Finite state technology

Dana Dannélls
dana.dannells@svenska.gu.se

Språkbanken, Department of Swedish
University of Gothenburg

5 November 2015

Today's lecture

Regular Expressions

Finite-State Automata

Finite State Transducers

Regular expressions and finite state automata

Regular languages

Sets of strings

{“ab” “bb” “cb”}
{“ac” “abc” “abbc” “abbc”}
{}
{“”}

A sequence of symbols; any sequence of alphanumeric characters.

Regular expressions

A language for specifying text search strings.

Formally, an algebric notation for characterizing a set of strings.

For example: [a b* c]

where

- atomic symbols (e.g. ‘a’) denote languages.
- operations (here: concatenation and kleene-star) are operations over languages.

Regular expressions

Set of operators

? “zero or one instances of the previous character”

* “zero or more occurrences of the immediately previous character or regular expression”

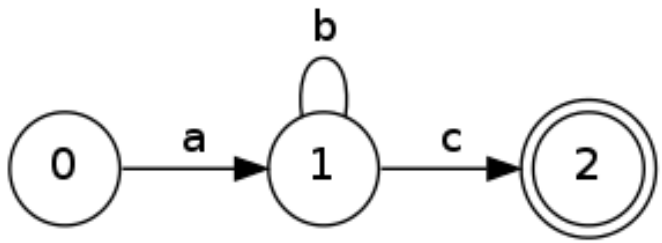
(/./) “any single character”

| “disjunction”

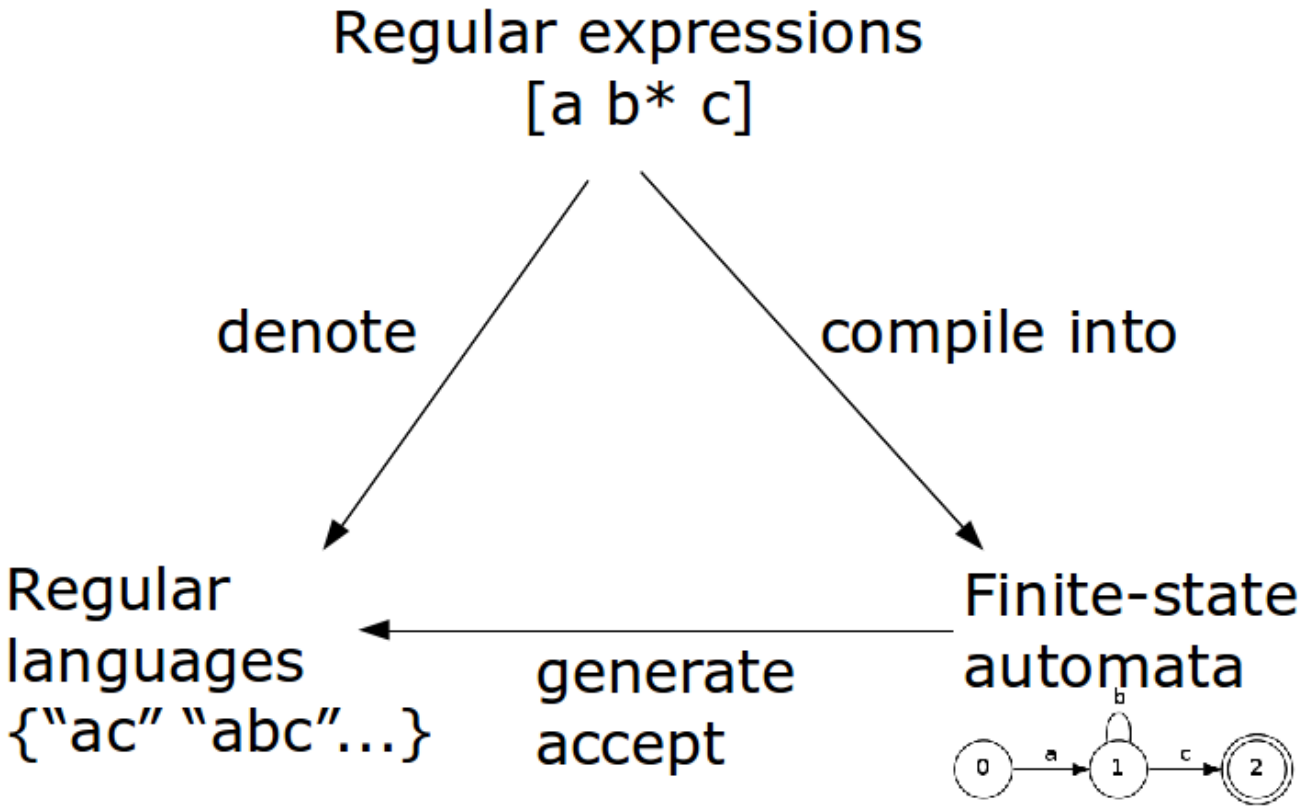
( ) “precedence”

Finite-State Automata (FSA)

Directed graphs consisting of states, and arcs labelled with symbols.



Regular expressions, regular languages and automata



FSA as graphs

The sheep language /baa*!/

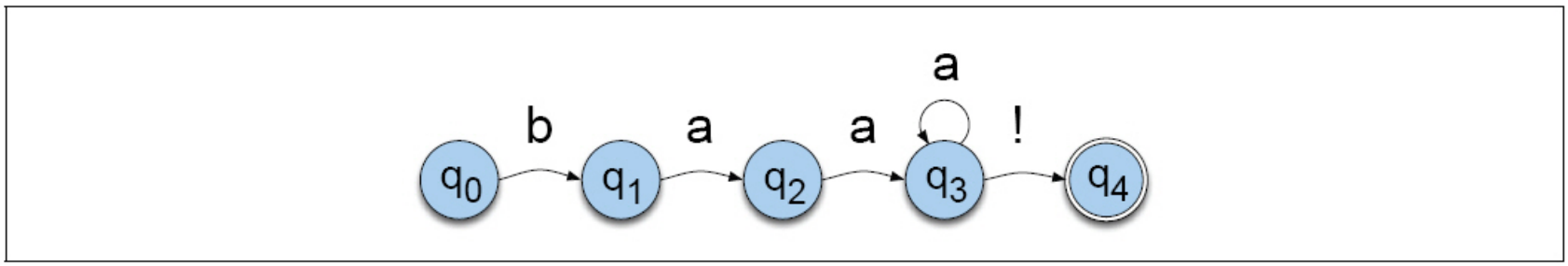


Figure 2.10: A finite-state automaton for talking sheep.

A finite-state automaton for talking sheep

We can say the following things about the talking sheep machine :

- It has 5 states
- b, a, and ! are in its alphabet
- q0 is the start state
- q4 is an accept state
- It has 5 transitions

but note There are other machines that correspond to this same language.

FSA represented with a state-transition table

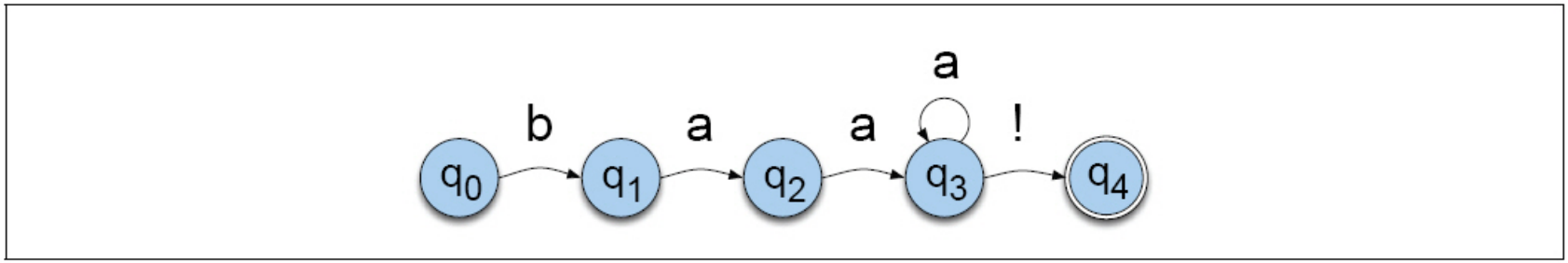
	Input		
State	b	a	!
0	1	Ø	Ø
1	Ø	2	Ø
2	Ø	3	Ø
3	Ø	3	4
4:	Ø	Ø	Ø

More formally

You can specify an FSA by enumerating the following things:

- The set of states: Q
- A finite alphabet: Σ
- A start state: q0
- A set of accept/final states: F, $F \subseteq Q$
- A transition function between states: δ(q,i), where δ is a relation from QxΣ to Q

Formal description for the talking sheep FSA



$Q = \{q0,q1,q2,q3,q4\}$

$\Sigma = \{a,b,!\}$

$F = \{q4\}$

$\delta(q,i)$

Formal language

A set of strings, each string composed of symbols from a finite symbol set called alphabet.

The alphabet for the sheep language is the set

$$\Sigma = \{\text{a,b,!}\}$$

About alphabets

The term *alphabet* 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.

Another example

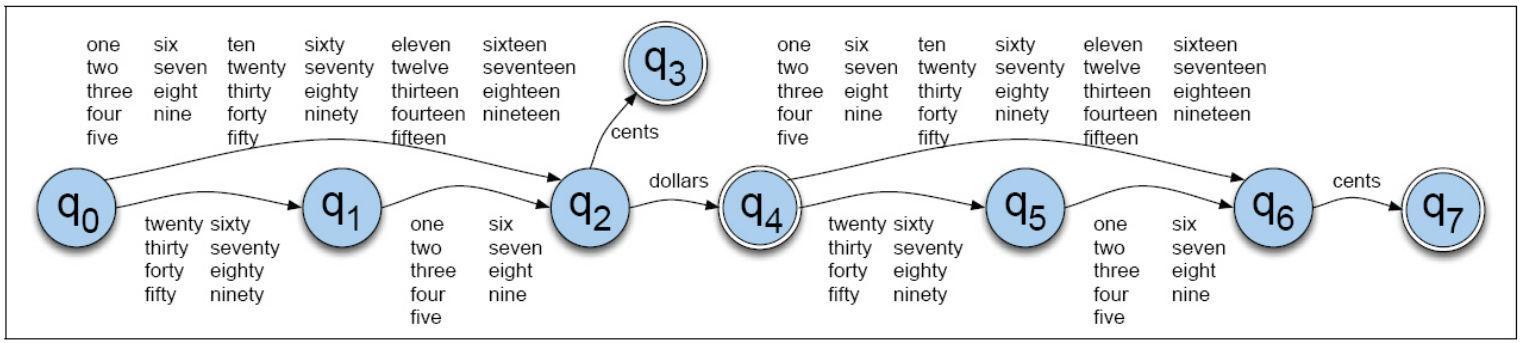


Figure 2.12: FSA for the simple dollars and cents.

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.

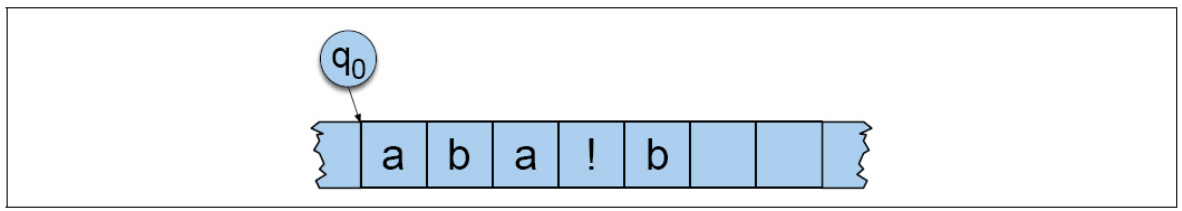


Figure 2.11: A tape with cells.

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.

Deterministic recognizer

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.

D-Recognize algorithm

```
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
```

Figure 2.12: An algorithm for deterministic recognition of FSAs.

Key points

Crudely therefore. . . matching strings with regular expressions (ala Perl, grep, etc.) is a matter of:

1. translating the regular expression into a machine (a table) and
2. passing the table and the string to an interpreter.

FSAs can be viewed from two perspectives:

1. Acceptors that can tell you if a string is in the language.
2. Generators to produce all and only the strings in the language.

Non-Deterministic FSAs (NFSAs)

An automaton whose behavior during recognition is fully determined by the state it is in and the symbol it is looking at.

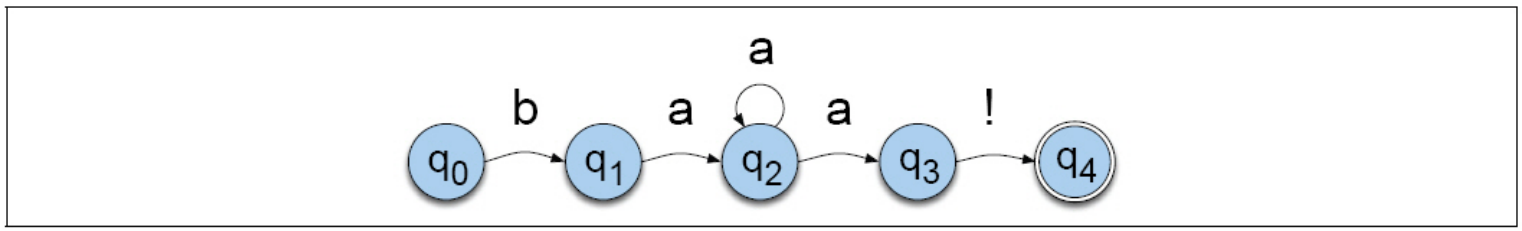
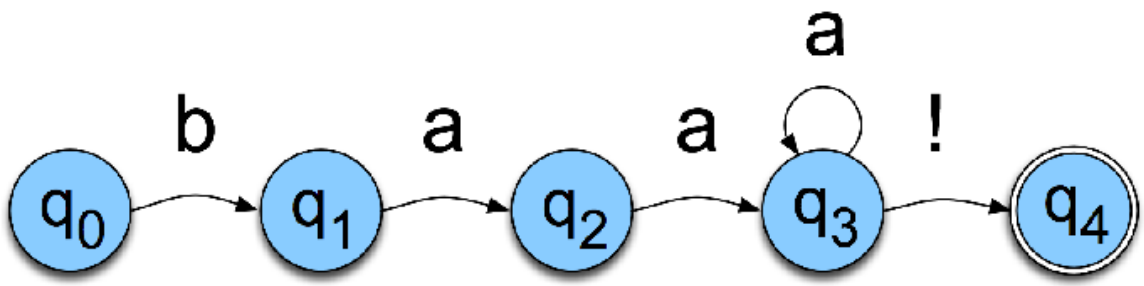
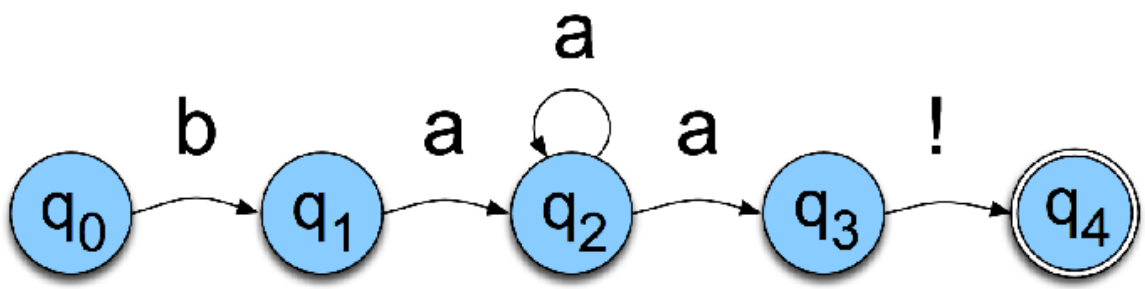


Figure 2.17: A non deterministic FSA for talking sheep.

Non-Determinism





Non-Determinism

Another type of non-deterministic automaton is one caused by arcs that have no symbols on them called *epsilon transitions*

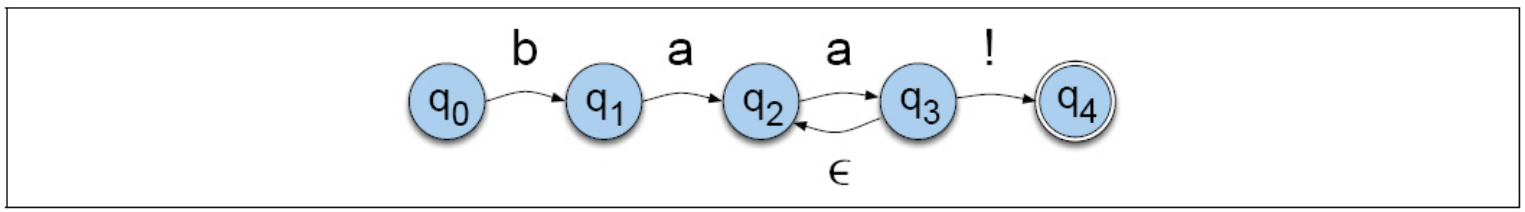


Figure 2.18: Another NFSA for the sheep language.

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.

Non-Deterministic 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 as it is).

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.

Non-Deterministic Recognition: Search

In a NDFSA 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

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.

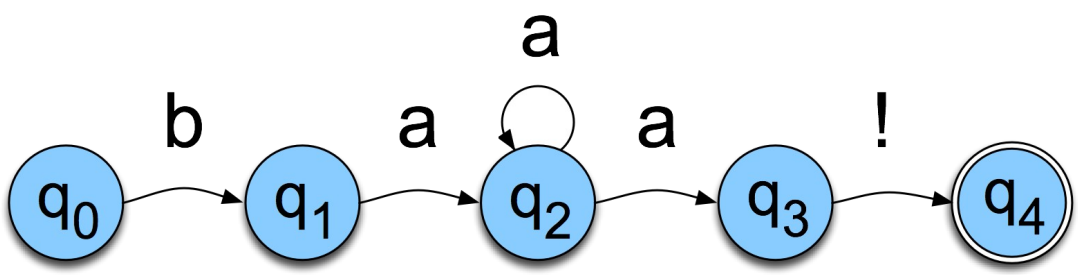
Three standard solutions

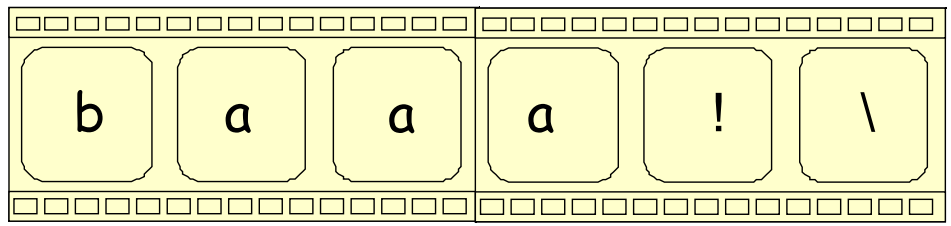
Backup: whenever we come to a choice point, we could put a marker to mark where we were in the input and what state the automaton was in. Then if it turns out that we look at the wrong choice, we could back up and try another path.

Look ahead: We could look ahead in the input to help us decide which path to take.

Parallelism: whenever we come to a choice point, we could look at every alternative path in parallel.

Example

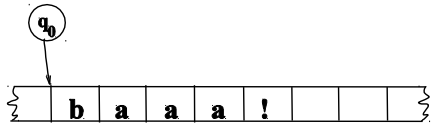


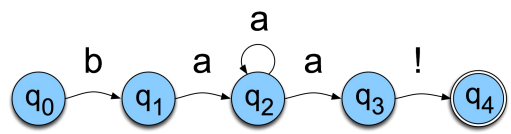


q_0 q_1 q_2 q_2 q_3 q_4

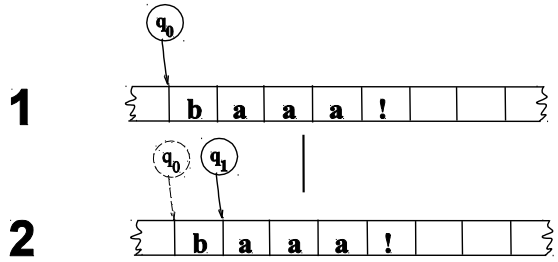
Example

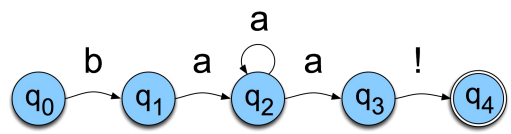
1



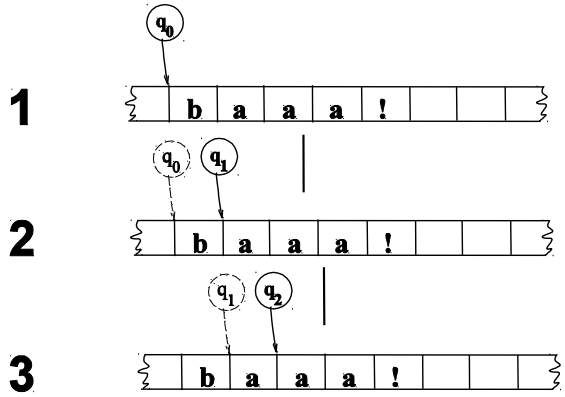


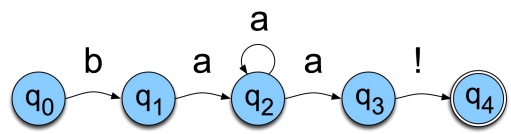
Example



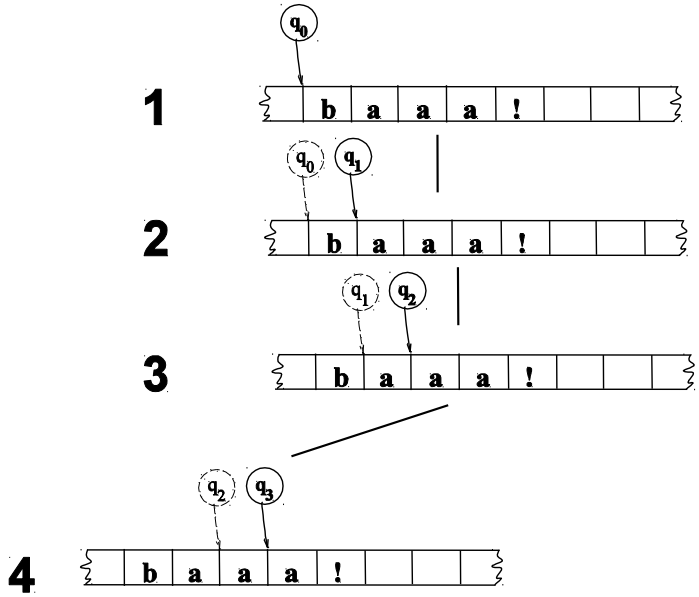


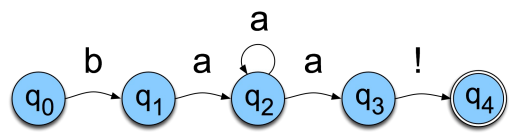
Example



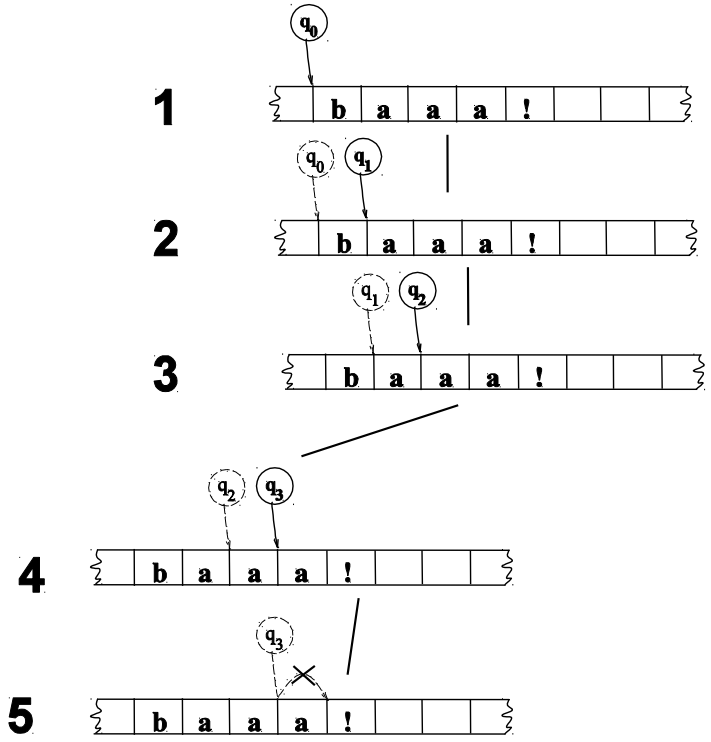


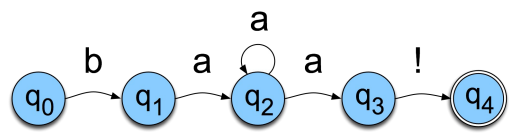
Example



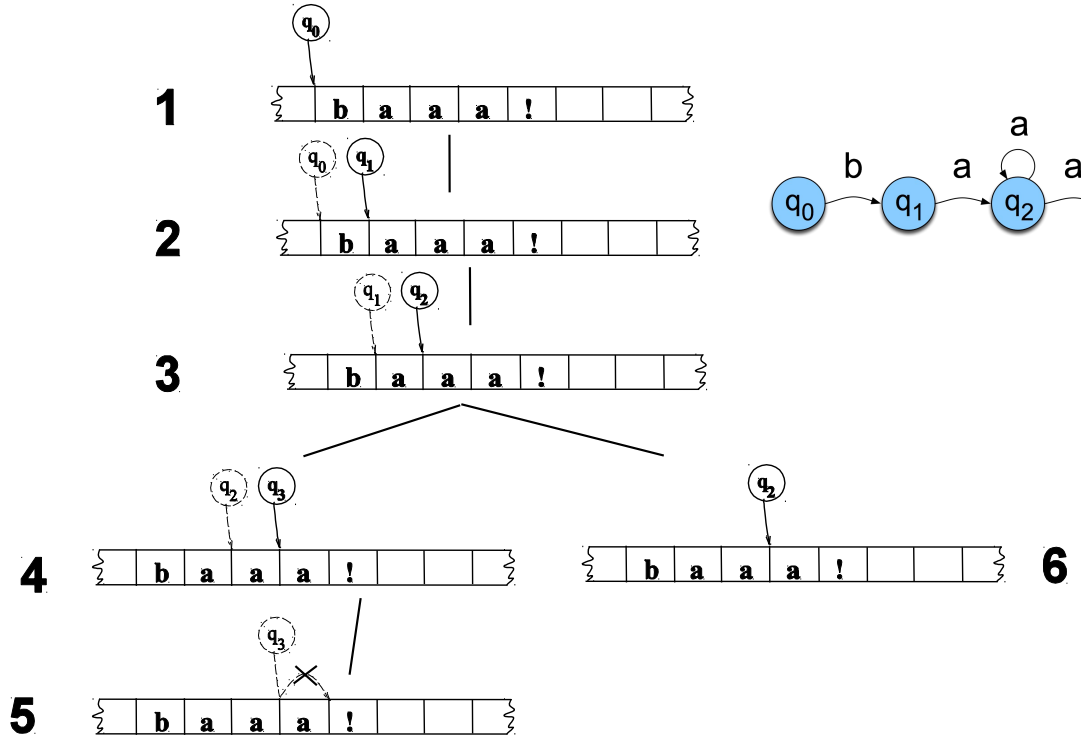


Example

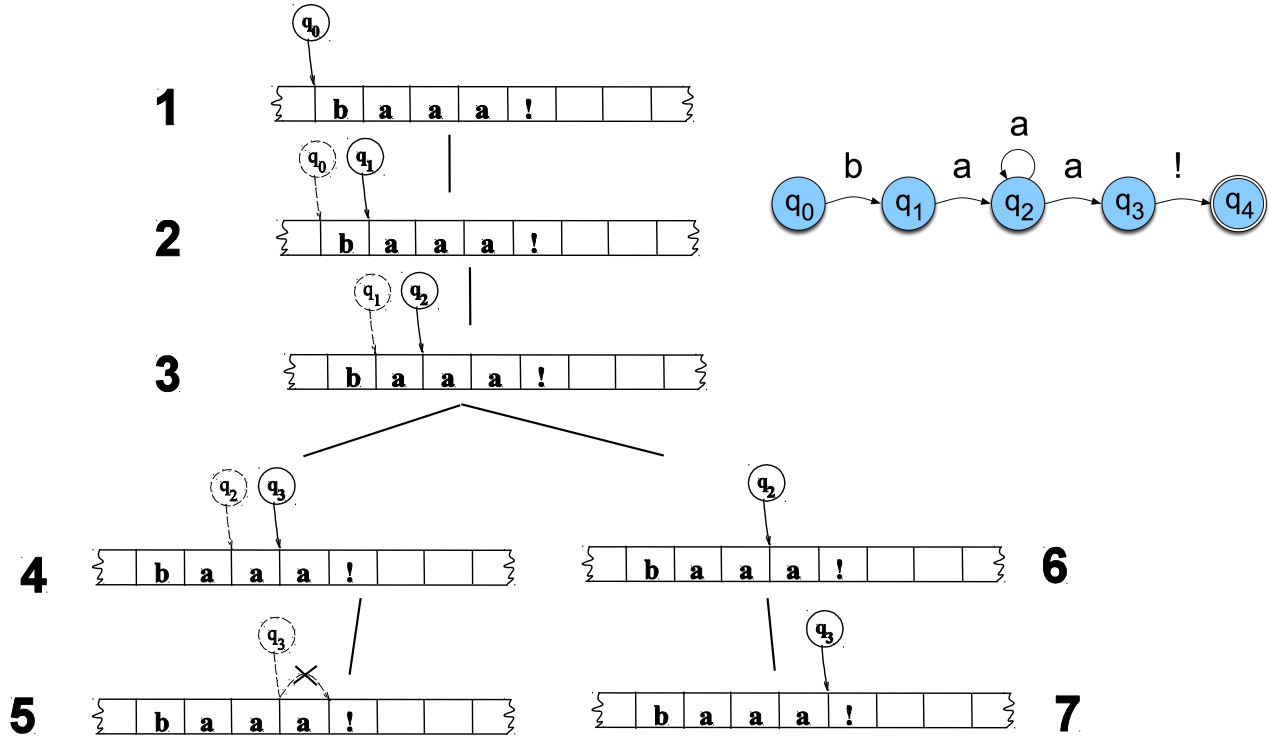




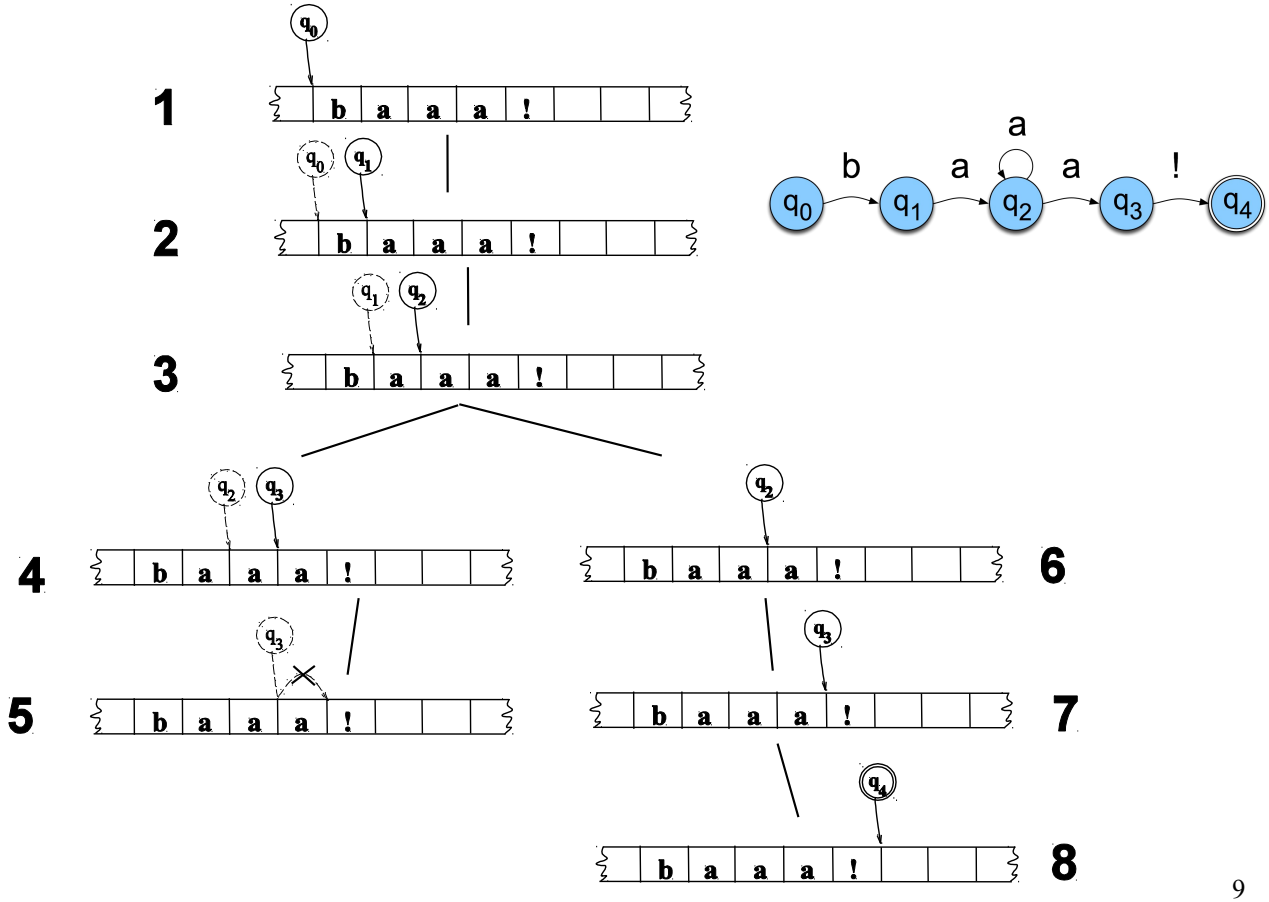
Example



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?

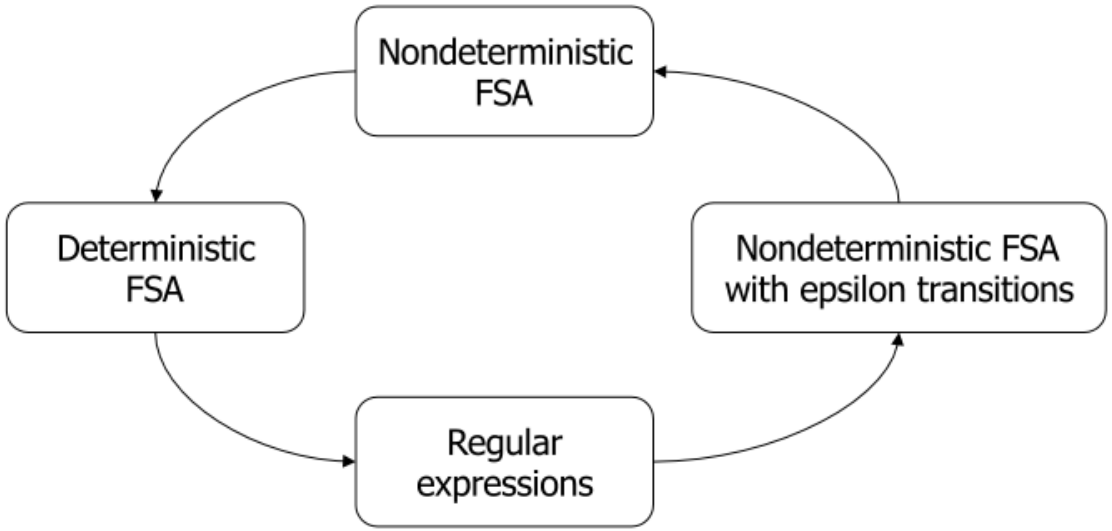
Non-determinism doesn't get us more formal power and it causes headaches so why bother?

Because more natural (understandable) solutions

Regular languages more formally

1. $\emptyset$ is a regular language
2. $\forall a \in \Sigma \cup \epsilon$, $\{a\}$ is a regular language
3. If L_1 and L_2 are regular languages, then so are:
   (a) $L_1 \cdot L_2 = \{xy | x \in L_1, y \in L_2\}$, the **concatenation** of L_1 and L_2
   (b) $L_1 \cup L_2$, the **union** or **disjunction** of L_1 and L_2
   (c) L_1^*, the **Kleene closure** of L_1

Regex to FSA to regex



Picture adapted from Hopcroft & Ullman 1979

From regular expressions to finite-state automata

The only really necessary operators:

- Union
- Concatenation
- Iteration

Sidenote: Compare regular grammars:

A $--> $ x B

A $--> $ x

(where A and B are nonterminals, and where x is a sequence of terminals)

Concatenation

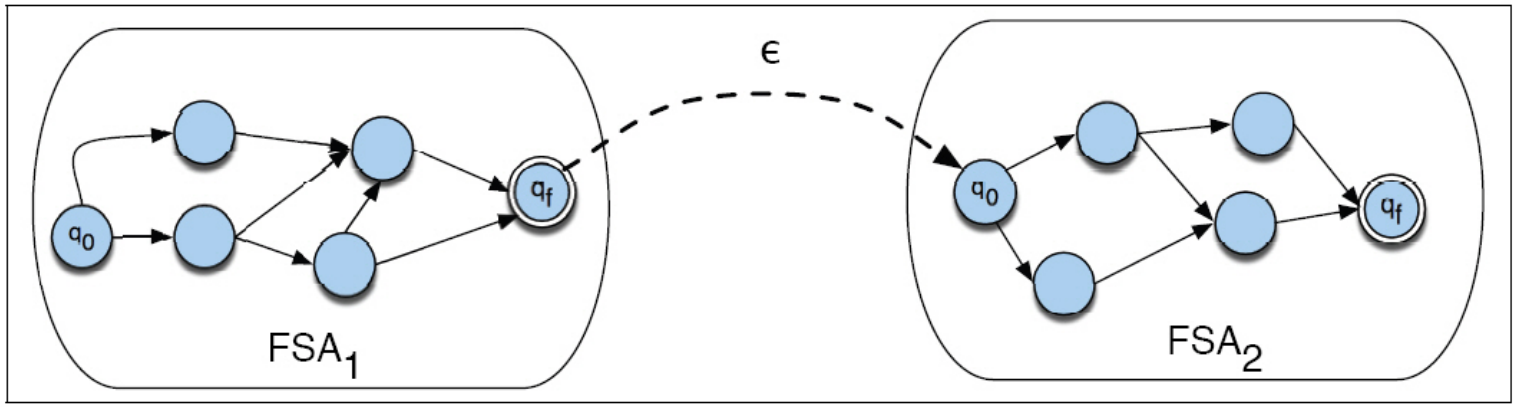


Figure 2.23: The concatenation of two FSAs

Iteration

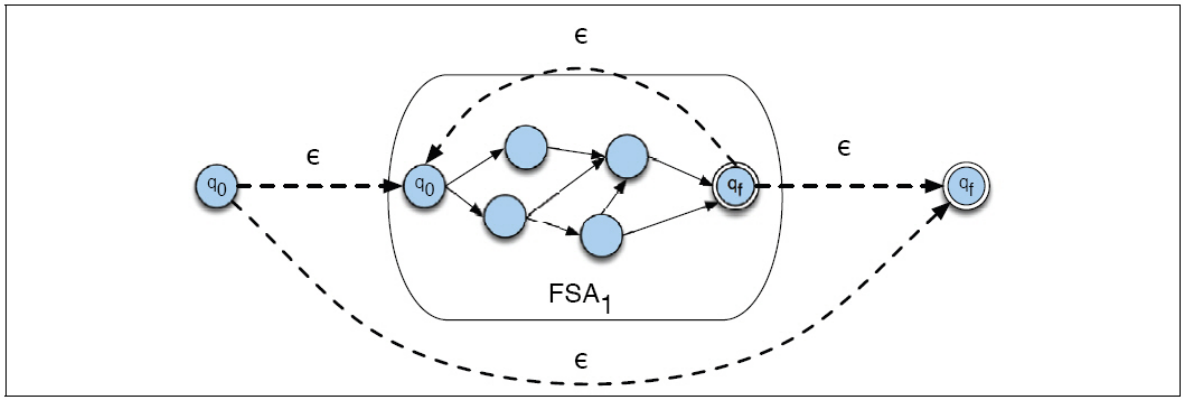


Figure 2.24: The closure (Kleene *) of an FSA

Union

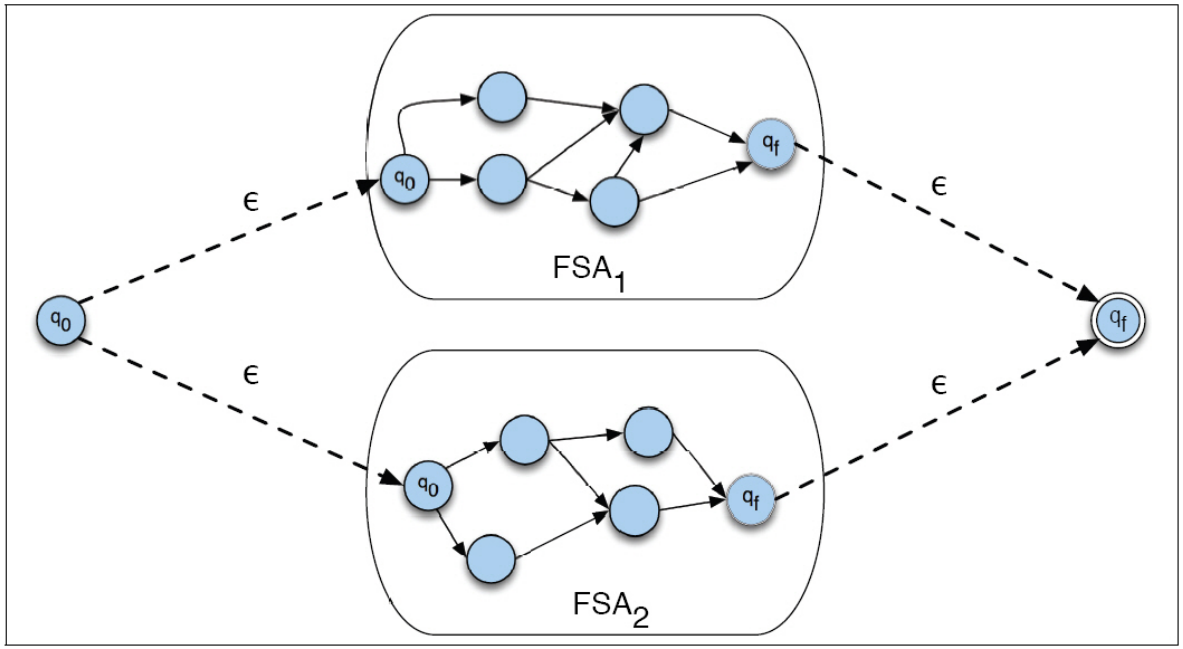


Figure 2.25: The union (|) of an FSA

Various equivalences

$$(\mathrm{A}) = \mathrm{A}|[]$$
$$\mathrm{A}+ = \mathrm{A}\ \mathrm{A}^*$$
$$\mathrm{A}+ = \mathrm{A}^*\ \text{-}\ []$$
$$\mathrm{A}\ \text{-}\ \mathrm{B} = \mathrm{A}\ \&\ {\sim}\mathrm{B}$$
$${\sim}\mathrm{A} = ?^*\ \text{-}\ \mathrm{A}$$
$${\sim}[\mathrm{A} \mid \mathrm{B}] = {\sim}\mathrm{A}\ \&\ {\sim}\mathrm{B}$$
$${\sim}[\mathrm{A}\ \&\ \mathrm{B}] = {\sim}\mathrm{A} \mid {\sim}\mathrm{B}$$

Various equivalences

$$\text{A - A} = \sim[?*]$$
$$\text{A} \mid \sim[?*] = \text{A}$$
$$\text{A } [] = \text{A}$$
$$\text{A} \sim[?*] = \sim[?*]$$
$$\text{A } \& \ ?* = \text{A}$$
$$\text{A} \mid ?* = ?*$$

Finite state transducers (FST)

Component technologies in FST

Word lists and lexica

Tokenisers

Morphological analysers

Part-of-speech taggers

Parsers

Finite state lexicon

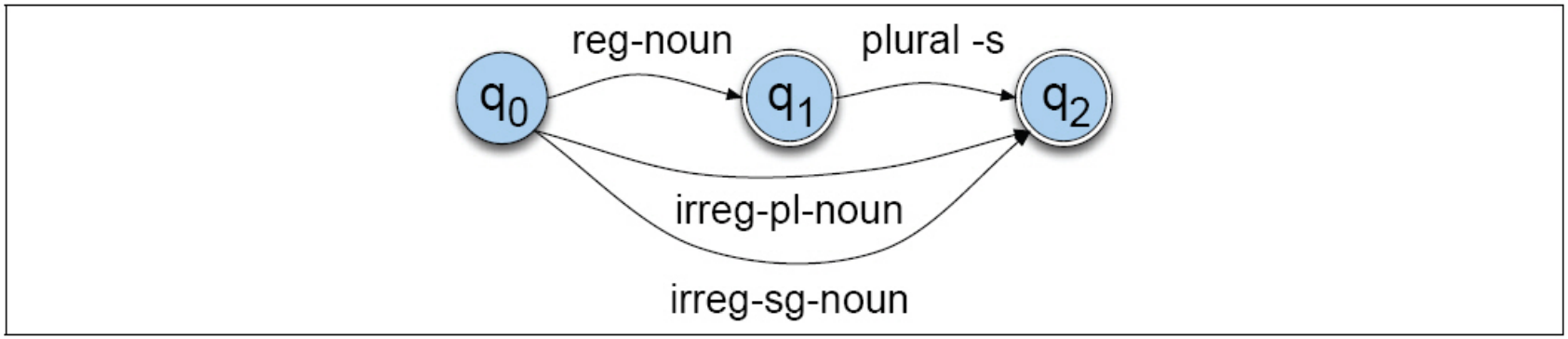


Figure 3.3: FSA for a English nominal inflection

Finite state lexicon

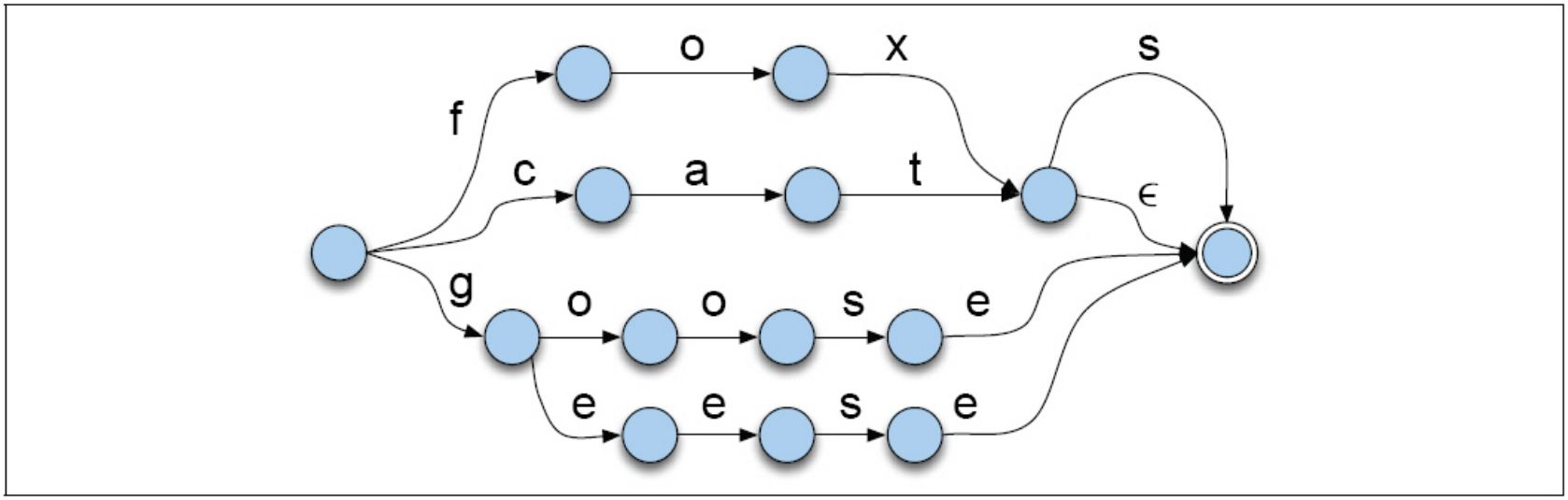


Figure 3.7: FSA for a few English nouns and their inflection

Applications of FST

Named-entity recognition

Information extraction

Corpus linguistics

Spelling- and grammar checking

Speech processing applications

Important theoretical results

Kleene's theorem (concerning FSAs)

Closure properties of regular languages and regular relations

Decidability

Kleene's theorem

Any language recognized by an FSA is denoted by a regular expression and any language denoted by a regular expression can be recognized by a FSA.

Closure properties of regular languages

A set is said to be closed under an operation iff applying the operation to members of the set will never take us outside the set.

Example: if A and B are regular languages, then [A|B] is always regular. Therefore regular languages are closed under union.

Decidability

Given one automaton A:

- Is the string S a string in L(A)?
- Does L(A) contain any strings at all?
- Is L(A) equivalent to ?*?

Given two automata A1 and A2:

- Is L(A1) a subset of L(A2)?
- Are L(A1) and L(A2) equivalent?
- Do L(A1) and L(A2) overlap?

Decidability

Determinization and minimalization lead to a 'normal form' that makes it easy to answer decidability questions!

XFST commands:

- test upper-universal (test lower-universal)
- test sublanguage
- test equivalent
- test overlap

Regular relations

Sets of pairs of strings

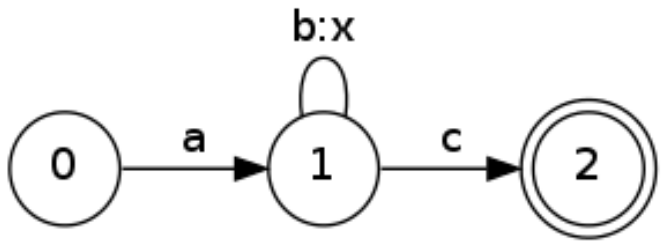
{ <“ac”:“ac”> <“abc”:“axc”> <“abbc”:“axxc”> . . . }

A regular relation has

- an upper language (e.g. { “ac” “abc” “abbc” . . . })
- a lower language (e.g. { “ac” “axc” “axxc” . . . })

Finite-State Transducers

Directed graphs consisting of states, and arcs labelled with pairs of symbols.



More formally

You can specify an FST by enumerating the following things:

- A finite set of states: Q
- A finite set corresponding to the input alphabet: Σ
- A finite set corresponding to the output alphabet: Δ
- The start state: q0 $\in$ Q
- The set of final states: F $\subseteq$ Q
- The transition function between states: δ(q,w)
- The output function giving the set of possible output strings for each state and input: σ(q,w)

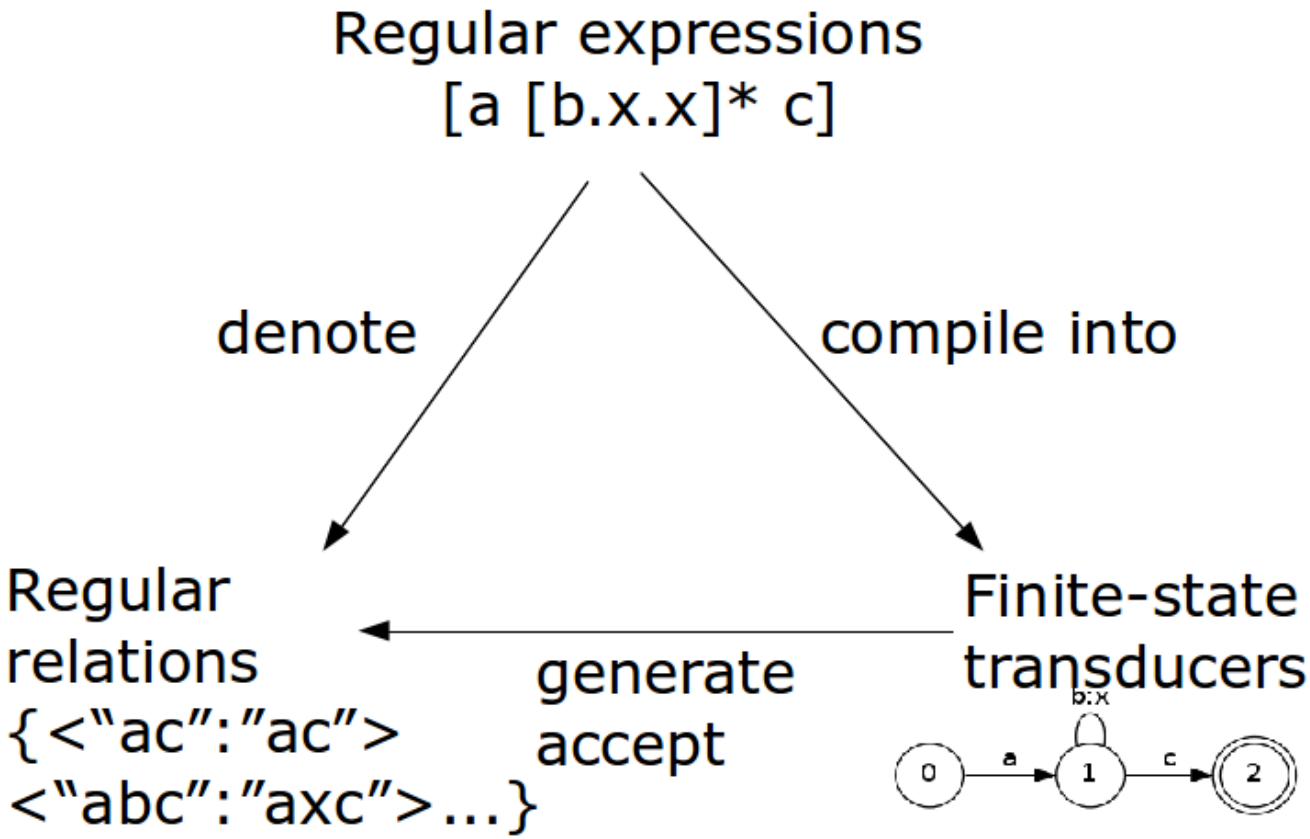
Regular expressions

[a [b .x.x] * c]

‘a’ denotes a regular relation <“a”:“a”>

concatenation and iteration are operations over regular relations

Regular expressions, relations and transducers



Lexical transducers

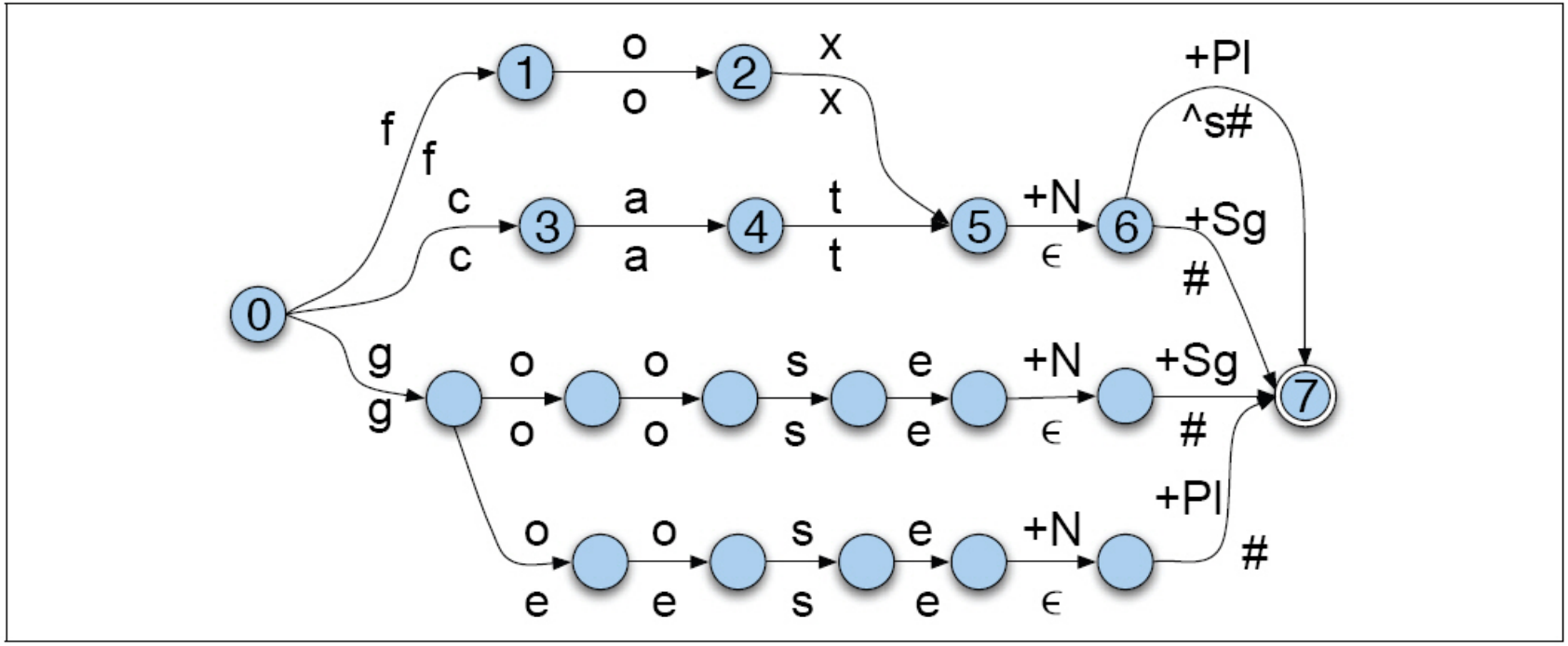


Figure 3.14: English nominal inflection FST

Closure properties of regular relations

Regular relations are closed under union, composition and inversion.

Regular relations in general are not closed under intersection, but relations where upper and lower strings always have the same length are (i.e. relations generated by epsilon-free transducers).

Transducers are not closed under difference or complement.

The Xerox Finite-State Tool

Compiles extended regular expressions into finite-state machines (automata and transducers).

Allows the user to display, examine and modify the machines.

Xerox tools for lexicon construction

LEXC

Lexicon compiler – for building finite-state lexica. Syntax and algorithms optimized for this task.

TWOLC

Compiler for two-level rules (Koskennemi 1983)