Regular Expressions

Definitions

Equivalence to Finite Automata

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.

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

RE's: Definition

- Basis 1: If *a* is any symbol, then **a** is a RE, and L(**a**) = {a}.
  - Note: {a} is the language containing one string, and that string is of length 1.
- Basis 2: ϵ is a RE, and L(ϵ) = {ϵ}.
- Basis 3: ∅ is a RE, and L(∅) = ∅.

RE's: Definition – (2)

- Induction 1: If E_1 and E_2 are regular expressions, then E_1+E_2 is a regular expression, and $L(E_1+E_2) = L(E_1) \cup L(E_2)$.
- Induction 2: If E_1 and E_2 are regular expressions, then E_1E_2 is a regular expression, and $L(E_1E_2) = L(E_1)L(E_2)$.
- Induction 3: If E is a RE, then E* is a RE, and $L(E^*) = (L(E))^*$.

Precedence of Operators

- Parentheses may be used wherever needed to influence the grouping of operators.
- Order of precedence is * (highest), then concatenation, then + (lowest).

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,... }.
- L((**0**+**10**)*(ϵ+**1**)) = all strings of 0's and 1's without two consecutive 1's.

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 (next slide).

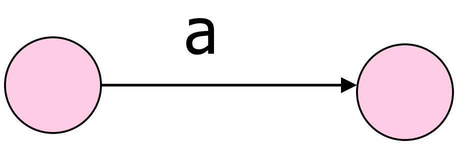
Form of ϵ-NFA's Constructed

Start state:
Only state
with external
predecessors

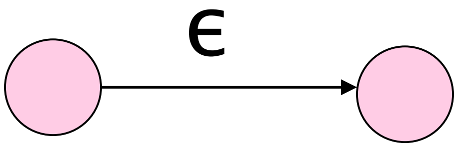
"Final" state:
Only state
with external
successors

RE to ϵ-NFA: Basis

- Symbol **a**:

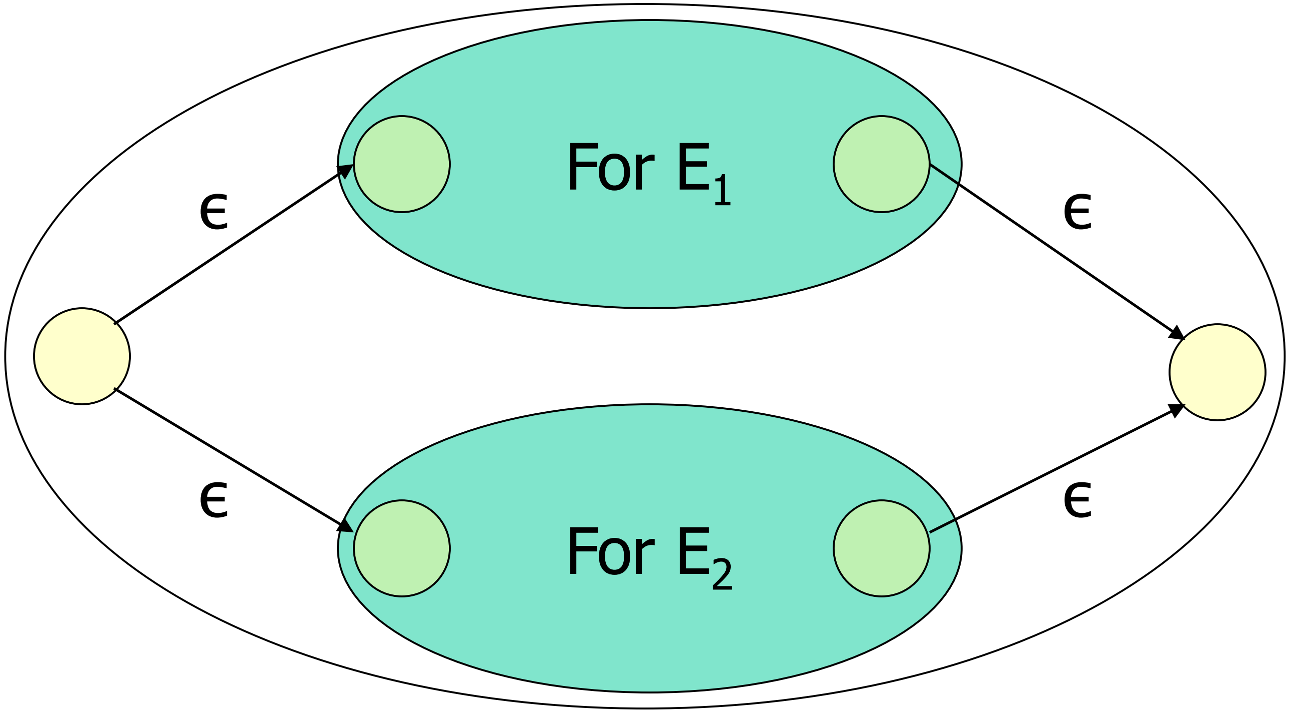


- ϵ:



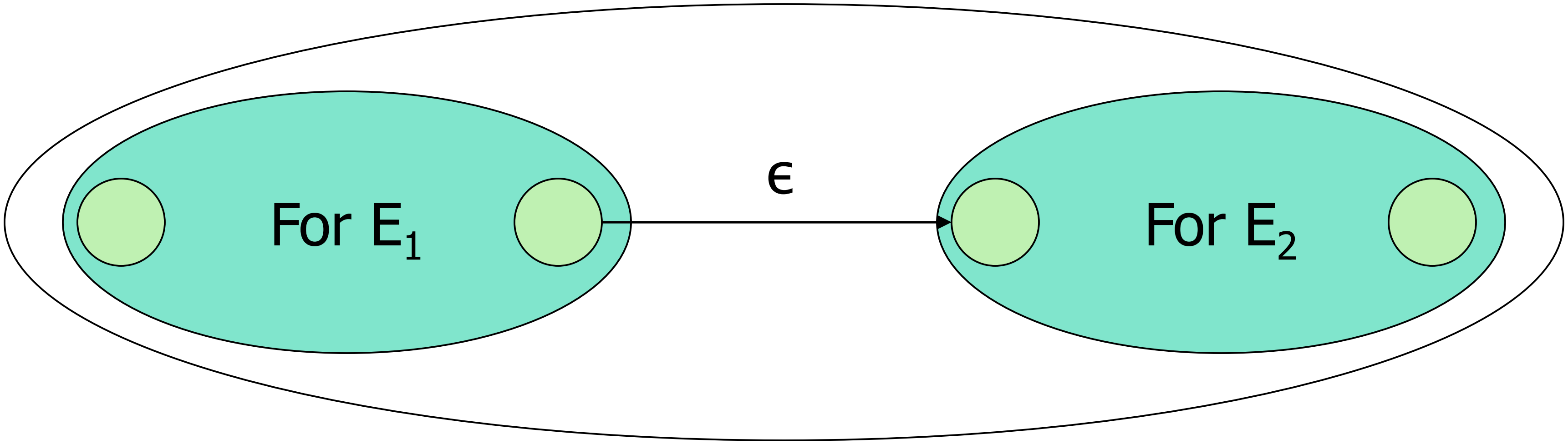
- ∅:

RE to ϵ-NFA: Induction 1 – Union



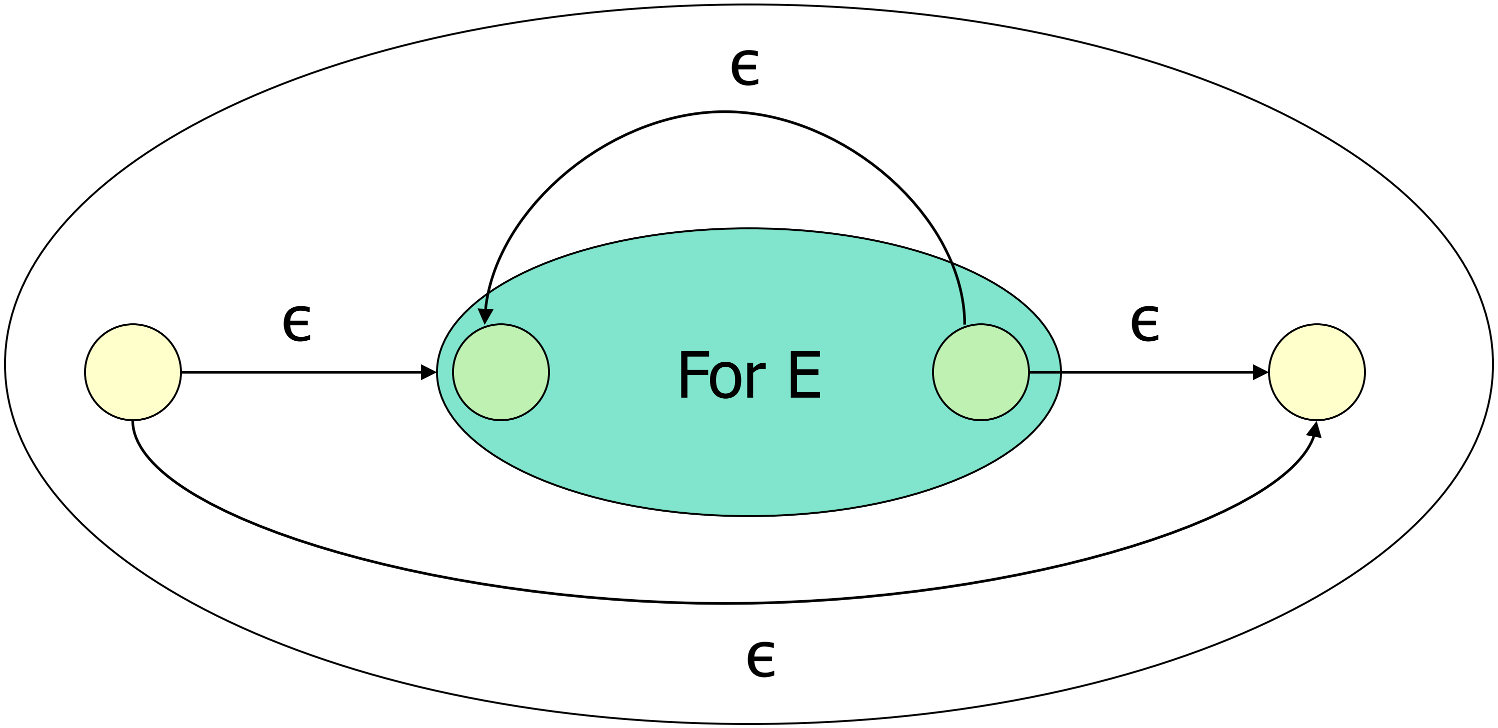
For $E_1 \cup E_2$

RE to ϵ-NFA: Induction 2 – Concatenation



For E_1E_2

RE to ϵ-NFA: Induction 3 – Closure



For E*

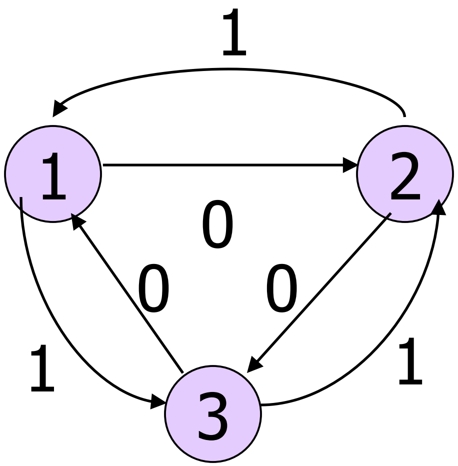
DFA-to-RE

- 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.
- 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 = ??

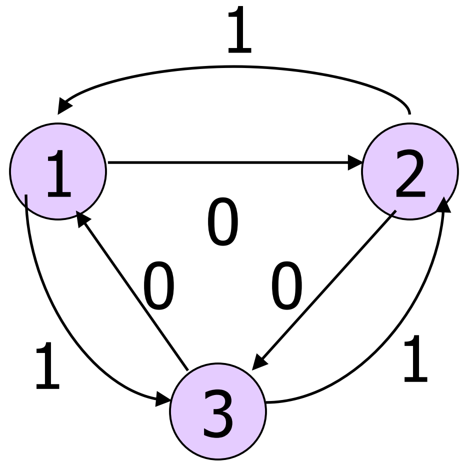
DFA-to-RE

- 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.
  - $\emptyset$ 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: $\emptyset$ plus anything = that thing.

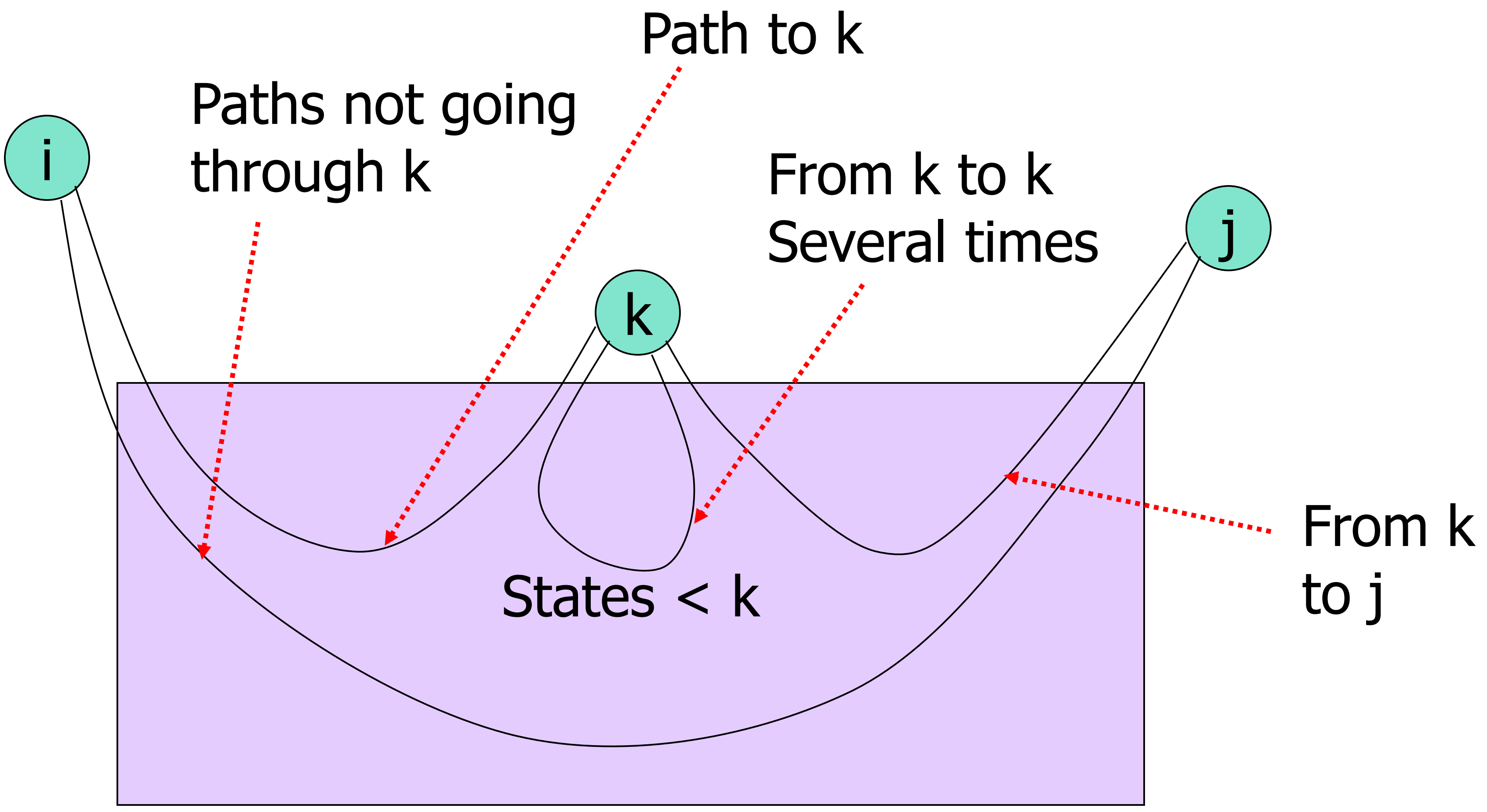
k-Path Inductive Case

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

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

- R_{ij}^{k-1}: Doesn't go through k
- R_{ik}^{k-1}: Goes from i to k the first time
- $(R_{kk}^{k-1})^{*}$: Zero or more times from k to k
- R_{kj}^{k-1}: Then, from k to j

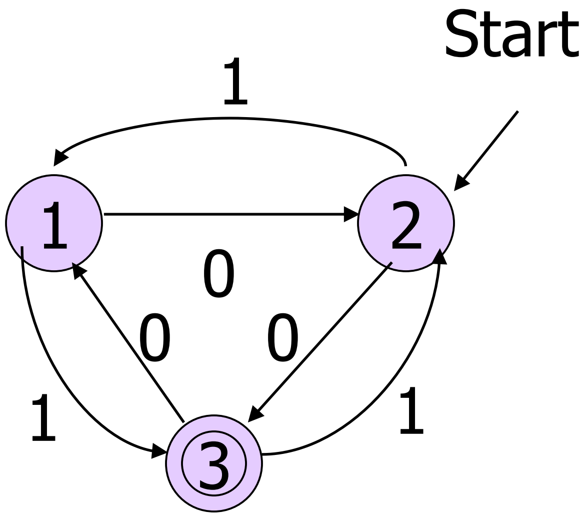
Illustration of Induction



Final Step

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

Example



- $R_{23}{}^3 = R_{23}{}^2 + R_{23}{}^2(R_{33}{}^2)^*R_{33}{}^2 = R_{23}{}^2(R_{33}{}^2)^*$
- $R_{23}{}^2 = (\mathbf{10})^*\mathbf{0}+\mathbf{1}(\mathbf{01})^*\mathbf{1}$
- $R_{33}{}^2 = \epsilon + \mathbf{0}(\mathbf{01})^*(\mathbf{1}+\mathbf{00}) + \mathbf{1}(\mathbf{10})^*(\mathbf{0}+\mathbf{11})$
- $R_{23}{}^3 = [(\mathbf{10})^*\mathbf{0}+\mathbf{1}(\mathbf{01})^*\mathbf{1}]\ [\epsilon + (\mathbf{0}(\mathbf{01})^*(\mathbf{1}+\mathbf{00}) + \mathbf{1}(\mathbf{10})^*(\mathbf{0}+\mathbf{11}))]^*$

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.

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 is the identity for concatenation.
  - ϵR = Rϵ = R.
- ∅ is the annihilator for concatenation.
  - ∅R = R∅ = ∅.

Applications of Regular Expressions

Unix RE's

Text Processing

Lexical Analysis

Some Applications

- RE's appear in many systems, often private software that needs a simple language to describe sequences of events.
- We'll use Junglee as an example, then talk about text processing and lexical analysis.

Junglee

- Started in the mid-90's by three of my students, Ashish Gupta, Anand Rajaraman, and Venky Harinarayan.
- Goal was to integrate information from Web pages.
- Bought by Amazon when Yahoo! hired them to build a comparison shopper for books.

Integrating Want Ads

- Junglee's first contract was to integrate on-line want ads into a queryable table.
- Each company organized its employment pages differently.
  - Worse: the organization typically changed weekly.

Junglee's Solution

- They developed a regular-expression language for navigating within a page and among pages.
- Input symbols were:
  - Letters, for forming words like "salary".
  - HTML tags, for following structure of page.
  - Links, to jump between pages.

Junglee's Solution – (2)

- Engineers could then write RE's to describe how to find key information at a Web site.
  - E.g., position title, salary, requirements,…
- Because they had a little language, they could incorporate new sites quickly, and they could modify their strategy when the site changed.

RE-Based Software Architecture

- Junglee used a common form of architecture:
  - Use RE's plus actions (arbitrary code) as your input language.
  - Compile into a DFA or simulated NFA.
  - Each accepting state is associated with an action, which is executed when that state is entered.

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."

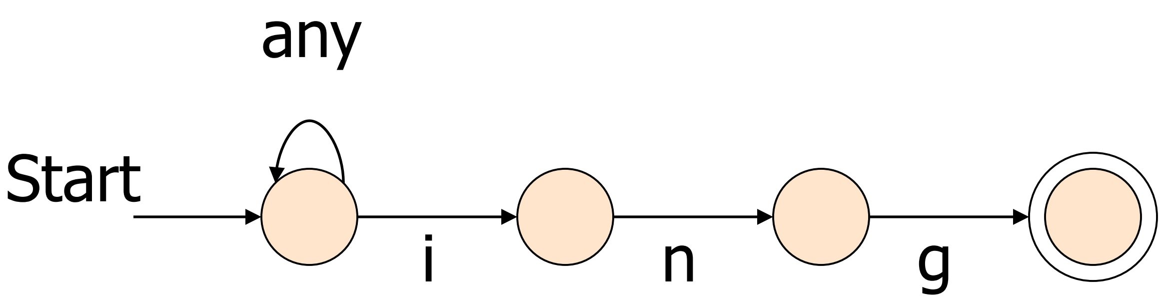
UNIX RE Notation – (2)

- | 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*."

Example: Text Processing

- Remember our DFA for recognizing strings that end in "ing"?
- It was rather tricky.
- But the RE for such strings is easy: **.*ing** where the dot is the UNIX "any".
- Even an NFA is easy (next slide).

NFA for "Ends in *ing*"



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.

Tricks for Combining Tokens

- There are some ambiguities that need to be resolved as we convert RE's to a DFA.
- Examples:
  1. "if" looks like an identifier, but it is a reserved word.
  2. < might be a comparison operator, but if followed by =, then the token is <=.

Tricks – (2)

- Convert the RE for each token to an ϵ–NFA.
  - Each has its own final state.
- Combine these all by introducing a new start state with ϵ-transitions to the start states of each ϵ–NFA.
- Then convert to a DFA.

Tricks – (3)

- If a DFA state has several final states among its members, give them priority.
- Example: Give all reserved words priority over identifiers, so if the DFA arrives at a state that contains final states for the “if” ϵ–NFA as well as for the identifier ϵ–NFA, if declares “if”, not identifier.

Tricks – (4)

- It's a bit more complicated, because the DFA has to have an additional power.
- It must be able to read an input symbol and then, when it accepts, put that symbol back on the input to be read later.

Example: Put-Back

- Suppose "<" is the first input symbol.
- Read the next input symbol.
  - If it is "=", accept and declare the token is <=.
  - If it is anything else, put it back and declare the token is <.

Example: Put-Back – (2)

- Suppose "if" has been read from the input.
- Read the next input symbol.
  - If it is a letter or digit, continue processing.
    - You did not have reserved word "if"; you are working on an identifier.
  - Otherwise, put it back and declare the token is "if".