Regular Expressions

Jurafsky and Martin, Chapter 2

Outline

Regular Expressions and Finite State Automata

Deterministic Recognition

Generative Grammars and Formal Languages

Non-Determinism

Recognition as Search

ND-Recognize

Fun with FSAs

Administration

Today in Technology History Archives, January 8 - Computer scientist Joseph Weizenbaum, inventor of ELIZA, born (1923)

- http://tecsoc.org/pubs/history/2003/jan8.htm

Words

- finite state automata (this chapter)
- finite state transducers
- English morphology
- other areas of AI (machine learning, reflex agents)

Regular Expressions

Useful ways to view Regular Expressions

- as a practical way to specify textual search strings
- as a way to specify the design of a particular kind of machine

As we will see, these are equivalent

Uses of Regular Expressions in NLP

Simple but powerful tools such as grep and Perl allow large corpus analysis and "shallow" processing

- what word is most likely to begin a sentence vs. a question?
- which pronouns are conjoined most often?
- in your own email, are you more or less polite than the people you correspond with (and with labeled data, how can you learn this)?
- which candidate for Governor is mentioned most often in the news (and with labeled data, which is mentioned most favorably)?

With other tools, we can also

- build simple interactive applications like Eliza

Regular Expressions: Text Searches

Everybody does it

- Perl, emacs, Word, sed, awk, grep, netscape, etc.

Basic regular expression patterns (Perl notation) - J&M pages 23-27

- sequence (slashes)
- disjunction (brackets)
- range (brackets plus dash)
- negation (open bracket plus caret)
- optionality (question-mark)
- Kleene star (asterisk)
- wildcard (period)
- anchors (caret, dollar-sign, \b, \B)

Brackets and Character Disjunction

RE	Match	Example Patterns
`/[wW]oodchuck/`	Woodchuck or woodchuck	"Woodchuck"
`/[abc]/`	'a', 'b', *or* 'c'	"In uomini, in soldati"
`/[1234567890]/`	any digit	"plenty of 7 to 5"

A Simple Example

Finding instances of the determiner "the" (using a pattern, over a corpus)

```
/the/
/[tT]he/
/\b[tT]h\b/
```

The second RE catches "the" when it begins a sentence, while the third RE does not match "the" when embedded in another word such as "other"

The Two Kinds of Errors

The process we just went through was based on fixing errors in the regular expression

- errors where some of the instances were missed (judged to be not instances when they should have been)
- errors where the instances were included (when they should not have been)

This is pretty much going to be the story of the rest of the course

REs: Text Searches (continued)

Disjunction, grouping, and precedence - J&M pages 27-28

Operators are greedy

Advanced operators - J&M pages 30-31

Substitution and memory - J&M pages 31-33

Will return to RE syntax using NLTK tutorial

ELIZA: Substitutions Using Memory

```
User: Men are all alike.

s/.* all .*/IN WHAT WAY/

ELIZA: IN WHAT WAY

User: They're always bugging us about something or other.

s/.* always .*/CAN YOU THINK OF A SPECIFIC EXAMPLE/

ELIZA: CAN YOU THINK OF A SPECIFIC EXAMPLE

User: My boyfriend says I'm depressed.

s/.* I'm (depressed|sad) .*/I AM SORRY TO HEAR YOU ARE \1/

ELIZA: I AM SORRY TO HEAR YOU ARE DEPRESSED
```

Finite State Automata

Regular Expressions (REs) can be viewed as a way to describe machines called Finite State Automata (FSA, also known as automata, finite automata).

FSAs and their close variants are a theoretical foundation of much of the field of NLP.

FSAs as Directed Graphs

A sheep language

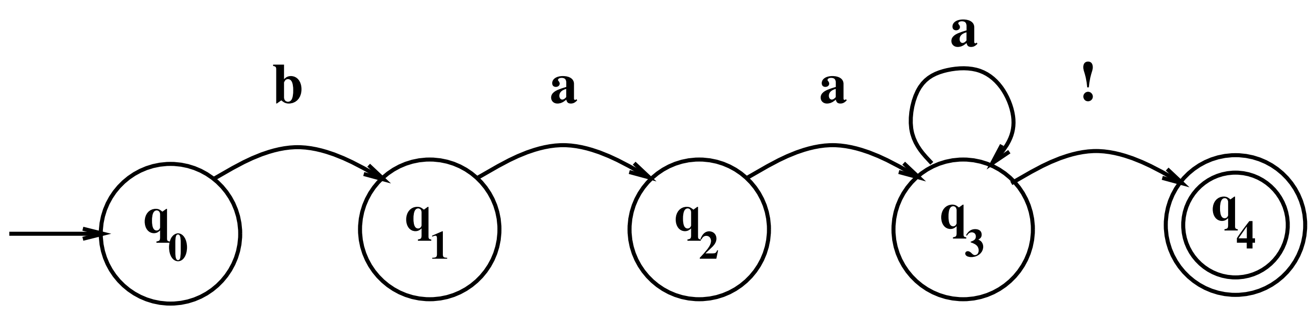
- baa!
- baaa!
- ...

A regular expression for the sheep language

- /baa+!/

FSAs as Directed Graphs (continued)

A graphical view of the sheep language

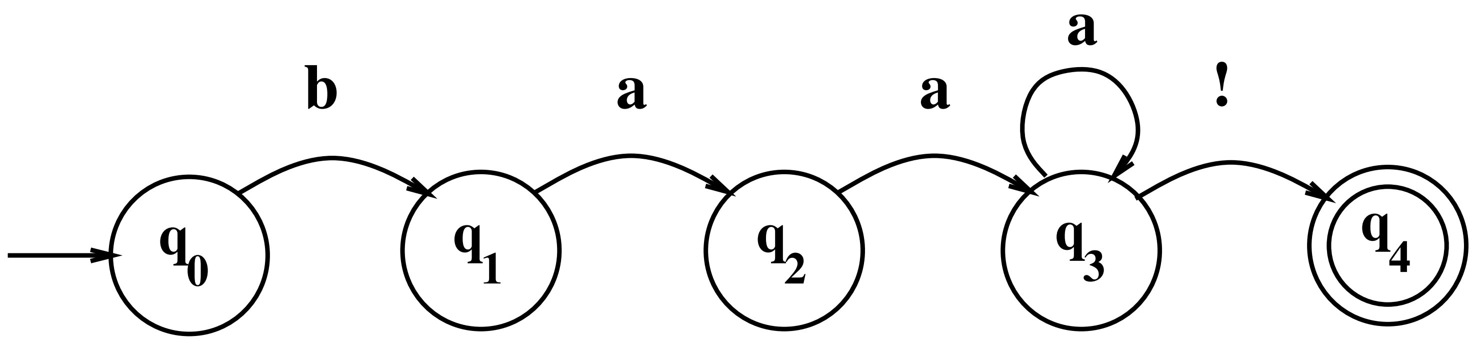


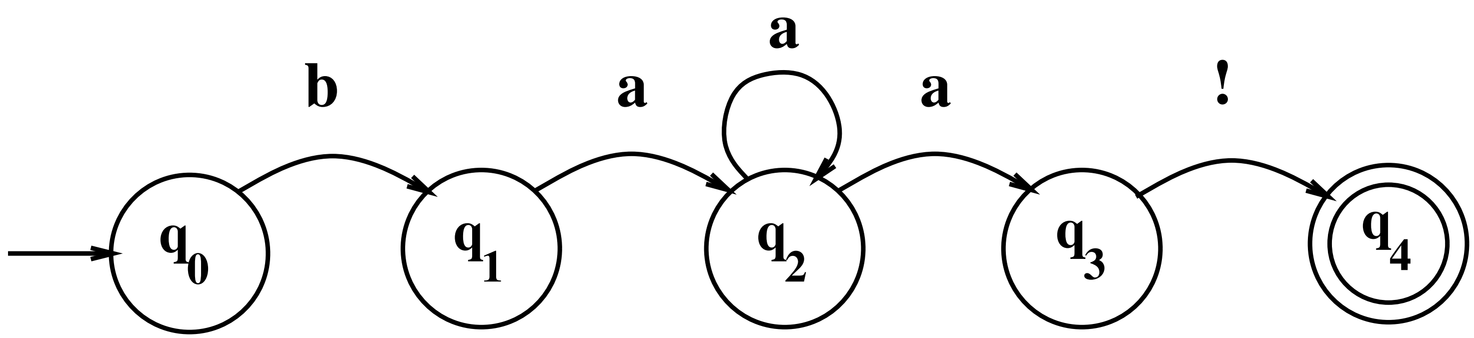
- five states (nodes in the graph)
- q0 is the start state (incoming arrow)
- q4 is the final state (double circle)
- five transitions (arcs in the graph)
- deterministic

Alternatives for our Sheep Language

/baa+!/

/baaa*!/





More on this later...

A More Formal View

FSAs can be formally specied as a 5-tuple

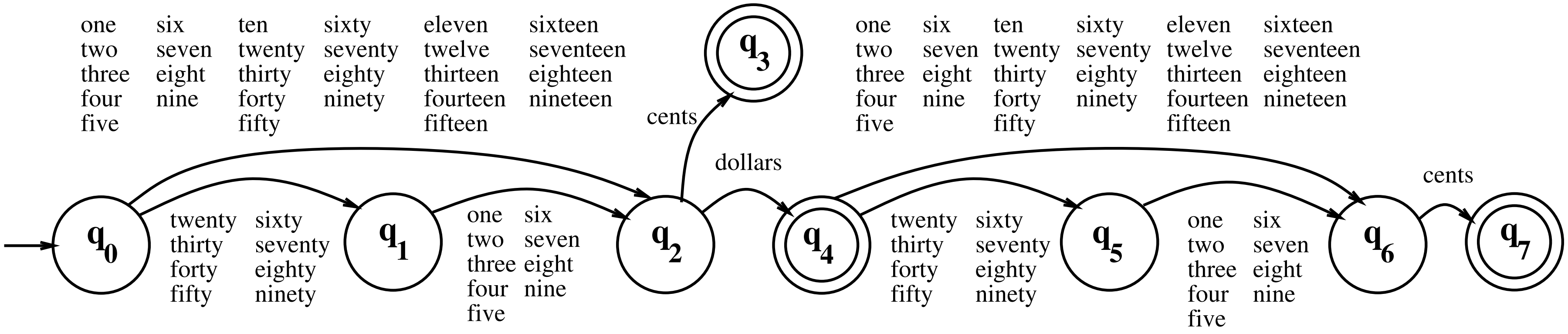
- a finite set of N states $q_0, q_1 \ldots q_n$: Q
- a finite input alphabet of symbols: Σ
- the start state: q_0
- the set of final states: F $\subseteq$ Q
- the transition function that maps Q $\times$ Σ to Q

A Note on Alphabets

You shouldn't view the term alphabet too narrowly. In particular, you don't have to limit it to letters. Any kind of symbol will do.

Furthermore, those symbols can stand for objects that themselves have internal structure.

Simple Dollars and Cents Example

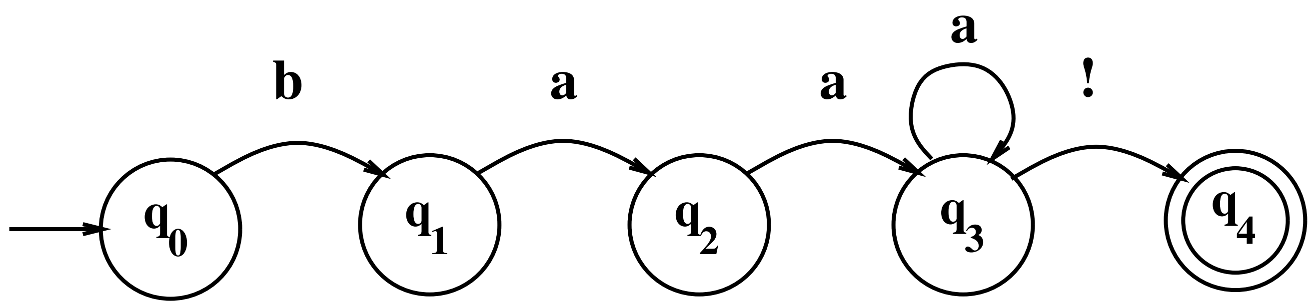


- alphabet of words
- accepts phrases like *ten cents, three dollars, twenty dollars thirty five cents*
- but...

– accepts *one dollars*

– doesn't accept *hundred dollars*

Yet Another Representation

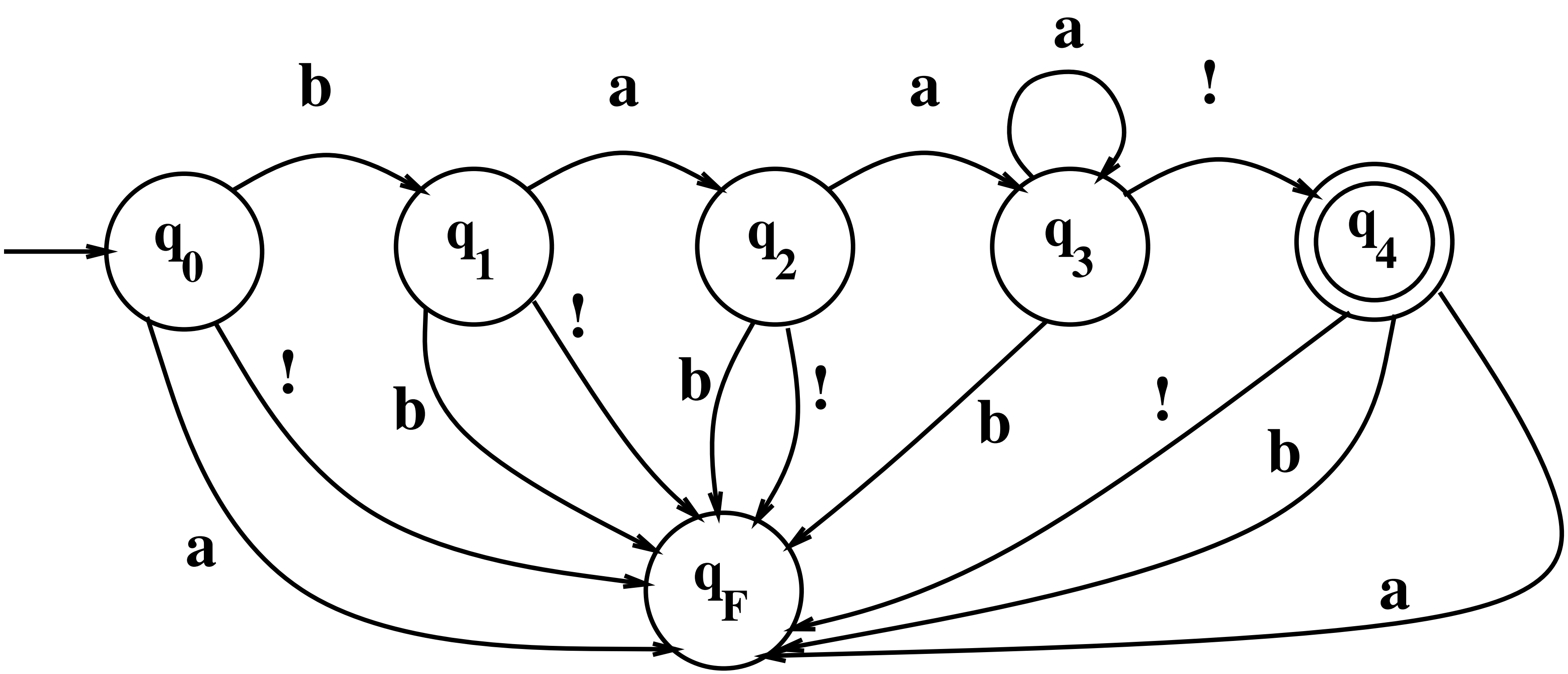


FSAs can also be encoded as *State Transition Tables*

- Q $= \{q_0, q_1, q_2, q_3, q_4\}$
- $\Sigma = \{$a,b,!$\}$
- F$=\{q_4\}$
- transition function $=$

	Input		
State	`b`	`a`	`!`
0	1	∅	∅
1	∅	2	∅
2	∅	3	∅
3	∅	3	4
4:	∅	∅	∅

Adding a Failing State

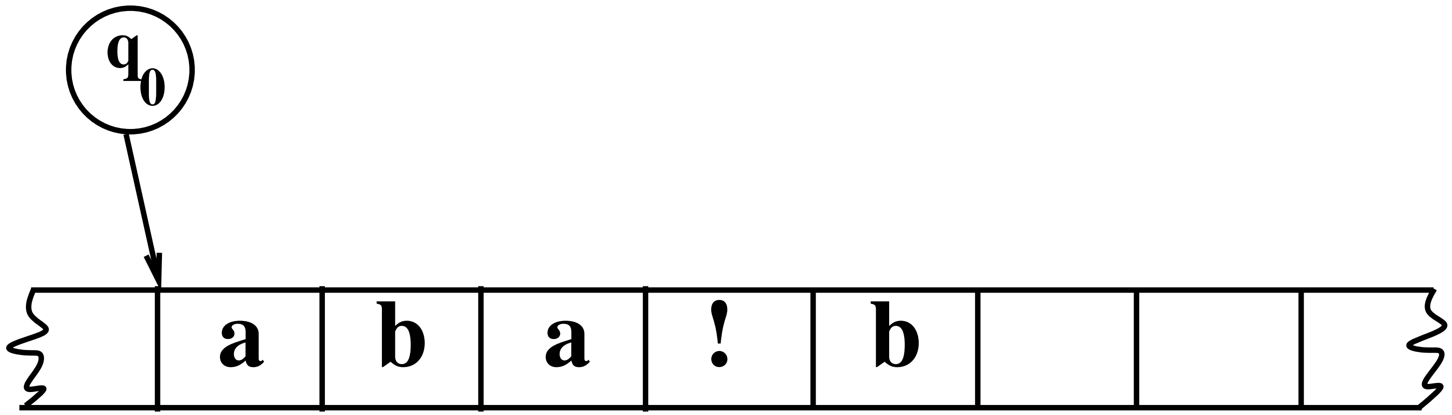


Recognition

Recognition (or acceptance) is the process of determining whether or not a given input should be accepted by a given machine

In terms of REs, its the process of determining whether or not a given input matches a particular regular expression.

Traditionally , recognition is viewed as processing an input written on a tape consisting of cells containing elements from the alphabet



D-Recognize

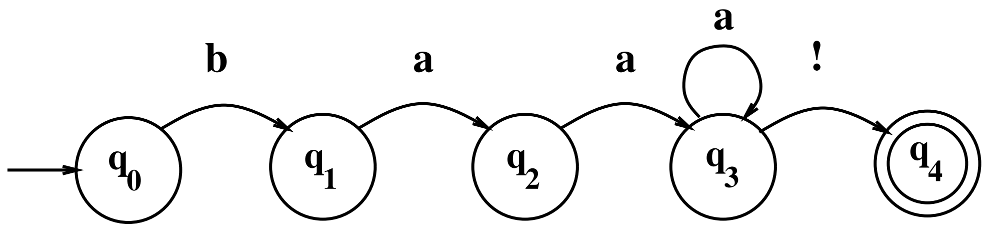
```
function D-RECOGNIZE(tape, machine) returns accept or reject

  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
```

D-Recognize (continued)

Key points about d-recognize

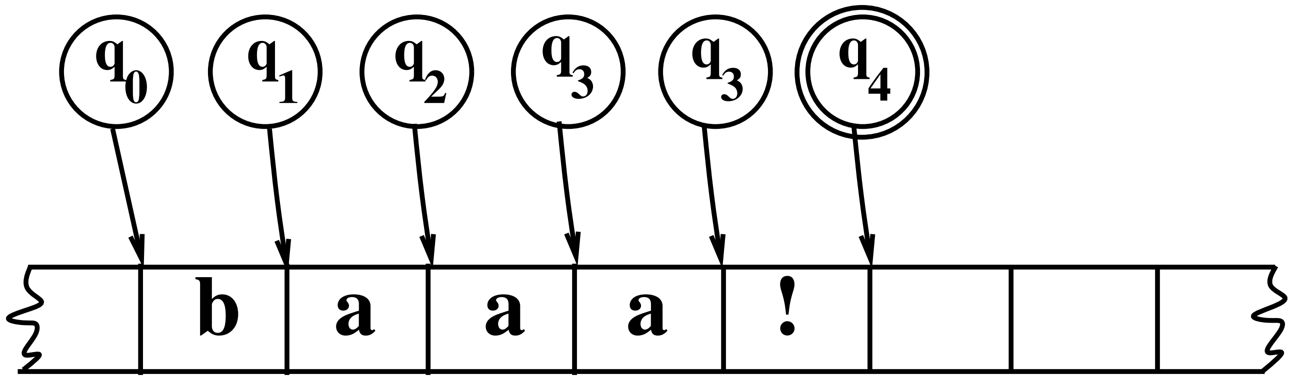
- Deterministic means that the code always knows what to do at each point in the process.
- d-recognize is essentially a crude table-driven interpreter.
- the recognition code is universal for all FSAs. To change to a new formal language, you merely change the alphabet and the table. The recognition code stays the same.
- thus, searching for a string using a RE involves compiling the RE into a table and passing the table to an interpreter.

An Example Trace

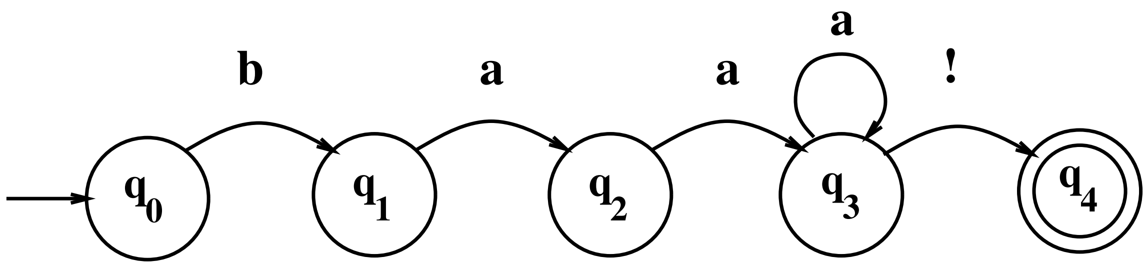


function D-RECOGNIZE(*tape, machine*) **returns** accept or reject

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


q0
q1
q2
q3
q3
q4
b
a
a
a
!

Another Example Trace



What about abc?

Formal Languages

A Formal Language is a set of strings composed of symbols from a finite set of symbols (the alphabet).

FSAs (and regular expressions) define formal languages (without having to explicitly enumerate the set).

Given a machine m (such as a particular FSA) L(m) means the formal language characterized by m.

- L(sheeptalk FSA) $= \{$baa!, baaa!, baaaa!, $\ldots\}$ (an infinite set)

Generative Grammars

The term Generative refers to the idea that FSAs can be viewed as generators of formal languages as well as acceptors. This dual view will pop up again and again.

To generate, you traverse the machine and write transition symbols on the tape rather than reading them.

Review

REs are compact textual representations of FSAs, themselves compact graphical representations of formal languages (sets of strings) called regular languages.

Recognition is the process of determining if an input is in the language accepted/generated/represented by a given FSA.

Recognition in deterministic machines is easy (table-driven).

FSAs can be useful tools for recognizing - and generating - *subsets* of natural language.

Review Examples

What would the machine that accepts the following dialect of sheep language (/baa+!/) look like?

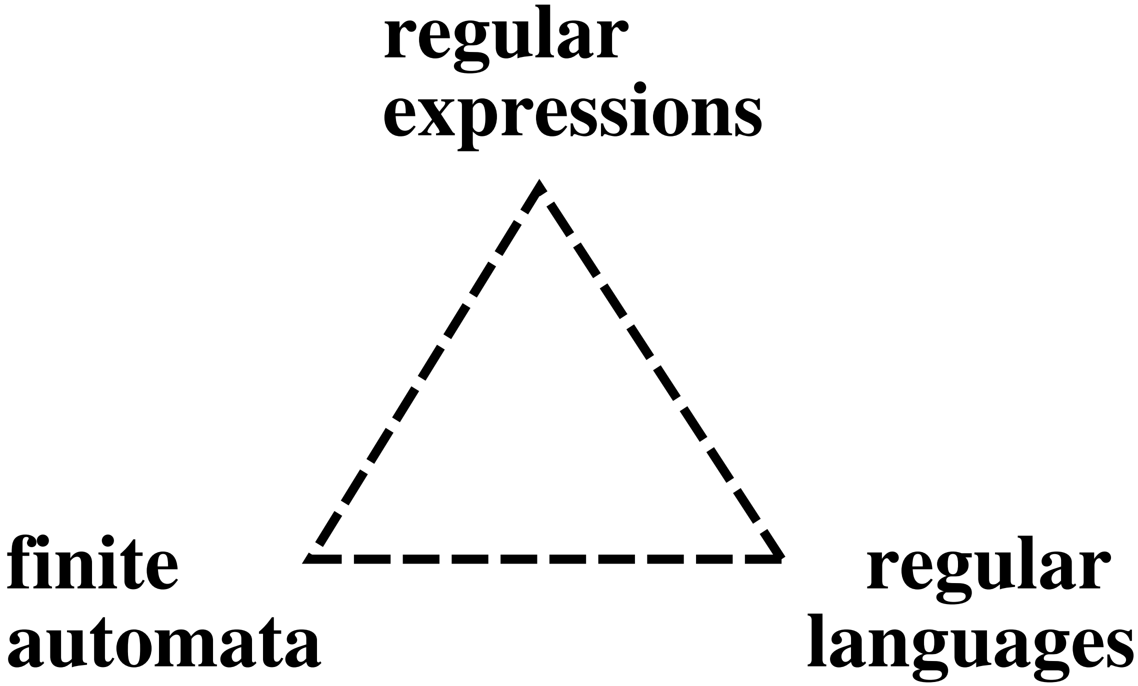
- /baa{49}!/

What does the following regular expression do to the input “DOES SHE LIKE CLASS”

```
s/([A-Z]+) +([A-Z]+)/ \2 \1 /
```

Three Views

There are three formally equivalent ways of (not including tables) looking at what we're doing.



Well Four

We'll talk later about representing regular languages with rules like these...

$S \rightarrow$ b a a A

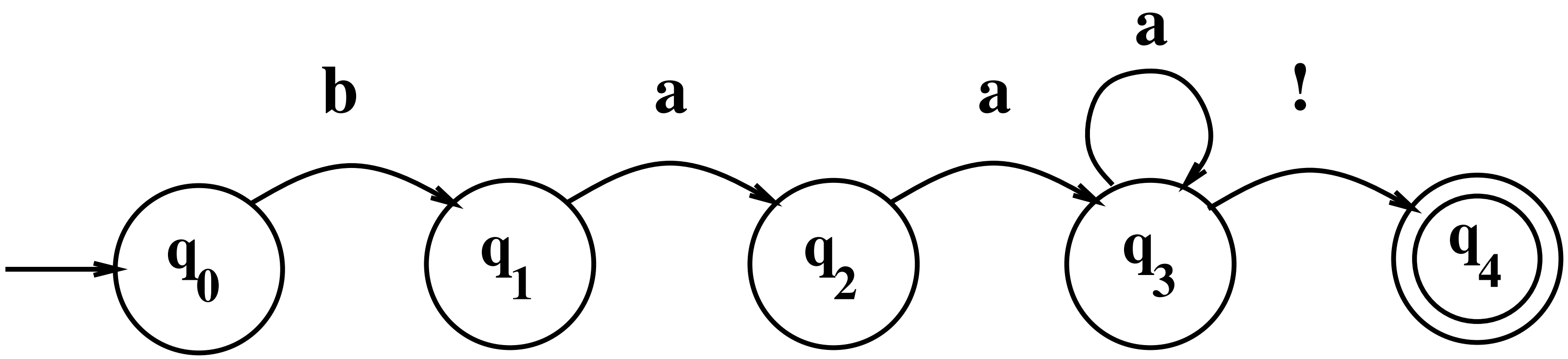
$A \rightarrow$ a A

$A \rightarrow$!

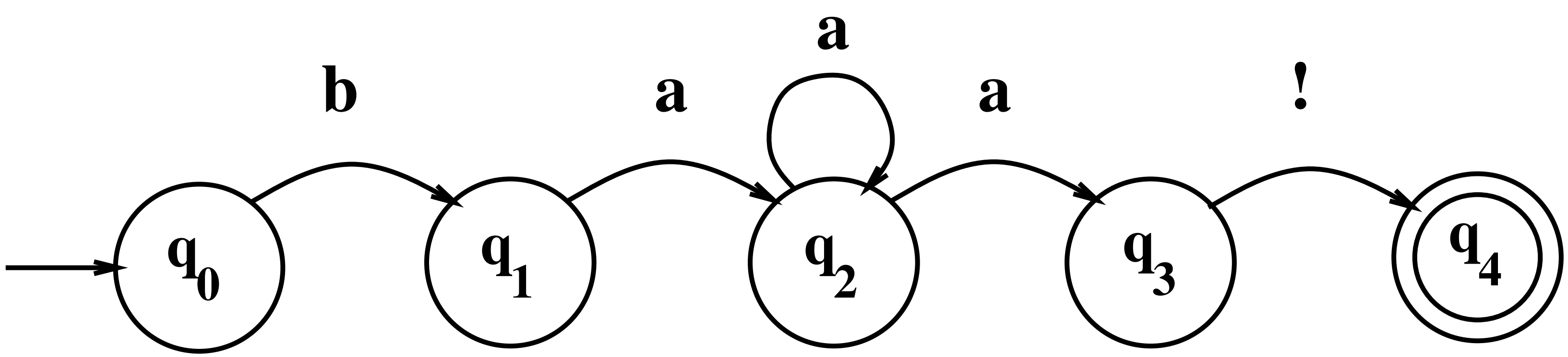
D-Recognize

```
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
```

Non-Deterministic FSAs (NFSAs)



- DFSA behavior is fully *determined* by current state and input

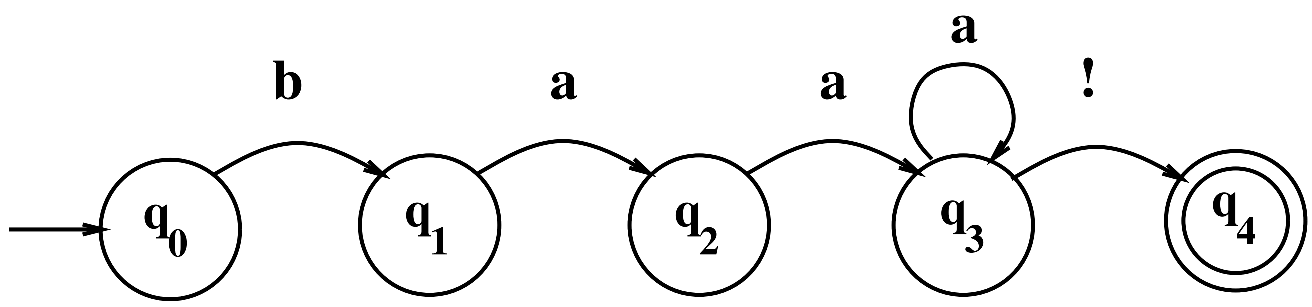


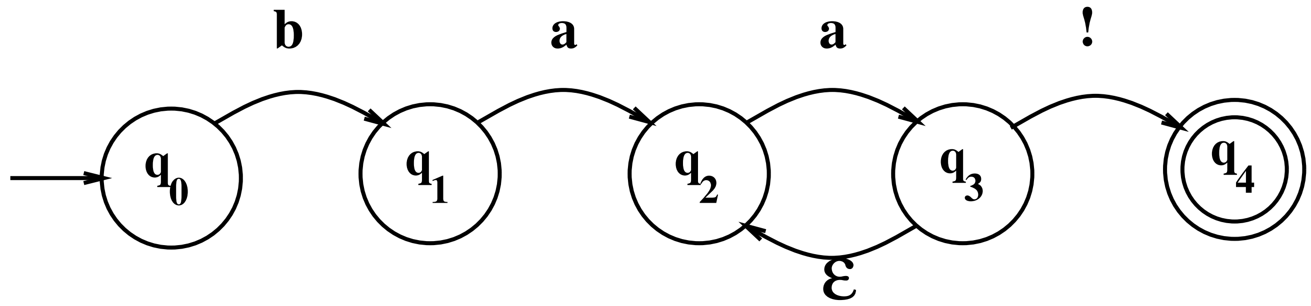
- NFSA1 is an automata with *decision points*

Non-Determinism (continued)

Another way is to use ϵ-transitions (arcs with no symbols)

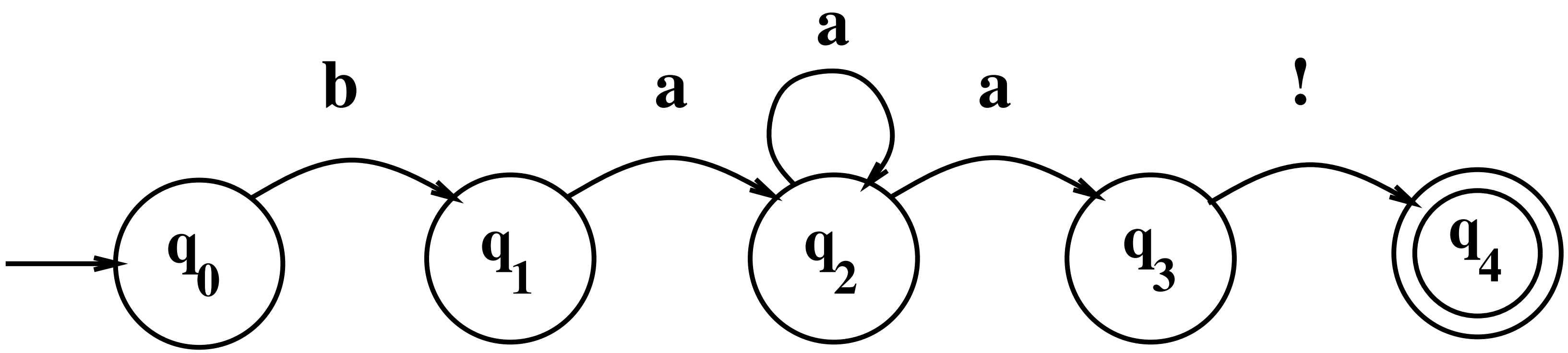
DFSA:





NFSA2: ϵ-transitions do not examine / advance the tape

NFSA Transition Tables



	Input			
State	b	a	!	ε
0	1	∅	∅	∅
1	∅	2	∅	∅
2	∅	2,3	∅	∅
3	∅	∅	4	∅
4:	∅	∅	∅	∅

- epsilon input column
- cells can be *lists* of nodes

Equivalence of D and ND Machines

Any ND machine can be converted to a D machine by a fairly simple construction (e.g., Hopcroft and Ullman 1979).

This means that formally they have the same power - they can recognize exactly the same classes of language. Non-determinism does not add formal power to FSAs.

It also means that one way of doing recognition would be to convert an ND-machine to a D-machine and then do recognition with D-Recognize.

Non-Deterministic Recognition

Back to recognition . . .

In a non-deterministic machine, there exists at least one path directed through the machine by a string in the language that leads to an accept condition.

There may exist paths driven by strings in the language that do no lead to an accept condition. Doesn't matter if some wrong choices lead to a reject condition.

In a non-deterministic machine no path directed through the machine by a string outside the language leads to an accept condition.

Non-Determinism (continued)

So... non-deterministic recognition succeeds whenever it is driven by an input to any accept condition.

However, being driven to a reject condition by an input does not imply it should be rejected.

To reject a string it has to be sure that there are no paths that can lead to an accept.

That means that the problem of non-deterministic recognition can be thought of as a search problem.

The Problem of Choice

Choice in non-deterministic models comes up again and again in NLP

Several standard solutions

- backup (search, this chapter)
  - mark input/state at choice points
  - if wrong choice, use marker to backup and try another choice
- lookahead
  - look ahead in the input to help make choice
- parallelism
  - look at all choices in parallel

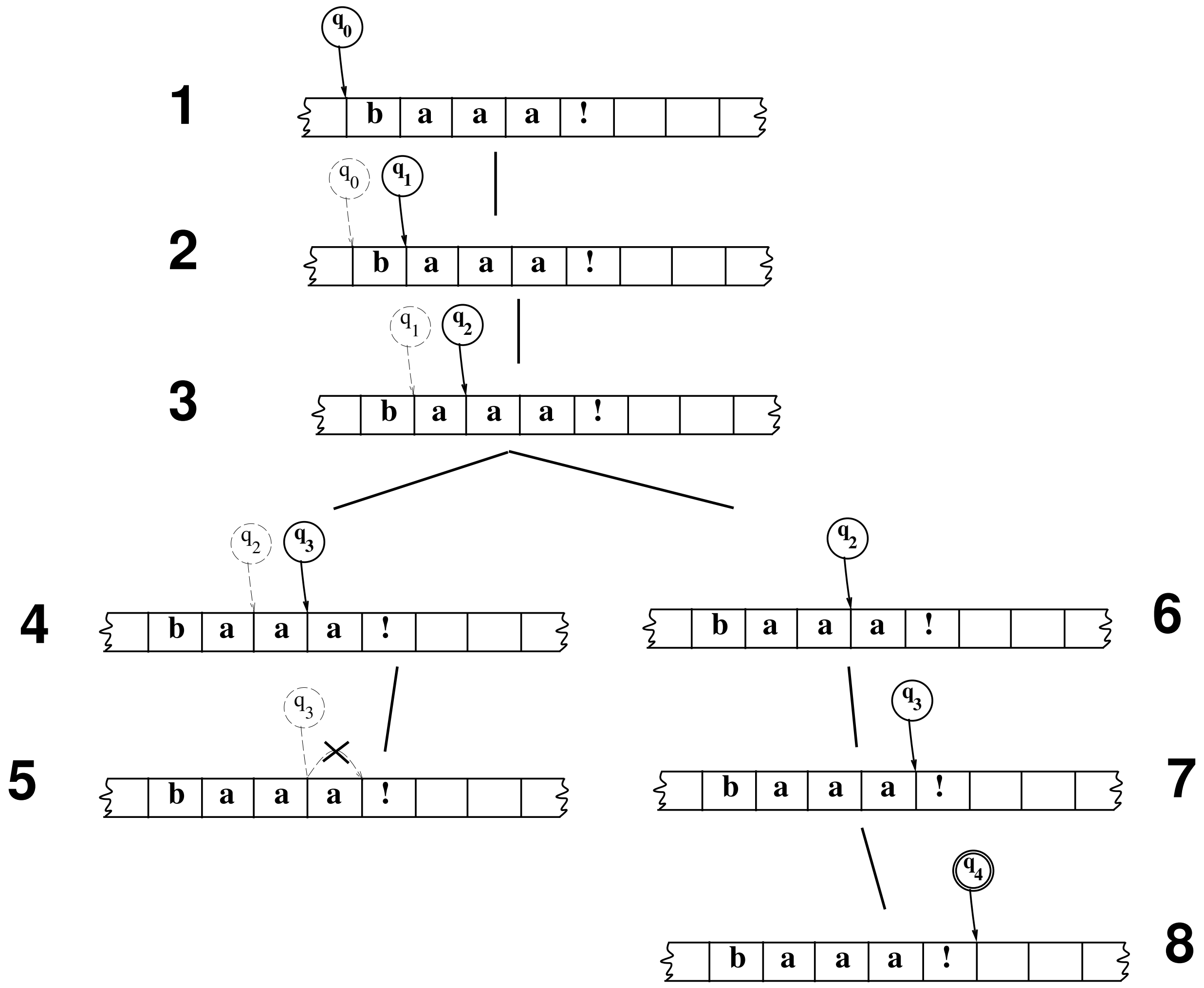
The Backup Approach

After a wrong choice leads to a dead-end (no input left, no legal transitions), return to a previous choice point and pursue another *unexplored* choice.

Thus, at each choice point, the search process needs to remember the search state:

- tape position
- FSA node (or *machine* state)

Backup Example



Key Points

A search-state is a pairing of a single machine-state with a position on the input tape.

By keeping track of not yet explored search-states, a recognizer can systematically explore all possible paths through a machine given some input.

ND-Recognize Code

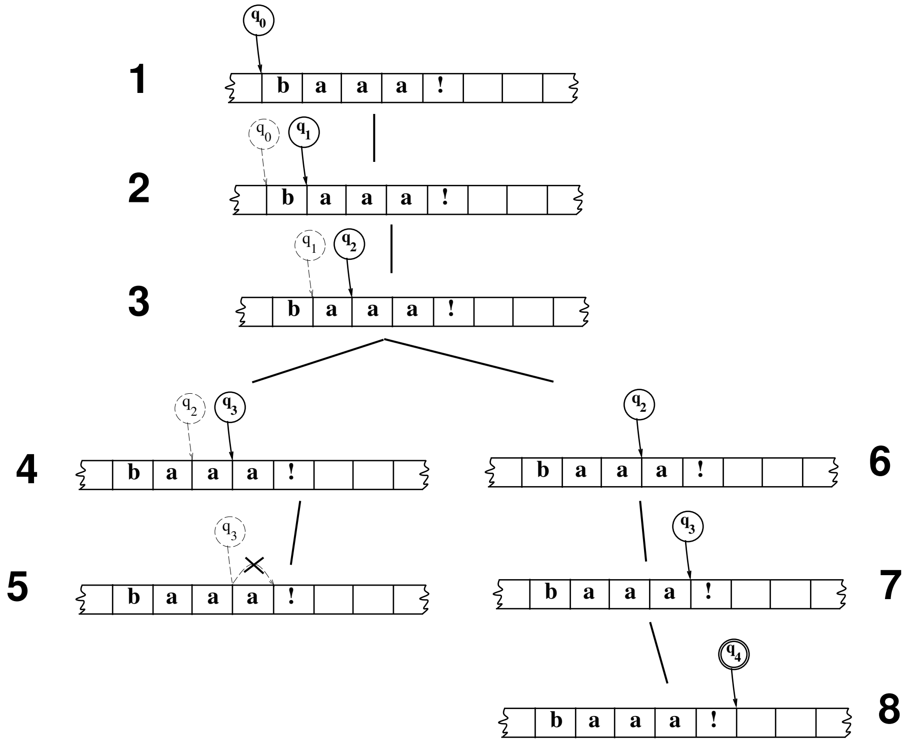
```
function ND-RECOGNIZE(tape, machine) returns accept or reject

  agenda ← {(Initial state of machine, beginning of tape)}
  current-search-state ← NEXT(agenda)
  loop
    if ACCEPT-STATE?(current-search-state) returns true then
       return accept
    else
       agenda ← agenda ∪ GENERATE-NEW-STATES(current-search-state)
    if agenda is empty then
       return reject
    else
       current-search-state ← NEXT(agenda)
  end

function GENERATE-NEW-STATES(current-state) returns a set of search-
states

  current-node ← the node the current search-state is in
  index ← the point on the tape the current search-state is looking at
  return a list of search states from transition table as follows:
    (transition-table[current-node,ε], index)
    ∪
    (transition-table[current-node, tape[index]], index + 1)

function ACCEPT-STATE?(search-state) returns true or false

  current-node ← the node search-state is in
  index ← the point on the tape search-state is looking at
  if index is at the end of the tape and current-node is an accept state of machine
then
    return true
  else
    return false
```

Why Bother

Given all of the above, non-determinism seems to be more trouble that its worth. Why introduce it at all?

- more intuitive to solve certain problems
- often lots smaller (n versus possibly 2^n)

Example Revisited



- each # is the current search state
- transition table for (q_2, a) examined twice, but tape pointer different

Recognition as Search

State-Space Search: Algorithms such as ND-Recognize, which systemically search for solutions

- problem definition creates a space of possible solutions
- goal is to explore the space
  - return answer when found
  - reject when space has been exhaustively explored

ND-Recognize as State-Space Search

Search states: pairings of machine states with tape position

State-space: all possible pairings given the machine

Search goal: find an accept state/end of tape pair

Ordering the Search

Note that NEXT is undefined in ND-Recognize

Depth-first search

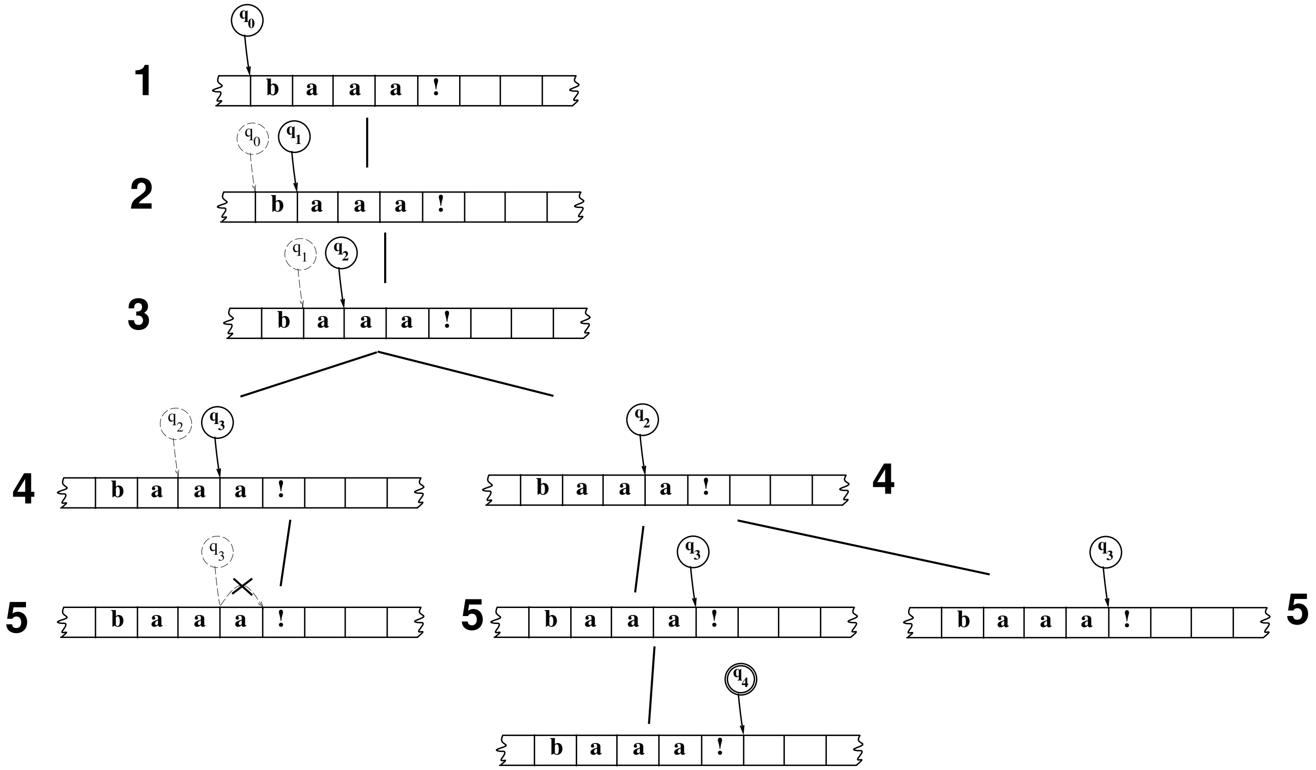
- most recently created states considered first
- agenda is a stack
- last in first out (LIFO)
- previous example trace

Breadth-first search

- states considered in creation order
- agenda is a queue
- first in first out (FIFO)

Dynamic programming / A*: CS1571

Breadth-First Trace



- instead of picking one choice and following it, examine all possible choices
- typically less efficient with respect to memory

Regular Languages

Equivalent

- the class of languages that are definable by regular expressions
- the class of languages that are characterizable by FSAs (either deterministic or not)

Formal Definition

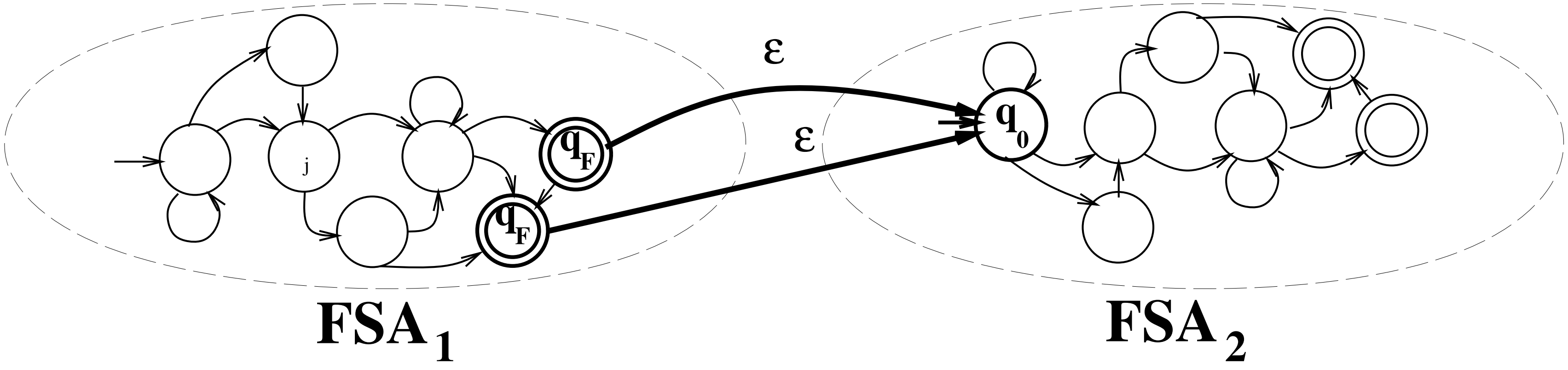
- if L_1 and L_2 are regular languages, so are the concatenation of L_1 and L_2, the union or disjunction of L_1 and L_2, and the Kleene closure of L_1
- thus, all the regular expression operators introduced in J&M Chapter 2 (except memory) can be implemented by these three operators

Fun With Automata

It turns out to be interesting (and useful) to consider the implications of various operations that combine regular languages into new languages.

In particular, we may want to know if the resulting combined language is still regular. Why?

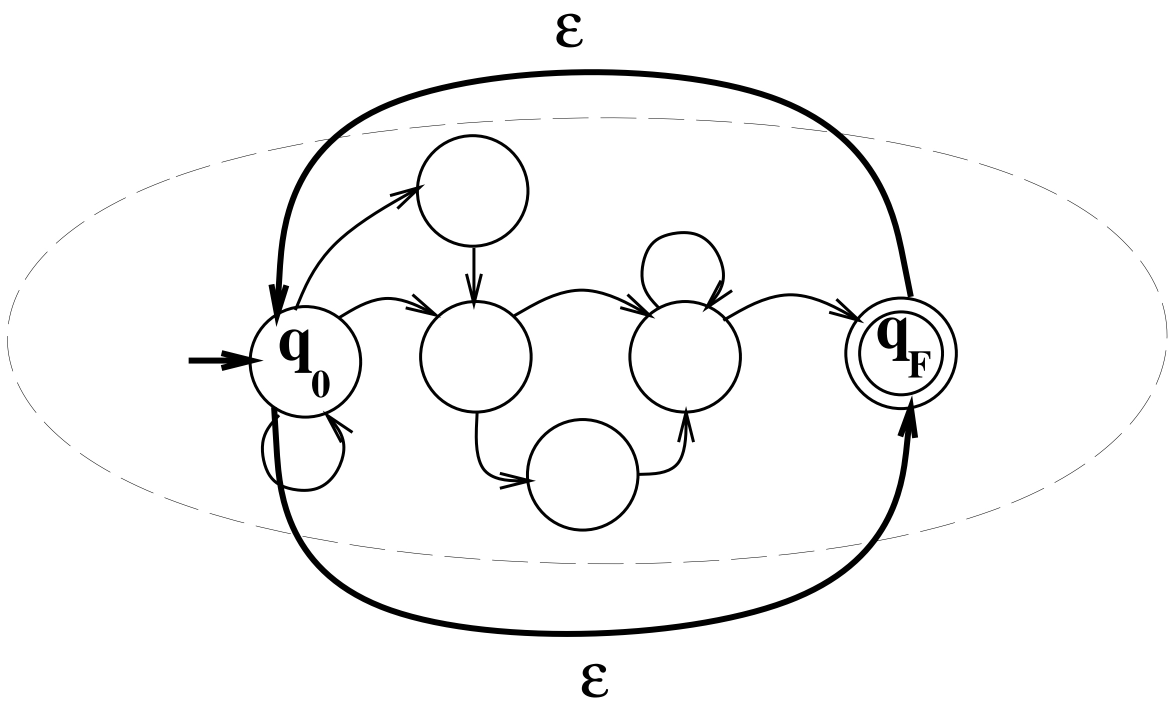
Concatenation



The primitive concatenation operation of a regular expression can be imitated by an automaton.

Use epsilon transitions to connect all final states of the first machine to the single initial state of the second.

Kleene *

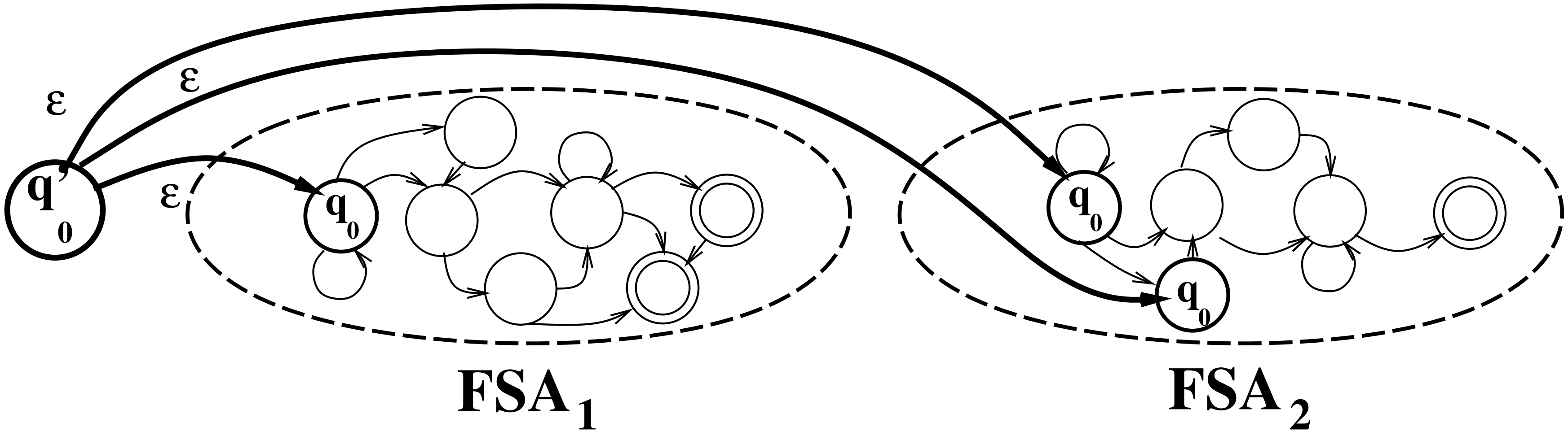


Connect all final states back to the initial state (repetition)

Directly link initial and final state (zero)

How would you implement Kleene + ?

Union



Add a new initial state with epsilon transitions to all the former initial states.

Closure (continued)

Regular languages are also closed under other operations (e.g., intersection) - see Jurafsky and Martin page 50.

Composing Automata

Consider the problem of recognizing dates like the following:

- January 27, 2000
- Wednesday January 27, 2000
- Monday February 29, 1900

One solution (a bad one) is to build one big complex regular expression.

What might a better solution be?

For Next Time

Homework 3

New topic: tagging