CS 3313
Foundations of Computing:

Regular Expressions and Regular Languages

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

Review…where are we ?

- Review of languages, proof methods
- Model for Deterministic finite state machines – Deterministic Finite Automata (DFA)
- Add non-determinism to a DFA
  - Machine has more than one choice of moves in a current configuration
    - From current state, read input and can go to one of several states
  - Machine can change states (make a move) on empty string
    - Without reading any input, it can change states
  - Non-determinism = examine all parallel paths at once
  - Machine accepts if at least one sequence/choice of moves leads to a final state
- Question: Can DFA simulate NFA ? i.e, are they equivalent in power? …..Proof (simulation) next class

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
    - Did you know – perl scripts for DNA sequence matching was a big thing
      - Knowing perl = well paid intern at BioInfo labs a decade ago
  - 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

3

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.

4

4

Recall Definitions and Notations:

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

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

5

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: $\{01,111,10\} \cup \{00, 01\} = \{01,111,10,00\}$.

6

6

Concatenation and Kleene star

- 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: $\{01,111,10\}.\{00, 01\} = \{0100, 0101, 11100, 11101, 1000, 1001\}$.

◆ 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.

◆ $L^* = \{\lambda\} \cup L \cup LL \cup LLL \cup \ldots$

◆ Example: $\{0,10\}^* = \{\lambda, 0, 10, 00, 010, 100, 1010, \ldots\}$

7

7

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

8

Definition: Languages & 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 $(L(r_1))^*$

Is this form of defining expressions ringing a bell with DB stuff?

9

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
- we have the following *precedence rules*:
  - star closure precedes concatenation
  - concatenation precedes union
- Parentheses are used to override the normal precedence of operators

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

10

Algebraic Laws for RE's

- Union and concatenation behave sort of like addition and multiplication.
  - + is commutative and associative; concatenation is associative.
    - $(r_1 + r_2) = (r_2+r_1)$ $r_1.(r_2.r_3) = (r_1.r_2).r_3$
  - Concatenation distributes over +
    - $r_1.(r_2 + r_3) = (r_1r_2 + r_1r_3)$
  - Exception: Concatenation is not commutative.
- $\emptyset$ is the identity for +.
  - $R + \emptyset = R$
- λ (ϵ) is the identity for concatenation.
  - $\lambda R = R \lambda = R$
- $\emptyset$ is the annihilator for concatenation.
  - $\emptyset R = R\emptyset = \emptyset$

11

11

Examples: RE's

- L(**01**) = {01}.
- L(**01**+**0**) = {01, 0}.
- **L(01. 0)** = {01}.{0}= {010}
- L(**0**(**1**+**0**)) = {01, 00}.
  - Note order of precedence of operators.
- L(**0***) = {λ , 0, 00, 000,… }.
- L((**01**)*) = {λ , 01, 0101, 010101,…..}

12

12

Examples…..

- $(a+b)^* = \{a,b\}^* =$
- $(ab)^* =$
- $a\,(bb)^* =$
- $L = \{a^i b^j\}$ =
- R.Exp for binary strings containing at least one pair of consecutive 0's =
- Binary strings of odd length =
- $L = \{a^i b^j \mid (a)\ i \text{ is even and } j \text{ is odd or } (b)\ i \text{ is odd and } j \text{ is even },\ i,j >= 0\}$

13

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 $(a^*b^*)^*$

14

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 $L=\{ w \mid w \in \{0,1,2\}^*$ *and* (a) w contains two consecutive 0's (and can be more than one such pair) or (b) $w=xy$ and x contains substring 101 and y ends with two 2's. }

3. Describe the language denoted by regular expression

   ((**0**+**10**)*(λ +**1**))

15

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."
  - Check out grep if you have not used it before…. `man grep`
- Most UNIX commands use an extended RE notation that still defines only regular languages.
- Did you know Python supports Reg.Ex. Search ?
  - `import re`

16

16

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."
- | 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 + λ.
  - Example: [ab]? = "an optional *a* or *b*."

17

17

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 "<=".
- Using a tool like Lex or Flex, one can write a regular expression for each different kind of token.
- Example: in UNIX notation, *identifiers* = [A-Za-z][A-Za-z0-9]*.
- Each RE has an associated action.
  - Example: return a code for the token found
    - How? Will get clearer when we cover Reg.Expr. To DFA !

18

18

Summary

- Next – is NFA = DFA ? Proog ? Constructive proof again!
  - This proof will not only provide an algorithm to generate DFA from a RE, but shows that the three formalisms (DFA, NFA, Reg.Expr) are equivalent and define Regular Languages
- Then what ?...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
- **Leads to**: 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

19

19

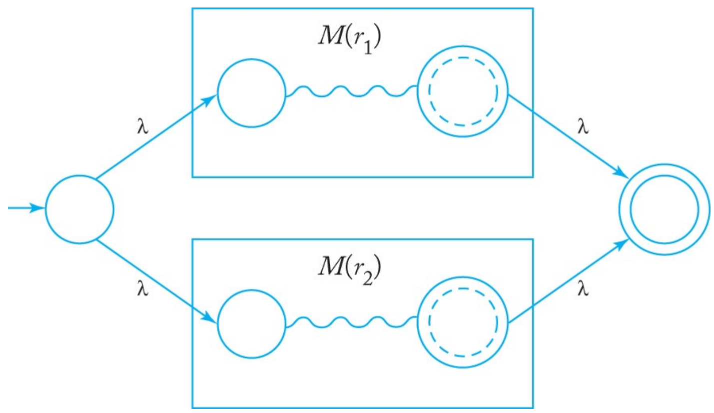
Regular Expressions and Regular Languages

- **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)***
- 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

20

Recall design of NFAs with λ-moves

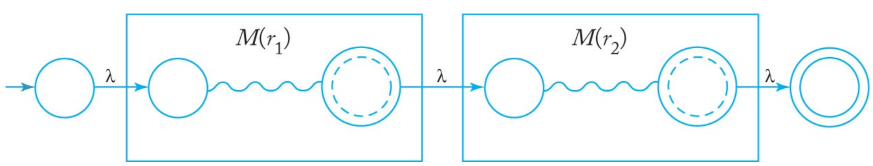
- What does this NFA accept, in terms of languages accepted by M1 and M2 ?
  - Notation: M1 is M(r1) and M2 is M(r2) ?



21

Recall Design of NFAs with E-moves

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



22

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
  - Why is a constructive proof "interesting" ?

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

23

Equivalence of RE's and Finite Automata

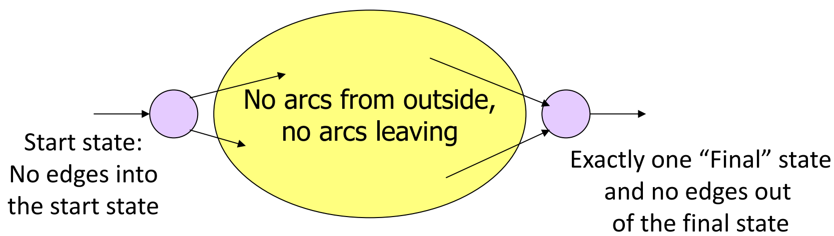
- We need to show that for every RE, there is a finite automaton that accepts the same language.
  - Pick the most powerful automaton type: the ϵ-NFA.
- And we need to show that for every finite automaton, there is a RE defining its language.
  - Pick the most restrictive type: the DFA.

24

24

Converting a RE to an

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

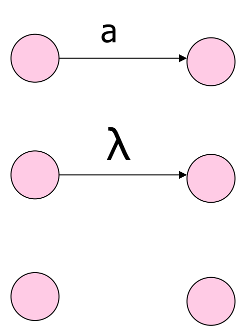


25

25

RE to λ-NFA : Basis

- Symbol **a**:
- ϵ (or λ):
- ∅:



26

26

Inductive Step

- Ind. Hypothesis: Assume statement holds for any expressions E_1, E_2 with n operators
  - There is a NFA M_1 such that $L(M_1) = L(E_1)$ and
    an NFA M_2 such that $L(M_2) = L(E_2)$
  - NFA's have exactly one final state – no transitions out of final, and no into start state
    - M_1 has start state q_1 and final state f_1
    - M_2 has start state q_2 and final state f_2
- *Prove we can construct NFA M that accepts:*
  - $E_1 + E_2$
  - $E_1 . E_2$
  - $(E_1)^*$

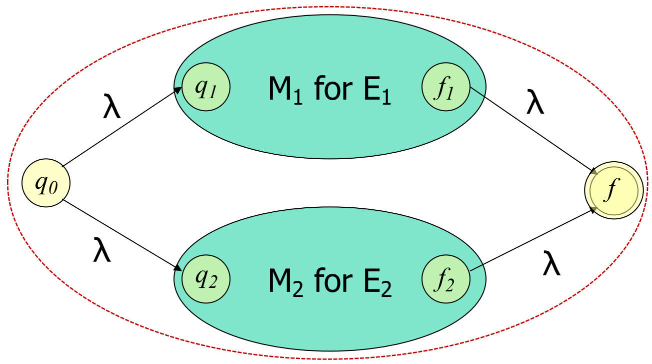
27

Recall Definition of acceptance

- A string w is accepted by a NFA M if and only if $\boldsymbol{\delta}(q_0, w) \in F$
- In terms of NFA M_1 such that $L(M_1) = L(E_1)$ and
  an NFA M_2 such that $L(M_2) = L(E_2)$:
  M_1 : function $\boldsymbol{\delta}_1$, start state q_1 and one final state f_1
  M_2 : function $\boldsymbol{\delta}_2$, start state q_2 and one final state f_2
- x $\in$ $L(E_2)$ if and only if $\boldsymbol{\delta}_1(q_1, x) = \{f_1\}$
- y $\in$ $L(E_2)$ if and only if $\boldsymbol{\delta}_2(q_2, y) = \{f_2\}$

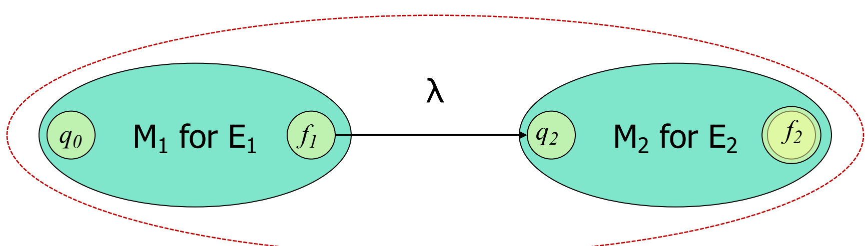
28

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



For $E_1 + E_2 = E_1 \cup E_2$

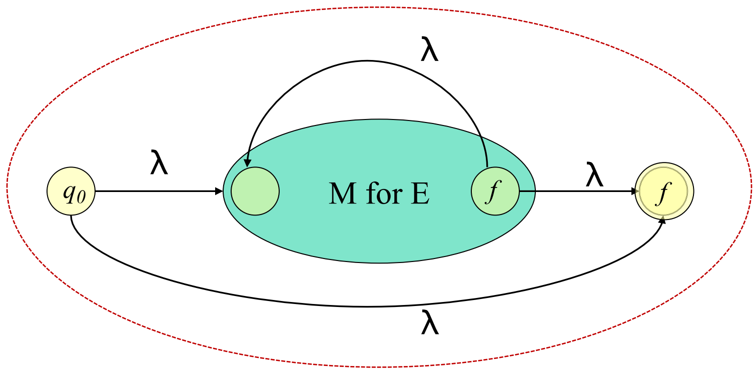
RE to ϵ-NFA: Induction 2: Concatenation



$q_0 = q_1$

For E_1E_2

RE to ϵ-NFA: Induction 3: Closure



For E*

31

31

RE to NFA: Example

- The constructive proof results in a procedure to generate an NFA from a regular expression
  - Not a very efficient NFA…..but other algorithms can be applied to reduce the number of states
- Example: (ab)* + (ba)*

32

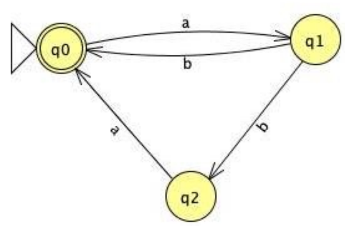
DFA/NFA to Regular Expression

- Find the labels of the paths from start state to each final state
  - Concatenate labels on the path
  - If we have two choices of paths with labels w_1 and w_2 then "or" the paths to get w_1+w_2
  - If there is a cycle, with path labelled w, then w^*

33

Example: Automaton to Reg. Expression

- Find expression for all paths from start state to a final state
- Example: paths from q_0 to q_0
  - q_0 to q_1 to q_0 = (ab)
  - q_0 to q_1 to q_2 to q_0 = (aba)
  - But: can repeat cycle from q_0 to q_0
  - q_0 to itself on empty string λ
- Therefore: *Reg. Exp.* = *(ab + aba)**



34

Algorithm to generate Regular Expression from Finite Automata

- 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*

35

DFA-to-RE Algorithm

- A strange sort of induction.
- States of the DFA are named $1,2,\ldots,n$.
- Induction is on k, the maximum state number we are allowed to traverse along a path.
- Derive set of strings (reg. exp.) that go from state q_i to q_j without passing through any state numbered k or greater
- Similar to the Floyd Warshall algorithm to computer all pairs shortest paths in a graph
  - Did you see this before ?.....VERY useful (and often used) algorithm!

36

36

Key Ideas for DFA-to-RE Algorithm

- DFA M= $(Q, \Sigma, \delta, q_1, F)$
- N states: $(q_1, q_2, \ldots q_n)$
- Start state: q_1
- Consider path from state q_i to q_j that pass through states numbered 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

37

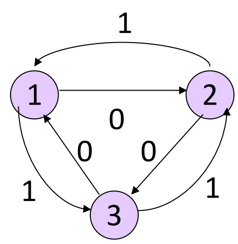
k-Paths

- 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 – can go through any state
- RE is the union of RE's for the *n-paths* from the start state to each final state.

38

38

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 = ??

39

39

DFA-to-RE

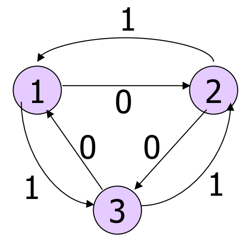
- Basis: k = 0; only arcs or a node by itself.
- Induction: construct RE's for paths allowed to pass through state k from paths allowed only up to k-1.

40

40

DFA to RE Constructive Proof: 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*.
  - $\varnothing$ if no such arc.
  - But add λ if $i=j$.
    - R_{12}^{0} = **0**.
    - $R_{11}^{0} = \varnothing + \lambda = \lambda$.



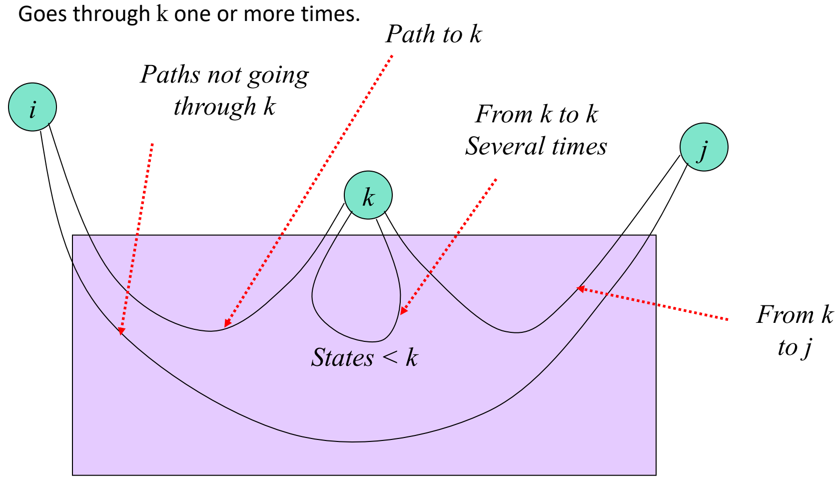
Notice algebraic law:
Ø plus/union anything =
that thing.

41

41

DFA to RE Constructive Proof: k-Path Induction

- Let R_{ij}^{k} (r.e. for *k-paths* from state i to state j).
- 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.



42

42

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 through at most k-1

Zero or more times from k to k

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

43

Algorithm:

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

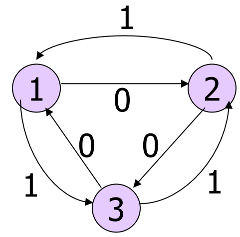
k=0

	1	2	3
1	λ	0	1
2	1	λ	0
3	0	1	λ

k=1

	1	2	3
1	λ	0	1
2	1	λ+10	0+11
3	0	1+00	λ+01

Example: k=1

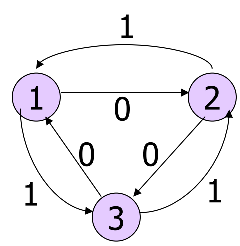


- $R_{12}^1 = R_{12}^0 + R_{11}^0 \, (R_{11}^0)^* \, R_{12}^0$
  - $0 + \lambda \, (\lambda)^* \, 0 \; = 0$
- $R_{22}^1 = R_{22}^0 + R_{21}^0 \, (R_{11}^0)^* \, R_{12}^0$
  - $\lambda + 1(\lambda)^* 0 = \lambda \; + 10$
- $R_{23}^1 = R_{23}^0 + R_{21}^0 \, (R_{11}^0)^* \, R_{13}^0$
  - $0 \; + \; 1 \;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

45

45

Example: k=2



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

	1	2	3
1	λ +(0(λ+10)*1	0+(λ+10)(10)*(λ+10) = 0+0(10)*= 0 (10)*	1+(0(λ+10)*(0+11)
2	1+(λ+10)(λ+10)*1= 1+(10)*1	(λ+10)+ (λ+10)(λ+10)*(λ+10)= (λ+10)*= (10)*	(0+11)+ (λ+10)(λ+10)*(0+11) = (0+11)+(10)*(0+11)
3	0 + (1+00)(λ+10)*(1)= 0 + (1+00)(10)*(1)	(1+00)+ ((1+00).(λ+10)*(λ+10)) = (1+00) (10)*	(λ+01) + ((1 +00) (λ+10)*(0+11))

46

DFA to RE: Algorithm - 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

47

47

Summary

- Next – is NFA = DFA ? Proof ? Constructive proof again!
  - This proof will not only provide an algorithm to generate DFA from a RE, but shows that the three formalisms (DFA, NFA, Reg.Expr) are equivalent and define Regular Languages
- Then what ?...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
- **Leads to**: 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

48

48