http://xkcd.com/208/

Lecture 9
Regexes, Finite State Automata

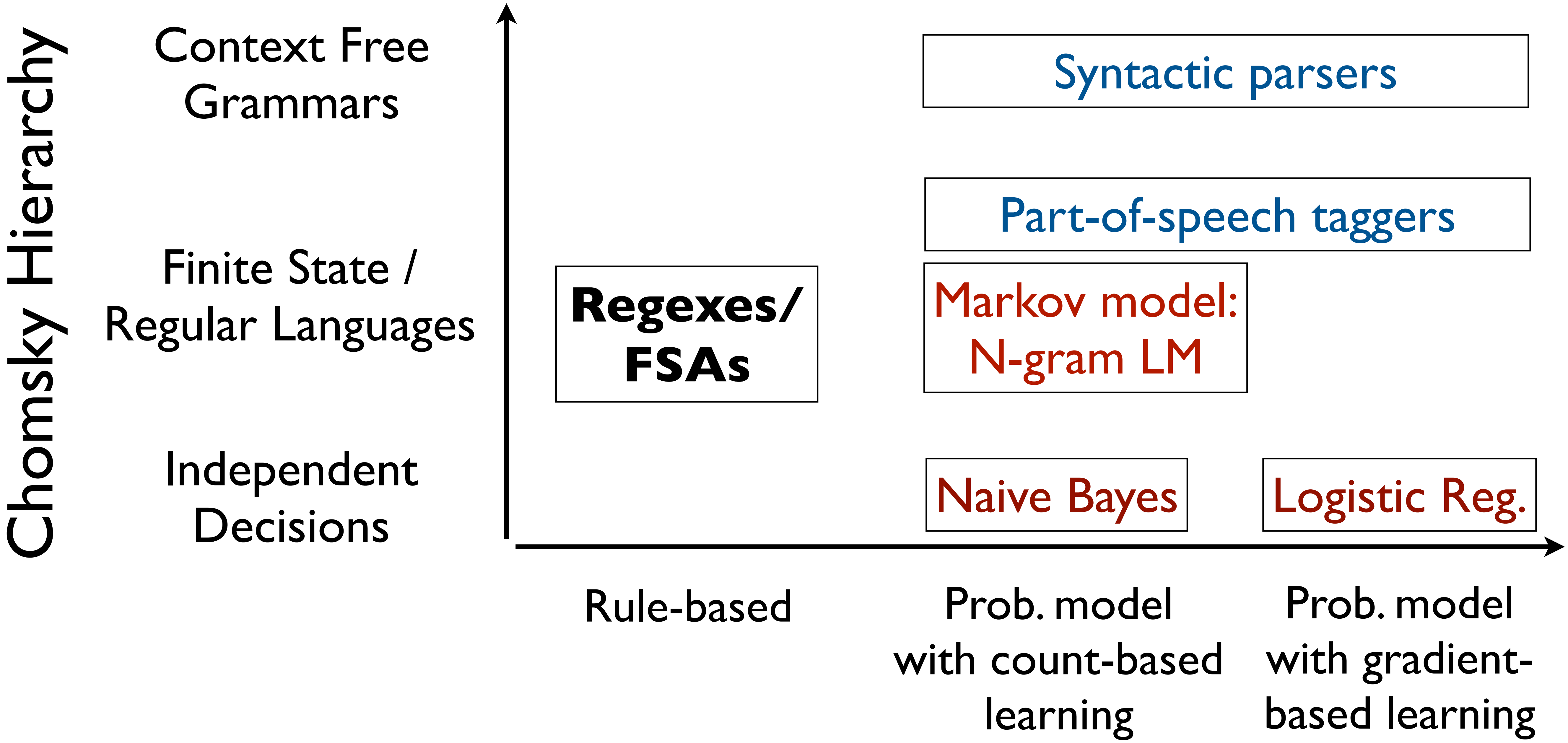
Intro to NLP, CS585, Fall 2014
http://people.cs.umass.edu/~brenocon/inlp2014/
Brendan O'Connor (http://brenocon.com)

- Exercise 5 out - due Thursday
    - http://people.cs.umass.edu/~brenocon/inlp2014/schedule.html
- PS2 coming
- Midterm review: Wed Oct 15?

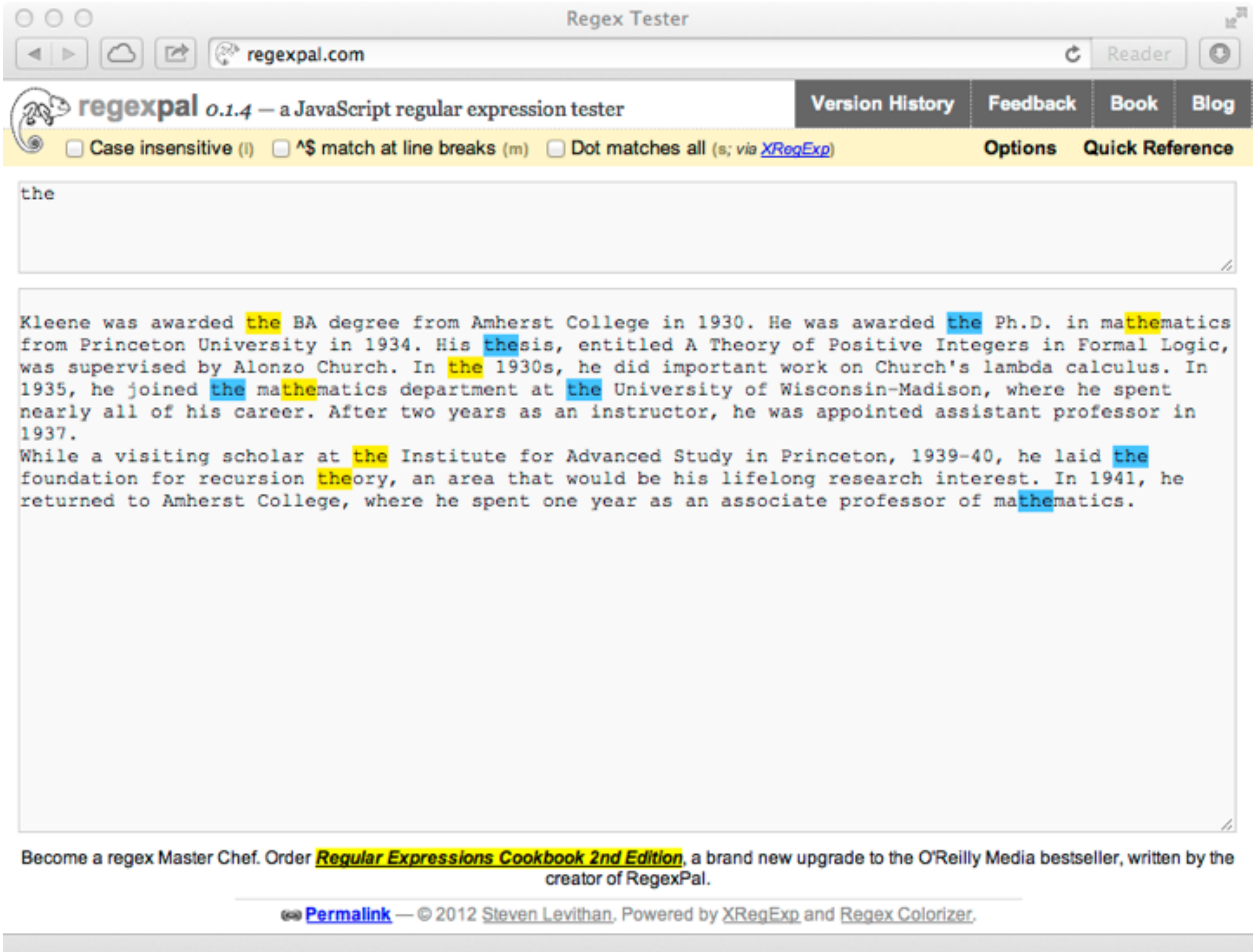
Today

- Regular expressions
- Finite-state automata

Computation/Statistics in NLP (in this course)



Regular expressions



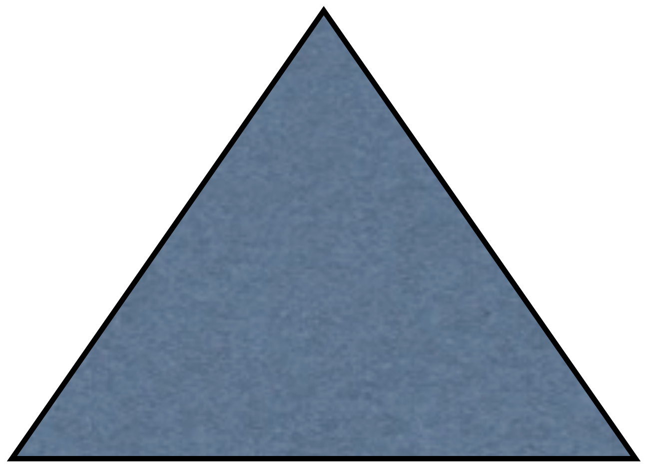
http://regexpal.com

Regular expressions

- Unix/Perl-style regular expressions
  - The current standard in all programming languages
  - *grep*, Python *re.search()*, JavaScript, Java...
  - Operations: Search, match, substitute
- Theory: regular expressions and finite-state automata
  - Just matching
- Next time: Finite-state transducers
  - Substitution

Equivalent ways of representing a regular language

Regular Language
the set of accepted strings



Finite-state Automaton
machinery for accepting

Regular Expression
a way to type the automata

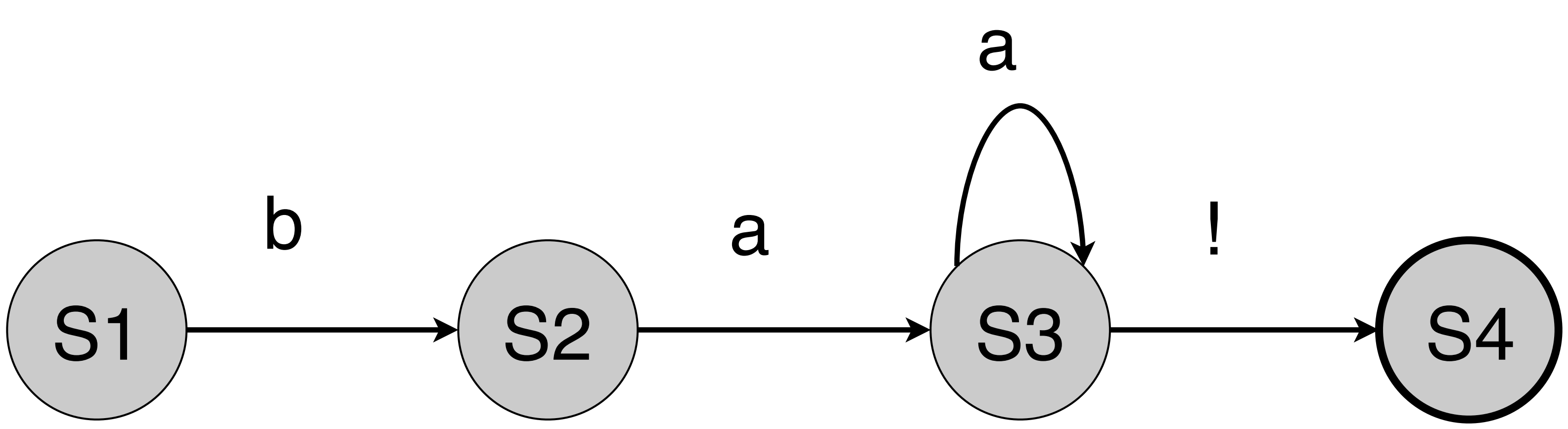
Example: Sheep Language

Set of strings

- In the language:
  "ba!", "baa!", "baaaaa!"
- Not in the language:
  "ba", "b!", "ab!", "bbaaa!", "alibaba!"

Finite-state Automata

is a state-machine, described by a list of rules



Rules
S1 → b S2
S2 → a S3
S3 → a S3
S3 → ! S4
S4 → *[[accept]]*

double circle
indicates "accept state"

Regular Expression `baa*!`

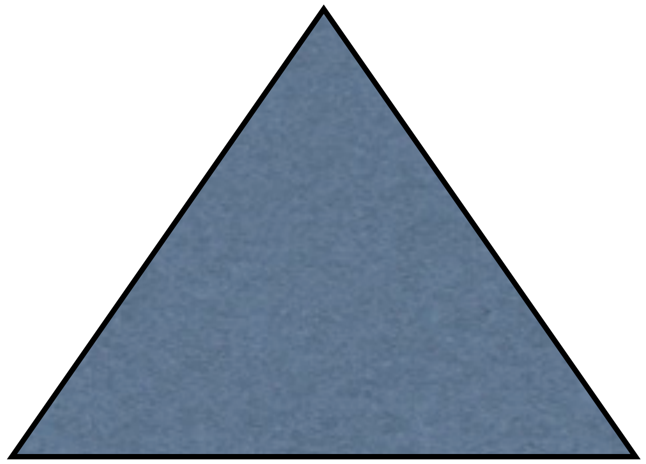
ndrew McCallum, UMass Amherst,
ncluding material from Chris Manning and Jason Eisner

* corresponds to the loop

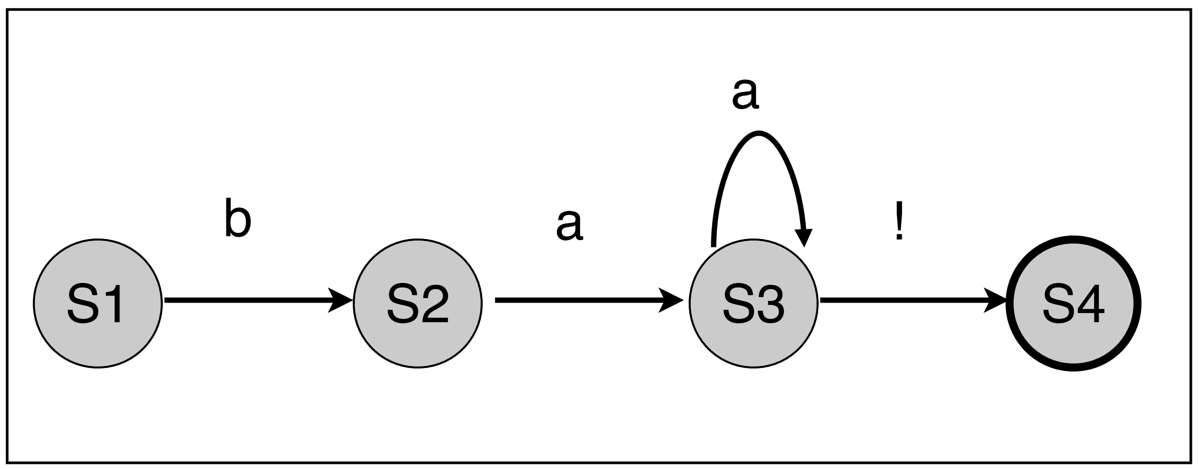
Equivalent ways of representing a regular language

Regular Language
the set of accepted strings

```
ba!
baa!
baaa!
...
```



Finite-state Automaton
machinery for accepting



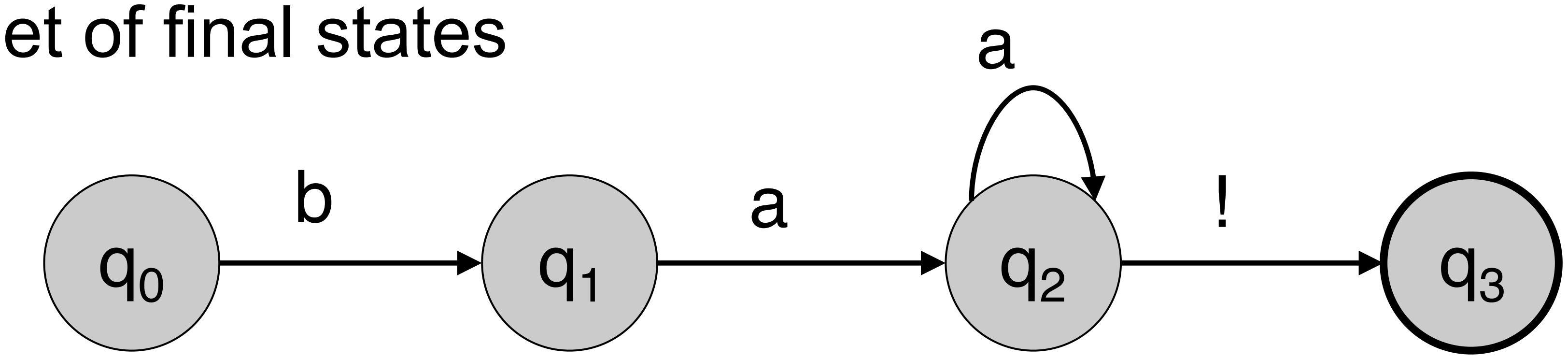
Regular Expression
a way to type the automata

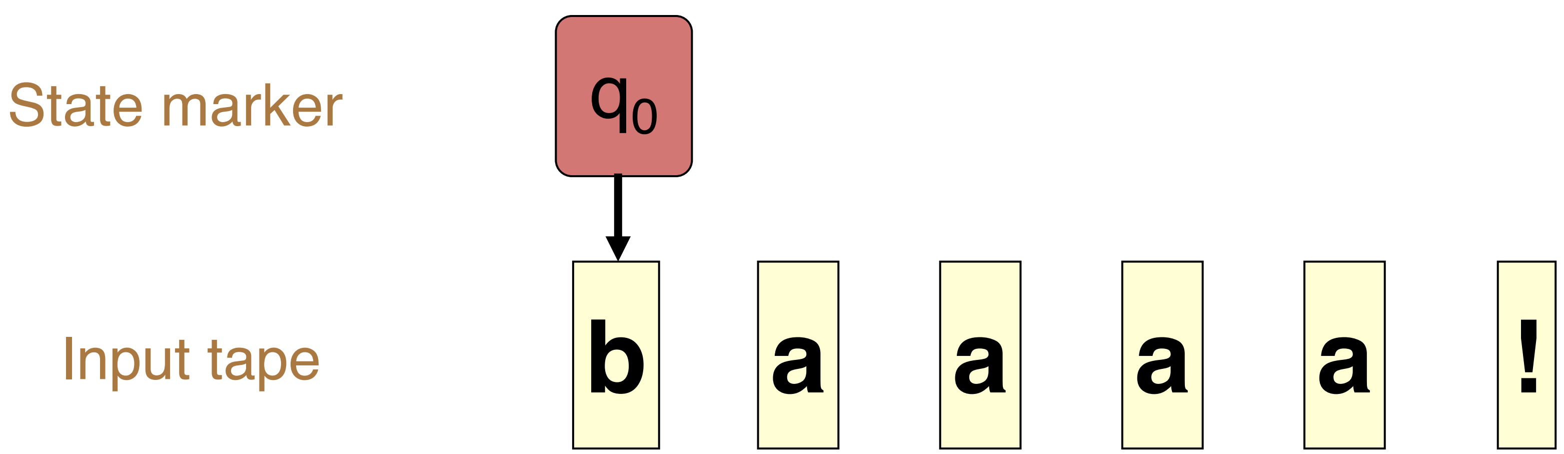
```
baa*!
```

Finite State Automata, more formally

- A finite state automata is a 5-tuple: $(Q, \Sigma, q_0, F, \delta(q,i))$
  - Q : finite set of N states, $q_0, q_1, q_2, \ldots q_N$
  - Σ : finite set of symbols: the vocabulary
  - $\delta(q,i)$: transition function, given state and input, returns next state (production rules)
  - q_0: the start state
  - F: the set of final states

The FSA





<u>Acceptance algorithm</u>:
Repeat:
- Traverse edge labeled with input to new state.
- If no such edge: REJECT
- ACCEPT if at one of the final states (here, q3).

Two types of characters in REs

- **Literal**
  - Every normal text character is an RE, and denotes itself.
- **Meta-characters**
  - Special characters that allow you to combine REs in various ways

REs: Fundamental operations

These <u>combine</u> or change regular languages into a new regular language

	Pattern	Matches
Concatenation	`abc`	*abc*
Disjunction	`a|b`	*a* *b*
	`(a|bb)d`	*ad* *bbd*
Kleene star	`a*`	ε *a aa aaa ...*
	`c(a|bb)*`	*ca cbba*

The empty string

cb ?

Operations as on FSAs: *ab* (Concatenation)

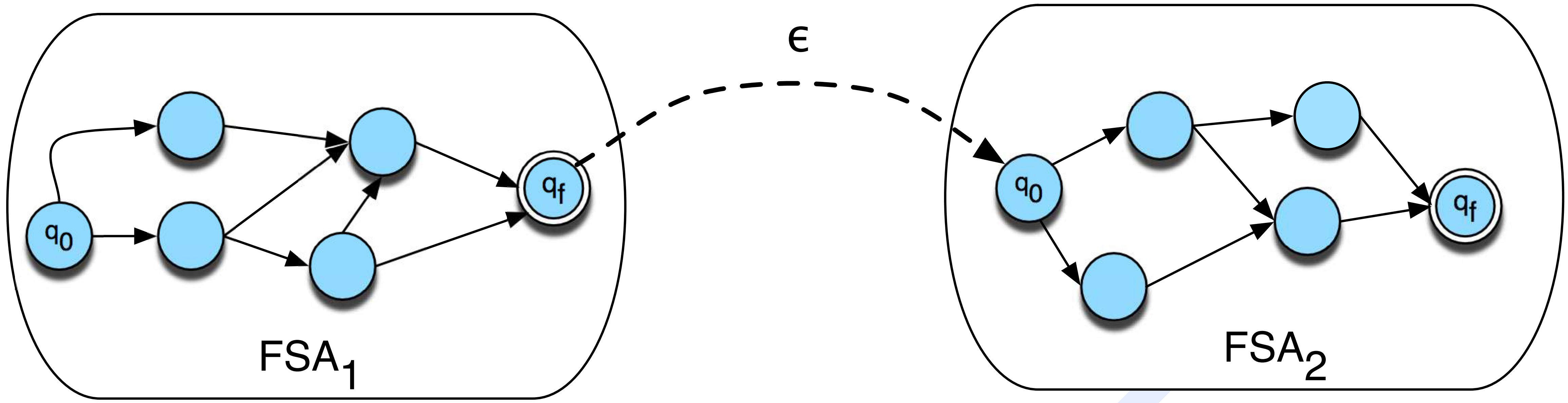


Figure 2.25 The concatenation of two FSAs.

Operations as on FSAs: *a** (Kleene star)

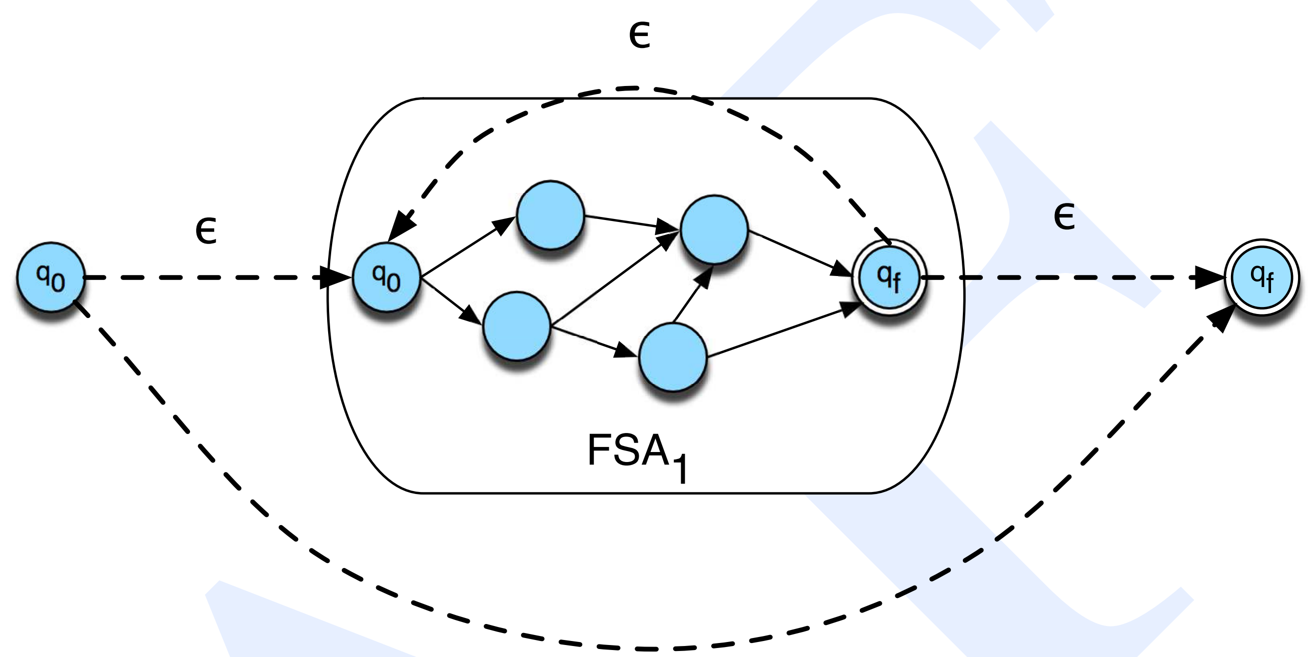
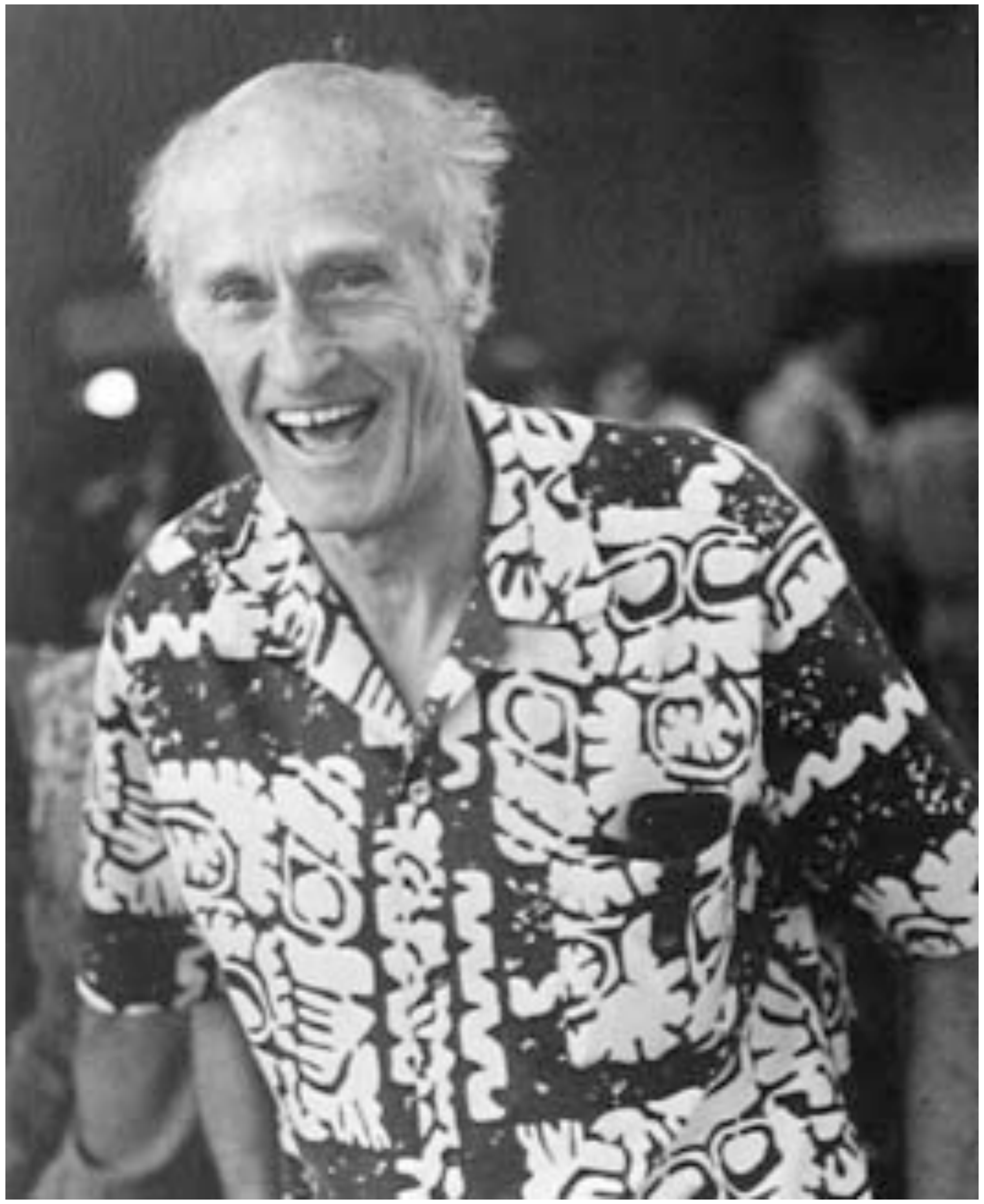


Figure 2.26 The closure (Kleene *) of an FSA.

Stephen Kleene, 1909 - 1994



Attended Amherst College!

Best known for *recursion theory* in mathematical logic, together with Alonzo Church, Alan Turing and others;

and for inventing regular expressions.

Practical Applications of RegEx's

- Web search
- Word processing, find, substitute
- Validate fields in a database (dates, email addr, URLs)
- Searching corpus for linguistic patterns
  - and gathering statistics...

- Finite state machines extensively used for
  - acoustic modeling in speech recognition
  - information extraction (e.g. people & company names)
  - morphology
  - ...

Unix/Perl-style RE basics

	Pattern	Matches
Character Concat	**went**	went
	Precedence marker ↓	
Alternatives	**(go|went)**	go went
	[aeiou]	a o u
disjunc. negation	**[^aeiou]**	b c d f g
wildcard char	**.**	a z &
Loops & skips	**a***	ε a aa aaa ...
one or more	**a+**	a aa aaa
zero or one	**colou?r**	color colour

ndrew McCallum, UMass Amherst,
ncluding material from Chris Manning and Jason Eisner

e.g. https://docs.python.org/2/library/re.html

Unix/Perl-style RE basics

- Aliases (shorthand)
  - **\d** digits [0-9]
  - **\D** non-digits [^0-9]
  - **\w** alphanumeric [a-zA-Z0-9_]
  - **\W** non-alphanumeric [^a-zA-Z0-9_]
  - **\s** whitespace [\t\n\r\f\v]
- Examples
  - `\d+ dollars` 3 dollars, 50 dollars, 982 dollars
  - `\w*oo\w*` food, boo, oodles
- Escape character
  - `\` is the general escape character; e.g. `\.` is not a wildcard, but matches a period `.`
  - if you want to use `\` in a string it has to be escaped `\\`

e.g. https://docs.python.org/2/library/re.html

Unix/Perl-style RE fancy stuff

- Anchors. a.k.a. "zero width characters".
- They match positions in the text.
  - `^` beginning of line
  - `$` end of line
  - `\b` word boundary, i.e. location with `\w` on one side but not on the other.
  - `\B` negated word boundary, i.e. any location that would not match `\b`
  - `\bthe\b` *the* NOT *together*

- Counters {1}, {1,2}, {3,}
  - `go{2,7}al` *goooooal* NOT *goal*

ndrew McCallum, UMass Amherst,
ncluding material from Chris Manning and Jason Eisner

e.g. https://docs.python.org/2/library/re.html

- Demo: things in tweets
  - phone numbers
  - dates, times
  - emoticons

- Some nice options for *grep*
  - grep --color=always
  - grep -P

Emoticons

useful trick: decompose the regex with nice names

`r'...'` in python just means "raw" string

```
NormalEyes = r'[:=]'
Wink = r'[;]'

NoseArea = r'(|o|O|-)'   ## rather tight precision, \S might be reasonable...

HappyMouths = r'[D\)\]]'
SadMouths = r'[\(\[]'
Tongue = r'[pP]'
OtherMouths = r'[doO/\\]'  # remove forward slash if http://'s aren't cleaned

Emoticon = (
    "("+NormalEyes+"|"+Wink+")" +
    NoseArea +
    "("+Tongue+"|"+OtherMouths+"|"+SadMouths+"|"+HappyMouths+")"
)
```

https://github.com/brendano/tweetmotif/blob/master/emoticons.py

half the battle in maintainability is just decomposing the rules with nice names. no one does this when you have the hacky perl mentality, but you totally can. here's one i wrote for emoticons.

note there are precision/recall tradeoffs with every decision you make when writing rules like this. for example, forward-slash for emoticon mouth gives horrible false positives if there are URLs in the text :/

Dates (Xerox FST syntax)

```
# DateParser.script
# Copyright (C) 2004  Lauri Karttunen

define Day     [{Monday} | {Tuesday} | {Wednesday} | {Thursday} |
                {Friday} | {Saturday} | {Sunday}] ;
define Month29 {February};
define Month30 [{April} | {June} | {September} | {December}];
define Month31 [{January} | {March} | {May} | {July} | {August} |
                {October} | {December}] ;
define Month   [Month29 | Month30 | Month31];

# Numbers from 1 to 31
define Date    [OneToNine | [1 | 2] ZeroToNine | 3 [%0 | 1]] ;
# Numbers from 1 to 9999
define Year [OneToNine ZeroToNine^<4];
# Day or [Month and Date] with optional Day and Year
define AllDates [Day | (Day {, }) Month { } Date ({, } Year)];

[...]
define ValidDates [AllDates & MaxDays & LeapDates];
```

open-source implementation: http://code.google.com/p/foma/wiki/ExampleScripts

Tuesday, September 30, 14

Lauri Karttunen is famous for lots of finite-state morphology stuff. i think this is a demo script he wrote for identifying dates in a text with an FST.

actually nearly all of it is just FSA-like. the key bit for how you use it is the bottom. it spits out these XML-ish tags around the strings matching ValidDates pattern. this is what FST's can do.

(note they do more complicated stuff for morphology)

this is actually an open-source implementation of Xerox's pattern language for FST's. it is fairly new. i believe it compiles to target OpenFST, a lower level algorithmic library for weighted FST's; it does all the unions and minimization and other finite state stuff, so compiles this pattern script into an FST that does date recognition. (OpenFST, in turn is a clone of the old AT&T finite state libraries.)

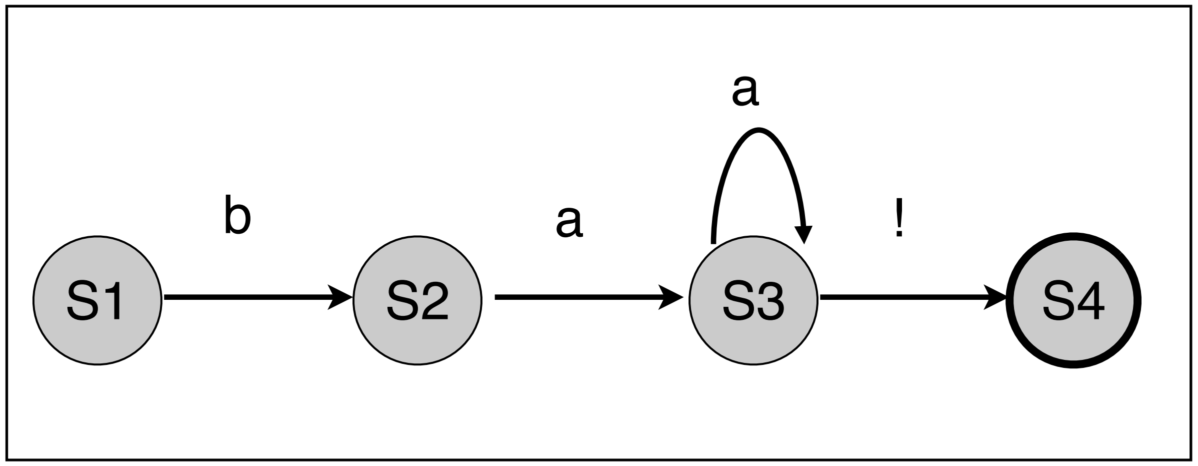
Evaluation

- Regexes are *rule-based systems*: text classifiers designed by hand, based on human knowledge.
- Statistical evaluation is not just for machine learning!
- If you have labeled data, evaluate with accuracy, precision, and recall, etc.
    - e.g. look at regex's matches in data. Percentage that are correct is the precision.
    - Precision is easy. Recall? Typically can only label within a high-recall filter (e.g. all \d matches for dates).
- Regexes very widely used for quick and dirty analysis without time for evaluation...
    - e.g. data cleaning

- Is there a relationship?
  - A regular language is a **boolean language model**: every string is either in it or not.
    - Described by an FSA.
  - A Markov model is a **probabilistic language model**: every string has a probability, and they sum to 1.
    - Described by P(nextword | context) probabilities?

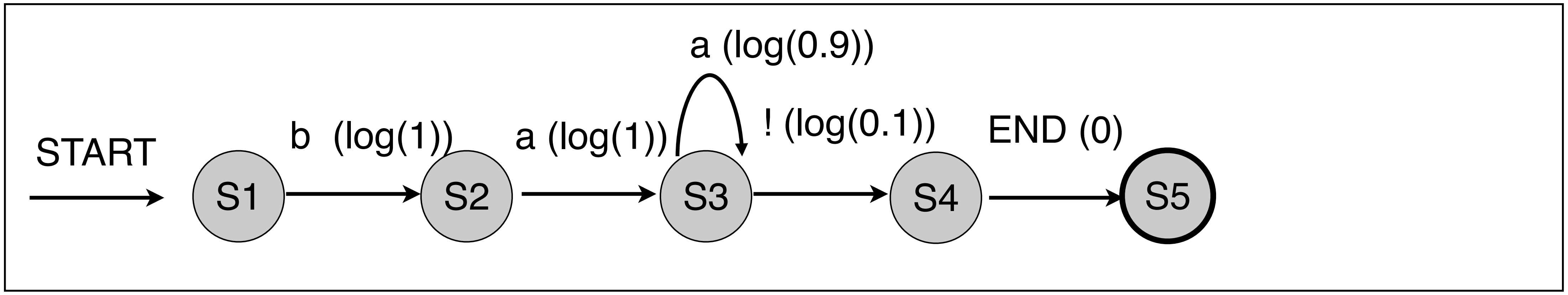
Boolean FSA A = (Q, Σ, q_0, F, δ(q,i))

accepts(A, string) --> {0, 1}



Weighted FSA B: with log-probability edge weights

logprob(B, string) --> (-inf, 0]



logprob of string = sum of traversed weights

What is the set of strings this gives non-zero probability to? Does it give the same prob for all strings in this set?