CS114: Regular Expressions and Automata

January 15, 2014
Lecture 2

Prof. Marie Meteer
Brandeis University

Models and Algorithms

- By models we mean the formalisms that are used to capture the various kinds of linguistic knowledge we need.
- Algorithms are then used to manipulate the knowledge representations needed to tackle the task at hand.

Models

- State machines
- Rule-based approaches
- Logical formalisms
- Probabilistic models

Algorithms

- Many of the algorithms that we'll study will turn out to be transducers; algorithms that take one kind of structure as input and output another.
- Unfortunately, ambiguity makes this process difficult. This leads us to employ algorithms that are designed to handle ambiguity of various kinds

Paradigms

- State-space search
  - To manage the problem of making choices during processing when we lack the information needed to make the right choice
- Dynamic programming
  - To avoid having to redo work during the course of a state-space search
    - CKY, Earley, Minimum Edit Distance, Viterbi, Baum-Welch
- Classifiers
  - Machine learning based classifiers that are trained to make decisions based on features extracted from the local context

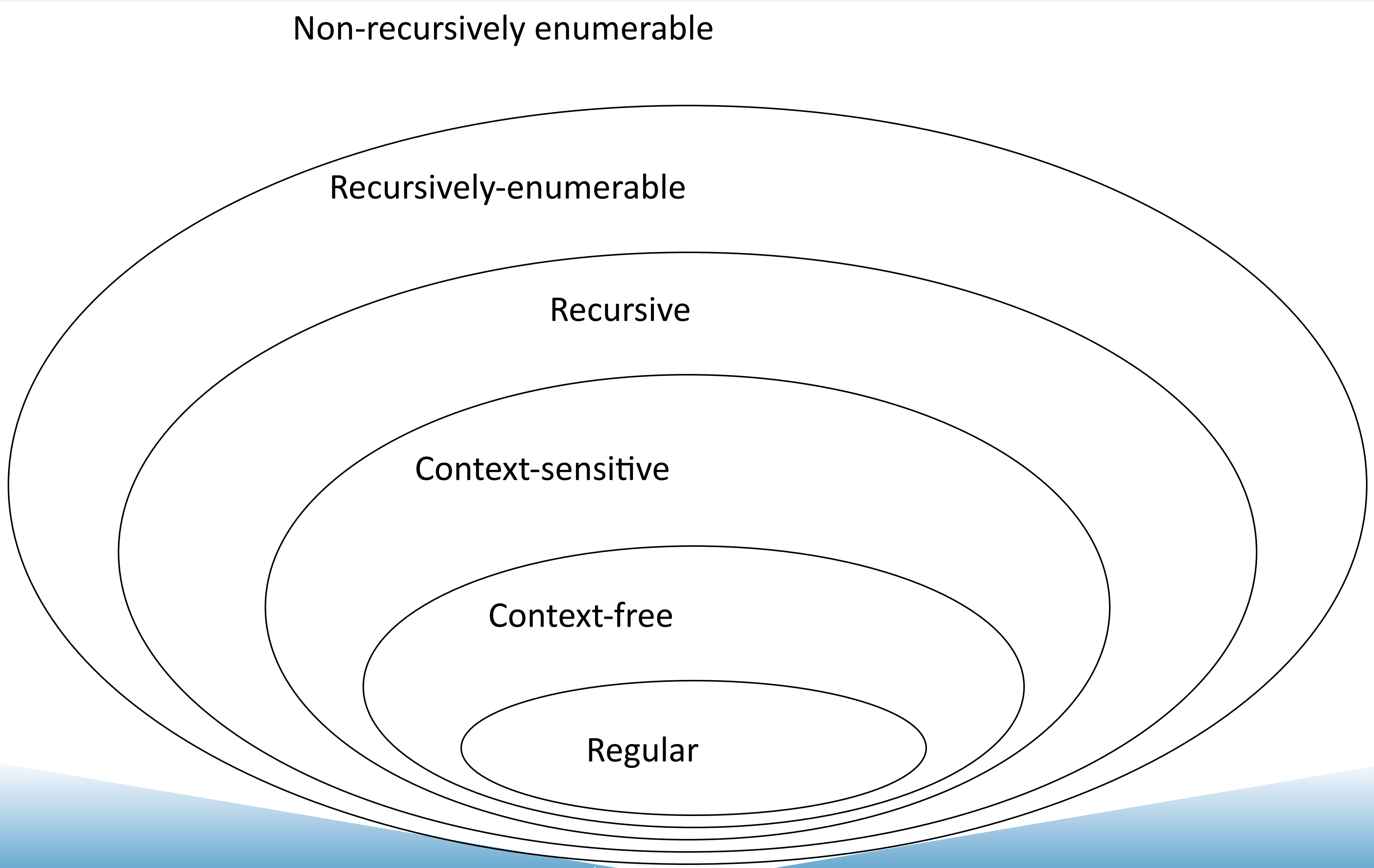
Languages and Grammars

- We can model a language with a grammar
  - Production rules: LHS→ RHS
  - NonTerminals indicate a production rule can be applied
  - Terminals make up the "strings" (sentences) of the language
- The grammar defines all the possible strings of terminals in the language
  - A "language" is generally an infinite number of finite strings
  - Any string can be "accepted"/parsed by the grammar
  - The grammar can generate all the strings

The "Generative Power" of Grammars

- Chomsky defined a hierarchy of language types distinguished by the characteristics of the grammars that can generate them
  - Finite State: A → Ab | b
  - Content Free: A → AB | a | b
  - Context Sensitive: bAc → bac
  - Recursively enumerable: No restrictions
- Many other important properties

Chomsky Hierarchy



Regular Expressions and Text Searching

- Everybody does it
  - Emacs, vi, perl, grep, etc..
- Regular expressions are a compact textual representation of a set of strings representing a language.

Example

- Find me all instances of the word "the" in a text (in perl)
  - `/the/`

    `Misses capitalized examples`
  - `/[tT]he/`
    - `Returns other or theology`
  - `/\b[tT]he\b/`
  - `/[^a-zA-Z][tT]he[^a-zA-Z]/`
  - `/(^|[^a-zA-Z])[tT]he[^a-zA-Z]/`

Slide from Dorr/Monz

Errors

- The process we just went through was based on two fixing kinds of errors
  - Matching strings that we should not have matched (there, then, other)
    - False positives (Type I)
  - Not matching things that we should have matched (The)
    - False negatives (Type II)

Errors

- We'll be telling the same story for many tasks, all semester. Reducing the error rate for an application often involves two antagonistic efforts:
  - Increasing accuracy, or precision
    - minimizing false positives
  - Increasing coverage, or recall
    - minimizing false negatives

Substitutions and Memory

- **Substitutions**
  - s/colour/color/
  - s/colour/color/g / ←global change
  - s/colour/color/i/ ←case independent
- **Memory ($1, $2, etc. refer back to matches)**
  - /the (.*)er they were, the $1er they will be/
  - /the (.*)er they (.*), the $1er they $2/

Eliza [Weizenbaum, 1966]

- User: Men are all alike
- ELIZA: IN WHAT WAY
- User: They're always bugging us about something or other
- ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE?
- User: Well, my boyfriend made me come here
- ELIZA: YOUR BOYFRIEND MADE YOU COME HERE
- User: He says I'm depressed much of the time
- ELIZA: I AM SORRY TO HEAR THAT YOU ARE DEPRESSED

Eliza-style regular expressions

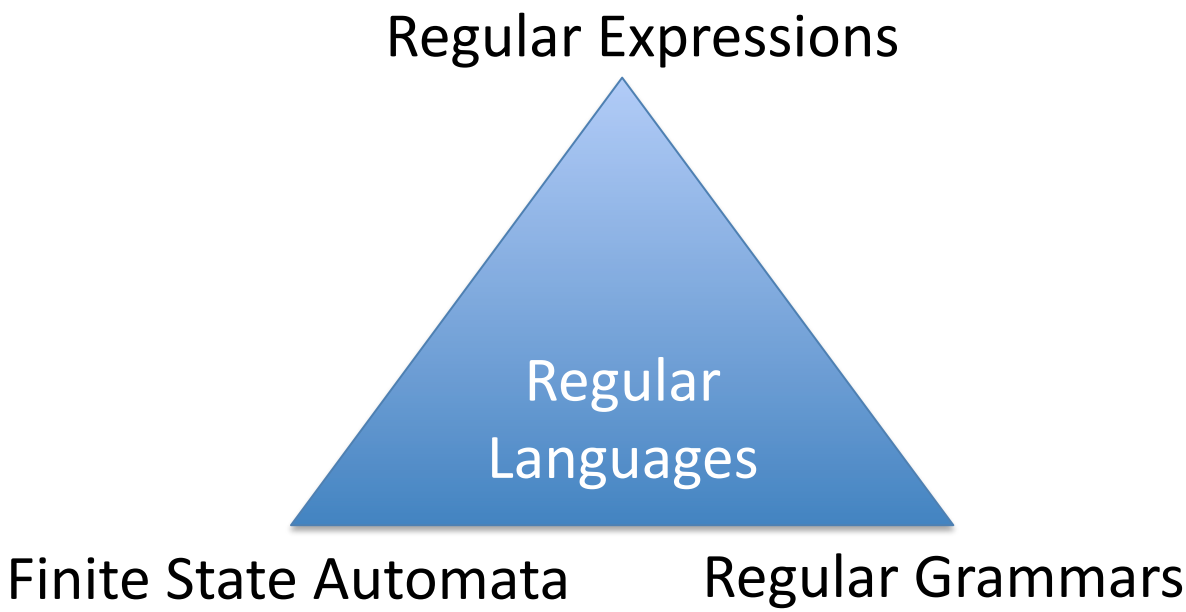
- **Step 1: replace first person with second person references**
  s/\bI('m| am)\b /YOU ARE/g
  s/\bmy\b /YOUR/g
  s/\bmine\b /YOURS/g
- **Step 2: use additional regular expressions to generate replies**
  s/.* YOU ARE (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/
  s/.* YOU ARE (depressed|sad) .*/WHY DO YOU THINK YOU ARE \1/
  s/.* all .*/IN WHAT WAY/
  s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/
- **Step 3:** use scores to rank possible transformations

Summary on REs so far

- Regular expressions are perhaps the single most useful tool for text manipulation
- Dumb but ubiquitous Eliza: you can do a lot with simple regular-expression substitutions

Three Views

- Three equivalent formal ways to look at what we're up to

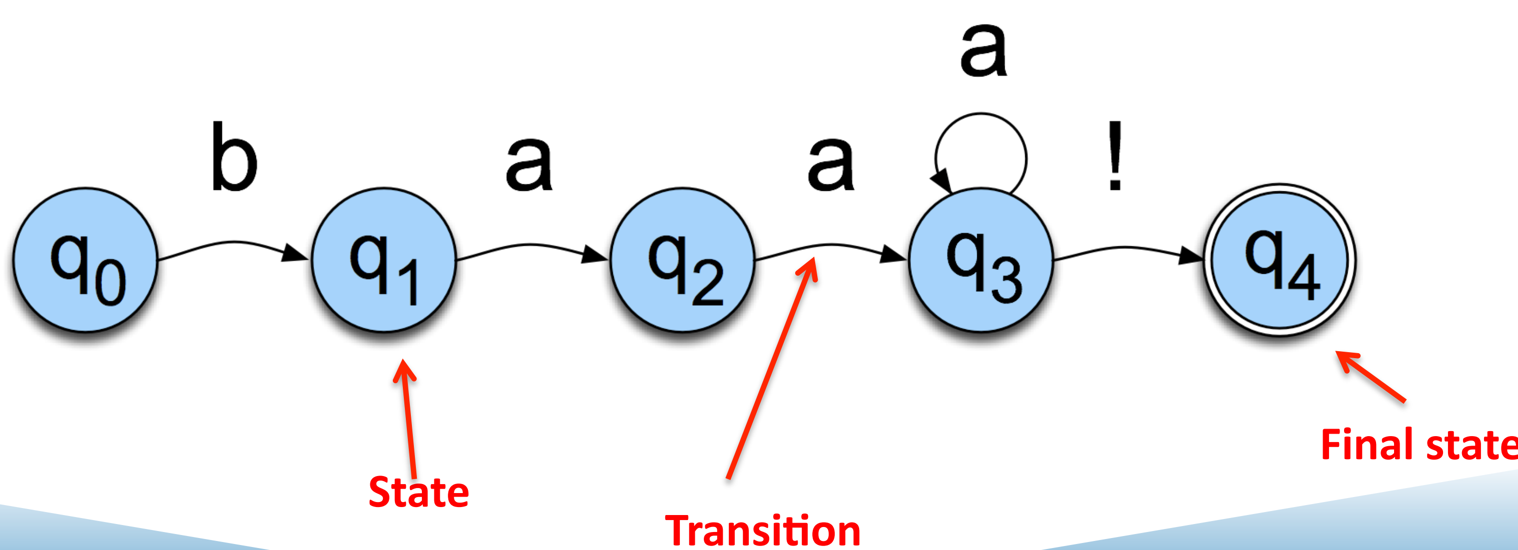


Finite State Automata

- Regular expressions can be viewed as a textual way of specifying the structure of finite-state automata.
- FSAs and their probabilistic relatives are at the core of much of what we'll be doing all semester.
- They also capture significant aspects of what linguists say we need for morphology and parts of syntax.

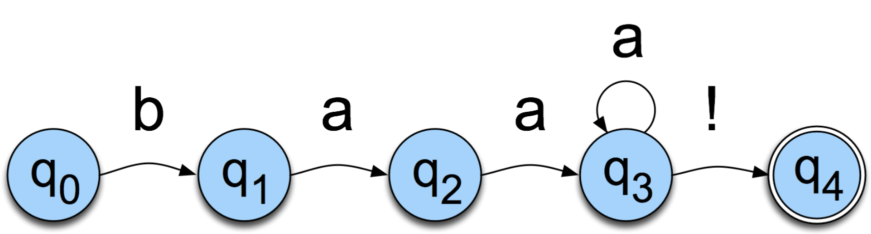
FSAs as Graphs

- Let's start with the sheep language
  `/baa+!/`



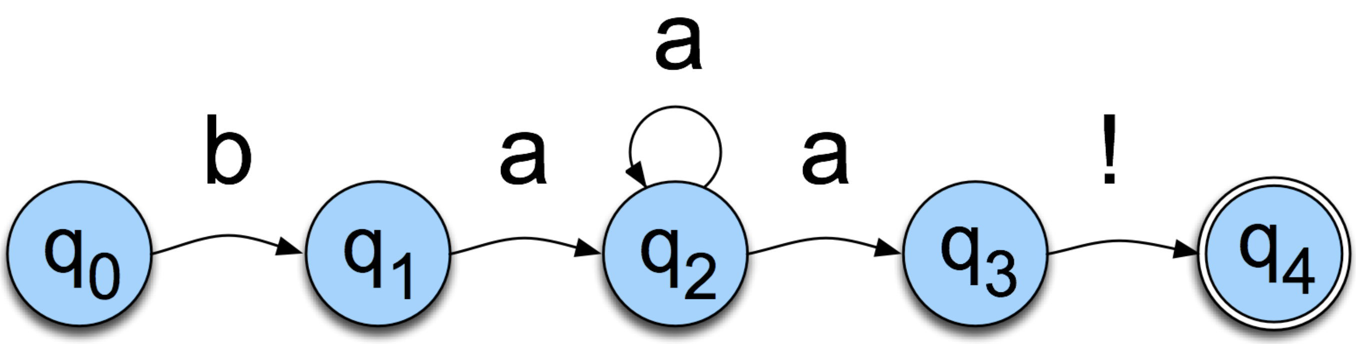
Sheep FSA

- We can say the following things about this machine
  - It has 5 states
  - b, a, and ! are in its alphabet
  - q_0 is the start state
  - q_4 is an accept state
  - It has 5 transitions



But Note

- There are other machines that correspond to this same language



- More on this one later

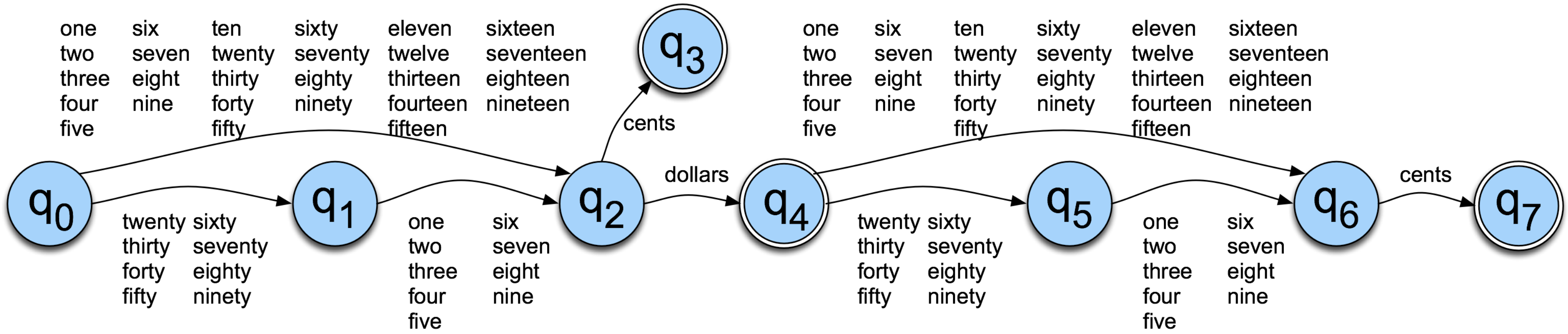
More Formally

- You can specify an FSA by enumerating the following things.
  - The set of states: Q
  - A finite alphabet: Σ
  - A start state
  - A set of accept/final states
  - A transition function that maps QxΣ to Q

About Alphabets

- Don't take term *alphabet* word too narrowly; it just means we need a finite set of symbols in the input.
- These symbols can and will stand for bigger objects that can have internal structure.

Dollars and Cents

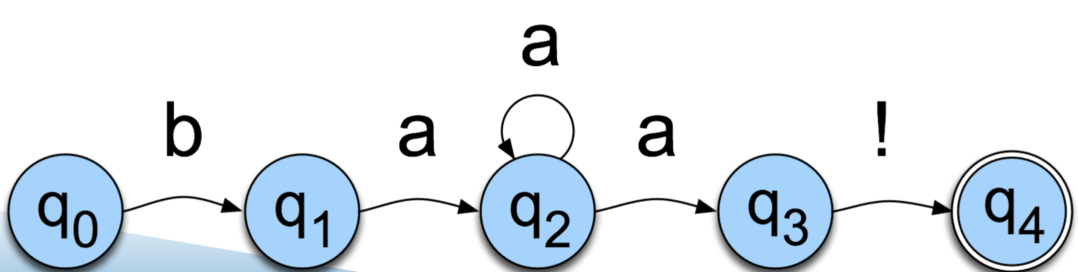


Yet Another View

- The guts of FSAs can ultimately be represented as tables

	b	a	!	e
0	1			
1		2		
2		2,3		
3			4	
4				

If you're in state 1 and you're looking at an a, go to state 2

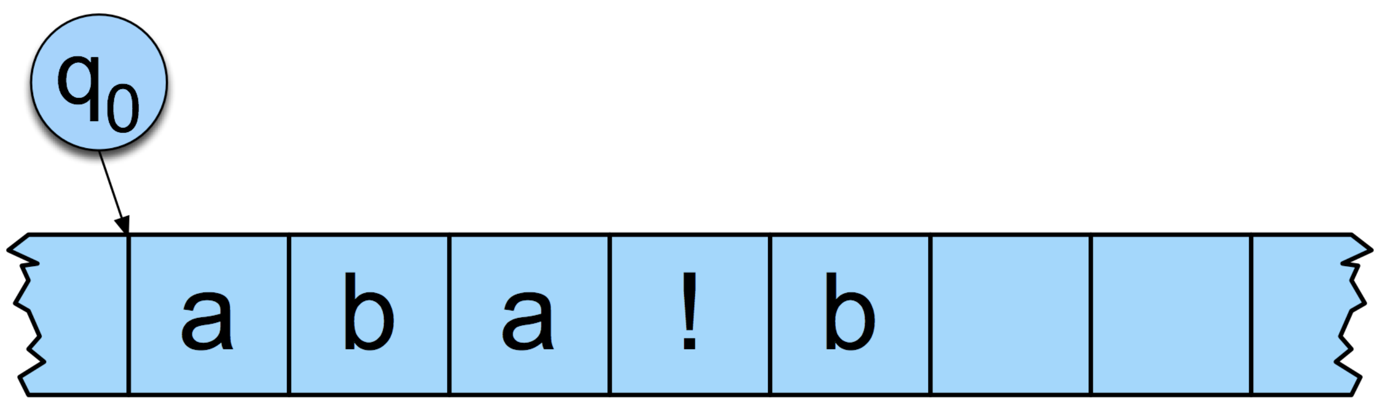


Recognition

- Recognition is the process of determining if a string should be accepted by a machine
- Or… it's the process of determining if a string is in the language we're defining with the machine
- Or… it's the process of determining if a regular expression matches a string
- Those all amount the same thing in the end

Recognition

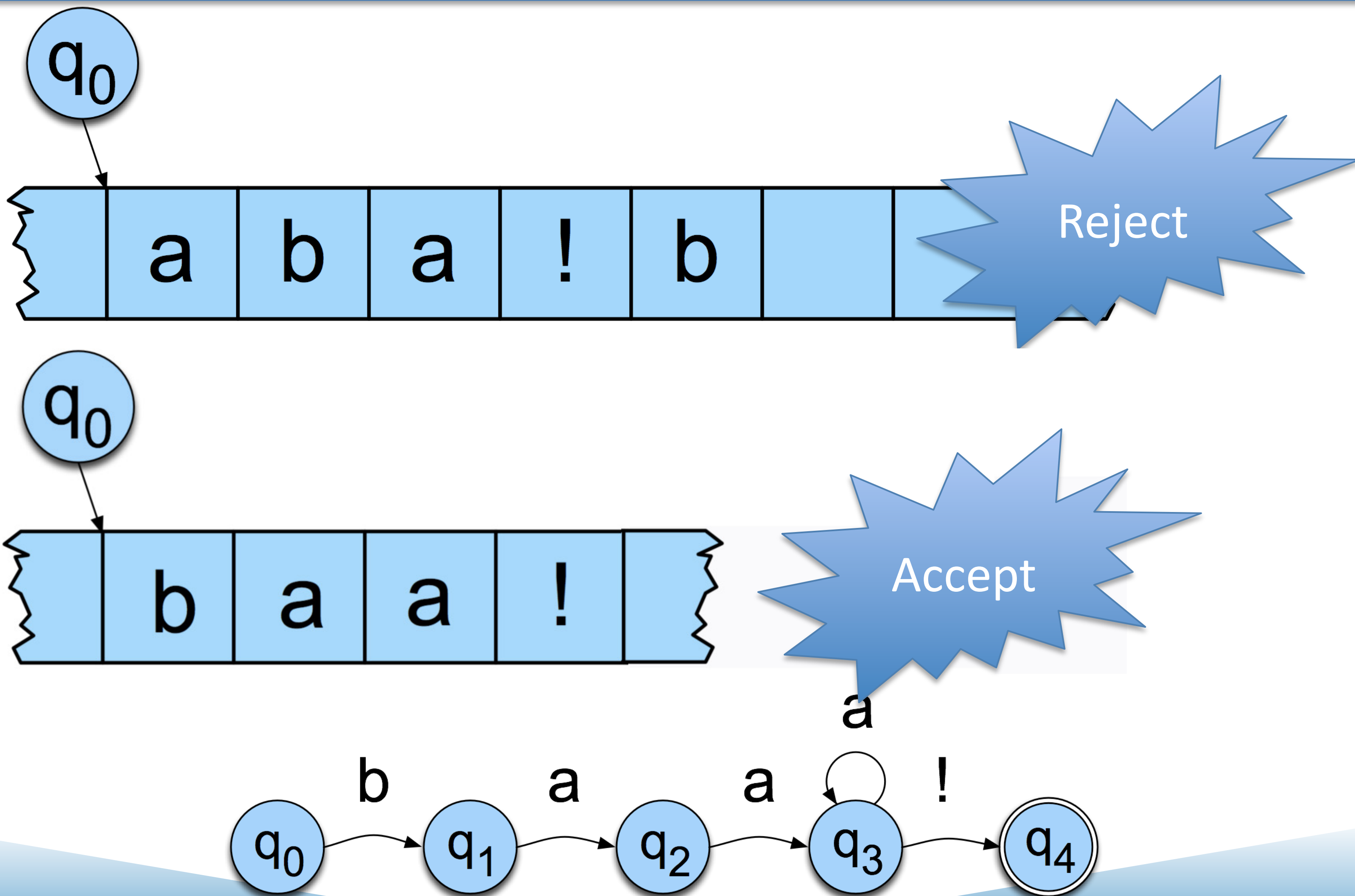
- Traditionally, (Turing's notion) this process is depicted with a tape.



Recognition

- Simply a process of starting in the start state
- Examining the current input
- Consulting the table
- Going to a new state and updating the tape pointer.
- Until you run out of tape.

Input tape



D-Recognize

```
function D-RECOGNIZE(tape, machine) returns accept or reject

  index ← Beginning of tape
  current-state ← Initial state of machine
  loop
   if End of input 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
```

Key Points

- Deterministic means that at each point in processing there is always one unique thing to do (no choices).
- D-recognize is a simple table-driven interpreter
- The algorithm is universal for all unambiguous regular languages.
  - To change the machine, you simply change the table.

Key Points

- Crudely therefore… matching strings with regular expressions (a la Perl, grep, etc.) is a matter of
  - translating the regular expression into a machine (a table) and
  - passing the table and the string to an interpreter

Recognition as Search

- You can view this algorithm as a trivial kind of state-space search.
- States are pairings of tape positions and state numbers.
- Operators are compiled into the table
- Goal state is a pairing with the end of tape position and a final accept state
- It is trivial because?

No ambiguity

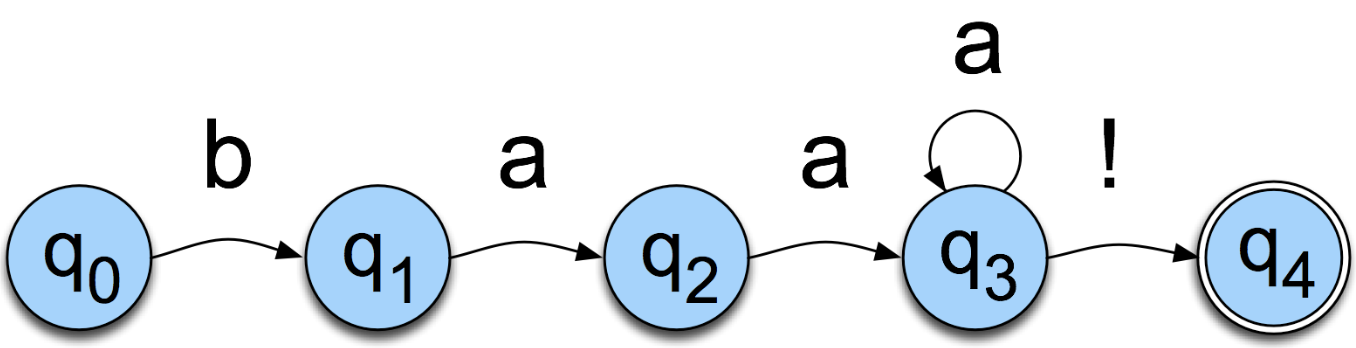
Generative Formalisms

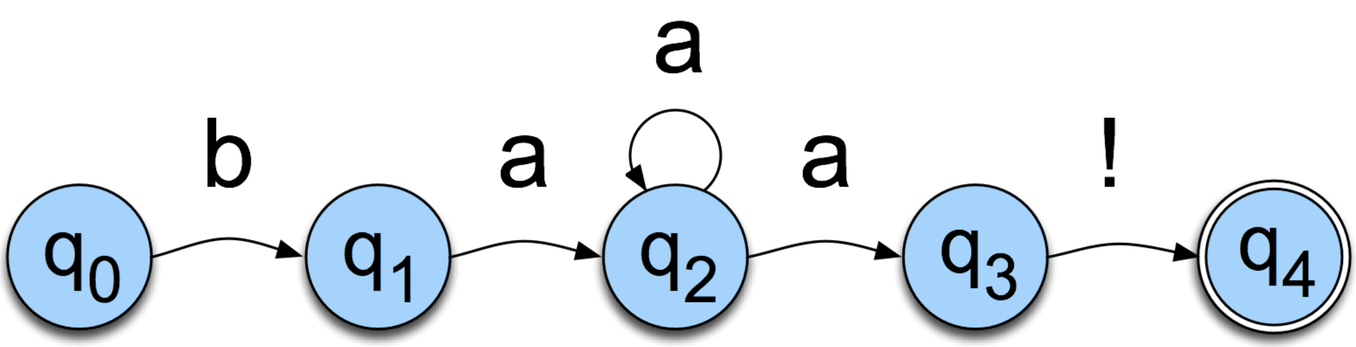
- *Formal Languages* are sets of strings composed of symbols from a finite set of symbols.
- Finite-state automata define formal languages (without having to enumerate all the strings in the language)
- The term *Generative* is based on the view that you can run the machine as a generator to get strings from the language.

Generative Formalisms

- FSAs can be viewed from two perspectives:
  - Acceptors that can tell you if a string is in the language
  - Generators to produce *all and only* the strings in the language

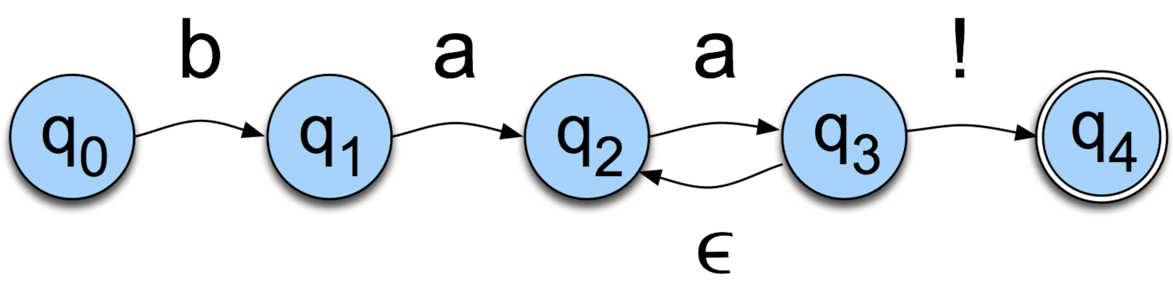
Non-Determinism





Non-Determinism cont.

- Yet another technique
  - Epsilon transitions
  - Key point: these transitions do not examine or advance the tape during recognition



Equivalence

- Non-deterministic machines can be converted to deterministic ones with a fairly simple construction
- That means that they have the same power; non-deterministic machines are not more powerful than deterministic ones in terms of the languages they can accept

ND Recognition

- Two basic approaches (used in all major implementations of regular expressions)
  1. Either take a ND machine and convert it to a D machine and then do recognition with that.
  2. Or explicitly manage the process of recognition as a state-space search (leaving the machine as is).

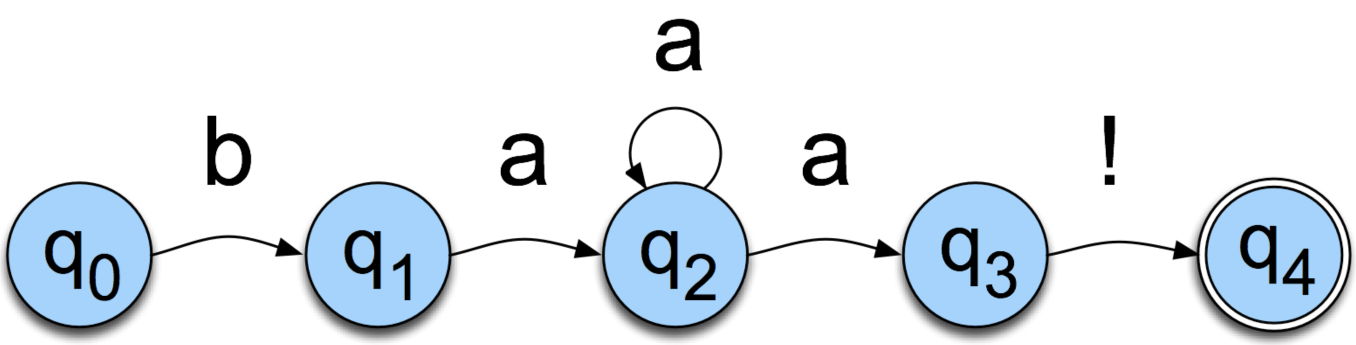
Non-Deterministic Recognition: Search

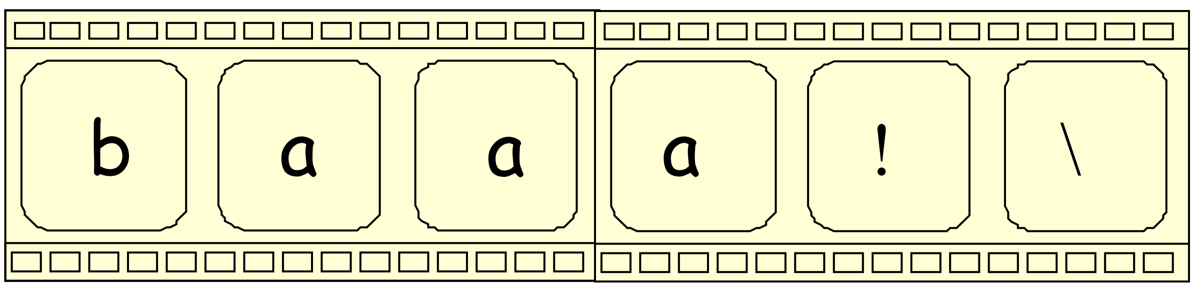
- In a ND FSA there exists at least one path through the machine for a string that is in the language defined by the machine.
- But not all paths directed through the machine for an accept string lead to an accept state.
- No paths through the machine lead to an accept state for a string not in the language.

Non-Deterministic Recognition

- So success in non-deterministic recognition occurs when a path is found through the machine that ends in an accept.
- Failure occurs when all of the possible paths for a given string lead to failure.

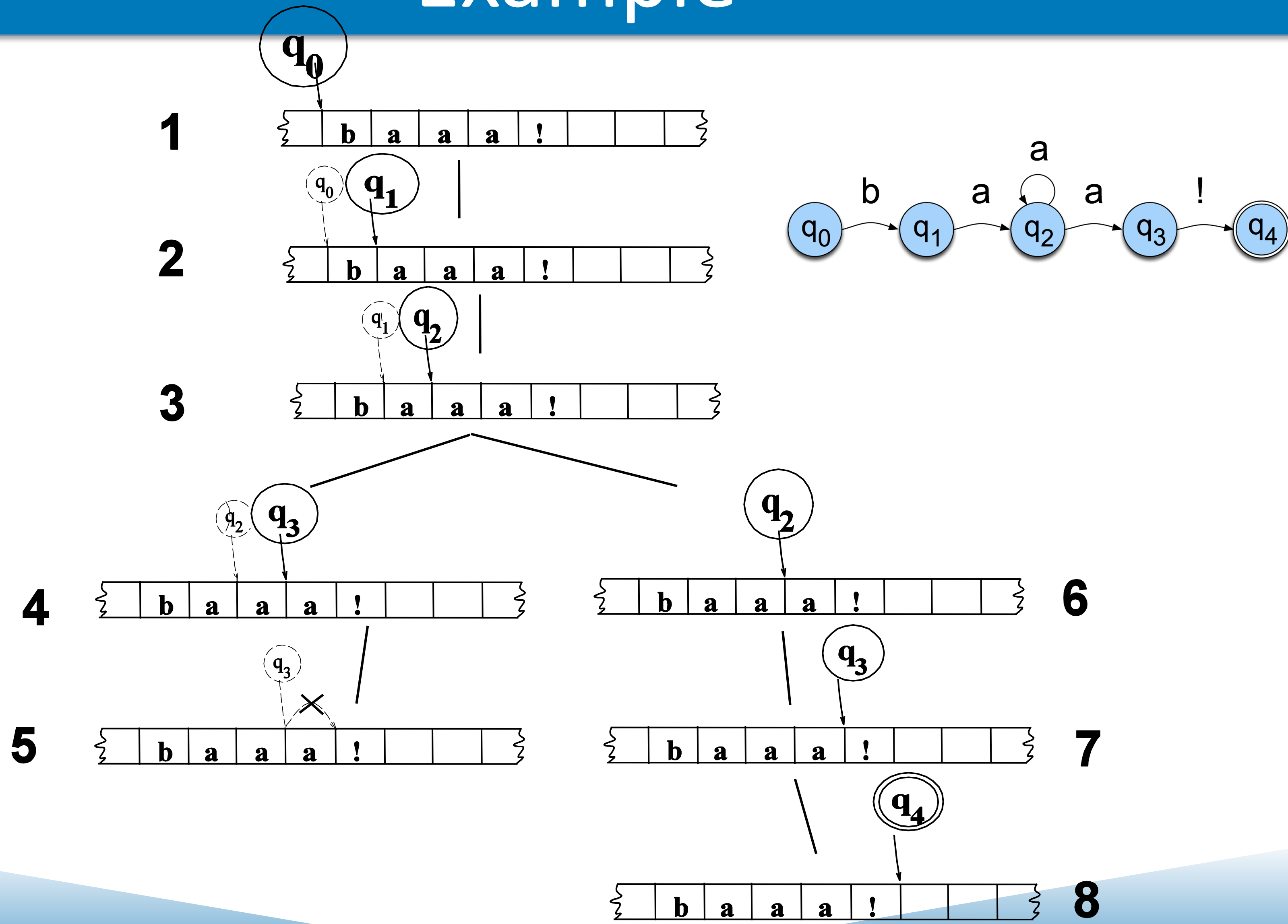
Example





q_0 q_1 q_2 q_2 q_3 q_4

Example



q0
1
b a a a !
q0 q1
2
b a a a !
q1 q2
3
b a a a !
q2 q3
4
b a a a !
q3
5
b a a a !
q2
6
b a a a !
q3
7
b a a a !
q4
8
b a a a !
b
a
a
a
!
q0
q1
q2
q3
q4

Key Points

- States in the search space are pairings of tape positions and states in the machine.
- By keeping track of as yet unexplored states, a recognizer can systematically explore all the paths through the machine given an input.

Why Bother?

- Non-determinism doesn't get us more formal power and it causes headaches so why bother?
  - More natural (understandable) solutions

Non-determinism

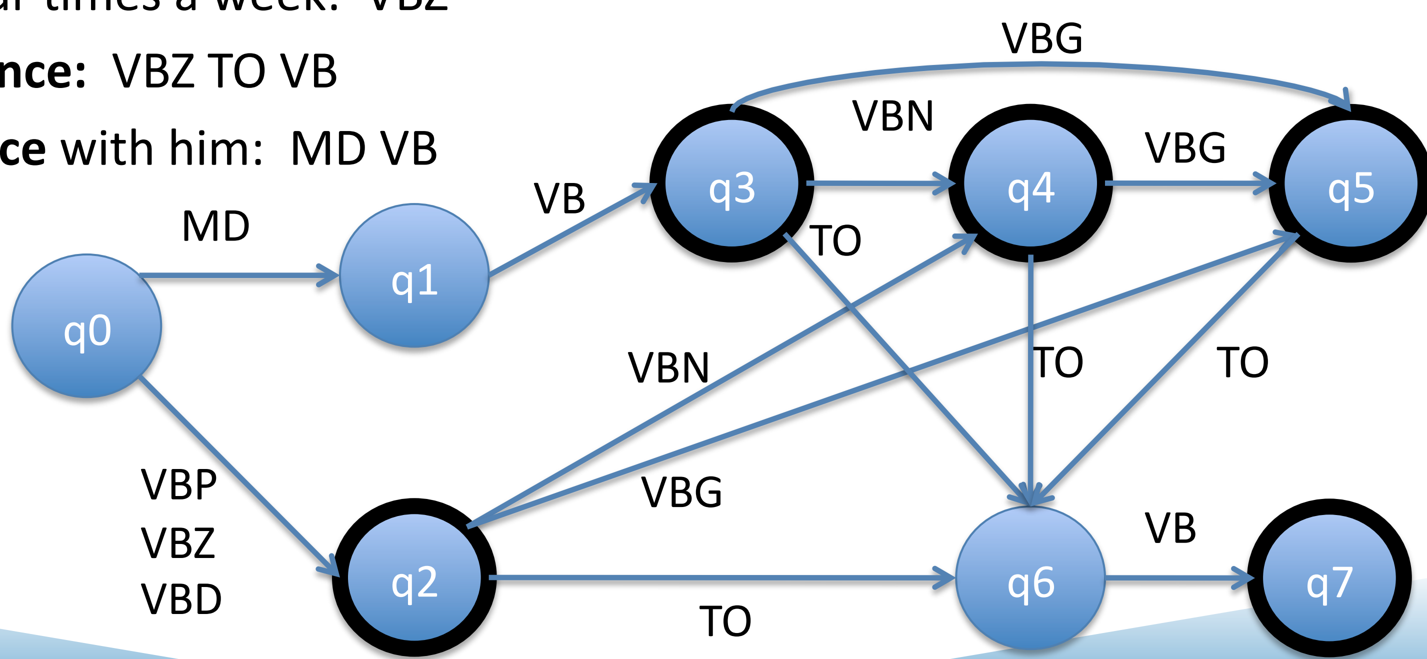
- Three ways to handle this:
  - Backup
  - Look ahead
  - Parallelism
- "Recognition" is search
  - Breadth first
  - Depth First
- Deterministic & nondeterministic equivalent
  - NFSA generally much cleaner
  - DFSA can have many more states
  - See textbook for discussion

Another FSA Example: Verb Groups in English

Tag	Part of speech	Example
MD	Modal	Could, would, will, might, …
VB	Verb base	Eat
VBD	Verb, past tense	Ate (with any subject) (I ate, he ate …)
VBZ	Verb, 3sng, pres	He eats (only he, she, it)
VBP	Verb, non-3sng, pres	I eat, We eat, they eat, you eat
VBG	Verb, gerund	I am eating (always with a form of “is”)
VBN	Verb, past participle	I have eaten (always with a form of “have”)
TO	“to”	“to” when marking a verb as in “to eat”

FSA for Verb Groups

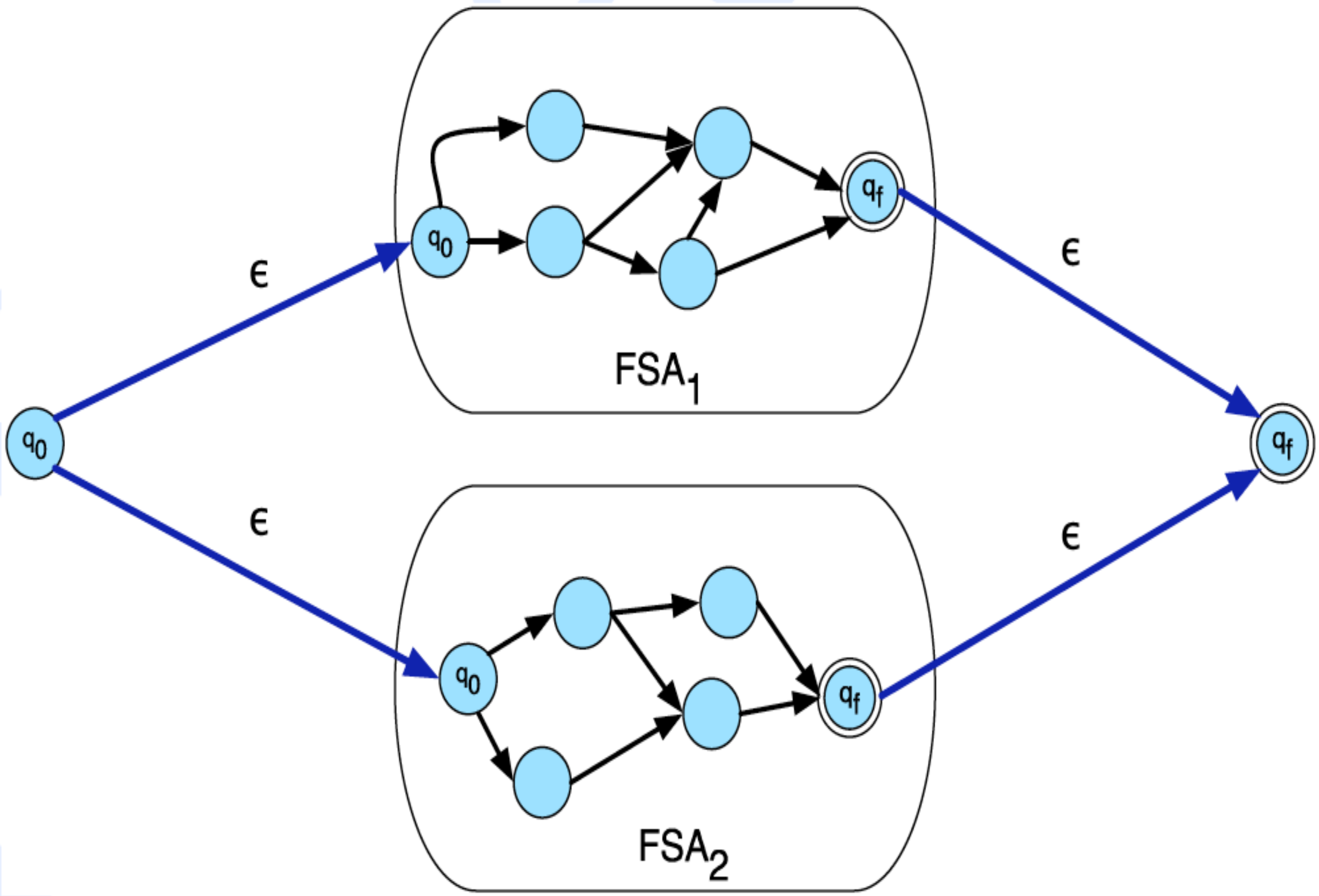
- I **could have danced** all night: MD VB VBN
- I **was dancing** when the lights went out: VBD VBG
- We **danced** the night away: VBD
- I **would have been dancing**, but …: MD VB VBN VBG
- He **has danced** his whole life: VBZ VBN
- She **dances** four times a week: VBZ
- He **loves to dance:** VBZ TO VB
- She **might dance** with him: MD VB



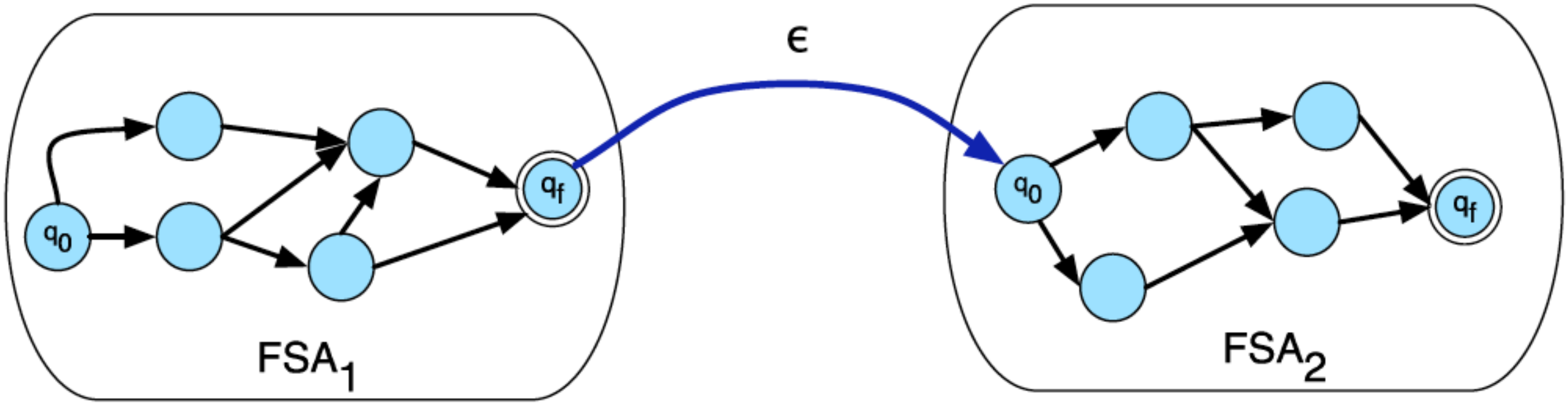
Compositional Machines

- Formal languages are just sets of strings
- Therefore, we can talk about various set operations (intersection, union, concatenation)
- This turns out to be a useful exercise

Union



Concatenation



Intersection

- Accept a string that is in both of two specified languages
- An indirect construction…
  – A^B = ~(~A or ~B)

(See details in SLP Ch 2)

Languages and Grammars

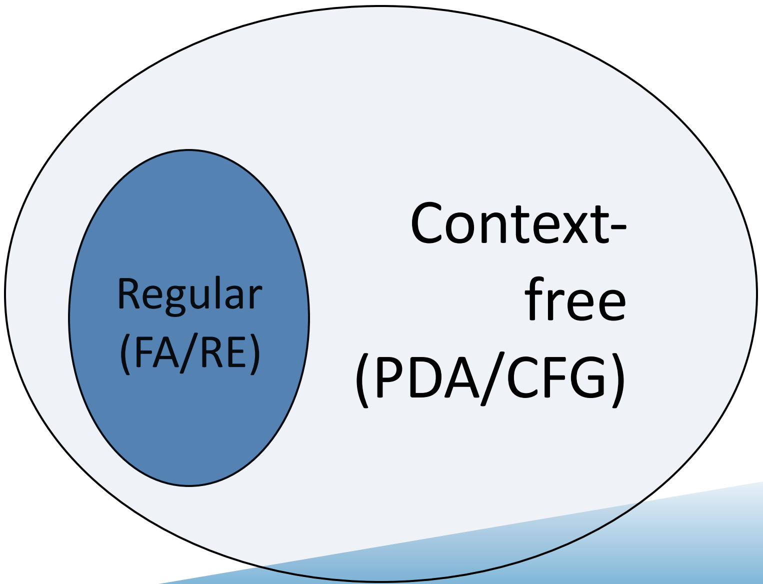
- We can model a language with a grammar
  - Production rules: LHS→ RHS
  - NonTerminals indicate a production rule can be applied
  - Terminals make up the "strings" (sentences) of the language
- The grammar defines all the possible strings of terminals in the language
  - A "language" is generally an infinite number of finite strings
  - Any string can be "accepted"/parsed by the grammar
  - The grammar can generate all the strings

Not all languages are regular

- So what happens to the languages which are not regular?
- Can we still come up with a language recognizer?
  - i.e., something that will accept (or reject) strings that belong (or do not belong) to the language?

Context-Free Languages

- A language class larger than the class of regular languages
- Supports natural, recursive notation called "context-free grammar"
- Applications:
  - Parse trees, compilers
  - XML



An Example

- A palindrome is a word that reads identical from both ends
  - E.g., madam, redivider, malayalam, 010010010
- Let L = { w | w is a binary palindrome}
- Is L regular?
  - No.
  - Proof:
    - Let $w=0^N10^N$
    - By Pumping lemma, w can be rewritten as xyz, such that xy^kz is also L (for any $k \geq 0$)
    - But $|xy| \leq N$ and $y \neq \varepsilon$
    - ==> $y=0^+$
    - ==> xy^kz *will NOT* be in L for k=0
    - ==> Contradiction

But the language of palindromes…

is a CFL, because it supports recursive substitution (in the form of a CFG)

- This is because we can construct a *"grammar"* like this:

1. A ==> ε
2. A ==> 0
3. A ==> 1
4. A ==> 0A0
5. A ==> 1A1

Productions

Terminal

Variable or non-terminal

Same as:
A => 0A0 | 1A1 | 0 | 1 | ε

How does this grammar work?

How does the CFG for palindromes work?

An input string belongs to the language (i.e., accepted) iff it can be generated by the CFG

- Example: 01110
- G can generate this input string as follows:

  – A=> 0A0

  => 01A10

  => 01110

G:
A => 0A0 | 1A1 | 0 | 1 | ε

Context-Free Grammar: Definition

- A context-free grammar G=(V,T,P,S), where:
  - V: set of variables or non-terminals
  - T: set of terminals (this is equal to the alphabet)
  - P: set of *productions,* each of which is of the form V ==> α_1 | α_2 | …
    - Where each α_i is an arbitrary string of variables and terminals
  - S ==> start variable

CFG for the language of binary palindromes:

- G=({A},{0,1},P,A)
- P: A ==> 0 A 0 | 1 A 1 | 0 | 1 | ε

More examples

- Parenthesis matching in code
- Syntax checking
- In scenarios where there is a general need for:
  - Matching a symbol with another symbol, or
  - Matching a count of one symbol with that of another symbol, or
  - Recursively substituting one symbol with a string of other symbols

Tag-Markup Languages

- Roll ==> <ROLL> Class Students </ROLL>
- Class ==> <CLASS> Text </CLASS>
- Text ==> Char Text | Char
- Char ==> a | b | … | z | A | B | .. | Z
- Students ==> Student Students | ε
- Student ==> <STUD> Text </STUD>

Chomsky Hierarchy