Speech and Language Technology

Formal Languages, Regular Expressions and Finite-State Automata

Topics

- Formal Languages in brief
- Regular Expressions
- Finite-State Automata (FSA)
- Non-Deterministic FSA (NFSA or NFA)
- Regular and Non-Regular Languages

Source

- Speech and Language Processing: An introduction to natural language processing, computational linguistics, and speech recognition. Daniel Jurafsky & James H. Martin. Draft of January 19, 2007.

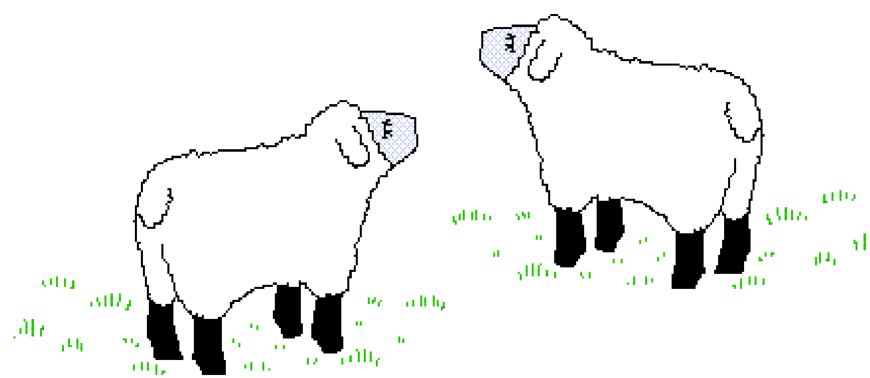
- An updated draft is available here: http://www.cs.vassar.edu/~cs395/docs/2.pdf

Formal Languages

- A **formal language** *L* over an **alphabet** Σ is a set of **words** (strings) over that alphabet.
  - L = {w1, w2, w3, ….}
  - Σ = {s1, s2, s3, …}

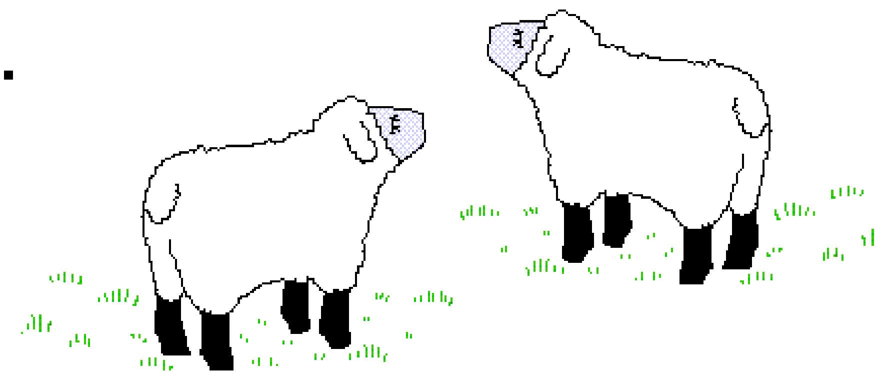
Formal Languages

- A **formal language** *L* over an **alphabet** Σ is a set of **words** (strings) over that alphabet.
  - L = {w1, w2, w3, ….}
  - Σ = {s1, s2, s3, …}
- For example, consider sheep-talk:
  - L = {“baa!”, “baaa!”, “baaaa!”, “baaaaa!”…}
  - Σ = {‘b’,’a’,’!’}



Formal Languages

- A **formal language** *L* over an **alphabet** Σ is a set of **words** (strings) over that alphabet.
  - L = {w1, w2, w3, ….}
  - Σ = {s1, s2, s3, …}
- For example, consider sheep-talk:
  - L = {“baa!”, “baaa!”, “baaaa!”, “baaaaa!”…}
  - Σ = {‘b’,’a’,’!’}
- L and Σ can be infinite.



Regular Expressions

- First developed by Kleene (1956)
- A regexp is a formula in a special language that is used for specifying classes of strings.

Regular Expressions

- First developed by Kleene (1956)
- A regexp is a formula in a special language that is used for specifying classes of strings.
- By definition, any regexp characterizes a language.

Regular Expressions

- First developed by Kleene (1956)
- A regexp is a formula in a special language that is used for specifying classes of strings.
- By definition, any regexp characterizes a language.
- Simple examples:
  - /ab/ – {“ab”}
  - /a[bc]/ – {“ab”,“ac”}
  - /ab./ – {“aba”,“abb”,“abc”,“abd”,…}

Regular Expressions – Use

- Regular Expressions are widely used for pattern recognition in search applications.
- General idea: the user specifies a regxp – a pattern that stands for a set of strings – and the application finds all matches in a given corpus.
- In a typical search application, each line that contains a match of the regexp is returned entirely.
- Implementation in unix-based systems: grep
- Examples will follow.

Regular Expressions – Syntax

- A regexp is sequence of characters:
  - /ab/
  - /a[bc]/
- Slashes are **not** part of a regexp definition; they are used to clarify what the boundaries of the expression are.
- A regexp can consist of a single character (e.g. /!/) or a sequence of characters (/urgl/)
- Regular expressions are **case sensitive.**

Regular Expressions – simple examples

- Examples (only the first match is marked):

Regexp	Example Patterns Matched
/woodchucks/	"interesting links to <u>woodchucks</u> and lemurs"
/a/	"M<u>a</u>ry Ann stopped by Mona's"
/Claire says,/	""Dagmar, my gift please," <u>Claire says,</u>"
/song/	"all our pretty <u>song</u>s"
/!/	""You've left the burglar behind again<u>!</u>" said Nori"

- Note that a blank space (character 0x20) can be used as is in a regexp (example 3).

Regular Expressions – Disjunction

- Disjunction of characters:
  - A string of characters inside the braces specify a disjunction of characters to match.
  - Examples:

Regexp	Match
/[wW]oodchuck/	Woodchuck or woodchuck
/[abc]/	‘a’, ‘b’, or ‘c’
/[1234567890]/	Any digit

Regular Expressions – Ranges

- Ranges are useful to simplify a cumbersome notation.
- They are defined using the dash (‘–’) character:

Regexp	Match	Example Patterns Matched
/[A–Z]/	An uppercase letter	“we should call it ‘Drenched Blossoms’”
/[a–z]/	A lowercase letter	“my beans were impatient to be hoed!”
/[0–9]/	A digit	“Chapter 1: Down the Rabbit Hole”

Regular Expressions – Negation

- Square brackets opened by the caret character – ‘^’ –can be used to specify characters that cannot be matched by a regexp:

Regexp	Match (single characters)	Example Patterns Matched
/[ˆA-Z]/	not an uppercase letter	“Oyfn pripetchik”
/[ˆSs]/	neither ‘S’ nor ‘s’	“I have no exquisite reason”
/[eˆ]/	either ‘e’ or ‘ˆ’	“look up ˆ now”
/aˆb/	the pattern ‘aˆb’	“look up aˆb now”

Regular Expressions – Predefined Ranges

- The regexp syntax includes some predefined ranges:

Regexp	Expansion	Match
/\d/	/[0-9]/	Any digit
/\D/	/[^0-9]/	Any non-digit
/\w/	/[a-zA-Z0-9_]/	Any alphanumeric or underscore
/\W/	/[^\w]/	A non-alphanumeric
/\s/	/[\r\t\n\f]/	Whitespace (space, tab)
/\S/	/[^\s]/	Non-whitespace

- Note: /\t/ stands for the tab character, /\n/ stands for new line, /\r/ stands for carriage return and /\f/ stands for page break.

Regular Expressions – Repetition

- The regexp syntax supports various kinds of repetitions:
  - To specify that a character (or a sequence of characters) may appear zero or one time, use the question mark ('?'):

Regexp	Match	Example Patterns Matched
/woodchucks?/	woodchuck or woodchucks	"woodchuck is"
/colou?r/	color or colour	any colour you like

Regular Expressions – Repetition

- The regexp syntax supports various kinds of repetitions:
  - To specify that a character (or a sequence of characters) may appear zero or more times, use the asterisk mark ('*') – called also Kleene* – pronounced as "cleany star":

Regexp	Match	Example Patterns Matched
/Wood*chucks/	woochuck or woodchucks or wooddchucks or …	"woochucks are bad, but woodchucks are nice"
/baaa*!/	baa! or baaa! or baaaa!...	"And then we heard another baaaa!..."

Regular Expressions – Repetition

- The regexp syntax supports various kinds of repetitions:
  - To specify that a character (or a sequence of characters) may appear one or more times, use the plus mark ('+') – called also Kleene+:

Regexp	Match	Example Patterns Matched
/Wood+chucks/	woodchucks or wooddchucks or woodddchucks or …	"woochucks are bad, but <u>woodchucks</u> are nice"
/baa+!/	baa! or baaa! or baaaa!...	"And then we heard another <u>baaaa!</u>..."

Regular Expressions – Repetition

- Summary:

*	zero or more occurrences of the previous char or expression
+	one or more occurrences of the previous char or expression
?	exactly zero or one occurrence of the previous char or expression
{n}	*n* occurrences of the previous char or expression
{n,m}	from *n* to *m* occurrences of the previous char or expression
{n,}	at least *n* occurrences of the previous char or expression

Regular Expressions – Repetition

- The regexp syntax supports various kinds of repetitions:
  - To specify specific amounts of repetitions, use the curly brackets:

Regexp	Match
/a{3}b{2}ca/	aaabbca
/a{3,}b{2}ca/	aaabbca or aaaabbca or aaaaabbca or …
/a{3,4}b{2}ca/	aaabbca or aaaabbca
/ba{3,}!/	baaa! or baaaa! or baaaaa!...

Regular Expressions – ‘.’

- The period character – ‘.’ – serves as a wildcard expression that matches any single character (except a carriage return):

Regexp	Match	Example Patterns
/beg.n/	Any string comprised of a single character between ‘beg’ and ‘n’.	began begin beg’n
/beg.*n/	Any string begins with ‘beg’ followed by one or more characters and ends with ‘n’.	begn begabcden begun beguun
/beg\.n/	The string ‘beg.n’	beg.n

Regular Expressions – Grouping

- Grouping of a sequence of characters allows us to define patterns with repeated and/or alternating sequences.
- Grouping is done by parenthesis.
- Patterns with repeated sequences:

Regexp	Match
/a(ba)+c/	abac or ababac or abababac or …
/(a(bc)+)*c/	c or abcc or abcbcc or …

Regular Expressions – Grouping

- Patterns with alternating sequences:

Regexp	Match
/gupp(y|ies)/	guppy or guppies
/b(i|ou)nd/	bind or bound

- Notice the use of pipe ‘|’ to separate the alternating sequences.
- Note that if the regexp is simple a list of alternating sequences then grouping is not required: /dog|cat/ matches ‘dog’ or ‘cat’.

Regular Expressions – Anchors

- Special characters that anchor regexps to particular places in a string.
- Line boundaries:
  - Beginning of line: ^
  - End of line: $
- Word boundaries: \b

Regexp	Match	
/^The/	the word *The* only at the start of a line	The bus was late
/ˆThe dog\.$/	The exact line 'The dog.'	The dog.
/\bthe\b/	the word *the*	Others than the...

Regular Expressions – Operator Precedence

- Why does /the*/ match ‘theeee’ and not ‘thethe’?
- Why does /the|any/ match ‘the’ or ‘any’ and not ‘theny’?
- The answers are in the operator precedence hierarchy defined for regular expressions:

Operator Precedence Hierarchy	
Parenthesis	( )
Counters	* + ? {}
Sequences and Anchors	the ^my end$
Disjunction	|

Regular Expressions – Greediness

- Consider the regexp /[a–z]*/ matched against the string 'hello'.
- The regexp can match zero or more letters and hence it's interpretation is apparently ambiguous.
- The ambiguity is resolved by favoring the *largest* string that can be matched, i.e. 'hello'.
- We say that patterns are greedy in the sense of expanding to cover as much of a string as they can.

Regular Expressions – Escaping

- Escaping is needed when meta-characters like ‘*’ or ‘.’ need to be matched as they are without being interpreted according to their special role in the regexp syntax
- Regexps escaping is done by the backslash character – ‘\’.

Escaped character	Character to be matched
\.	.
*	*
\+	+

Regular Expressions – Summary

- A regexp is a formula in a special language that is used for specifying classes of strings.
- Any regexp characterizes some language.
- A typical search application takes a document and a regexp as an input and returns the list of lines from the document in which the regexp can be matched.

Regular Expressions – Summary

- Regexp: /woodchucks?/
- Text:

Imagine that you have become a passionate fan of woodchucks.

Desiring more information on this celebrated woodland creature, you turn to your favorite Web browser and type in woodchuck.

Your browser returns a few sites.

You have a flash of inspiration and type in woodchucks.

Regular Expressions – Summary

- Regexp: /woodchucks?/ (– {woodchuck,)
- Text: woodchucks}

Imagine that you have become a passionate fan of woodchucks.

Desiring more information on this celebrated woodland creature, you turn to your favorite Web browser and type in woodchuck.

Your browser returns a few sites.

You have a flash of inspiration and type in woodchucks.

Regular Expressions – More

- Resources:
  - http://www.regular-expressions.info/
  - http://en.wikipedia.org/wiki/Regular_expression
  - http://www.zytrax.com/tech/web/regex.htm

Finite State Automata

- Finite State Automata are a specific type of state machines: A set of states and transitions that may reach an *Accept* or *Reject* state according to a given input.

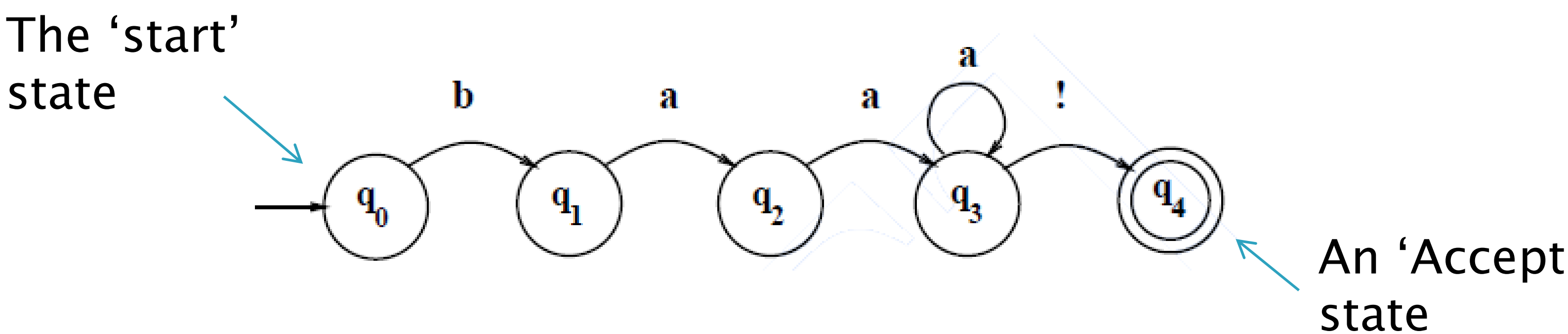
- Finite State Automata are commonly used to recognize formal languages and are computationally equivalent to regular expressions.

- Any language that a regexp can characterize, an FSA can characterize as well (and vice versa)

- Singular: Automaton; Plural: Automata

Finite State Automata

- Visually, finite state automata are drawn as graphs with nodes that stand for the states and links that stand for the transitions per input. For example:



- Q: What language does this automaton recognize?

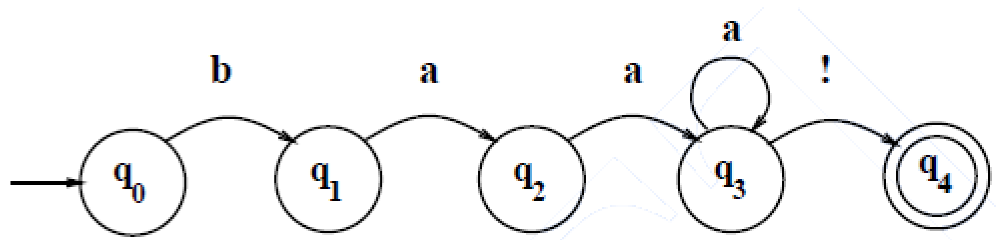
Finite State Automata

- Formally, an FSA is defined as follows:
  - $Q = q_0 q_1 q_2 \ldots q_{N-1}$ a finite set of N **states**
  - Σ – a finite **input alphabet** of symbols
  - q_0 – the **start state**
  - F – the set of **accepting (final) states**, $F \subseteq Q$
  - $\delta(q, i)$ the **transition function** or transition matrix between states.

Finite State Automata

- For example, the FSA below is defined as follows:
  - $Q = \{q_0, q_1, q_2, q_3, q_4\}$
  - $\Sigma = \{$'a','b','!'$\}$
  - q_0 – the start state
  - F – q_4
  - $\delta(q, i) =$

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



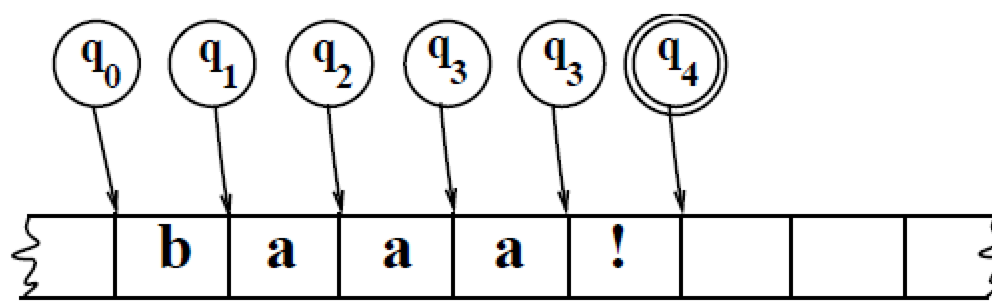
Finite State Automata

- How an FSA recognizes a language:
- On the surface, an FSA is only a set of states and transitions. It describes relations between states according to user input.
- A function is needed to feed it input and use the transition function to change states.
- The D-RECOGNIZE function.

Finite State Automata

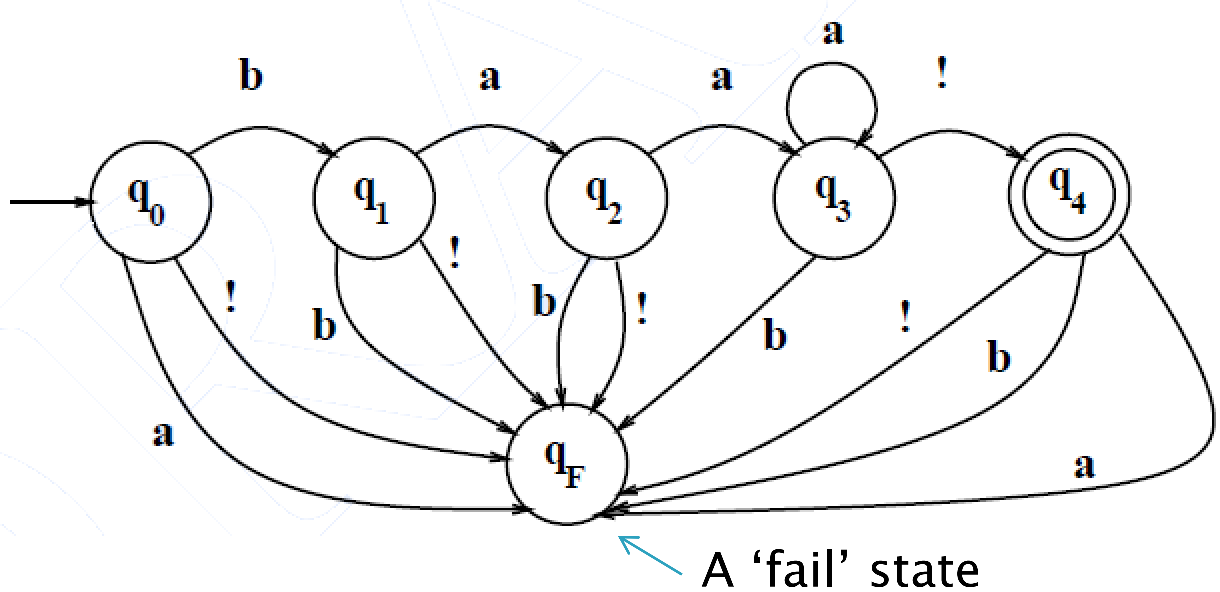
- **The D-RECOGNIZE function:**

```
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 Loop
end
```



Finite State Automata

- Two ways to handle rejected strings:
  - By empty slots in the transition table that stand for 'unsupported input' and treated accordingly by D-recognize (as we seen above)
  - By a dedicated 'fail' state in the automaton:



Intermediate Summary

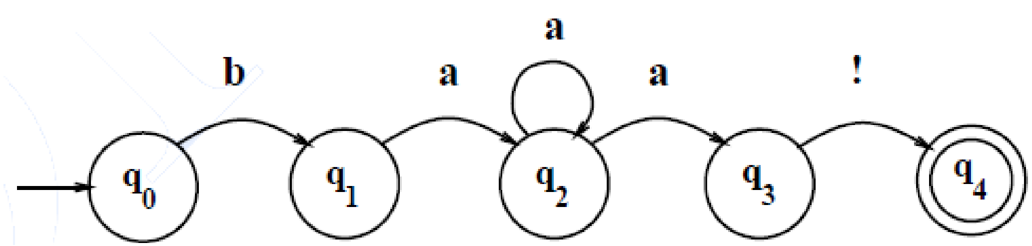
- So far we have seen regular expressions and finite state automata.
- Both are used to characterize formal languages:
  - A Regexp describes a pattern for which the matched strings constitute the language.

    → A regexp characterizes a language by generating it from a pattern.
  - An FSA describes a set of states and transitions that determine the set of strings (i.e. a language) that are accepted.

    → An FSA characterizes a language by recognizing it.

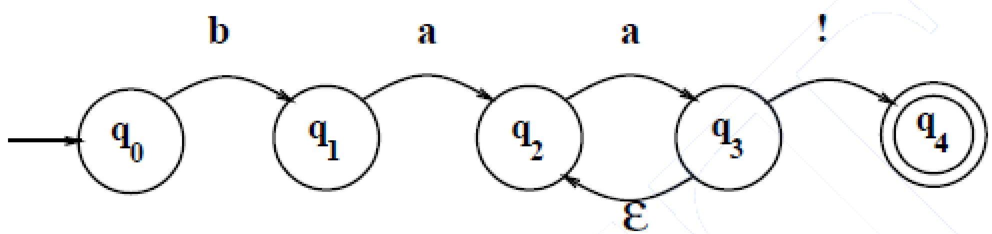
Non-Deterministic FSA

- Automata with decision points like in q_2 in the automaton below are called **non-deterministic FSAs** (or **NFSAs** or NFAs).



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

- Non-determinism may appear also by the use of epsilon transitions ($q_3 \rightarrow q_2$) that allow the recognizer to switch states without any input:



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

Non-Deterministic FSA

- Accepting strings is more complex in the non-deterministic case
- Since there is more than one choice at some point, we might take the wrong choice.
- Several solutions:
  - Backup strategy: a *marker* is placed in each choice point.Then if it turns out that we took the wrong choice, we could back up and try another path.
  - Look-ahead strategy: We could look ahead in the input to help us decide which path to take.
  - Parallelism strategy: Whenever we come to a choice point, we could look at every alternative path in parallel.
  - Alternative: convert the NFSA to an FSA and then accept the strings. But Is this possible?

Non-Deterministic FSA

- NFSAs may seem to have more computational power in the sense of allowing more complex languages to be defined.
- However, it turns out that in terms of computational power they are equivalent.
- Formally, any non-deterministic FSA is translatable to a deterministic FSA.
- The translated FSA may require more memory space but nonetheless it would accept the same language as the NFSA.

From NFSA to FSA

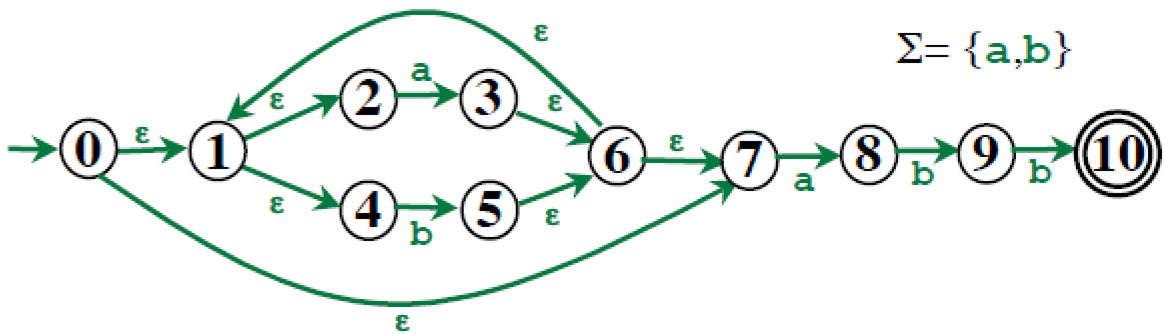
- Slides by **Harry H. Porter, 2005**
- http://web.cecs.pdx.edu/~harry/compilers/slides/LexicalPart3.pdf

- General idea:
  - Construct an FSA by simulating a parallel transition on the original NFSA
  - Each state in the FSA will correspond to a set of NFSA states.

- Full example in the original slides.

From NFSA to FSA

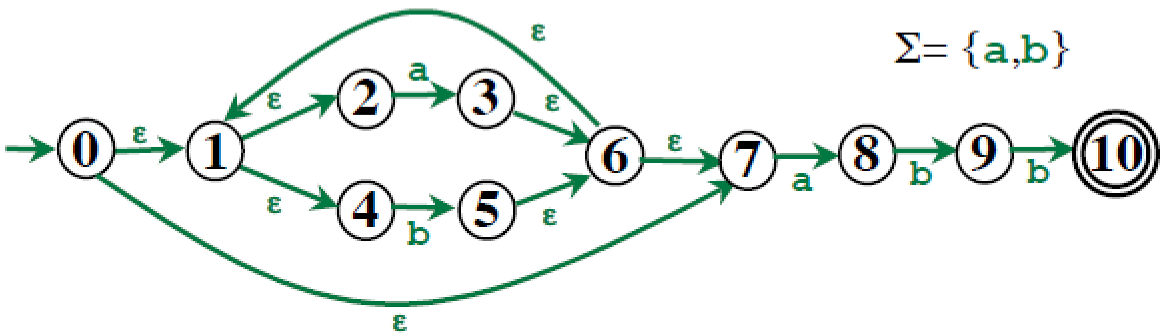
- Consider the following NFSA:



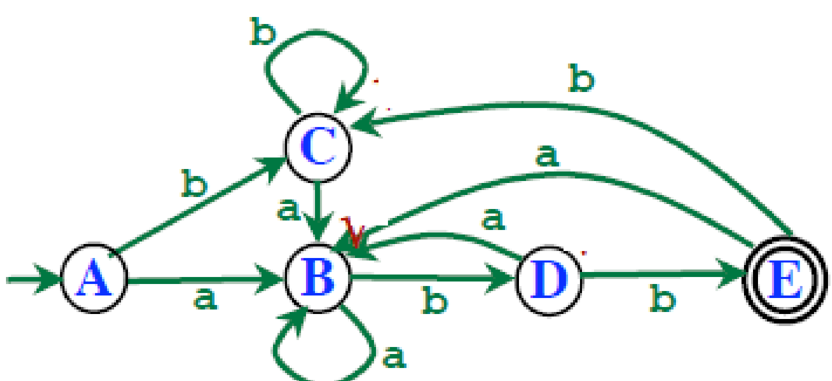
- It accepts strings such as ‘aabb’, ‘abb’, ‘bbb’, etc.

From NFSA to FSA

- Consider the following NFSA:



- A translation to an FSA:



A={0,1,2,4,7}

B={1,2,3,4,6,7,8}

C={1,2,4,5,6,7}

D={1,2,4,5,6,7,9}

E={1,2,4,5,6,7,10}

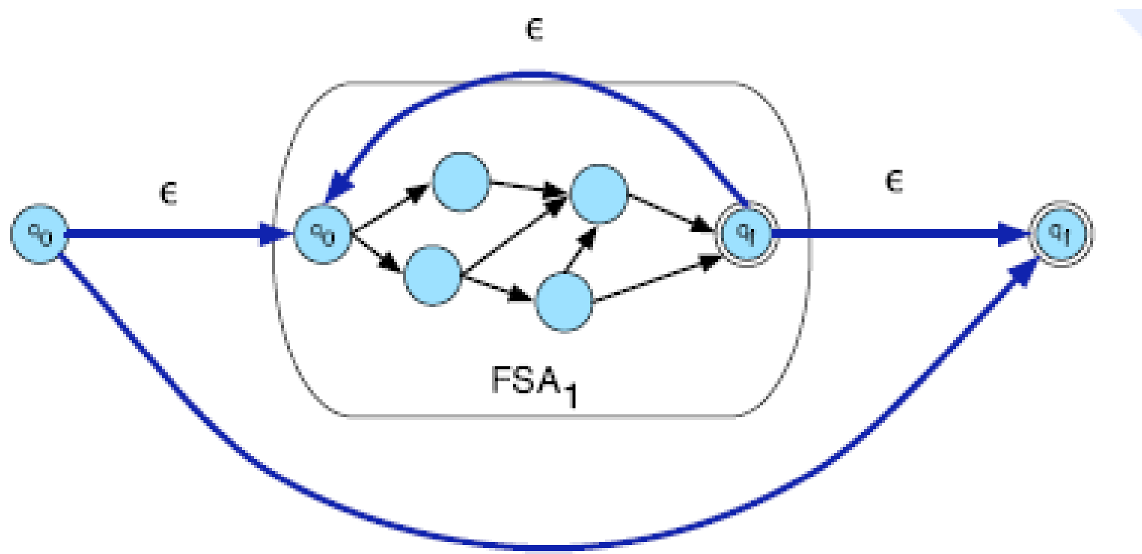
From Regexp to NFSA

- The general idea is to create an NFSA for each basic sequence in a regexp and then to connect all NFSAs by epsilon links.
- For basic sequences:

(a) r=ϵ (b) r=∅ (c) r=a

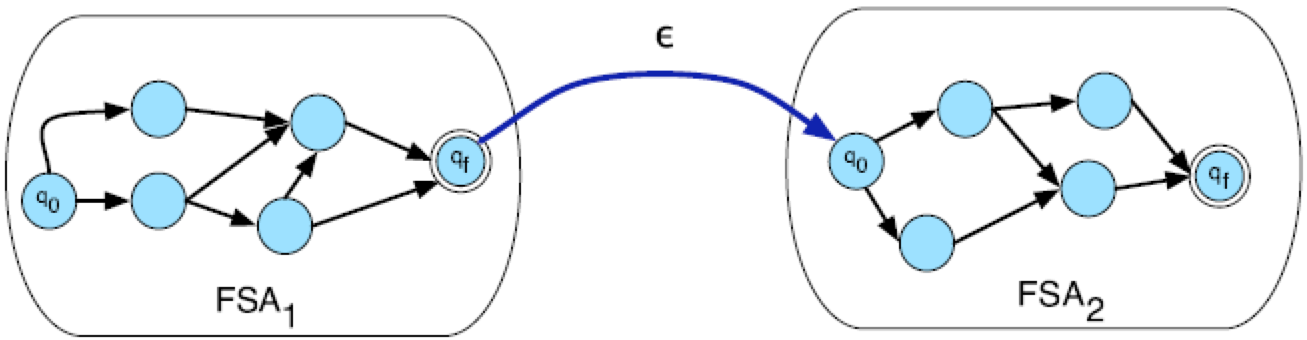
From Regexp to NFSA

- For Kleene*: We create a new final and initial state, connect the original final states of the FSA back to the initial states by e-transitions and then put direct links between the new initial and final states by e-transitions.



From Regexp to NFSA

- For example, **concatenation**: We just string two FSAs next to each other by connecting all the final states of FSA_2 by epsilon links



Regular Languages

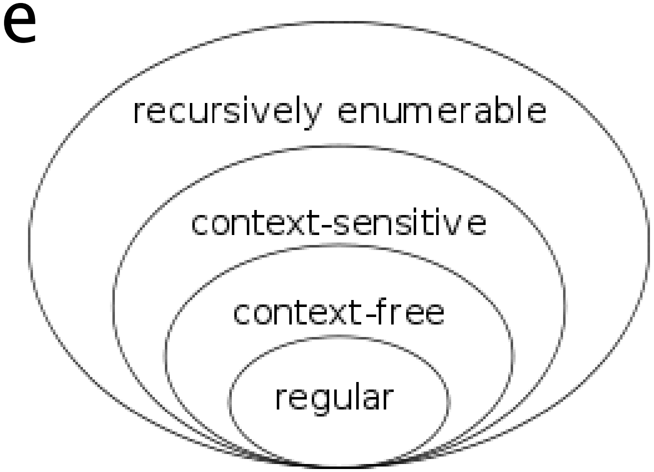
- The class of languages that can be defined by regular expressions is exactly the same as the class of languages that can be characterized by finite-state automata (whether deterministic or non-deterministic).
- Because of this, we call these languages the **regular languages**.

Non-Regular Languages

- It turns out that not all languages are regular.
- For example: $L = \{a^n b^n : n \geq 1\}$
- The automaton/regexp needs to 'remember' the exact number of 'a's in order to match it with the number of 'b's.
- This cannot be achieved without some sort of on-the-fly memory resource
- Theory of computation:

  Diagram Source: Wikipedia

  http://en.wikipedia.org/wiki/Regular_language



More Resources

- Michael Sipser (1997). *Introduction to the Theory of Computation*. PWS Publishing. ISBN 0-534-94728-X.

- Hopcroft, John E.; Motwani, Rajeev; Ullman, Jeffrey D. (2000). *Introduction to Automata Theory, Languages, and Computation* (2nd ed.). Addison-Wesley.