# Ling/CSE 472: Introduction to Computational Linguistics

3/30/17

Regular expressions & Finite State Automata

# Overview

- Formal definition of regular languages
- Regular expression examples
- Extension to regular expressions
- Substitution in Python
- Finite state automata
- Recognition with FSA -- pseudocode
- DFSA v. NFSA
- Reading questions

# Announcement

- Assignment 1 is posted
- Elizalike.py (version 1) due 4/5 at 10am
- Complete assignment due 4/7 at 11:59pm

# Book vs. Lecture

- J&M talk about
  - regular expressions in the context of search,
  - then FSAs,
  - then regular languages
- We'll do
  - regular languages,
  - then regular expressions describing regular languages (with an aside about search)
  - then FSAs

# Formal languages

- From the point of view of formal languages theory, a language is a set of strings defined over some alphabet.

- Chomsky hierarchy: a description of classes of formal languages:

  - regular < context-free < context-sensitive < all languages

- Languages from a single level in the hierarchy can be described in terms of the same formal devices.

# Three views on the same object

*Regular languages can be described by regular expressions and by finite-state automata.*

- Regular language: a set of strings
- Regular expression: an expression from a certain formal language which describes a regular language
- Finite-state automaton: a simple computing machine which accepts or generates a regular language

# Formal definition of regular languages: symbols

- ϵ is the empty string
- ø is the empty set
- Σ is an alphabet (set of symbols)
- Examples of alphabets:
    - Σ = {a,b,!}
    - Σ = {c,a,t,d,o,g,f,x}
    - Σ = {cat, dog, fox}
    - Σ = {1, 3, 5, 7}

# Formal definition of regular languages

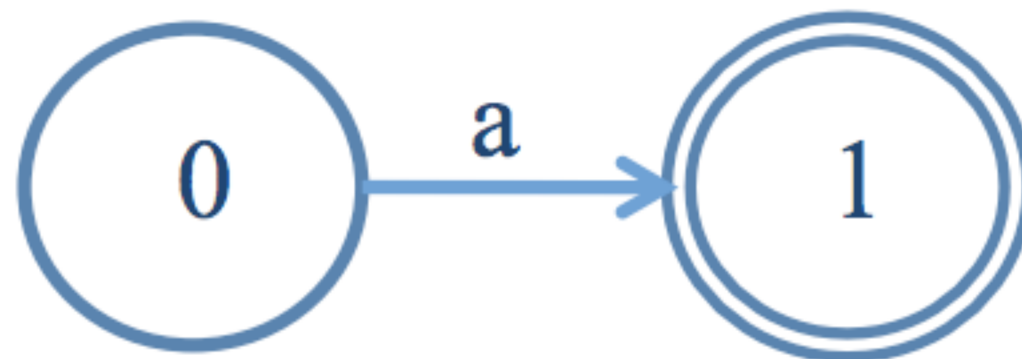
The class of regular languages over $\Sigma$ is formally defined as:

- $\emptyset$ is a regular language
- $\forall a \in \Sigma \cup \epsilon$, $\{a\}$ is a regular language
- If L1 and L2 are regular languages, then so are:
    - $L1 \cdot L2 = \{xy \mid x \in L1, y \in L2\}$ (concatenation)
    - $L1 \cup L2$ (union or disjunction)
    - $L1^*$ (Kleene closure)

(Jurafsky & Martin 2009:39)

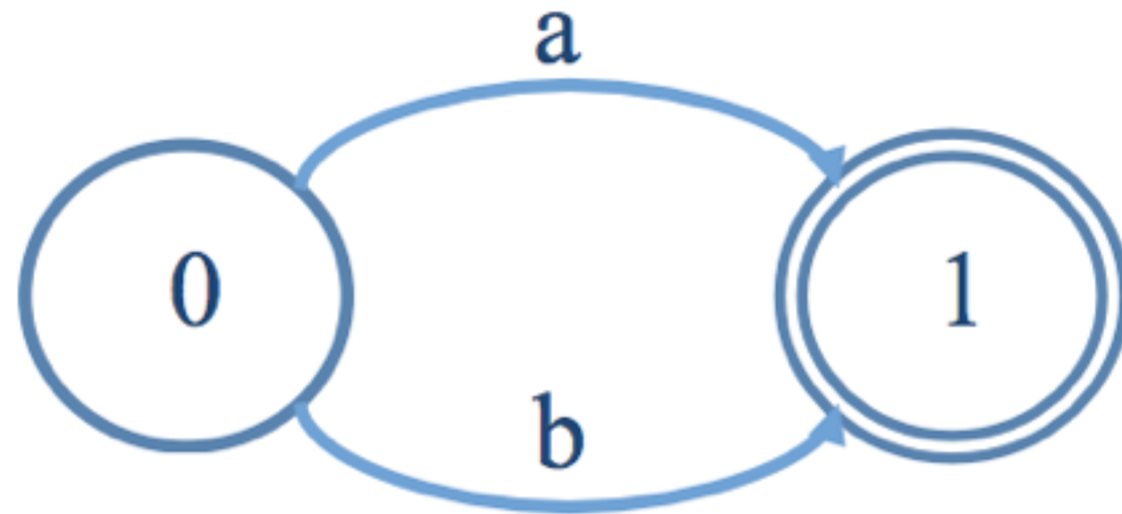
# Examples: Single Character

- Alphabet: $\Sigma = \{a\}$
- Language (set of strings): L = {a}
- Regular expression: /a/
- Finite state machine:



# Examples: Disjunction/union

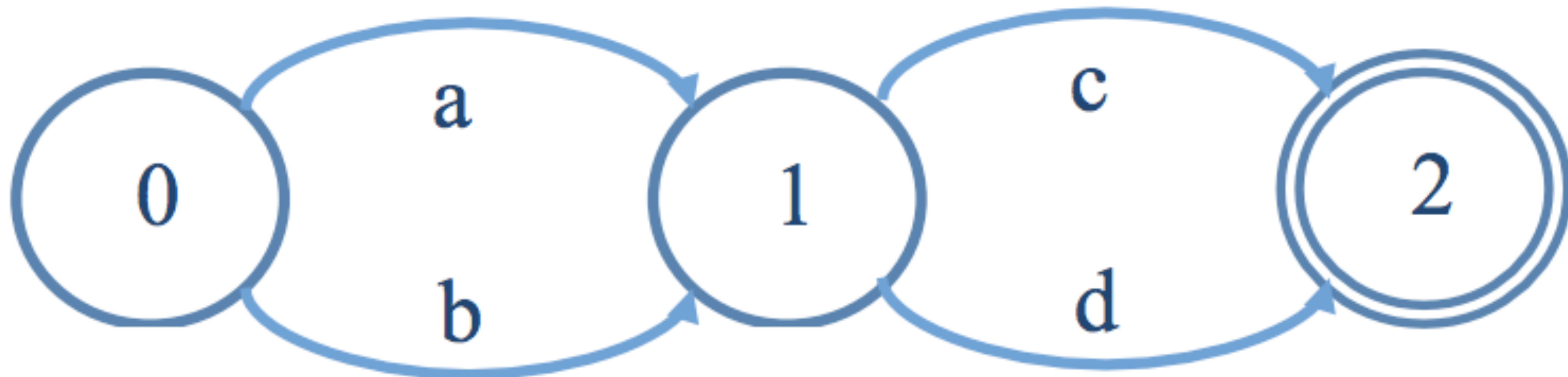
- Alphabet: Σ = {a, b}
- Language (set of strings): {a, b}
- Regular expression: /(a|b)/
- Finite state machine:




- This language/regexp/FSA also reflects the union of two languages: {a} ∪ {b}

# Examples: Concatenation

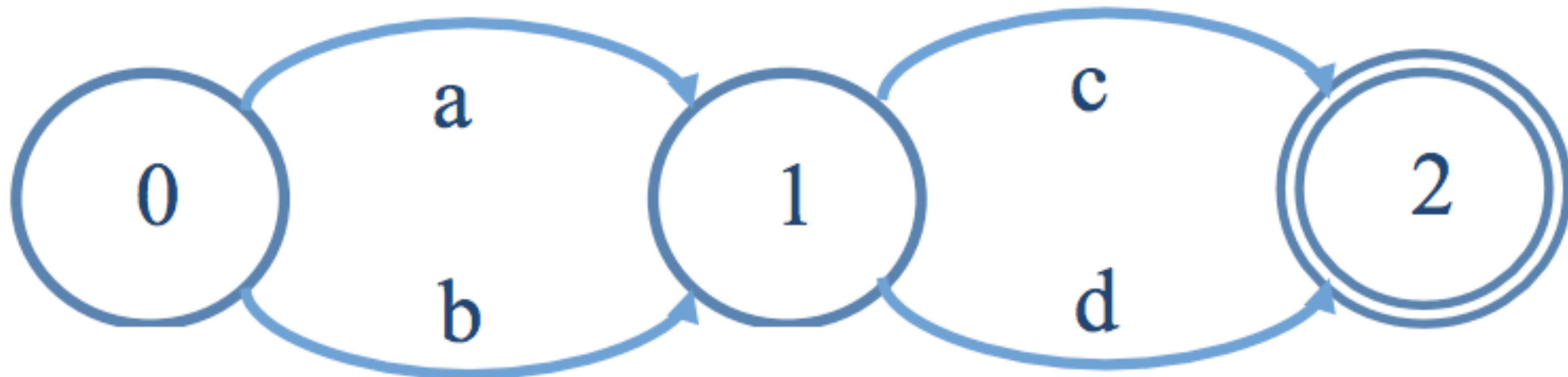
- Σ = {a, b, c, d}
  - L = {ac, ad, bc, bd}
  - Regular expression: /(a|b)(c|d)/
  - Finite state machine:



# Examples: Concatenation

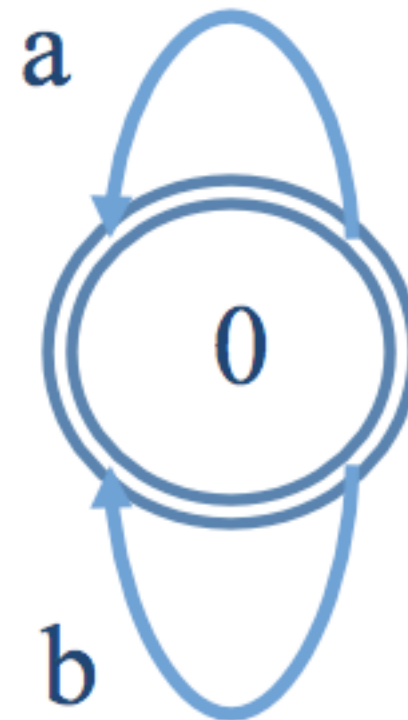
- Note: This language/regexp/FSA also reflects the concatenation of two languages:

  - L1 = {a, b} · L2 = {c, d}



# Examples: Kleene closure

- Σ = {a, b} (again)
- L = {ϵ, a, b, aa, bb, ab, ba, bb, ...}
- Regular expression: /(a|b)*/
- Finite state machine:



# Regular expression examples

- abc
- a|bc
- (a|bb)c
- a[bc]
- a*b
- a?b
- [^a]*th[aeiou]+[a-z]*
- ^a
- ^a.*z$
- \bthe\b

# Regular expressions: An extension

- Parentheses allow you to ‘save’ part of a string and access it again.
- In general:
  - ( ) creates a group that can be referenced
  - *n* refers back to the *n*th group in a regular expression
- For example, to search for repeated words: /([a-z]+)\1 \1/
- That will find any group of one or more lower case alphabetic characters repeated twice (and separated by a space): ‘I called kitty kitty kitty.’

# Regular expressions: An extension

- Note: This extension to Python/Perl/grep/MS “regular expression” syntax actually takes them beyond the realm of regular expressions in the Chomsky hierarchy sense.

- The languages generated by “regular expressions” augmented with this kind of memory device are NOT regular languages, i.e., cannot be recognized by FSAs.

- Why?

# So what are regular expressions good for?

- Search
- Search and replace
- Morphotactics: Defining classes of words as sequences of morphemes
- Chunk parsing: Finding units within running text that satisfy patterns such as D A* N

# Regular expression substitution in Python

- import re
- re.sub(pattern, replacement, string); use r to mark pattern as a regex
- \g<n> references the nth group

```
>>input = 'The Kiwis is a team.'
>>input =
re.sub(r'(\W)is(\W)','\g<1>are\g<2>',input)
>>print input
'The Kiwis are a team.'
```

# Back to FSAs

- An abstract computing machine
- Consists of a set of states (or nodes in a directed graph) and a set of transitions (labeled arcs in the graph)
- Three kinds of states: plain, start, final

# FSA as transition table

- An FSA can also be represented as a transition table:

| | Input | | |
|---|---|---|---|
| State | a | b | c |
| 0 | 1 | 3 | - |
| 1: | 1 | 2 | 3 |
| 2: | - | 3 | - |
| 3: | - | - | - |

# Recognizing a regular language

FSAs can be used to recognize a regular language.

- Take the FSA and a “tape” with the string to be recognized.
- Start with the start of the tape and the FSA’s start state.
- For each symbol on the tape, attempt to take the corresponding transition in the machine.
- If no transition is possible: reject.
- When the string is finished, check whether the current state is a final state.
- Yes: accept. No: reject.

# Notes on pseudocode

- Some basic components of algorithms:
  - Loops (e.g., while x is true; for every x in …)
  - Conditionals (e.g., if … do this, else do this)
  - Variable assignment (e.g., i = i+1)
  - Evaluating expressions (e.g. if x = 2, print x + 3)
  - Input values

# D-RECOGNIZE in pseudo-code

```
function D-Recognize(tape, transition-table) returns accept or reject
  index ← beginning of tape
  current-state ← initial state on transition-table
  loop
    if end of tape has been reached then
      if current-state is an accept state then
        return accept
      else
        return reject
    elsif transition-table[current-state, tape[index]] is empty then
      return reject
    else
      current-state ← transition-table[current-state, tape[index]]
      index ← index + 1
  end
```

Note: This is a slightly modified version of J&M, Figure 2.12, pg29

# Non-deterministic FSA (NFSA)

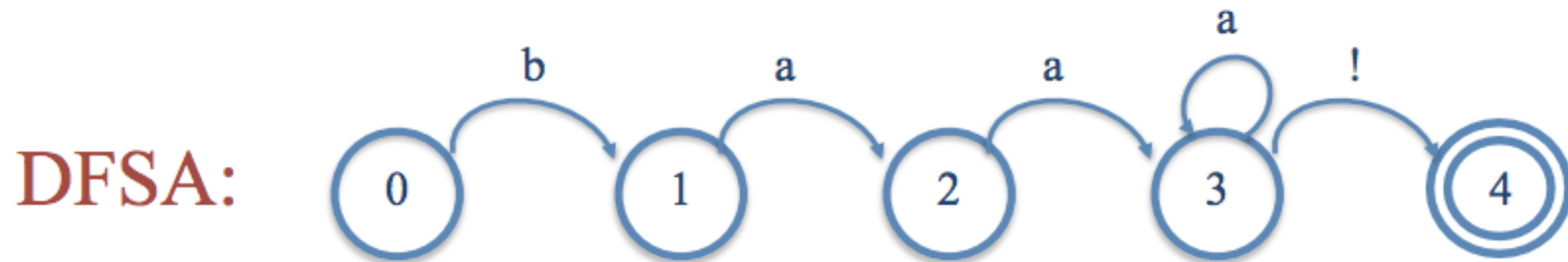
- The FSAs considered so far are deterministic (DFSA): there is only one choice at each node.

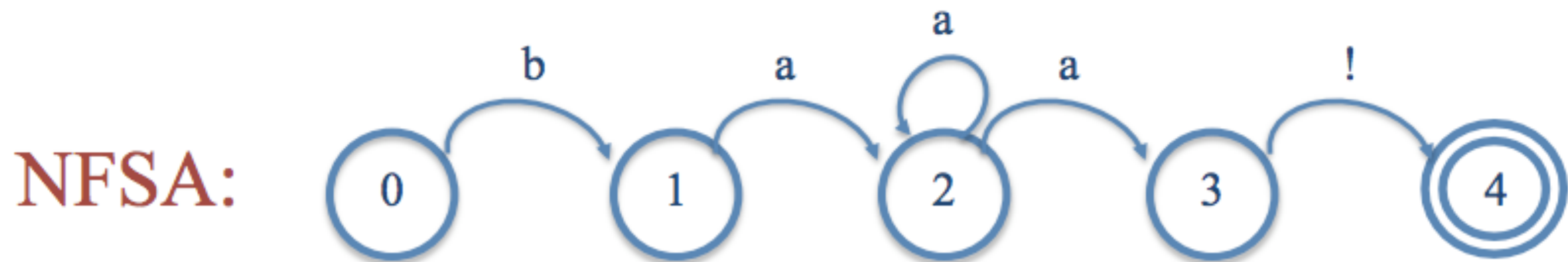
- NFSAs include more than one choice at (at least) one node.

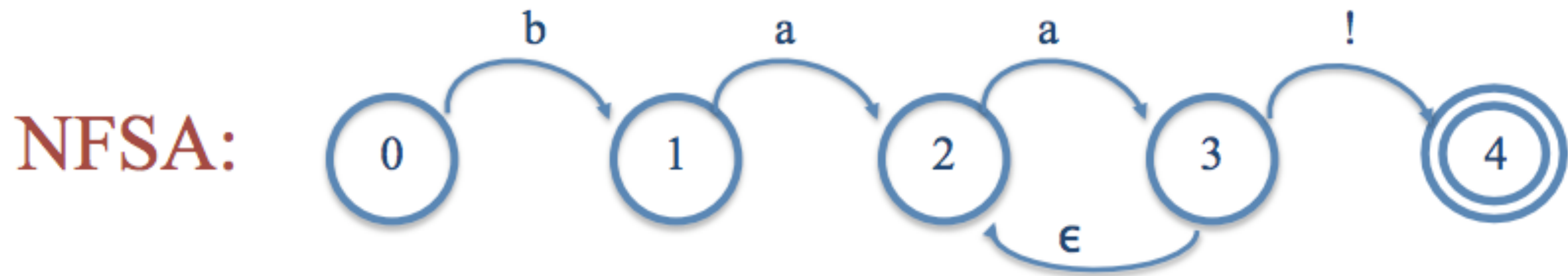
- Those choices might include $\epsilon$-transitions, i.e., unlabeled arcs that allow one to jump from one node to another without reading any input.

- Note: Any NFSA can be rewritten as a DFSA.

# Equivalent (D)FSA and NFSA



# Problem with Non-determinism

- Non-determinism can lead to dead ends when recognizing strings. How?
- Types of solutions:
  - backup
  - look-ahead
  - parallelism
- The backup technique is essentially to record a search-state at each decision point consisting of the position on the input tape and the state (node) of the machine. Then paths to dead ends can be retraced and different decisions made.
- What's the use/point of NFSAs, if there's always an equivalent DFSA?

# Overview

- Formal definition of regular languages
- Regular expression examples
- Extension to regular expressions
- Substitution in Python
- Finite state automata
- Recognition with FSA -- pseudocode
- DFSA v. NFSA
- Reading questions

# Reading Questions

- What are the minimal set of symbols needed for defining regex?
    - Concatenation, disjunction, *
- What's the point of .* if it will match any string?
- What's the point of * -- when do you ever want 0 instances of something?
- What's the difference between an "accepting state" and a "final state"?

# Reading Questions

- What makes one of the sheep talk FSAs deterministic but not the other?
- What are automata and how do they work? Why is that so abstract rather than describing a particular implementation?
- What's the difference between FSAs and regular expressions?

# Reading Questions

- Does the regular expression /the dog/ locate patterns that look like “dog the”? Is there any way to write a regular expression that says “give me any strings where all of these certain words show up in any order, anywhere in the string”?
- How would you to regular expressions in other alphabets?
- The alphabets used to define automata are particularly interesting to me. In particular, I wonder what restrictions exist on the contents of these alphabets. Can we have an alphabet of grammar structures at all levels? That is, could we have an alphabet with syntactic structures as well as morphological structures, or will the automaton only test for same-level structures? Is there room for a precedence of sorts, or would these two analyses require separate automata?

# Reading Questions

- For the list of operations under which regular languages are closed, why is it important (rather than just interesting) that the reversal is also regular? Also, are there any other important operations or ways of combining regular languages?