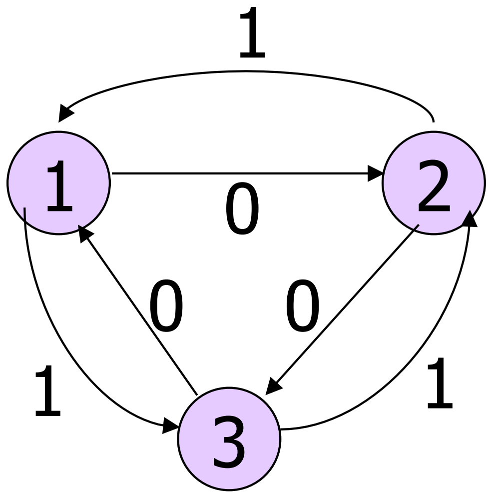
- $R_{12}^{2} = R_{12}^{1} + R_{12}^{1}( R_{22}^{1})R_{22}^{1}$
    - $0 + 0(\epsilon + 10)^{*} (\epsilon + 10) = 0 + 0 ( 10)^{*}$

- Compute table for R(i,j,k) for k=0,1,2…n

k=1

| | **1** | **2** | **3** |
|---|---|---|---|
| 1 | e | 0 | 1 |
| 2 | 1 | e+10 | 0+11 |
| 3 | 0 | 1+00 | e+01 |

# Example: k=2



- $R_{12}^{2} = R_{12}^{1} + R_{12}^{1}(R_{22}^{1})R_{22}^{1}$
    - $0 + 0(\epsilon + 10)^{*} (\epsilon + 10) = 0 + 0 ( 10)^{*}$
- $R_{31}^{2} = R_{31}^{1} + R_{32}^{1}(R_{22}^{1})^{*}R_{21}^{1}$
- $R_{32}^{2} = R_{32}^{1} + R_{32}^{1}(R_{22}^{1})^{*} R_{22}^{*}$
- $R_{23}^{3} = R_{23}^{2} + R_{23}^{2}(R_{33}^{2})^{*}R_{33}^{2}$

- Compute table for R(i,j,k) for k=0,1,2…n

k=2

| | **1** | **2** | **3** |
|---|---|---|---|
| 1 | e+(0(e+10)*1 | 0+(e+10)(10)*(e+10) = 0+0(10)*= 0 (10)* | 1+(0(e+10)*(0+11) |
| 2 | 1+(e+10)(e+10)*1= 1+(10)*1 | (e+10)+ (e+10)(e+10)*(e+10)= (e+10)*= (10)* | (0+11)+ (e+10)(e+10)*(0+11) = (0+11)+(10)*(0+11) |
| 3 | 0 + (1+00)(e+10)*(1)= 0 + (1+00)(10)*(1) | (1+00)+ ((1+00).(e+10)*(e+10)) = (1+00) (10)* | (e+01) + ( (1 +00) (e+10)*(0+11)) |

# Final Step

- The RE with the same language as the DFA is the sum (union) of $R_{1j}^{n}$, where:
  1. n is the number of states; i.e., paths are unconstrained.
  2. 1 ($q_1$) is the start state.
  3. j is one of the final states.

- In terms of an algorithm, $R_{ij}^{k}$ is R(i,j,k) with 1 <= i,j <= n and 0 <= k <= n.
  - Implies $O(n^3)$ algorithm

# Summary

- Each of the three types of automata (DFA, NFA, ϵ-NFA) we discussed, and regular expressions as well, define exactly the same set of languages: the regular languages.

- Question: what kinds of languages & properties are regular languages
  - what kinds of language properties can be defined using Res
  - What kinds of "problems" can be solved using DFAs

- **Next**: Properties of Regular languages
  - Closure properties – what happens when we perform set operations?
  - Decision properties – can we automate checking some properties?
  - Non-regular lang – how do we prove that a language is not regular

# Applications of RE

# UNIX Regular Expressions

- UNIX, from the beginning, used regular expressions in many places, including the “grep” command.
  - Grep = “Global (search for a) Regular Expression and Print.”
- Most UNIX commands use an extended RE notation that still defines only regular languages.

# UNIX RE Notation

- $[a_1a_2\ldots a_n]$ is shorthand for $a_1+a_2+\ldots+a_n$.
- *Ranges* indicated by first-dash-last and brackets.
  - Order is ASCII.
  - Examples: [a-z] = "any lower-case letter," [a-zA-Z] = "any letter."
- Dot = "any character."

# UNIX RE Notation – (2)

- | is used for union instead of +.
- But + has a meaning: “one or more of.”
  - E+ = EE*.
  - Example: [a-z]+ = “one or more lower-case letters.
- ? = “zero or one of.”
  - E? = E + ϵ.
  - Example: [ab]? = “an optional *a* or *b*.”

# Lexical Analysis

- The first thing a compiler does is break a program into *tokens* = substrings that together represent a unit.
  - Examples: identifiers, reserved words like “if,” meaningful single characters like “;” or “+”, multicharacter operators like “<=”.

# Lexical Analysis – (2)

- Using a tool like Lex or Flex, one can write a regular expression for each different kind of token.
- Example: in UNIX notation, identifiers are something like [A-Za-z][A-Za-z0-9]*.
- Each RE has an associated action.
  - Example: return a code for the token found.