CS 3313
Foundations of Computing:

Regular Expressions and Regular Languages

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

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 λ 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 $(L(r_1))^*$

Identities and Annihilators

- ϵ (λ) is the identity for concatenation.
  - $\epsilon R = R\epsilon = R$.
- $\varnothing$ is the annihilator for concatenation.
  - $\varnothing R = R\varnothing = \varnothing$.
- $\varnothing$ is the identity for +.
  - $R + \varnothing = R$.

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

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

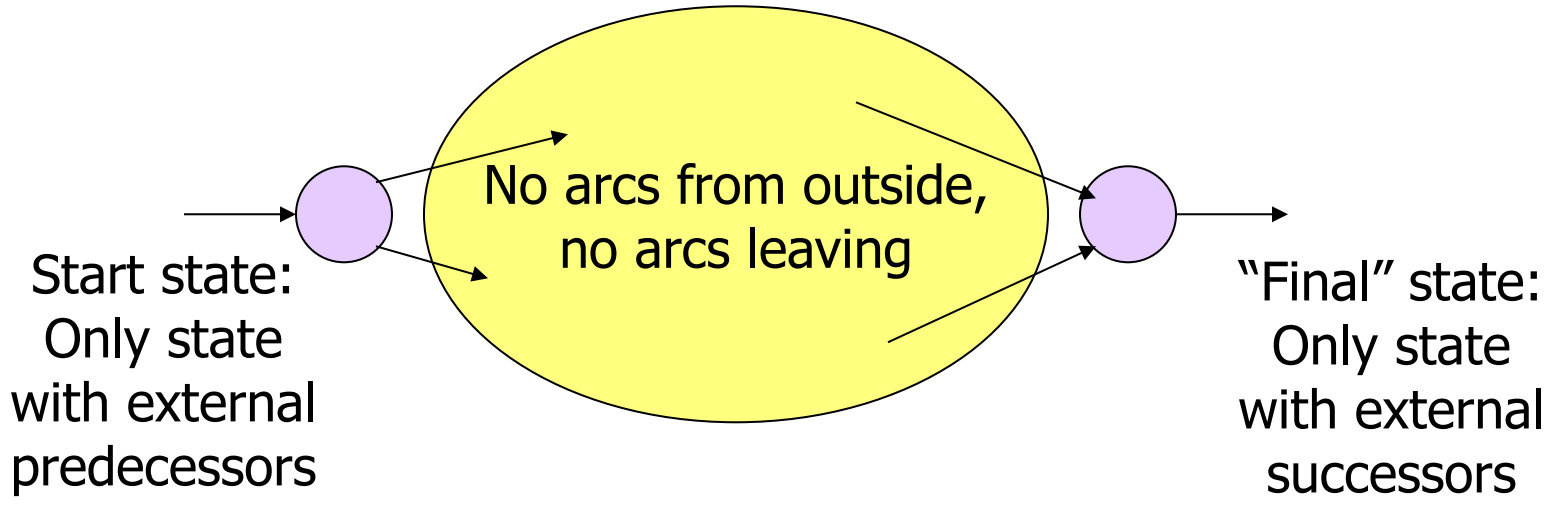
Two regular expressions are equivalent if they denote the same language. Consider, for example, (a + b) and (a*b*)**

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

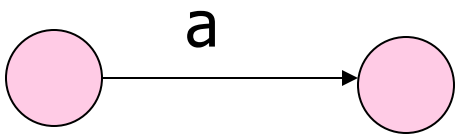
Recap: Converting a RE to an ϵ-NFA

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



RE to ϵ-NFA: Basis

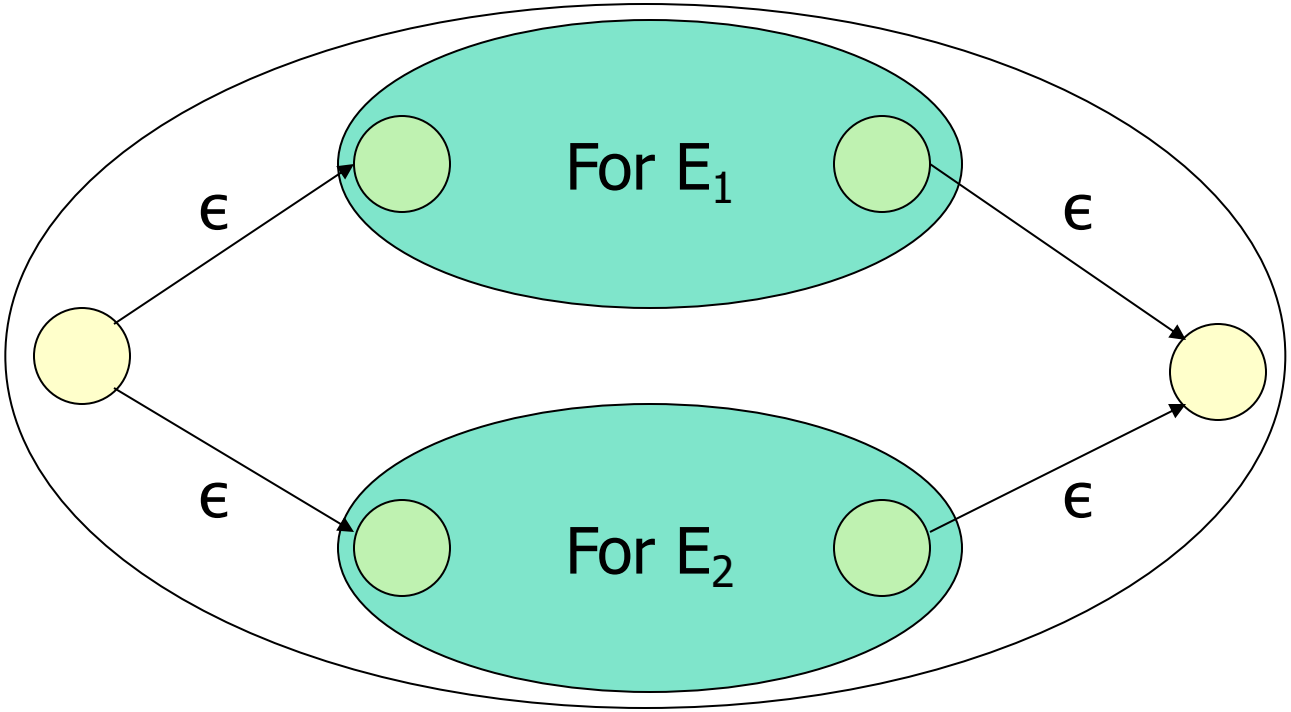
- Symbol **a**:



- ϵ (or λ):

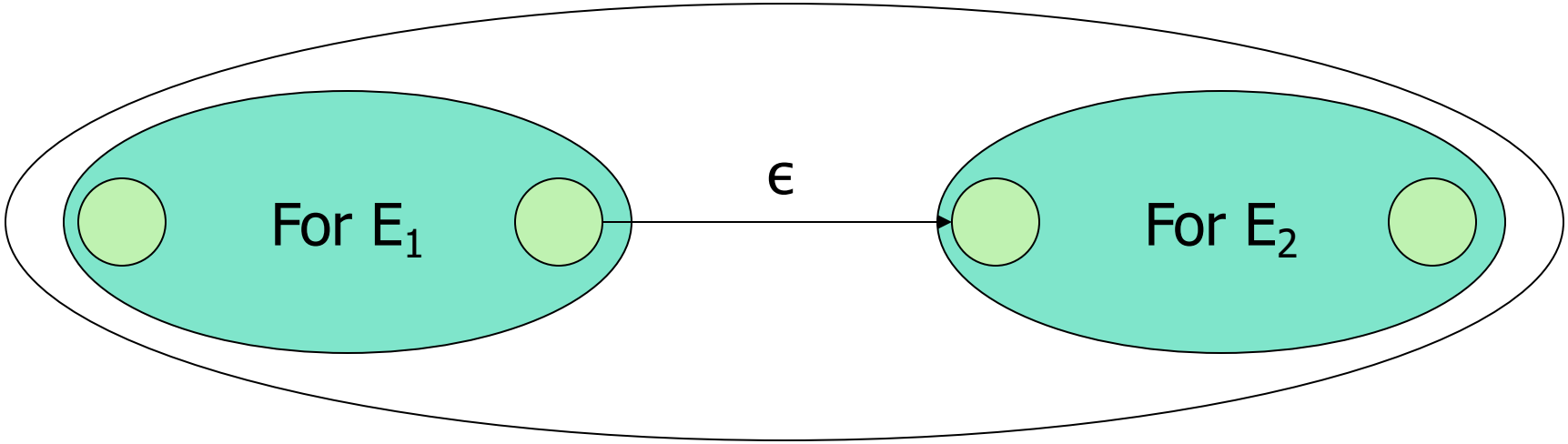
- ∅:

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



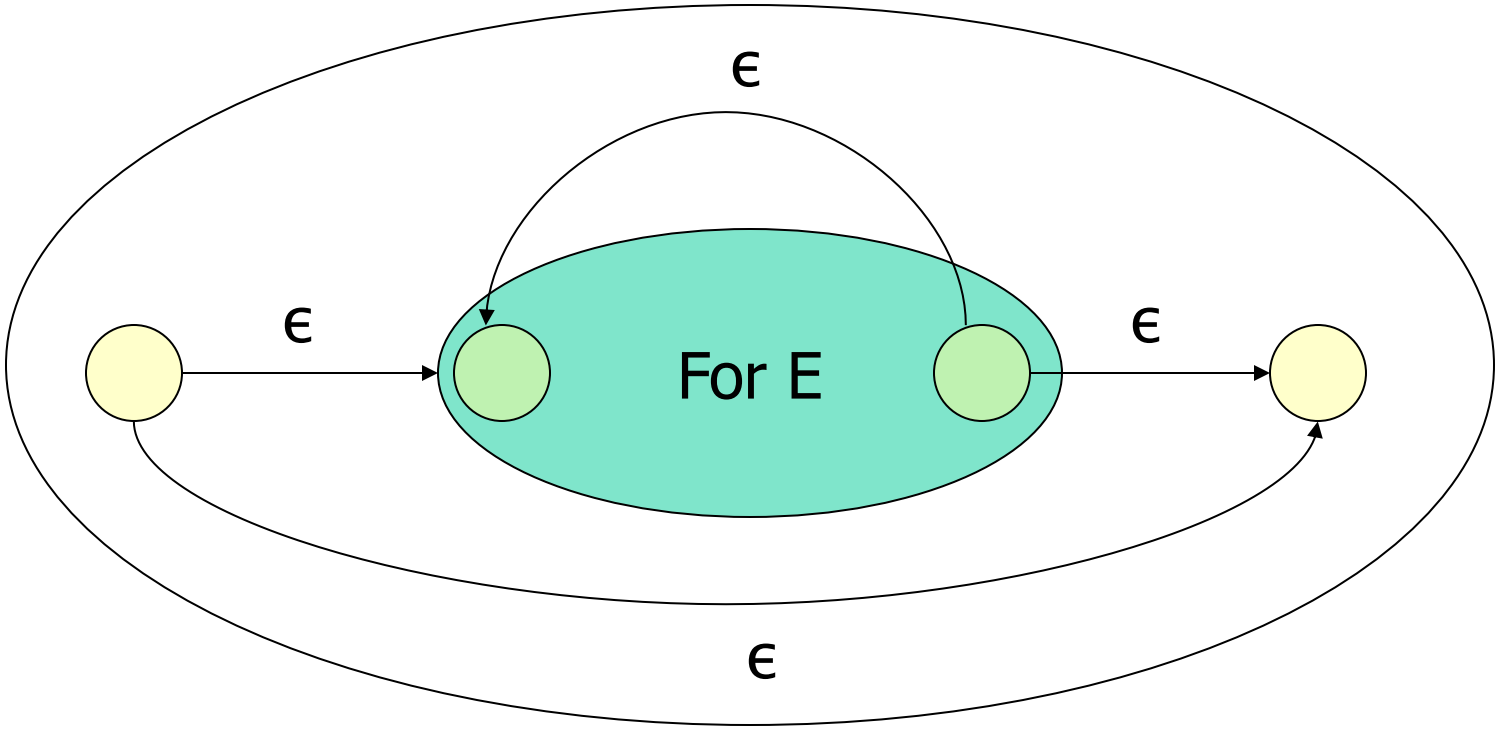
For $E_1 + E_2 = E_1 \cup E_2$

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*
  - *The procedure for generating an RE from a DFA is similar to the Floyd-Warshall Dynamic Programming solution to finding "All pairs shortest paths in a graph".*

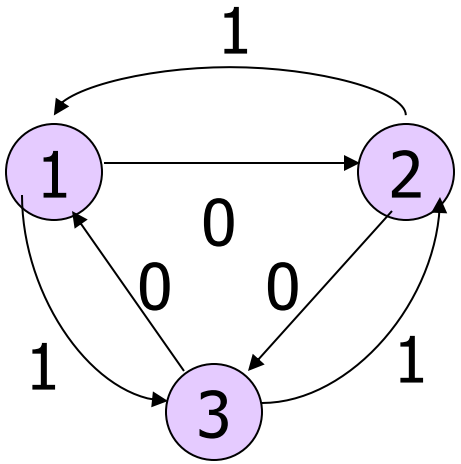
DFA-to-RE

- 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.

k-Paths

- A k-path is a path through the graph of the DFA that goes through no state numbered higher than k; i.e., only states 1~k can be used as intermediate states.
- Endpoints are not restricted; they can be any state.
- n-paths are unrestricted, where |Q| = n.
- RE is the union of RE's for the n-paths from the start state to each final state.

Example: k-Paths



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

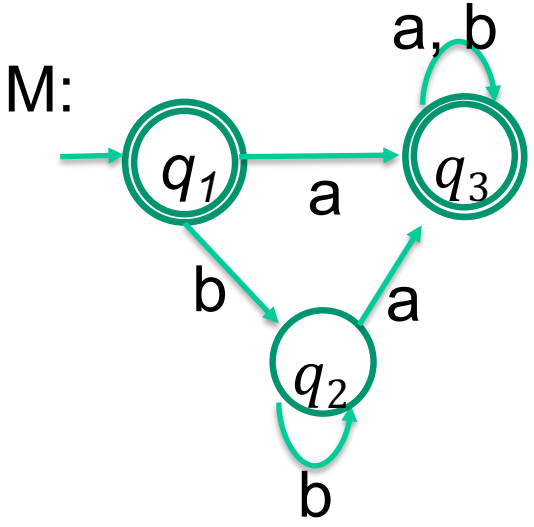
0-paths from 2 to 3:
RE for labels = **0**.

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**
or
(**10**)*(**0**+**11**)

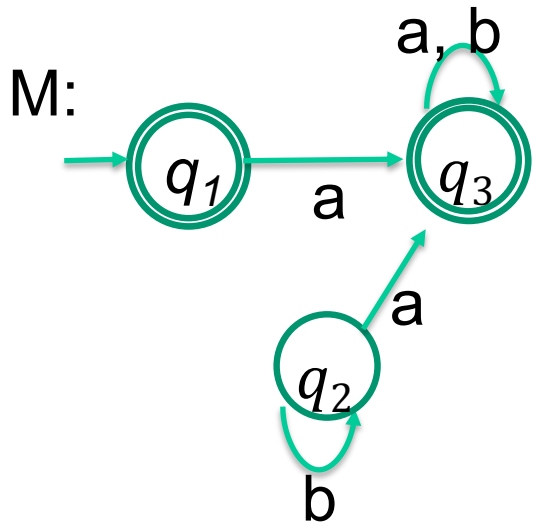
DFA-to-RE

- Basis: k = 0; only **direct** 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.

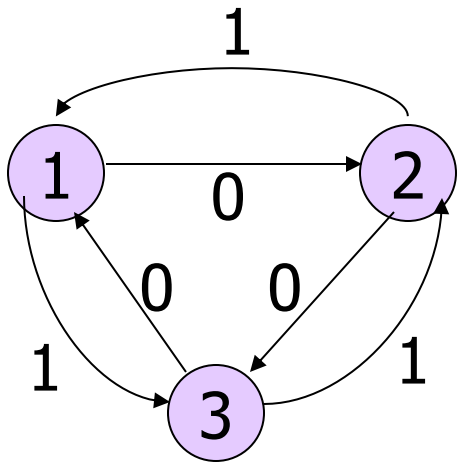


k-Path Induction

- 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

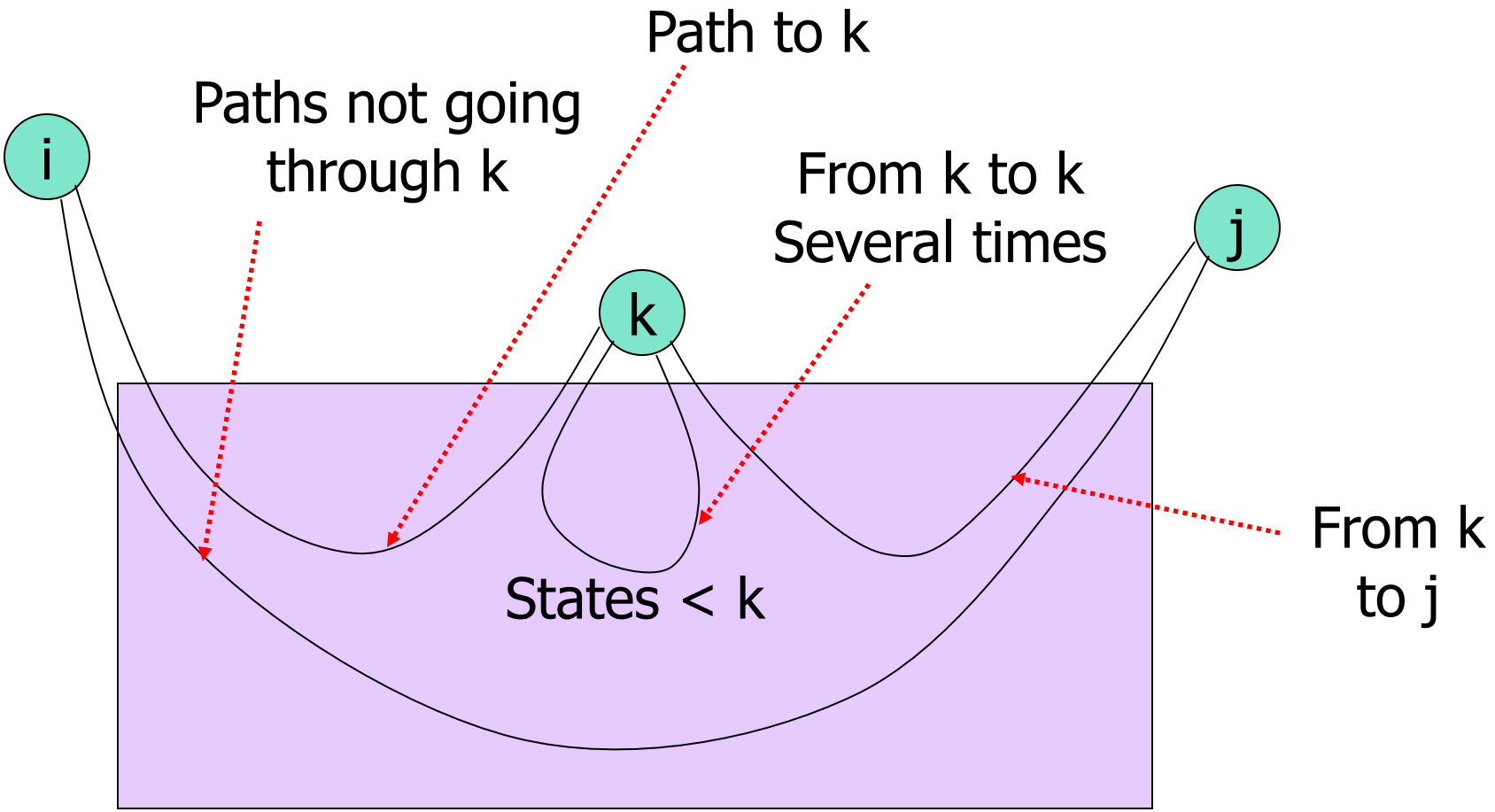


- $R_{12}^0 = \mathbf{0}$.
- $R_{11}^0 = \varnothing + \epsilon = \epsilon$.

Notice algebraic law:
$\varnothing$ plus anything =
that thing.

Illustration of Induction

R_{ij}^{k} =



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.

$$R_{ij}^{k} = R_{ij}^{k-1} + R_{ik}^{k-1}(R_{kk}^{k-1})^{*} R_{kj}^{k-1}.$$

Doesn't go through k

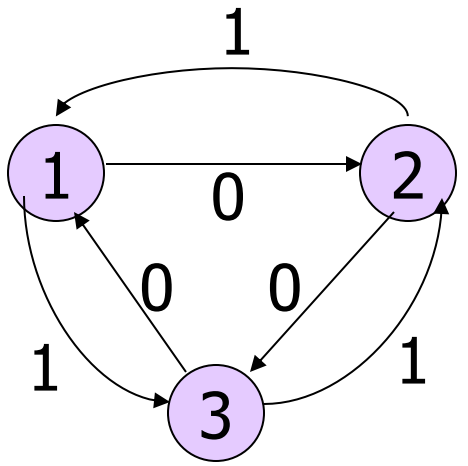
Goes from i to k the first time

Zero or more times from k to k

Then, from k to j going through at most k-1

Example: k=1

k=0	j=1	j=2	j=3
i=1	ϵ	0	1
i=2	1	ϵ	0
i=3	0	1	ϵ

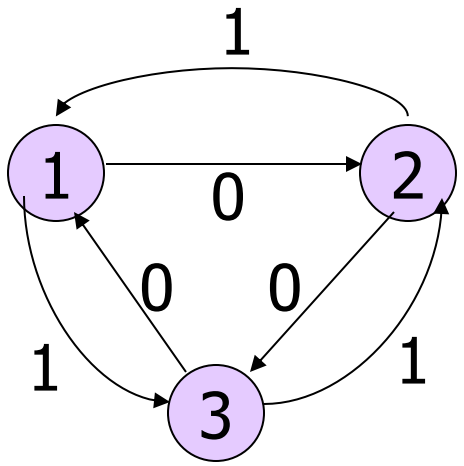


- $R_{12}^{1} = R_{12}^{0} + R_{11}^{0} (R_{11}^{0})^{*} R_{12}^{0}$
  - $0 + \epsilon (\epsilon)^{*} 0 = 0$
- $R_{22}^{1} = R_{22}^{0} + R_{21}^{0} (R_{11}^{0})^{*} R_{12}^{0}$
  - $\epsilon + 1(\epsilon)^{*}0 = \epsilon + 10$
- $R_{23}^{1} = R_{23}^{0} + R_{21}^{0} (R_{11}^{0})^{*} R_{13}^{0}$
  -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

k=1	j=1	j=2	j=3
i=1			
i=2			
i=3			

Example: k=2

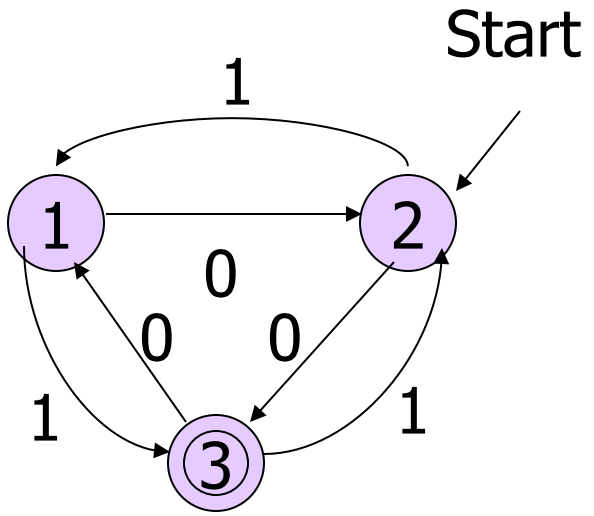
k=1	j=1	j=2	j=3
i=1	…	0	…
i=2	…	$\epsilon + 10$	…
i=3	…	…	…



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

k=2	j=1	j=2	j=3
i=1			
i=2			
i=3			

Example



- $R_{23}^{3} = R_{23}^{2} + R_{23}^{2}(R_{33}^{2})^{*}R_{33}^{2}$
- $= R_{23}^{2} + R_{23}^{2}(R_{33}^{2})^{+} = R_{23}^{2}(R_{33}^{2})^{*}$

- $R_{23}^{2} = (\mathbf{10})^{*}(\mathbf{0+11})$
- $R_{33}^{2} = (\epsilon + \mathbf{01}) + (\mathbf{1+00})(\mathbf{10})^{*}(\mathbf{0+11})$

- $R_{23}^{3} = (\mathbf{10})^{*}(\mathbf{0+11})[(\epsilon + \mathbf{01}) + (\mathbf{1+00})(\mathbf{10})^{*}(\mathbf{0+11})]^{*}$
- $= (\mathbf{10})^{*}(\mathbf{0+11})[\mathbf{01} + (\mathbf{1+00})(\mathbf{10})^{*}(\mathbf{0+11})]^{*}$

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

Answer to DFA→RE Problems (show your work)

k=0	1	2	3
1	e	0	1
2	1	e	0
3	0	1	e

k=1	1	2	3
1	e	0	1
2	1	e+10	0+11
3	0	1+00	e+01

k=2	1	2	3
1	e + 0(10)*1	0 (10)*	1 + 0(10)*(0+11)
2	(10)*1	(10)*	(10)*(0+11)
3	0 + (1+00)(10)*1	(1+00)(10)*	(e+01) + ((1+00)(10)*(0+11))

- R_{23}^{3} = **(10)*(0+11)[01 + (1+00)(10)*(0+11)]***

Note on DFA to RE

- Our constructive proof provides the basis for an algorithm that takes as input a DFA and generates an equivalent RE

- The result may be a complex RE
- We could instead provide a simpler RE directly from examining the DFA/NFA.

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.

HW2 Q&A

- To prove RE1 = RE2
  - Need to prove RE1 subset RE2, and RE2 subset RE1.
- Constructing REs from constraints
  - Consider the characteristics of the constraints.
  - Consider different situations/branches.
- NFA to DFA
  - $\hat{\delta}(q,a) = CL\left(\ \delta\left(\ \hat{\delta}(q,\epsilon),a\ \right)\right) = CL\left(\ \delta(\ CL(q),a\ )\right)$
  - Construct table for q's over all a values, where |q| = 1; then apply union for p's over all all a values, where |p| > 1.