Natural Language Processing

Lecture 2—1/15/2015

Susan W. Brown

Today

- Regular expressions
- Finite-state methods

Regular Expressions and Text Searching

- Regular expressions are a compact textual representation of a set of strings that constitute a language
  - In the simplest case, regular expressions describe regular languages
    - Here, a language means a set of strings given some alphabet.
- Extremely versatile and widely used technology
  - Emacs, vi, perl, grep, etc.

Example

- Find all the instances of the word “the” in a text.
  - `/the/`
  - `/[tT]he/`
  - `/\b[tT]he\b/`

Errors

- The process we just went through was based on fixing two 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 with respect to evaluation for many tasks. 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).

3 Formalisms

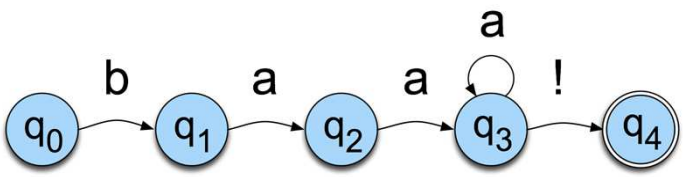
- Recall that I said that regular expressions describe languages (sets of strings)
- Turns out that there are 3 formalisms for capturing such languages, each with their own motivation and history
  - Regular expressions
    - Compact textual strings
      - Perfect for specifying patterns in programs or command-lines
  - Finite state automata
    - Graphs
  - Regular grammars
    - Rules

3 Formalisms

- These three approaches are all equivalent in terms of their ability to capture regular languages. But, as we'll see, they do inspire different algorithms and frameworks

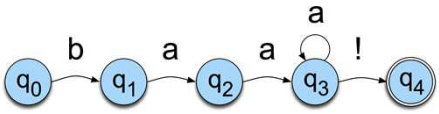
FSAs as Graphs

- Let's start with the sheep language from Chapter 2
  - /baa+!/



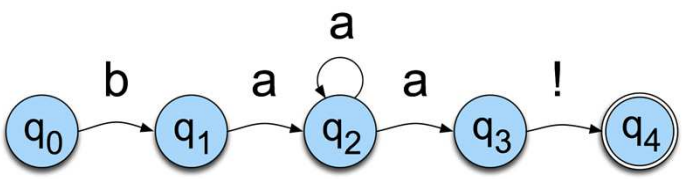
Sheep FSA

- 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

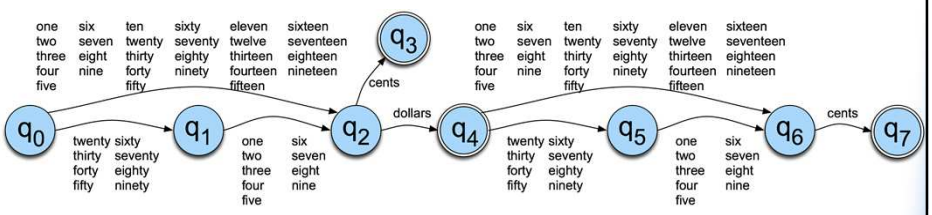
More Formally

- 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 states
  - A transition function that maps QxΣ to Q

About Alphabets

- Don't take term *alphabet* 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 may in turn have internal structure
  - Such as another FSA

Dollars and Cents

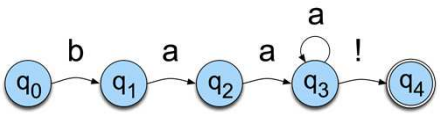


Yet Another View

- The guts of an FSA can ultimately be represented as a table

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

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

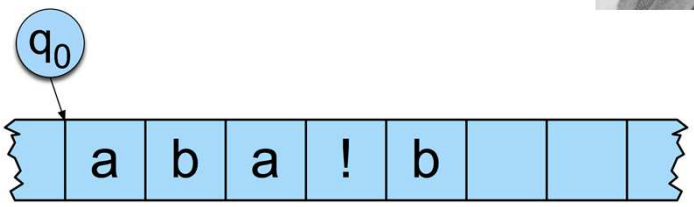


Recognition

- 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 to the same thing in the end

Recognition

- Traditionally, (Turing's notion) this process is depicted with an input string written on 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.

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; no ambiguity).
- 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.

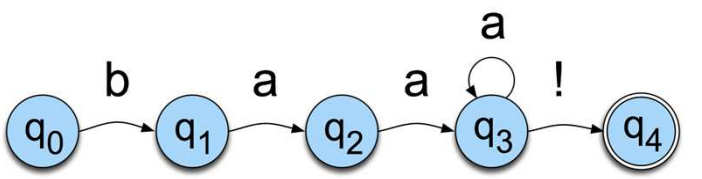
Key Points

- Crudely therefore… matching strings with regular expressions (ala 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 that implements D-recognize (or something like it)

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?

Non-Determinism



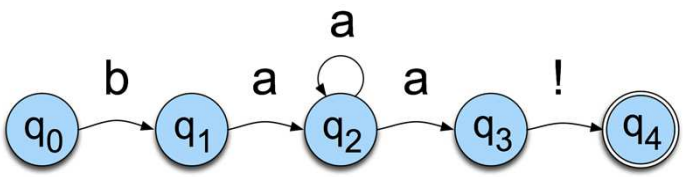
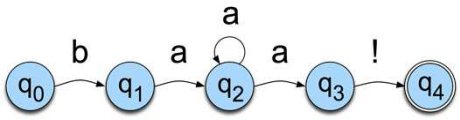


Table View

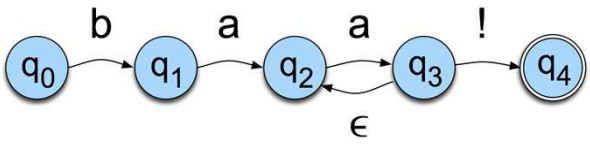
Allow multiple entries in the table to capture non-determinism

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



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 and can't characterize

ND Recognition

- Two basic approaches (used in all major implementations of regular expressions, see Friedl 2006)
  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/table as is).

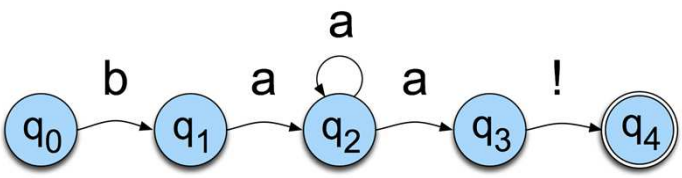
Non-Deterministic Recognition: Search

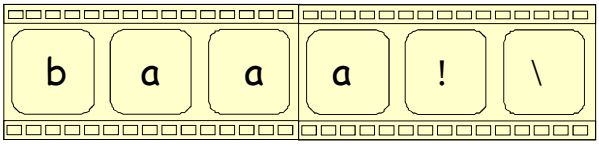
- In a ND FSA there exists at least one path through the machine for a string that is in the language defined by the machine.
- But not all paths directed through the machine for an accept string lead to an accept state.
- No paths through the machine lead to an accept state for a string not in the language.

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

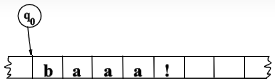


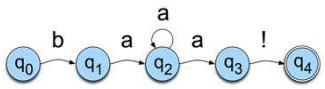


q_0 q_1 q_2 q_2 q_3 q_4

Example

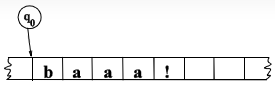
1



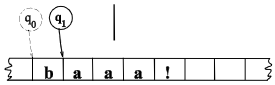


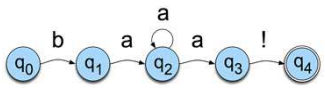
Example

1

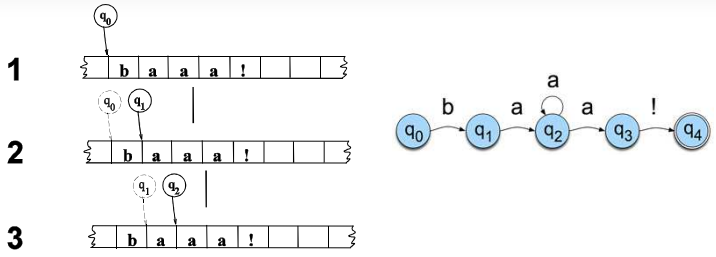


2

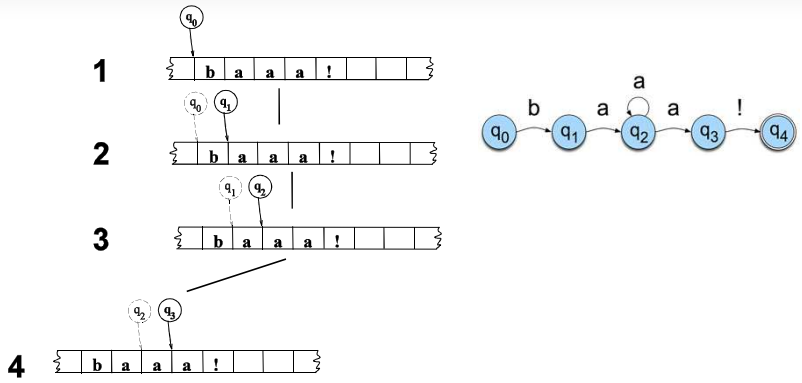




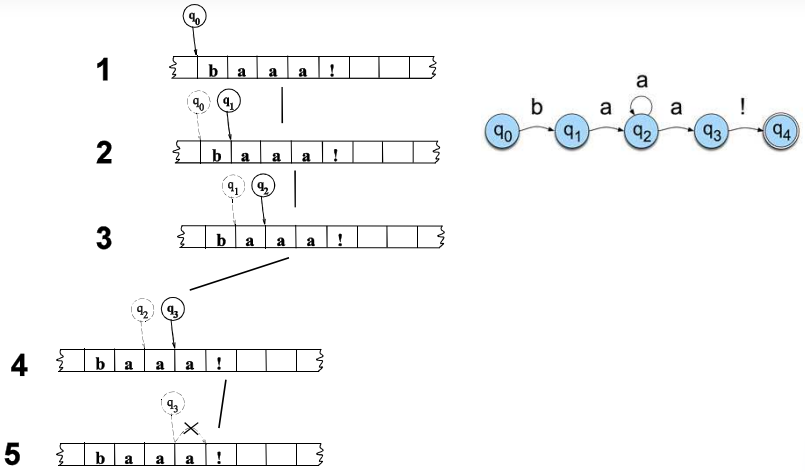
Example



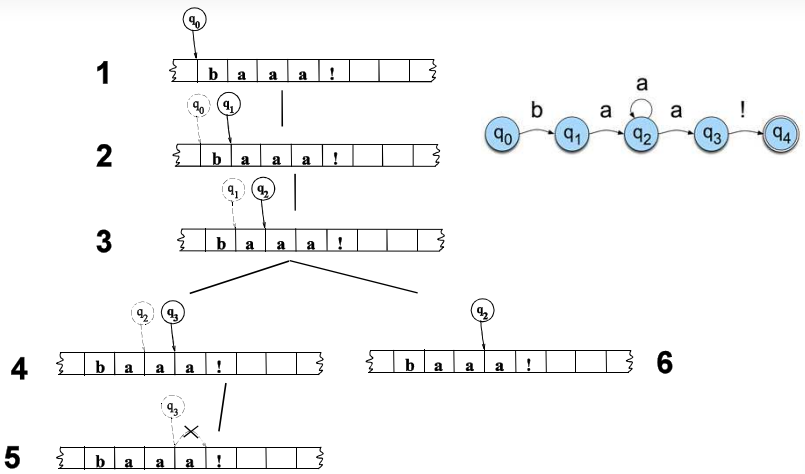
Example



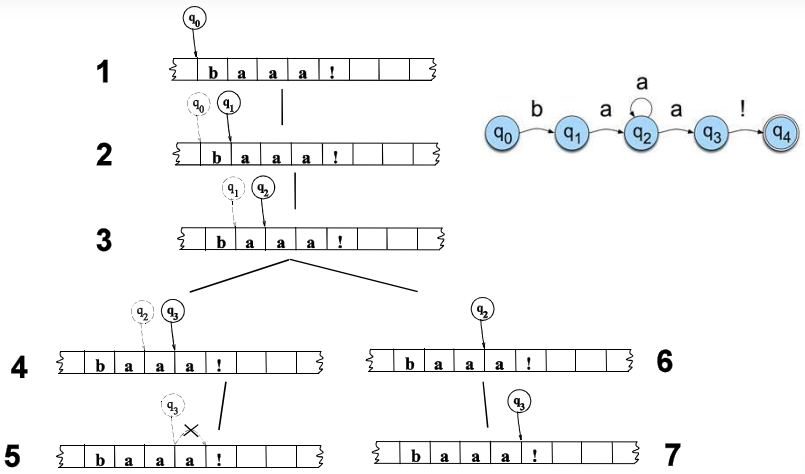
Example



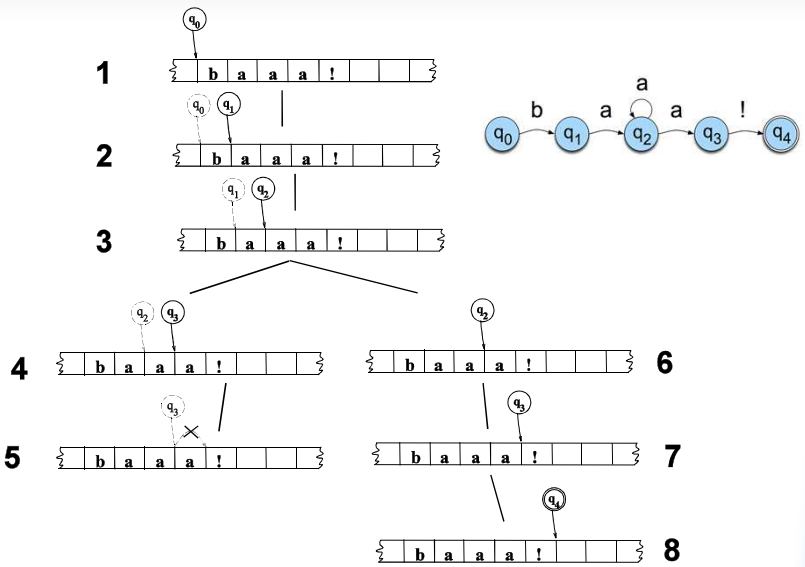
Example



Example



Example



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.

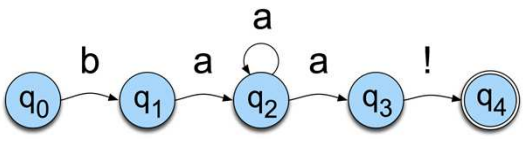
Why Bother?

- Non-determinism doesn't get us more formal power and it causes headaches so why bother?
  - More natural (understandable) solutions
  - Not always obvious to users whether the regex that they've produced is deterministic or not
    - Better to not make them worry about it

Admin Questions?

Converting NFAs to DFAs

- The Subset Construction is the means by which we can convert an NFA to a DFA automatically.
- The intuition is to think about being in multiple states at the same time. Let's go back to our earlier example where we're in state q_2 looking at an "a"



Subset Construction

- So the trick is to simulate going to both q2 and q3 at the same time
- One way to do this is to imagine a new state of a new machine that represents the state of being in states q2 and q3 at the same time
  - Let's call that new state {q2,q3}
    - That's just the name of a new state but it helps us remember where it came from
  - That's a subset of the original set of states
- The construction does this for all possible subsets of the original states (the powerset).
  - And then we fill in the transition table for that set

Subset Construction

- Given an NFA with the usual parts: Q, Σ, transition function δ, start state q_0, and designated accept states
- We'll construct a new DFA that accepts the same language where
  - States of the new machine are the powerset of states Q: call it Q_D
    - Set of all subsets of Q
  - Start state is $\{q_0\}$
  - Alphabet is the same: Σ
  - Accept states are the states in Q_D that contain any accept state from Q

Subset Construction

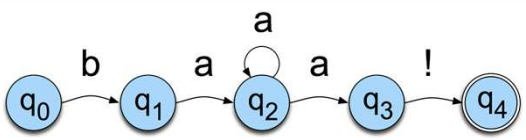
- What about the transition function?
  - For every new state we'll create a transition on a symbol α from the alphabet to a new state as follows
  - $\delta_D(\{q_1,...,q_k\}, \alpha)$ = is the union over all i = 1,...,k of $\delta_N(q_i, \alpha)$ for all α in the alphabet

Baaa!

- How does that work out for our example?
  - Alphabet is still "a", "b" and "!"
  - Start state is {q0}
  - Rest of the states are: {q1}, {q2},... {q4}, {q1,q2}, {q1,q3}... {q0,q1,q2,q3,q4,q5}
    - All 2^5-1 subsets of states in Q
- What's the transition table going to look like?

Lazy Method

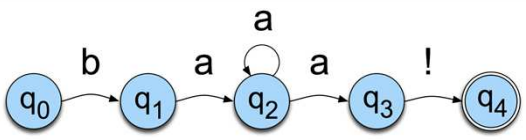
	b	a	!	
q0	q1			
q1		q2		
q2		q2,q3		
q3			q4	
q4				



	b	a	!	
{q0}				

Baaa!

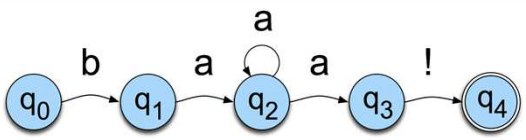
	b	a	!	
q0	q1			
q1		q2		
q2		q2,q3		
q3			q4	
q4				



	b	a	!	
{q0}	{q1}			

Baaa!

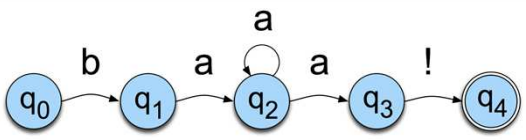
	b	a	!	
q0	q1			
q1		q2		
q2		q2,q3		
q3			q4	
q4				



	b	a	!	
{q0}	{q1}			
{q1}				

Baaa!

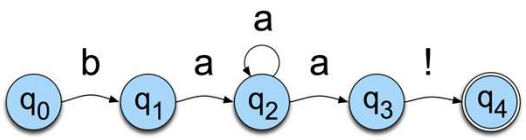
	b	a	!	
q0	q1			
q1		q2		
q2		q2,q3		
q3			q4	
q4				



	b	a	!	
{q0}	{q1}			
{q1}		{q2}		
{q2}				

Baaa!

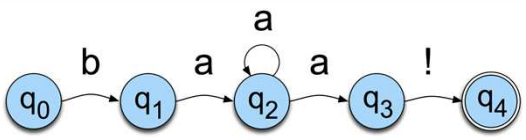
	b	a	!	
q0	q1			
q1		q2		
q2		q2,q3		
q3			q4	
q4				



	b	a	!	
{q0}	{q1}			
{q1}		{q2}		
{q2}		{q2,q3}		
{q2,q3}				

Baaa!

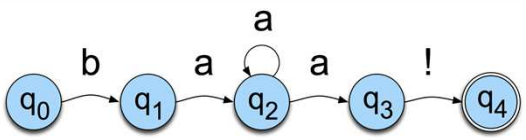
	b	a	!	
q0	q1			
q1		q2		
q2		q2,q3		
q3			q4	
q4				



	b	a	!	
{q0}	{q1}			
{q1}		{q2}		
{q2}		{q2,q3}		
{q2,q3}		{q2,q3}		

Baaa!

	b	a	!	
q0	q1			
q1		q2		
q2		q2,q3		
q3			q4	
q4				



	b	a	!	
{q0}	{q1}			
{q1}		{q2}		
{q2}		{q2,q3}		
{q2,q3}		{q2,q3}	{q4}	
{q4}				

Couple of Notes

- We didn't come close to needing 2^Q new states. Most of those were unreachable. So in theory there is the potential for an explosion in the number of states. In practice, it may be more manageable.
- Draw the new deterministic machine from the table on the previous slide... It should look familiar.

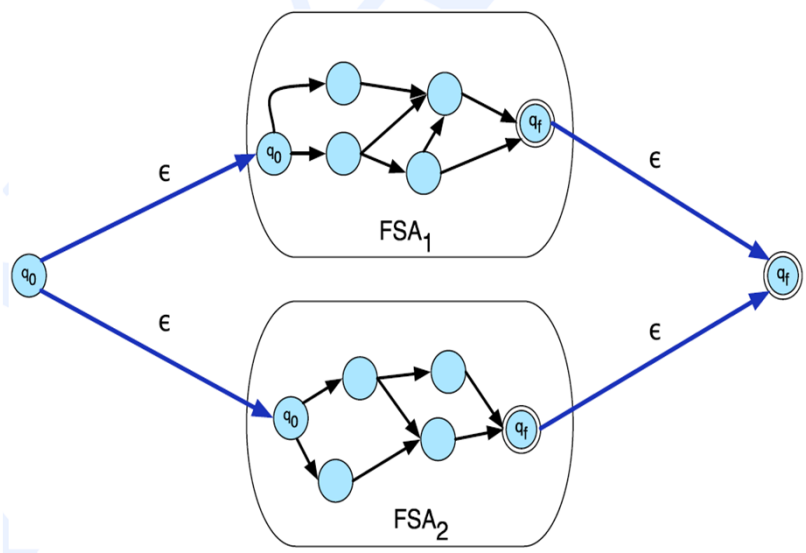
Compositional Machines

- Recall that formal languages are just sets of strings
- Therefore, we can talk about set operations (intersection, union, concatenation, negation) on languages
- This turns out to be a very useful
  - It allows us to decompose problems into smaller problems, solve those problems with specific languages, and then compose those solutions to solve the big problems.

Example

- Create a regex to match all the ways that people write down phone numbers. For just the U.S. that needs to cover
  - (303) 492-5555
  - 303.492.5555
  - 303-492-5555
  - 1-303-492-5555
  - (01) 303-492-5555
- You could write a big hairy regex to capture all that, or you could write individual regex's for each type and then OR them together into a new regex/machine.
- How does that work?

Union (Or)



Negation

- Construct a machine M2 to accept all strings not accepted by machine M1 and reject all the strings accepted by M1
  - Invert all the accept and not accept states in M1
- Does that work for non-deterministic machines?

Intersection (AND)

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

Problem Set 1

- From the Jurafsky and Martin book, end of chapter 2
- Problem 2.1; subsections 2, 4, and 6
- Problem 2.4
- Due Thursday, Jan. 22, in class

Next Week

- On to Chapter 3
  - Crash course in English morphology
  - Finite state transducers
  - Applications
    - Lexicons
    - Segmentation