Finite-State Automata and Algorithms

Bernd Kiefer, kiefer@dfki.de

Many thanks to Anette Frank for the slides

MSc. Computational Linguistics Course, SS 2009

Overview

- Finite-state automata (FSA) – What for?
  - Recap: Chomsky hierarchy of grammars and languages
  - FSA, regular languages and regular expressions
  - Appropriate problem classes and applications
- Finite-state automata and algorithms
  - Regular expressions and FSA
  - Deterministic (DFSA) vs. non-deterministic (NFSA) finite-state automata
  - Determinization: from NFSA to DFSA
  - Minimization of DFSA
- Extensions: finite-state transducers and FST operations

Finite-state automata: What for?

Chomsky Hierarchy of Languages

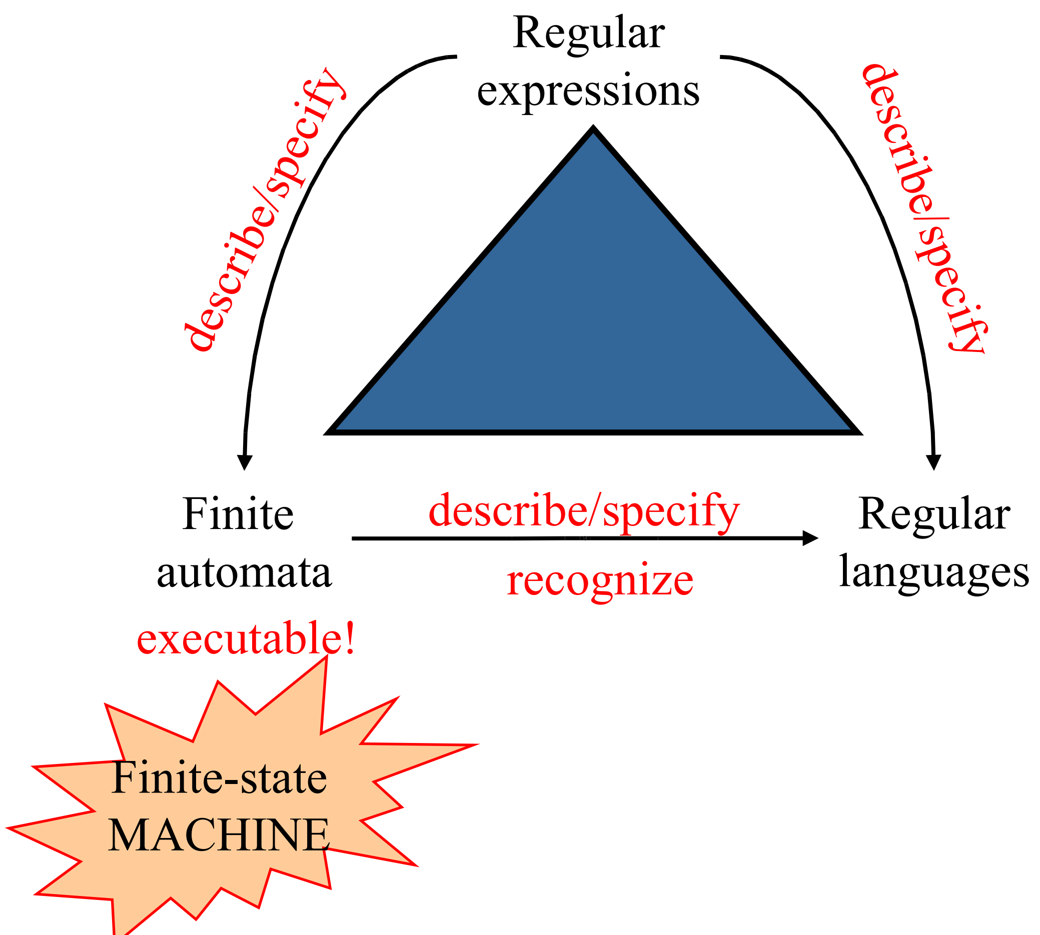
- Regular languages (Type-3)
- Context-free languages (Type-2)
- Context-sensitive languages (Type-1)
- Type-0 languages

Hierarchy of Grammars and Automata

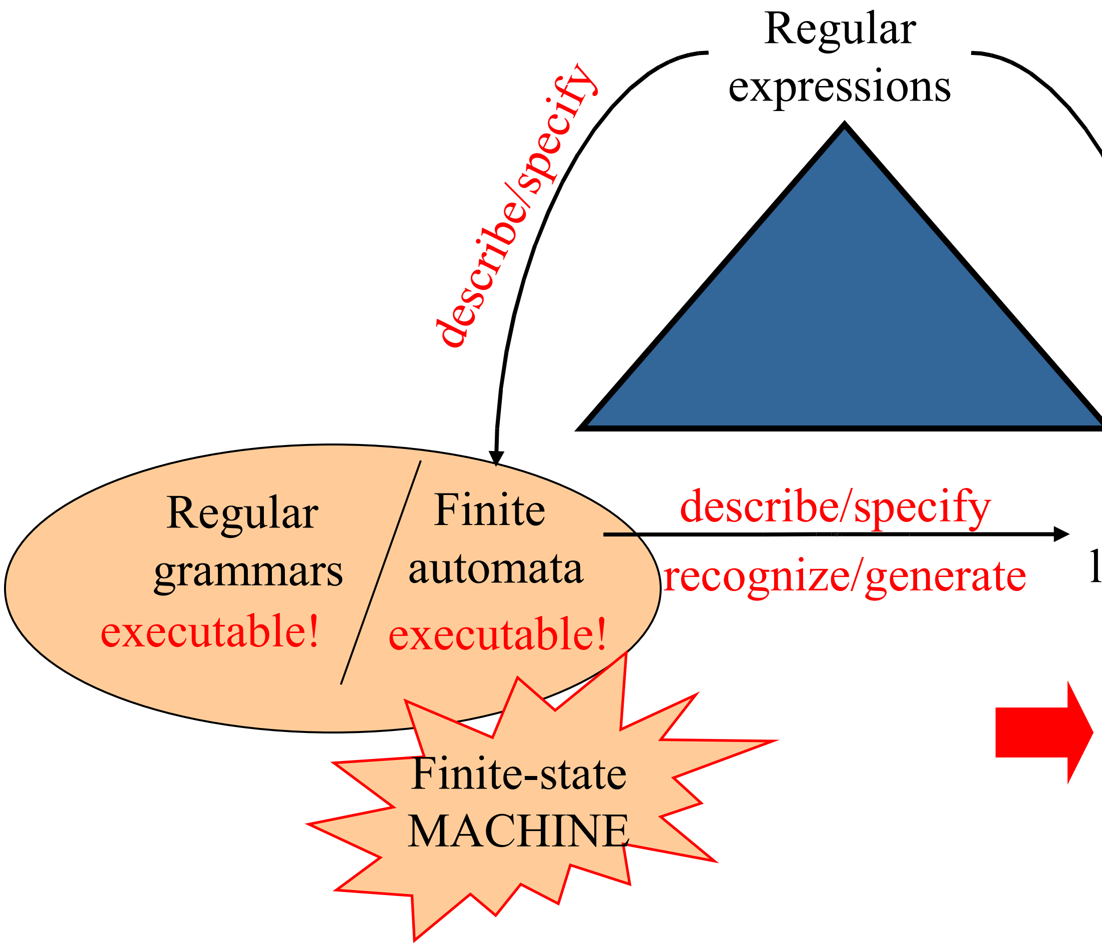
- Regular PS grammar
  Finite-state automata
- Context-free PS grammar
  Push-down automata
- Tree adjoining grammars
  Linear bounded automata
- General PS grammars
  Turing machine

computationally more complex
less efficient

Finite-state automata model regular languages



Finite-state automata model regular languages



- properties of regular languages
- appropriate problem classes
- algorithms for FSA

Languages, formal languages and grammars

Strings

- *Alphabet* Σ : finite set of symbols
- *String* : sequence $x_1 \ldots x_n$ of symbols x_i from the alphabet Σ
  - Special case: empty string ε
- *Language over* Σ : the *set of strings* that can be generated from Σ
  - Sigma star Σ^* : set of all possible strings over the alphabet Σ
    $\Sigma = \{a, b\}$ $\Sigma^* = \{\varepsilon, a, b, aa, ab, ba, bb, aaa, aab, \ldots\}$
  - Sigma plus $\Sigma+$: $\Sigma+ = \Sigma^* - \{\varepsilon\}$
  - Special languages: $\varnothing = \{\}$ (empty language) $\neq$ $\{\varepsilon\}$ (language of empty string)
- *A formal language* : a subset of Σ^*
- Basic operation on strings: *concatenation* •
  - If $a = x_i \ldots x_m$ and $b = x_{m+1} \ldots x_n$ then $a \cdot b = ab = x_i \ldots x_m x_{m+1} \ldots x_n$
  - Concatenation is associative but not commutative
  - ε is identity element : $a\varepsilon = \varepsilon a = a$
- A *grammar* of a particular type generates a *language* of a corresponding type

Recap on Formal Grammars and Languages

- A *formal grammar* is a tuple G = < Σ , Φ , S, R>
  - Σ alphabet of *terminal symbols*
  - Φ alphabet of *non-terminal symbols* ($\Sigma \cap \Phi = \emptyset$)
  - S the *start symbol*
  - R finite set of *rules* $R \subseteq \Gamma^* \times \Gamma^*$ *of the form* $\alpha \rightarrow \beta$
    where $\Gamma = \Sigma \cup \Phi$ and $\alpha \neq \varepsilon$ and $\alpha \notin \Sigma^*$
- The *language L(G) generated by a grammar G*
  - set of strings $w \subseteq \Sigma^*$ that can be *derived* from S according to G=<Σ ,Φ, S, R>
- *Derivation:* given G=<Σ, Φ, S, R> and $u,v \in \Gamma^* = (\Sigma \cup \Phi)^*$
  - a *direct derivation* (1 step) $\boldsymbol{w \Rightarrow_G v}$ holds iff
    $u_1, u_2 \in \Gamma^*$ exist such that $w = u_1 \alpha u_2$ and $v = u_1 \beta u_2$, and $\alpha \rightarrow \beta \in R$ exists
  - a *derivation* $\boldsymbol{w \Rightarrow_{G^*} v}$ holds iff either $w = v$
    or $z \in \Gamma^*$ exists such that $w \Rightarrow_{G^*} z$ *and* $z \Rightarrow_G v$
- A *language generated by a grammar G:* $L(G) = \{w : S \Rightarrow_{G^*} w \;\&\; w \in \Sigma^*\}$
  I.e., L(G) strongly depends on R !

Chomsky Hierarchy of Grammars

- Classification of languages generated by formal grammars
  - A language is of type *i* (*i* = *0,1,2,3*) iff it is generated by a type-*i* grammar
  - Classification according to increasingly restricted types of production rules
    L-type-0 ⊃ L-type-1 ⊃ L-type-2 ⊃ L-type-3
  - Every grammar generates a unique language, but a language can be generated by several different grammars.
  - Two grammars are
    - *(Weakly) equivalent* if they generate the same string language
    - *Strongly equivalent* if they generate both the same string language and the same tree language

Chomsky Hierarchy of Grammars

Type-0 languages: general phrase structure grammars

- no restrictions on the form of production rules: arbitrary strings on LHS and RHS of rules
- A grammar G = <Σ, Φ, S, R> generates a language L-type-0 iff
  - all rules R are of the form $\alpha \rightarrow \beta$, where $\alpha \in \Gamma+$ and $\beta \in \Gamma^*$ (with $\Gamma = \Sigma \cup \Phi$)
  - I.e., LHS a nonempty sequence of NT or T symbols with at least one NT symbol and RHS a possibly empty sequence of NT or T symbols
- Example:

  $G = <\{S,A,B,C,D,E\},\{a\},S,R>$, $L(G) = \{a^{2^n} \mid n \geq 1\}$

  $S \rightarrow ACaB.$ $\quad CB \rightarrow E.$ $\quad aE \rightarrow Ea.$

  $Ca \rightarrow aaC.$ $\quad aD \rightarrow Da.$ $\quad AE \rightarrow \varepsilon.$

  $CB \rightarrow DB.$ $\quad AD \rightarrow AC.$

  $a^{2^2} = aaaa \in L(G)$ iff $S \Rightarrow^* aaaa$

Chomsky Hierarchy of Grammars

Type-1 languages: context-sensitive grammars

- A grammar $G = <\Sigma, \Phi, S, R>$ generates a language L-type-1 iff
  - all rules R are of the form $\alpha A \gamma \rightarrow \alpha\beta\gamma$, or $S \rightarrow \varepsilon$ (with no S symbol on RHS) *where* $A \in \Phi$ and $\alpha, \beta, \gamma \in \Gamma^*$ ($\Gamma = \Sigma \cup \Phi$), $\beta \neq \varepsilon$
  - I.e., LHS: non-empty sequence of NT or T symbols with at least one NT symbol
    and RHS a nonempty sequence of NT or T symbols (exception: $S \rightarrow \varepsilon$)
  - For all rules LHS → RHS : |LHS| ≤ |RHS|
- Example:
  $L = \{ a^n b^n c^n \mid n \geq 1 \}$
- $R = \{ S \rightarrow a\,S\,B\,C,\quad a\,B \rightarrow a\,b,$
  $S \rightarrow a\,B\,C,\quad b\,B \rightarrow b\,b,$
  $C\,B \rightarrow B\,C,\quad b\,C \rightarrow b\,c,\quad c\,C \rightarrow c\,c \}$

$a^3b^3c^3 = aaabbbccc \in L(G)$ iff $S \Rightarrow^* aaabbbccc$

Chomsky Hierarchy of Grammars

Type-2 languages: context-free grammars

- A grammar G = <Σ, Φ, S, R> generates a language L-type-2 iff
  - all rules R are of the form $A \rightarrow \alpha$,
    where $A \in \Phi$ and $\alpha \in \Gamma^*$ ($\Gamma = \Sigma \cup \Phi$)
  - I.e., LHS: a single NT symbol; RHS a (possibly empty) sequence of NT or T symbols
- Example:
  $L = \{ a^n b a^n \mid n \geq 1 \}$
  $R = \{ S \rightarrow A\ S\ A,\ S \rightarrow b,\ A \rightarrow a \}$

Chomsky Hierarchy of Grammars

Type-3 languages: regular or finite-state grammar

- A grammar $G = \langle \Sigma, \Phi, S, R \rangle$ is called right (left) linear (or regular) iff
  - all rules R are of the form
    - $A \rightarrow w$ or $A \rightarrow wB$ *(or* $A \rightarrow Bw$*), where* $A, B \in \Phi$ and $w \in \Sigma^*$
  - i.e., LHS: a single NT symbol; RHS: a (possibly empty) sequence of T symbols, optionally followed (preceded) by a NT symbol
- Example:

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

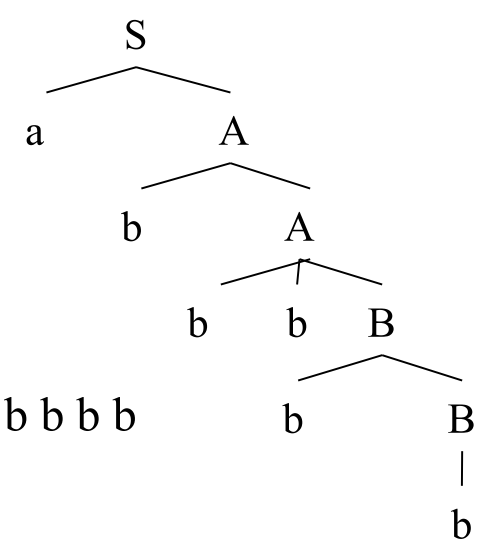
$\Phi = \{ S, A, B \}$

$R = \{ S \rightarrow a\, A, \quad B \rightarrow b\, B,$

$A \rightarrow a\, A, \quad B \rightarrow b$

$A \rightarrow b\, b\, B \quad \}$

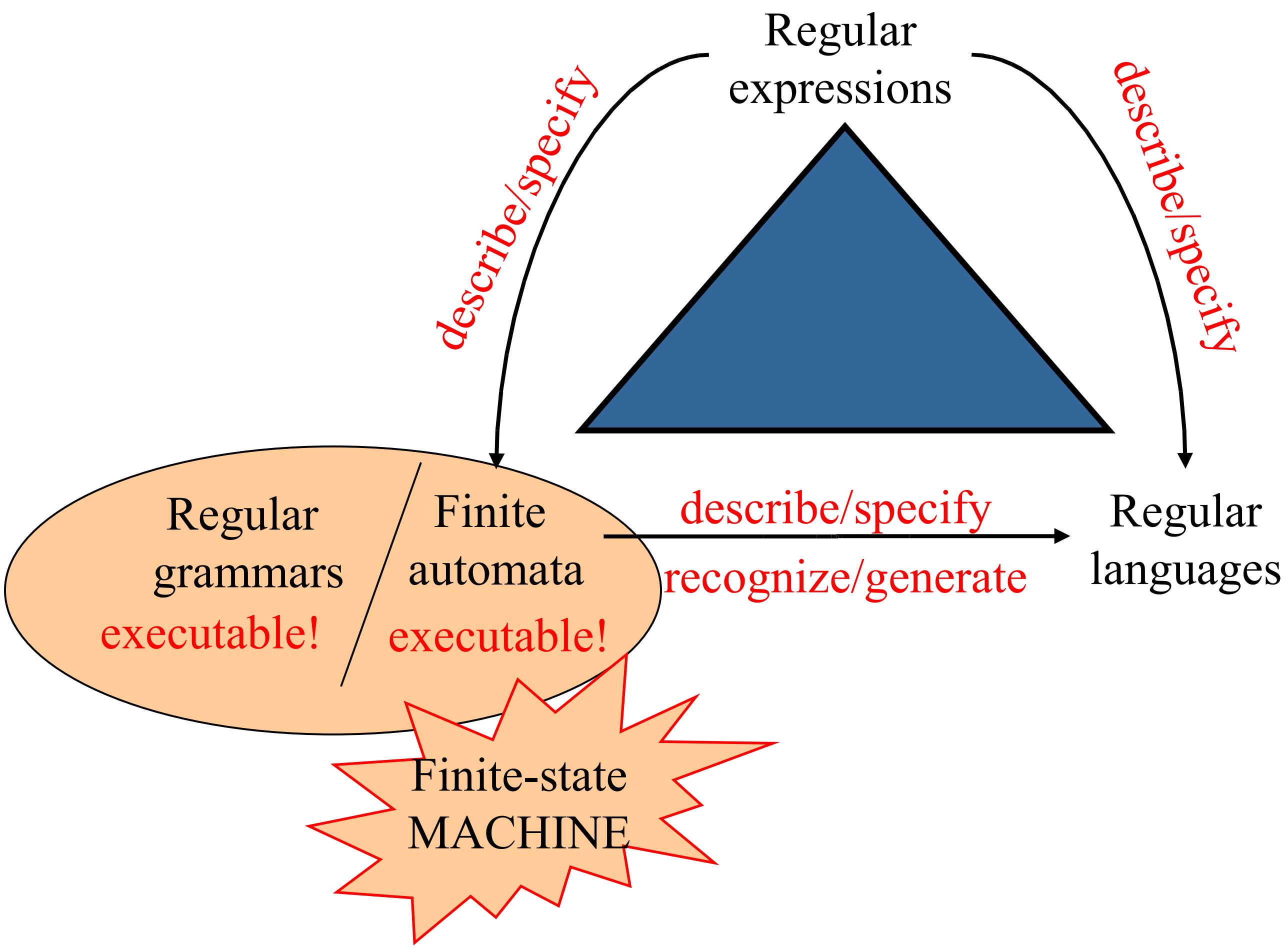
$S \Rightarrow a\, A \Rightarrow a\, a\, A \Rightarrow a\, a\, b\, b\, B \Rightarrow a\, a\, b\, b\, b\, B \Rightarrow a\, a\, b\, b\, b\, b$



Operations on languages

- Typical set-theoretic operations on languages
  - Union: $L_1 \cup L_2 = \{ w : w \in L_1 \text{ or } w \in L_2 \}$
  - Intersection: $L_1 \cap L_2 = \{ w : w \in L_1 \text{ and } w \in L_2 \}$
  - Difference: $L_1 - L_2 = \{ w : w \in L_1 \text{ and } w \notin L_2 \}$
  - Complement of $L \subseteq \Sigma^*$ wrt. Σ^*: $L^{-} = \Sigma^* - L$
- Language-theoretic operations on languages
  - Concatenation: $L_1L_2 = \{w_1w_2 : w_1 \in L_1 \text{ and } w_2 \in L_2\}$
  - Iteration: $L^0=\{\varepsilon\}$, $L^1=L$, $L^2=LL$, ... $L^*= \cup_{i\geq 0} L^i$, $L^+ = \cup_{i>0} L^i$
  - Mirror image: $L^{-1} = \{w^{-1} : w \in L\}$
- Union, concatenation and Kleene star are called regular operations
- Regular sets/languages: languages that are defined by the regular operations: concatenation $(\cdot)$, union $(\cup)$ and kleene star $(*)$
- Regular languages are *closed* under *concatenation, union, kleene star, intersection and complementation*

Regular languages, regular expressions and FSA

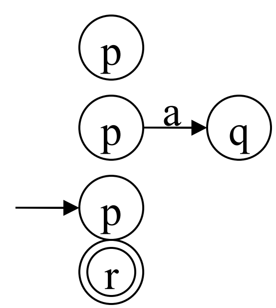


Regular languages and regular expressions

- Regular sets/languages can be specified/defined by regular expressions
  Given a set of terminal symbols Σ, the following are regular expressions
  - ε is a regular expression
  - For every $a \in \Sigma$, a is a regular expression
  - If R is a regular expression, then R* is a regular expression
  - If Q,R are regular expressions, then QR $(Q \cdot R)$ and $Q \cup R$ are regular expressions
- Every regular expression denotes a regular language
  - $L(\varepsilon) = \{\varepsilon\}$
  - $L(a) = \{a\}$ for all $a \in \Sigma$
  - $L(\alpha\beta) = L(\alpha)L(\beta)$
  - $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$
  - $L(\alpha^*) = L(\alpha)^*$

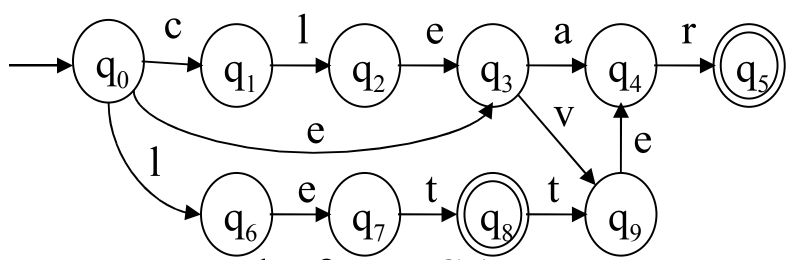
Finite-state automata (FSA)

- Grammars: generate (or recognize) languages
  Automata: recognize (or generate) languages
- Finite-state automata recognize regular languages
- A finite automaton (FA) is a tuple $A = <\Phi, \Sigma, \delta, q_0, F>$
  - Φ a finite non-empty set of states
  - $\mathbf{\Sigma}$ a finite alphabet of input letters
  - δ a transition function $\Phi \times \mathbf{\Sigma} \rightarrow \Phi$
  - $q_0 \in \Phi$ the initial state
  - $F \subseteq \Phi$ the set of final (accepting) states
- Transition graphs (diagrams):
  - states: circles — $p \in \Phi$
  - transitions: directed arcs between circles — $\delta(p, a) = q$
  - initial state — $p = q_0$
  - final state — $r \subseteq F$

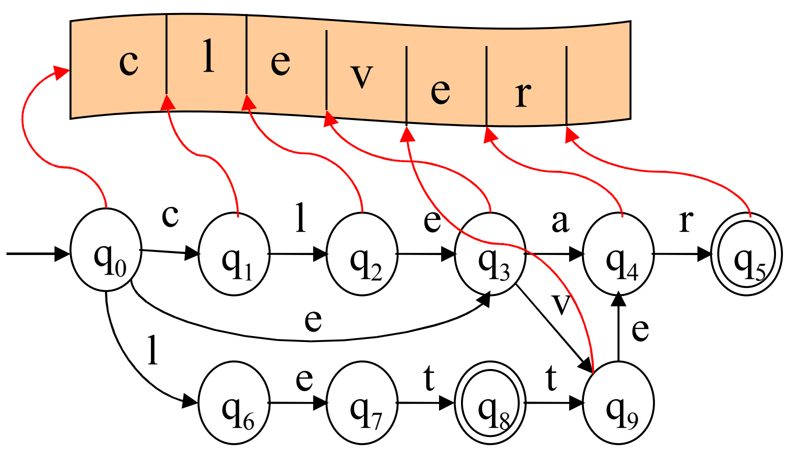


FSA transition graphs and traversal

- Transition graph



- Traversal of an FSA
  = Computation with an FSA



$S = q_0$ $F = \{q_5, q_8\}$

Transition function $\delta: \Phi \times \Sigma \rightarrow \Phi$

$\delta(q_0,c)=q_1$

$\delta(q_0,e)=q_3$

$\delta(q_0,l)=q_6$

$\delta(q_1,l)=q_2$

$\delta(q_2,e)=q_3$

$\delta(q_3,a)=q_4$

$\delta(q_3,v)=q_9$

$\delta(q_4,r)=q_5$

$\delta(q_6,e)=q_7$

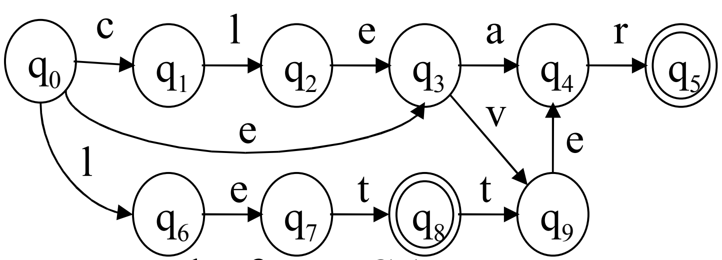
$\delta(q_7,t)=q_8$

$\delta(q_8,t)=q_9$

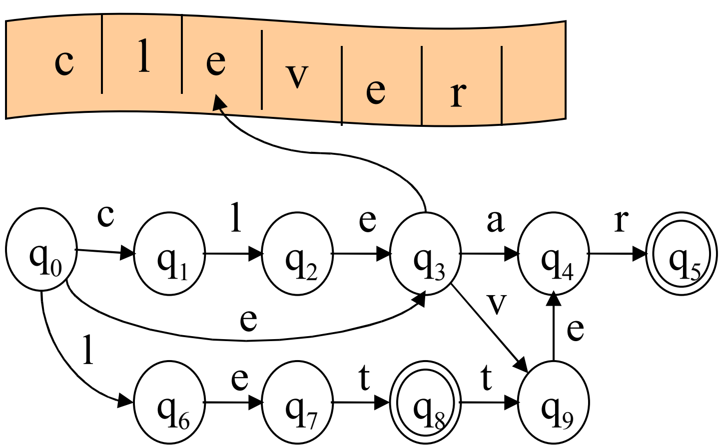
$\delta(q_9,e)=q_4$

FSA transition graphs and traversal

- Transition graph



- Traversal of an FSA
  = *Computation with an FSA*



State diagram

δ	a	c	e	l	r	t	v
q_0	0	q_1	q_3	q_6	0	0	0
q_1	0	0	0	q_2	0	0	0
q_2	0	0	q_3	0	0	0	0
q_3	q_4	0	0	0	0	0	q_9
q_4	0	0	0	0	q_5	0	0
q_5	0	0	0	0	0	0	0
q_6	0	0	q_7	0	0	0	0
q_7	0	0	0	0	0	q_8	0
q_8	0	0	0	0	0	q_9	0
q_9	0	0	q_4	0	0	0	0

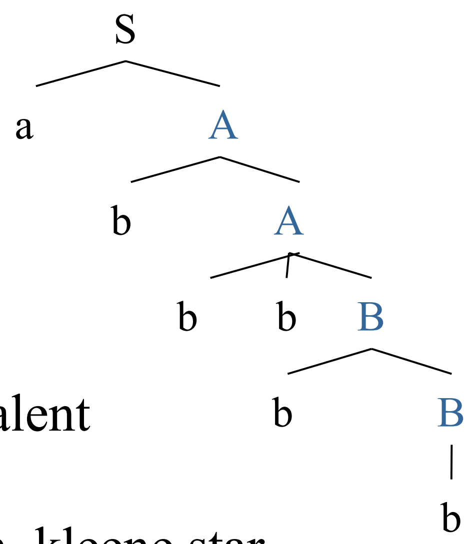
FSA can be used for
- acceptance (recognition)
- generation

FSA traversal and acceptance of an input string

- *Traversal* of a (deterministic) FSA
  - FSA *traversal* is defined by states and transitions of A, relative to an input string $w \in \Sigma^*$
  - A *configuration* of A is defined by the current state and the unread part of the input string: $(q, w_{i,})$, with $q \in \Phi$, w_i suffix of w
  - A *transition*: a binary relation between configurations
    $(q,w_i) \vdash_A (q',w_{i+1})$ iff $w_i = zw_{i+1}$ for $z \in \Sigma$ and $\delta(q,z)= q'$
    (q,w_i) *yields* (q',w_{i+1}) in a single transition step
  - Reflexive, transitive closure of $\vdash_A$: $(q, w_i) \vdash^*_A (q', w_j)$
    (q, w_i) *yields* (q', w_j) in zero or a finite number of steps
- *Acceptance*
  - Decide whether an input string *w* is in the language L(A) defined by FSA A
  - An FSA A *accepts* a string w iff $(q_0,w) \vdash^*_A (q_f, \varepsilon)$, with q_0 initial state, $q_f \subseteq F$
  - The *language L(A) accepted by FSA A* is the *set of all strings accepted by A*
    I.e., $w \in L(A)$ iff there is some $q_f \subseteq F_A$ such that $(q_0,w) \vdash^*_A (q_f, \varepsilon)$

Regular grammars and Finite-state automata

- A grammar G = <Σ, Φ, S, R> is called right linear (or regular) iff all rules R are of the form $A \rightarrow w$ or $A \rightarrow wB$, *where* $A,B \in \Phi$ and $w \in \Sigma^*$
  - $\Sigma=\{a, b\}$, $\Phi=\{S,A,B\}$, $R=\{S \rightarrow aA,\ A \rightarrow aA,\ A \rightarrow bbB,\ B \rightarrow bB,\ B \rightarrow b\}$
    $S \Rightarrow aA \Rightarrow aaA \Rightarrow aabbB \Rightarrow aabbbB \Rightarrow aabbbb$
  - The NT symbol corresponds to a state in an FSA: the future of the derivation only depends on the identity of this state or symbol and the remaining production rules.
  - *Correspondence of type-3 grammar rules with transitions in a (non-deterministic) FSA:*
    - $A \rightarrow w\ B \quad \equiv \quad \delta(A,w)=B$
    - $A \rightarrow w \quad \equiv \quad \delta(A,w)=q,\ q \in \Phi$
  - Conversely, we can construct an FSA from the rules of a type-3 language



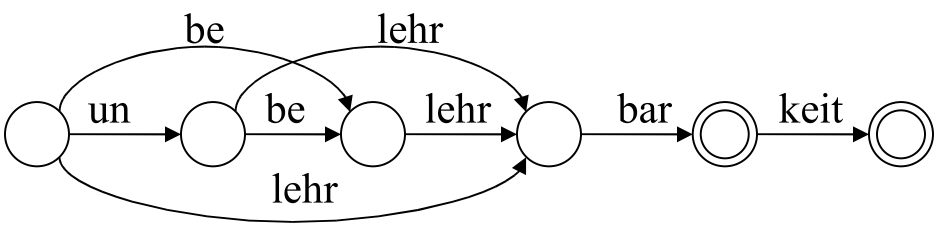
- Regular grammars and FSA can be shown to be equivalent
- Regular grammars generate regular languages
- Regular languages are defined by concatenation, union, kleene star

Deterministic finite-state automata

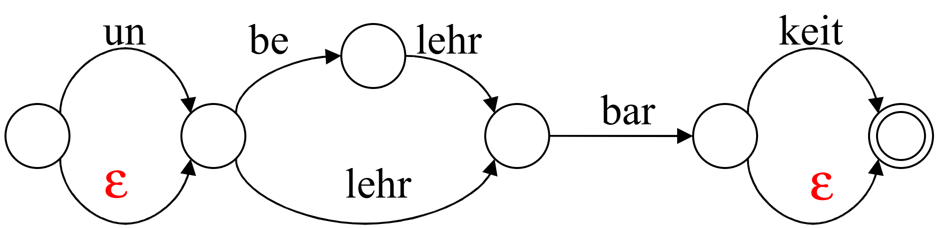
- Deterministic finite-state automata (DFSA)
  - at *each state*, there is *at most one transition* that can be taken to read the next input symbol
  - the next state (transition) is *fully determined by current configuration*
  - δ is functional (and there are no ε-transitions)
- Determinism is a useful property for an FSA to have!
  - Acceptance or rejection of an input can be computed in *linear time 0(n)* for inputs of length n
  - Especially important for processing of LARGE documents
- Appropriate problem classes for FSA
  - Recognition and acceptance of *regular languages*,
    in particular *string manipulation*, regular phonological and morphological processes
  - Approximations of non-regular languages in morphology, shallow finite-state parsing, …

Multiple equivalent FSA

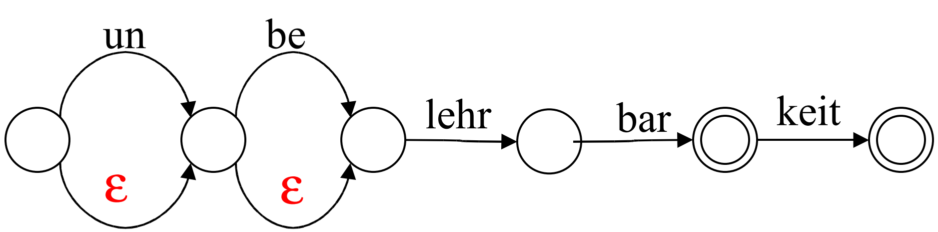
- FSA for the language L_{lehr} = { *lehrbar, lehrbarkeit, belehrbar, belehrbarkeit, unbelehrbar, unbelehrbarkeit, unlehrbar, unlehrbarkeit* }
- DFSA for L_{lehr}



- Regular expression and FSA for L_{lehr} : (un | ε) (be lehr | lehr) bar (keit | ε) (non-deterministic)



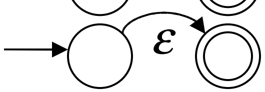
- Equivalent FSA (non-deterministic)



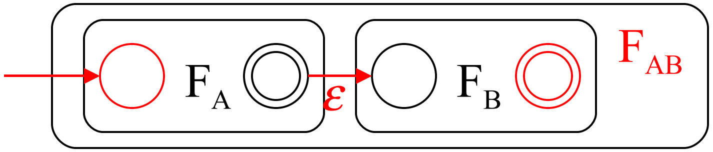
Defining FSA through regular expressions

- FSA for even mildly complex regular languages are best constructed from regular expressions!
- Every regular expression denotes a regular language
  - $L(\varepsilon) = \{\varepsilon\}$
  - $L(a) = \{a\}$ for all $a \in \Sigma$
    - $L(\alpha\beta) = L(\alpha)L(\beta)$
    - $L(\alpha \cup \beta) = L(\alpha) \cup L(\beta)$
    - $L(\alpha^*) = L(\alpha)^*$
- Every regular expression translates to a FSA (Closure properties)
  - An FSA for a (with $L(a) = \{a\}$), $a \in \Sigma$:

  - An FSA for ε (with $L(\varepsilon) = \{\varepsilon\}$), $\varepsilon \in \Sigma$:



  - Concatenation of two FSA F_A and F_B:



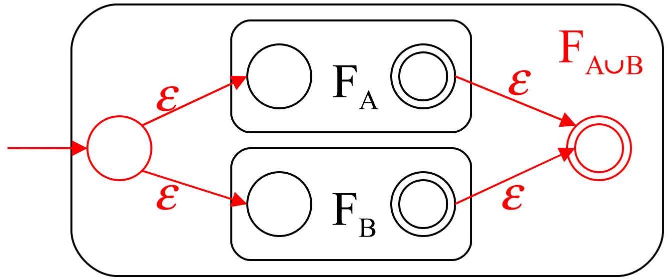
    - $\Sigma_{AB} = \Sigma_A$ (Σ initial state)
    - $\Phi_{AB} = \Phi_B$ (Φ set of final states)

$$\forall\ \delta_{AB} = \delta_A \cup \delta_B \cup \{\delta(<q_i,\varepsilon>,q_j) \mid q_i \in \Phi_A,\ q_j = \Sigma_B\}$$

Defining FSA through regular expressions

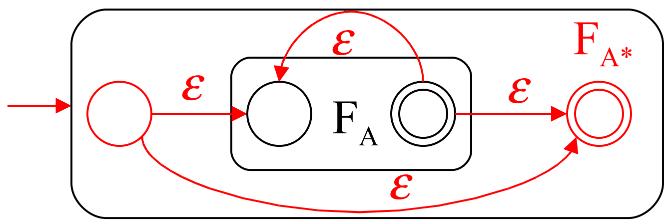
- union of two FSA F_A and F_B:
  - $S_{AB} = s_0$ (new state)
  - $F_{AB} = \{ s_j \}$ (new state)

  $\forall\ \delta_{AB} = \delta_A \cup \delta_B$
  $\cup\ \{\delta(<q_0,\varepsilon>,q_z) \mid q_0 = S_{AB}, (q_z = S_A \text{ or } q_z = S_B)\}$
  $\cup\ \{\delta(<q_z,\varepsilon>,q_j) \mid (q_z \in F_A \text{ or } q_z \in F_B), q_i \in F_{AB}\}$



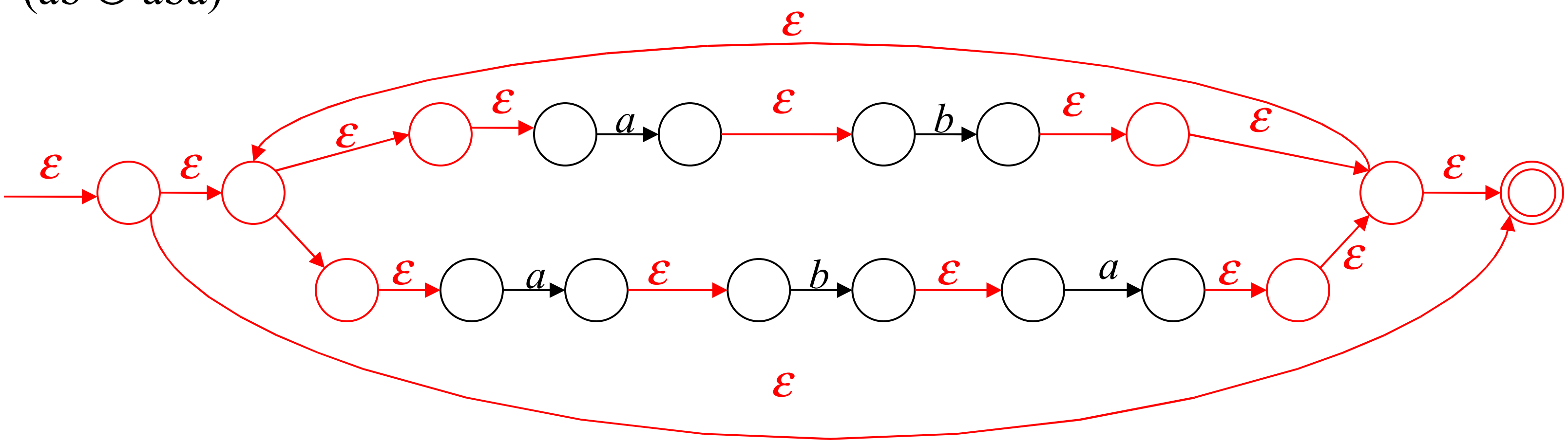
- Kleene Star over an FSA F_A :
  - $S_{A*} = s_0$ (new state)
  - $F_{A*} = \{ q_j \}$ (new state)

  $\forall\ \delta_{AB} = \delta_A \cup$
  $\cup\ \{\delta(<q_j,\varepsilon>,q_z) \mid q_j \in F_A, q_z = S_A)\}$
  $\cup\ \{\delta(<q_0,\varepsilon>,q_z) \mid q_0 = S_{A*}, (q_z = S_A \text{ or } q_z = F_{A*})\}$
  $\cup\ \{\delta(<q_z,\varepsilon>,q_j) \mid q_z \in F_A , q_j \in F_{A*}\}$



Defining FSA through regular expressions

(ab ∪ aba)*



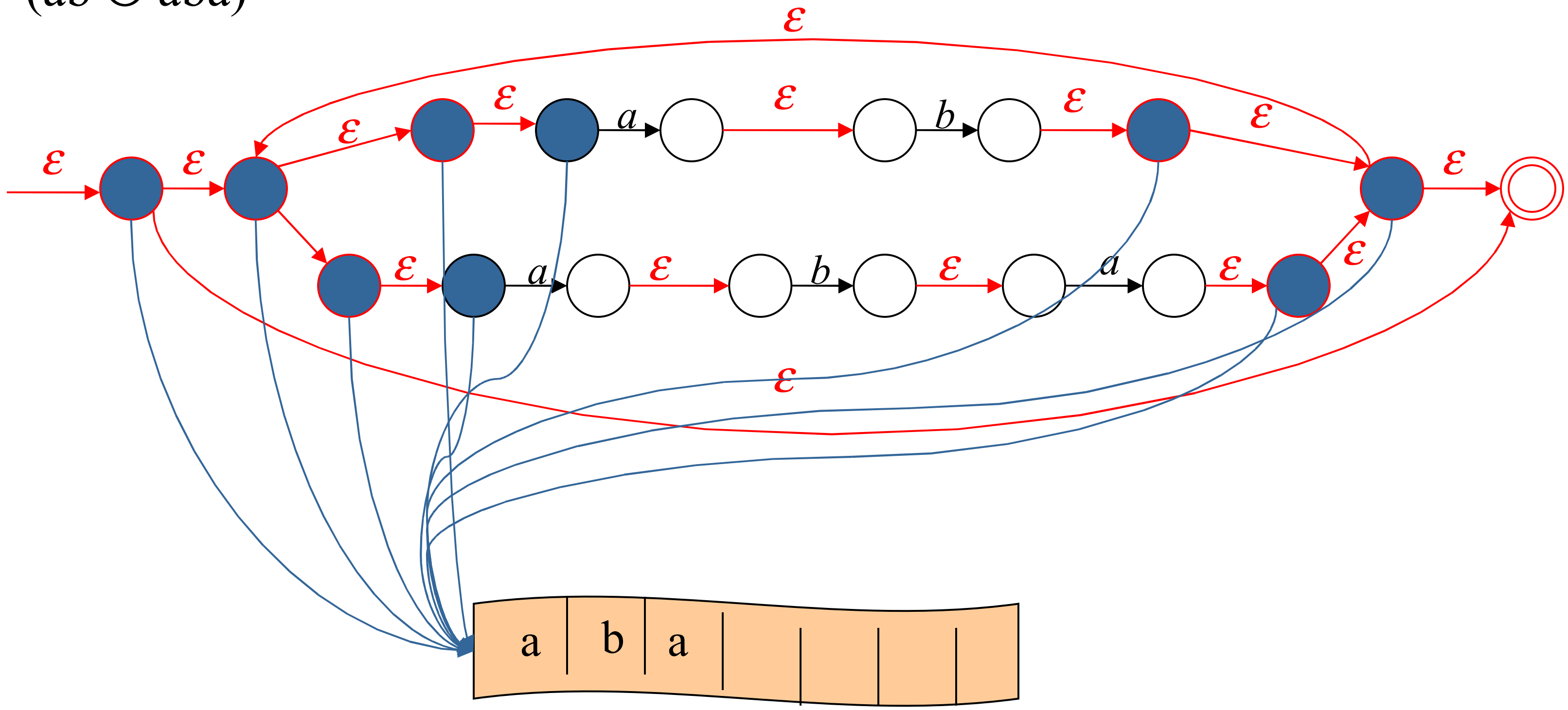
- *ε-transition:* move to δ(q, ε) without reading an input symbol
- FSA construction from regular expressions yields a non-deterministic FSA (NFSA)
  - Choice of next state is *only partially determined* by the current configuration, i.e., we cannot always predict which state will be the next state in the traversal

Non-deterministic finite-state automata (NFSA)

- *Non-determinism*
  - Introduced by ε-transitions and/or
  - Transition being a *relation* Δ over $\Phi \times \Sigma^* \times \Phi$, i.e. a set of triples $<q_{source},z,q_{target}>$
    Equivalently: Transition function δ maps to a *set of states*: $\delta: \Phi \times \Sigma \to \wp(\Phi)$
- A non-deterministic FSA (NFSA) is a tuple $A = <\Phi,\Sigma, \delta, q_0,F>$
  - Φ a finite non-empty set of states
  - $\boldsymbol{\Sigma}$ a finite alphabet of input letters
  - δ a transition function $\Phi \times \boldsymbol{\Sigma}^* \to \wp(\Phi)$ (or a finite relation over $\Phi \times \Sigma^* \times \Phi$)
  - $q_0 \in \Phi$ the initial state
  - $F \subseteq \Phi$ the set of final (accepting) states
- Adapted definitions for *transitions and acceptance* of a string by a NFSA
  - $(q,w) \vdash_A (q',w_{i+1})$ iff $w_i = zw_{i+1}$ for $z \in \Sigma^*$ and $q' \in \delta(q,z)$
  - An NDFA (w/o ε) *accepts* a string w iff there is *some traversal* such that $(q_0,w) \vdash^*_A (q', \varepsilon)$ and $q' \subseteq F$.
  - A string w is *rejected* by NDFA A iff A does not accept w,
    i.e. *all configurations* of A for string w are rejecting configurations!

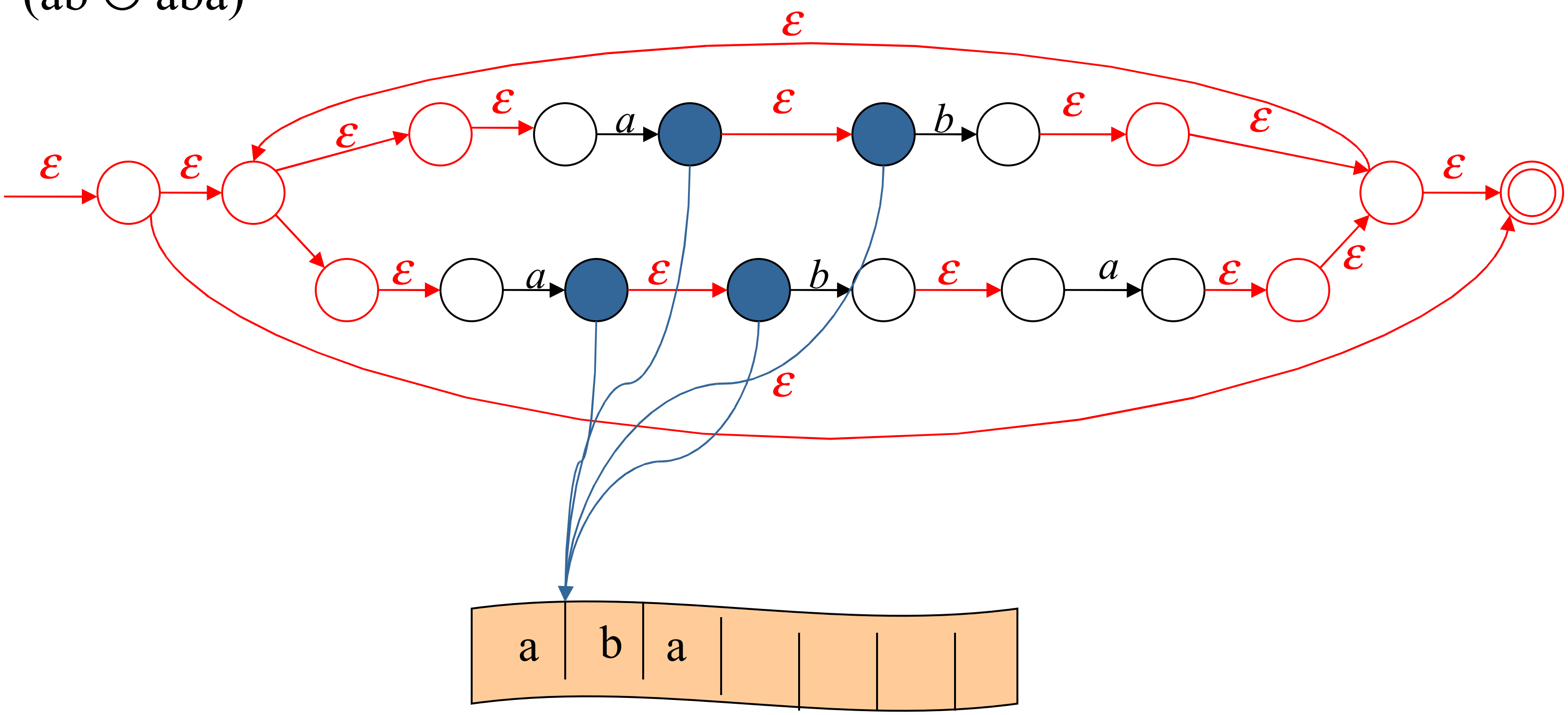
Non-determinism in FSA

(ab ∪ aba)*



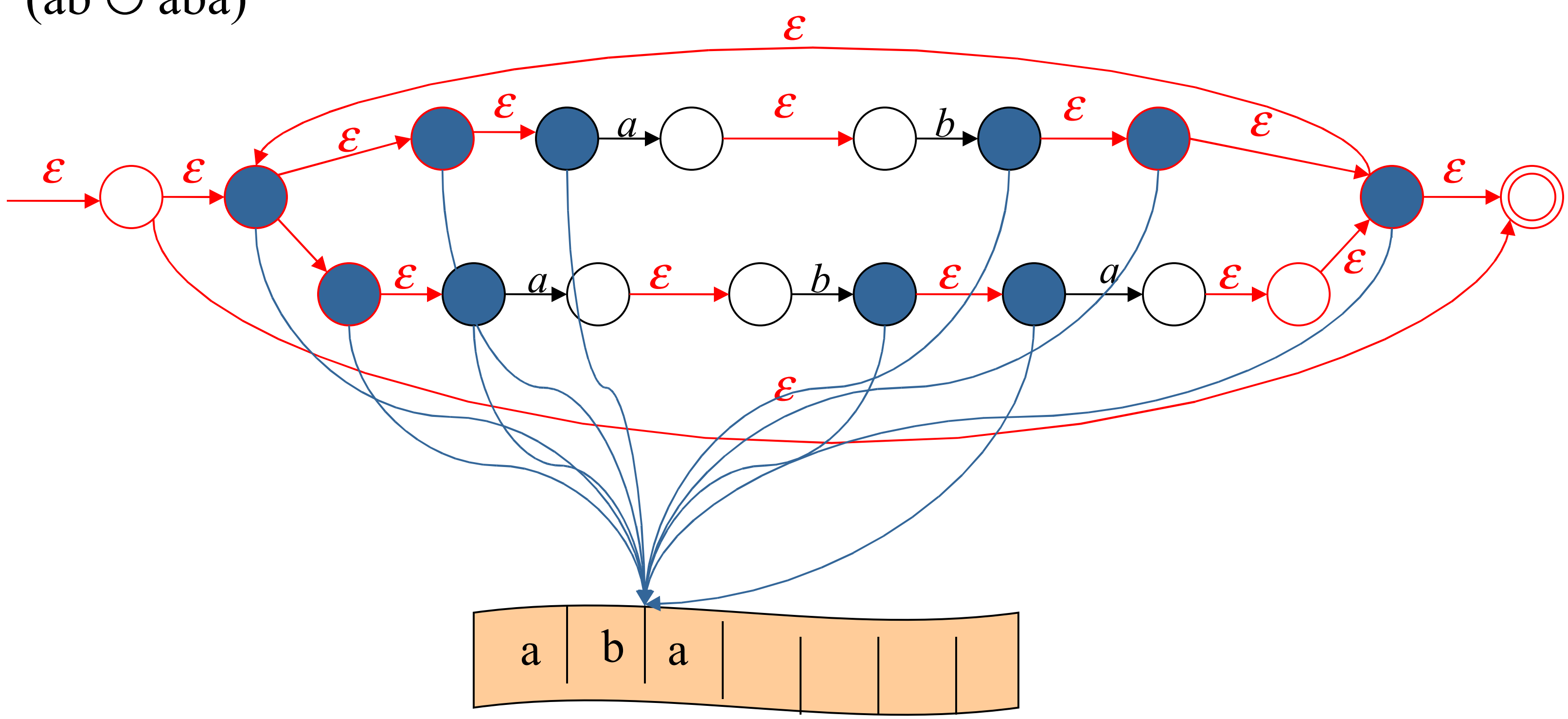
Non-determinism in FSA

(ab ∪ aba)*



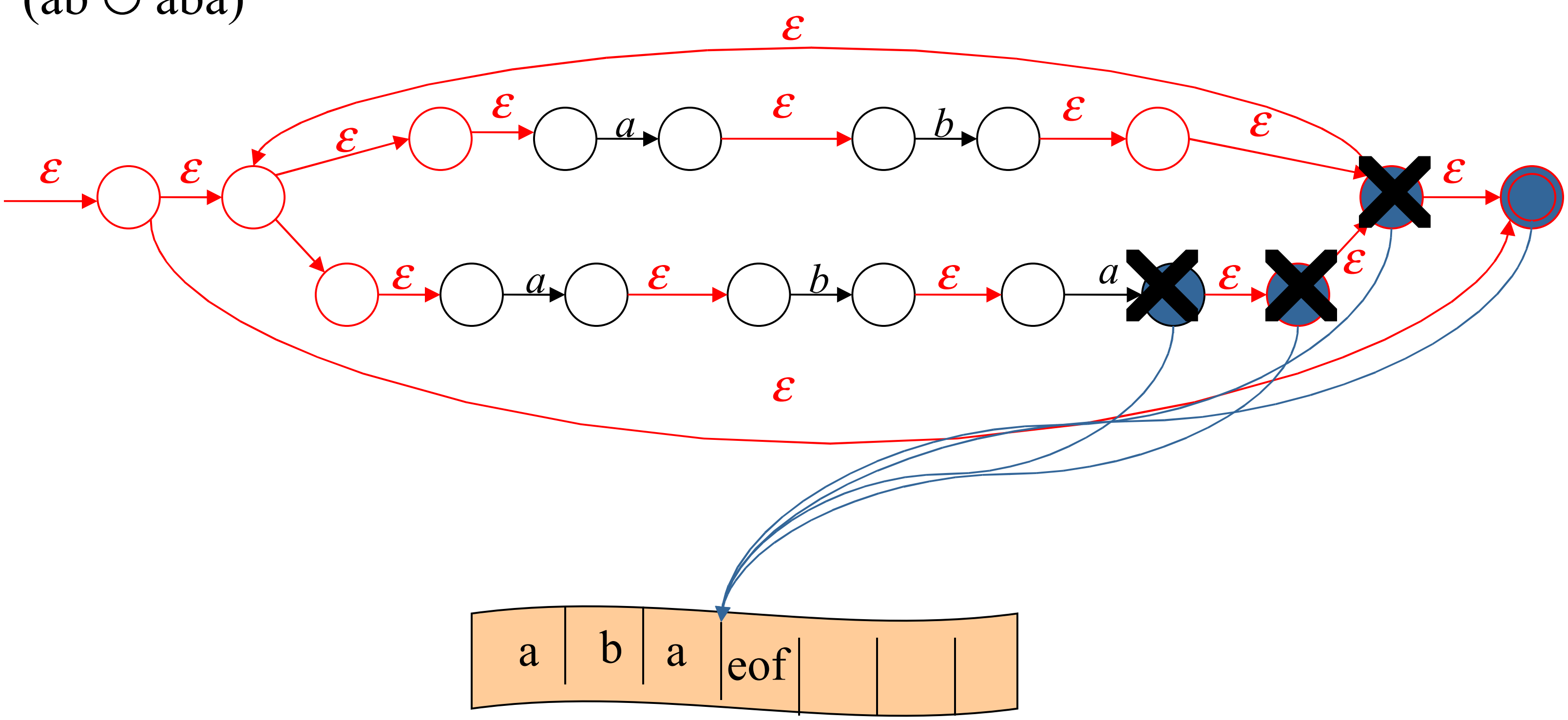
Non-determinism in FSA

(ab ∪ aba)*



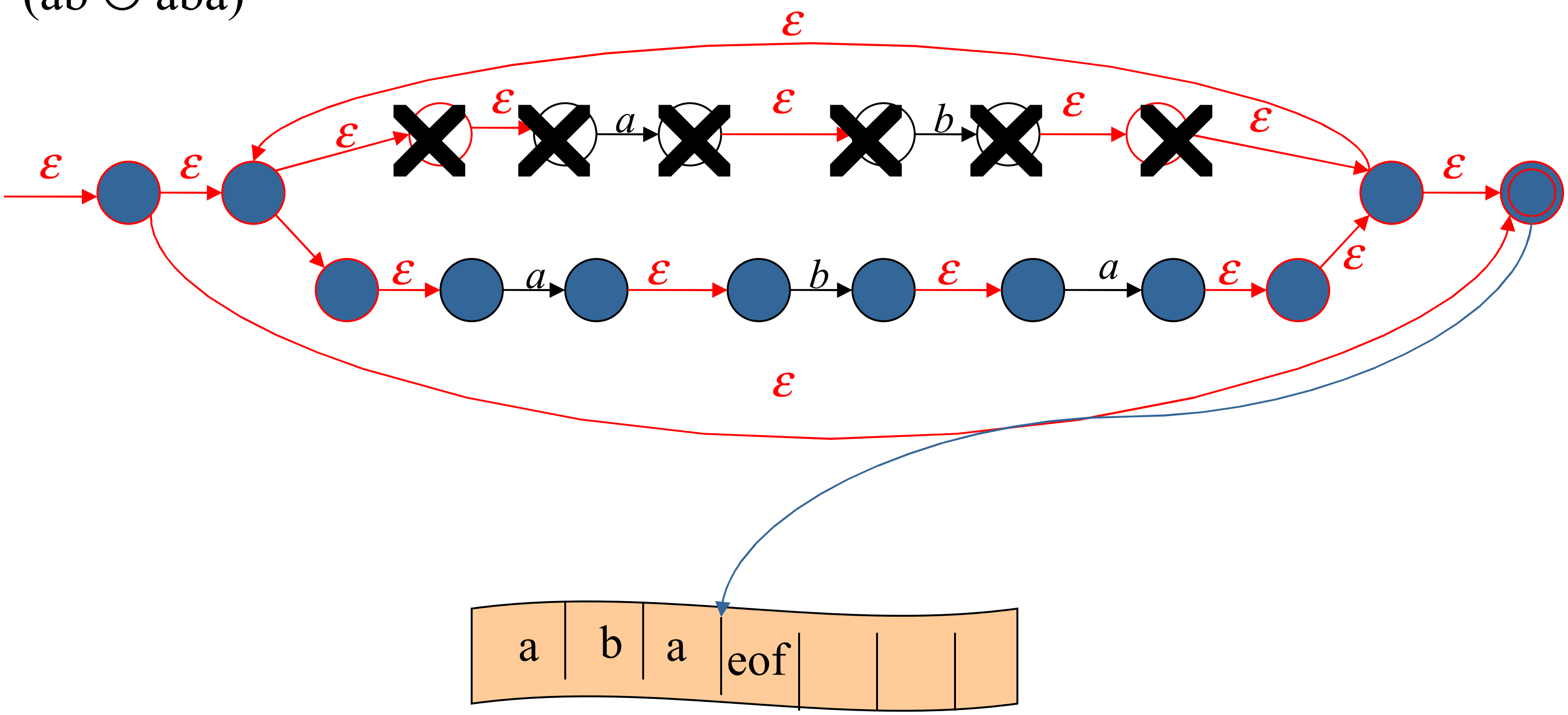
Non-determinism in FSA

(ab ∪ aba)*



Non-determinism in FSA

(ab ∪ aba)*



Equivalence of DFSA and NFSA

- Despite non-determinism, NFSA are not more powerful than DFSA: they accept the same class of languages: regular languages
- For every non-deterministic FSA there is deterministic FSA that accepts the same language (and vice versa)
  - The corresponding DFSA has in general more states, in which it models the sets of possible states the NFSA could be in in a given traversal
- There is an algorithm (via subset construction) that allows conversion of an NFSA to an equivalent DFSA

Efficiency considerations: an FSA is most efficient and compact iff

- It is a DFSA (efficiency) → Determinization of NFSA
- It is minimal (compact encoding) → Minimization of FSA

Equivalence of DFSA and NFSA

- FSA A_1 and A_2 are equivalent iff $L(A_1) = L(A_2)$
- Theorem: for each NFSA there is an equivalent DFSA
- Construction: $A = < \Phi, \Sigma, \delta, q_0, F >$ a NFSA over Σ
  - define $eps(q) = \{ p \in \Phi \mid (q, \varepsilon, p) \in \delta \}$
  - define an FSA $A' = <\Phi', \Sigma, \delta', q_0', F'>$ over sets of states, with

    $\Phi' = \{B \mid B \subseteq \Phi\}$

    $q_0' = \{eps(q_0)\}$

    $\delta'(B,a) = \cup \{ eps(p) \mid q \in B \text{ and } \exists\, p \in B \text{ such that } (q, a, p) \in \delta \}$

    $F' = \{B \subseteq \Phi \mid B \cap F \neq \emptyset\}$
- A' satisfies the definition of a DFSA. We need to show that $L(A) = L(A')$
- Define $D(q, w) := \{ p \in \Phi \mid (q, w) \vdash^*_A (p, \varepsilon) \}$ and

$$D'(Q, w) := \{ P \in \Phi' \mid (Q, w) \vdash^*_{A'} (P, \varepsilon) \}$$

Equivalence of DFSA and NFSA: Proof

Prove: $D(q_0, w) = D'(\{q_0\}, w)$ by induction over length of w

- $|w| = 0$: by definition of D and D'
- Induction step: $|w| = k+1$, $w = v$ a, by hypothesis:

  $D(q_0, v) = D'(\{q_0\}, v) = \{p_1, p_2, \dots , p_k\} = P$

  by def. of D: $D(q_0, w) = \bigcup_{p \in P} \{eps(q) \mid (p, a, q) \in \delta\}$

  by def. of δ': $D'(\{p_1, p_2, \dots , p_k\}, a) = \bigcup_{p \in P} \{eps(q) \mid (p, a, q) \in \delta\}$

  it follows:

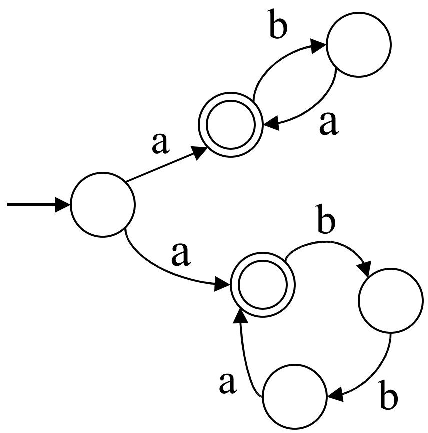
  $D'(\{q_0\}, w) = \delta'(D'(\{q_0\}, w), a) = D'(\{p_1, p_2, \dots , p_k\}, a)$

  $= \bigcup_{p \in P} \{eps(q) \mid (p, a, q) \in \delta\} = D(q_0, w)$ q.e.d.
- Finally, A and A' only accept if $D'(\{q_0\}, w) = D(q_0, w)$ contain a state $\in F$

Determinization by subset construction

NFSA $A=\langle\Phi,\Sigma,\delta,q_0,F\rangle$

$A'=\langle\Phi',\Sigma,\delta',q_0',F'\rangle$



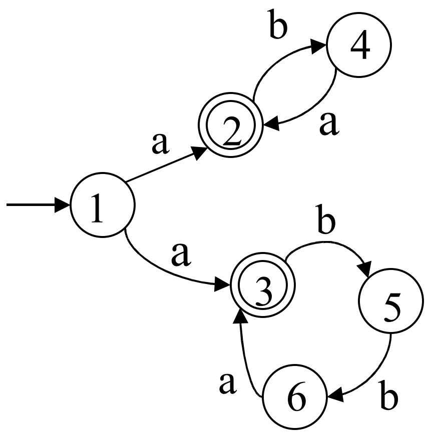
Subset construction:

Compute δ' from δ
for all subsets $S \subseteq \Phi$ and $a \in \Sigma$ s.th.
$\delta'(S,a) = \{ s' \mid \exists s \in S \text{ s.th. } (s,a,s') \in \delta \}$

$L(A) = a(ba)^* \cup a(bba)^*$

Determinization by subset construction

NFSA $A=<\Phi,\Sigma,\delta,q_0,F>$ $A'=<\Phi',\Sigma,\delta',q_0',F'>$



$L(A)=a(ba)^* \cup a(bba)^*$

$\Phi' = \{ B \mid B \subseteq \{1,2,3,4,5,6\}$

$q_0'=\{1\}$,

$\delta'(\{1\},a)=\{2,3\}$,

$\delta'(\{1\},b)=\emptyset$,

$\delta'(\{2,3\},a)=\emptyset$,

$\delta'(\{2,3\},b)=\{4,5\}$,

$\delta'(\{4,5\},a)=\{2\}$,

$\delta'(\{4,5\},b)=\{6\}$,

$\delta'(\{2\},a)=\emptyset$,

$\delta'(\{2\},b)=\{4\}$,

$\delta'(\{6\},a)=\{3\}$,

$\delta'(\{6\},b)=\emptyset$,

$\delta'(\{4\},a)=\{2\}$,

$\delta'(\{4\},b)=\emptyset$,

$\delta'(\{3\},a)=\emptyset$,

$\delta'(\{3\},b)=\{5\}$,

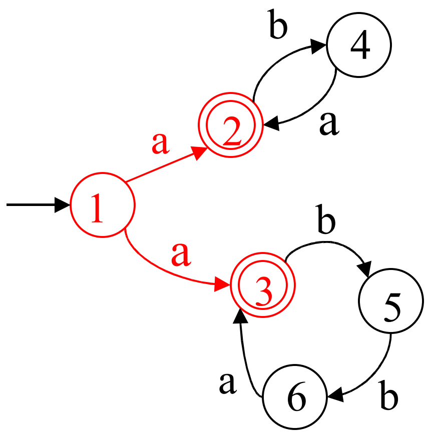
$\delta'(\{5\},a)=\emptyset$,

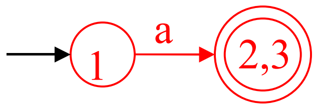
$\delta'(\{5\},b)=\{6\}$

$F'=\{\{2,3\},\{2\},\{3\}\}$

Determinization by subset construction

NFSA $A=<\Phi,\Sigma,\delta,q_0,F>$ $A'=<\Phi',\Sigma,\delta',q_0',F'>$





$\Phi' = \{ B \mid B \subseteq \{1,2,3,4,5,6\}$

$q_0' = \{1\}$,

$\delta'(\{1\},a)=\{2,3\}$,

$\delta'(\{1\},b)=\varnothing$,

$\delta'(\{2,3\},a)= \varnothing$,

$\delta'(\{2,3\},b)= \{4,5\}$,

$\delta'(\{4,5\},a)= \{2\}$,

$\delta'(\{4,5\},b)= \{6\}$,

$\delta'(\{2\},a)= \varnothing$,

$\delta'(\{2\},b)= \{4\}$,

$\delta'(\{6\},a)= \{3\}$,

$\delta'(\{6\},b)= \varnothing$,

$\delta'(\{4\},a)= \{2\}$,

$\delta'(\{4\},b)= \varnothing$,

$\delta'(\{3\},a)= \varnothing$,

$\delta'(\{3\},b)= \{5\}$,

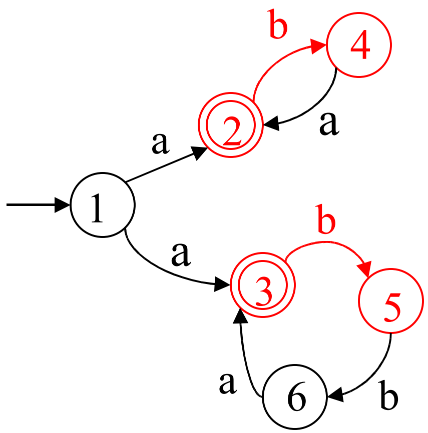
$\delta'(\{5\},a)= \varnothing$,

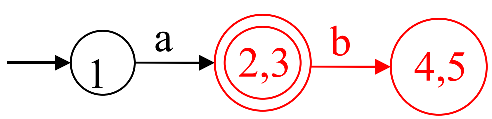
$\delta'(\{5\},b)= \{6\}$

$F'= \{\{2,3\},\{2\},\{3\}\}$

Determinization by subset construction

NFSA $A=<\Phi,\Sigma,\delta,q_0,F>$ $A'=<\Phi',\Sigma,\delta',q_0',F'>$





$\Phi' = \{ B \mid B \subseteq \{1,2,3,4,5,6\}$

$q_0' = \{1\}$,

$\delta'(\{1\},a)=\{2,3\}$,

$\delta'(\{1\},b)=\varnothing$,

$\delta'(\{2,3\},a)=\varnothing$,

$\delta'(\{2,3\},b)=\{4,5\}$,

$\delta'(\{4,5\},a)=\{2\}$,

$\delta'(\{4,5\},b)=\{6\}$,

$\delta'(\{2\},a)=\varnothing$,

$\delta'(\{2\},b)=\{4\}$,

$\delta'(\{6\},a)=\{3\}$,

$\delta'(\{6\},b)=\varnothing$,

$\delta'(\{4\},a)=\{2\}$,

$\delta'(\{4\},b)=\varnothing$,

$\delta'(\{3\},a)=\varnothing$,

$\delta'(\{3\},b)=\{5\}$,

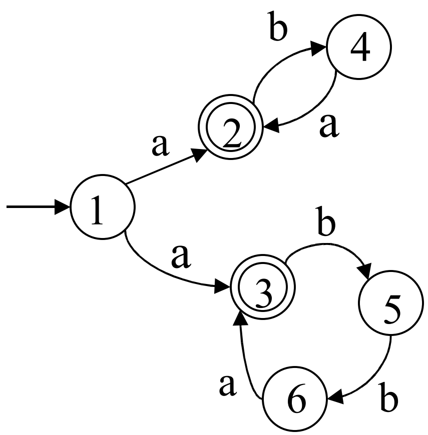
$\delta'(\{5\},a)=\varnothing$,

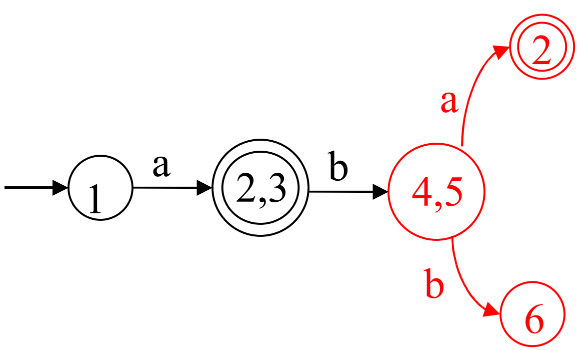
$\delta'(\{5\},b)=\{6\}$

$F' = \{\{2,3\},\{2\},\{3\}\}$

Determinization by subset construction

NFSA $A=\langle\Phi,\Sigma,\delta,q_0,F\rangle$





$A'=\langle\Phi',\Sigma,\delta',q_0',F'\rangle$

$\Phi' = \{ B \mid B \subseteq \{1,2,3,4,5,6\}$

$q_0'=\{1\}$,

$\delta'(\{1\},a)=\{2,3\}$,

$\delta'(\{1\},b)=\varnothing$,

$\delta'(\{2,3\},a)=\varnothing$,

$\delta'(\{2,3\},b)=\{4,5\}$,

$\delta'(\{4,5\},a)=\{2\}$,

$\delta'(\{4,5\},b)=\{6\}$,

$\delta'(\{2\},a)=\varnothing$,

$\delta'(\{2\},b)=\{4\}$,

$\delta'(\{6\},a)=\{3\}$,

$\delta'(\{6\},b)=\varnothing$,

$\delta'(\{4\},a)=\{2\}$,

$\delta'(\{4\},b)=\varnothing$,

$\delta'(\{3\},a)=\varnothing$,

$\delta'(\{3\},b)=\{5\}$,

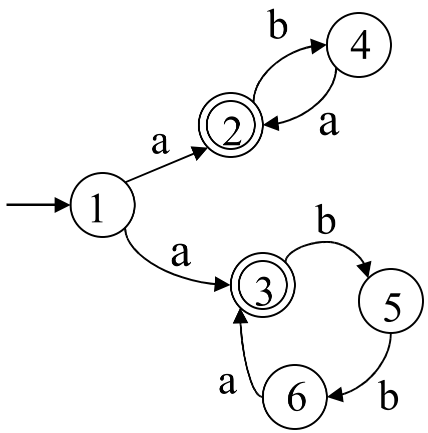
$\delta'(\{5\},a)=\varnothing$,

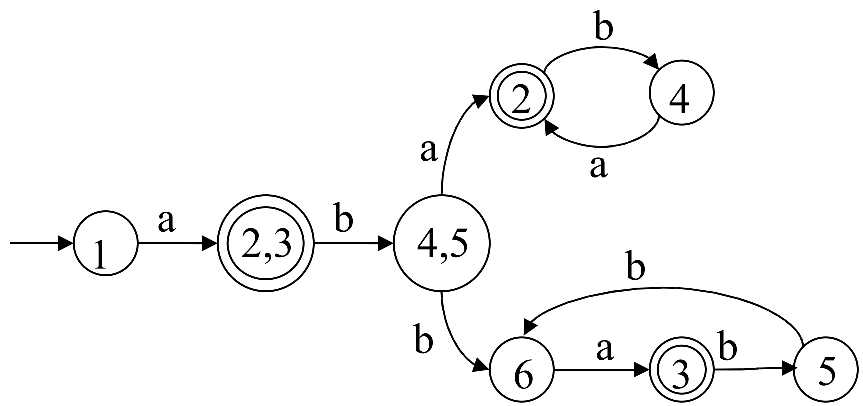
$\delta'(\{5\},b)=\{6\}$

$F'=\{\{2,3\},\{2\},\{3\}\}$

Determinization by subset construction

NFSA $A=<\Phi,\Sigma,\delta,q_0,F>$ DFSA $A'=<\Phi',\Sigma,\delta',q_0',F'>$





$L(A) = L(A') = a(ba)^* \cup a(bba)^*$

ε-transitions and ε-closure

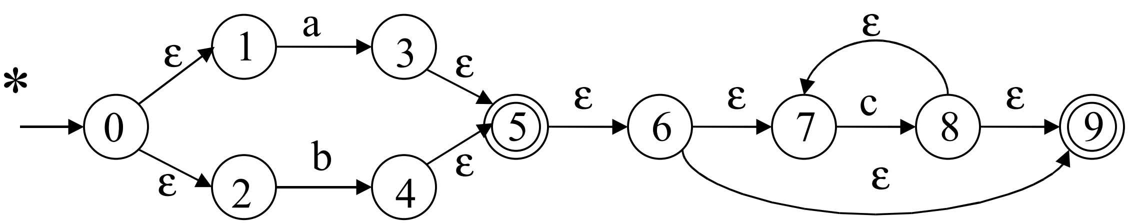
- Subset construction must account for ε-transitions
- ε-closure
  - The ε-closure of some state q consists of q as well as all states that can be reached from q through a sequence of ε-transitions
    - $q \in \varepsilon\text{–closur}\varepsilon(q)$
    - If $r \in \varepsilon\text{–closure}(q)$ and $(r, \varepsilon, q') \in \delta$, then $q' \in \varepsilon\text{–closure}(q)$,
  - ε-closure defined on sets of states

$$\forall\ \varepsilon\text{-closure}(R) = \bigcup_{q \in R} \varepsilon\text{-closure}(q) \qquad (\text{with } P \subseteq \Phi)$$

- Subset construction for ε-NFSA
  - Compute δ’ from δ for all subsets $S \subseteq \Phi$ and $a \in \Sigma$ s.th.
    $\delta'(S,a) = \{\ s'' \mid \exists s \in S \text{ s.th. } (s,a,s') \in \delta \text{ and } s'' \in \varepsilon\text{-closure}(s')\ \}$

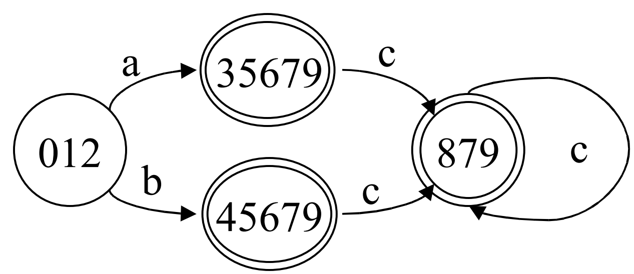
Example

- ε-NFSA for (a|b)c*



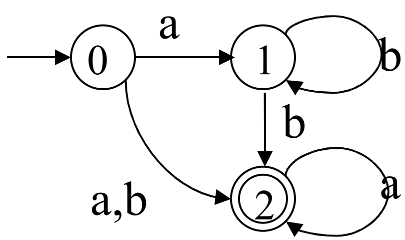
ε-closure for all s∈ Φ:
ε-closure(0)={0,1,2},
ε-closure(1)={1},
ε-closure(2)={2},
ε-closure(3)={3,5,6,7,9},
ε-closure(4)={4,5,6,7,9},
ε-closure(5)={5,6,7,9},
ε-closure(6)={6,7,9},
ε-closure(7)={7},
ε-closure(8)={8,7,9},
ε-closure(9)={9}

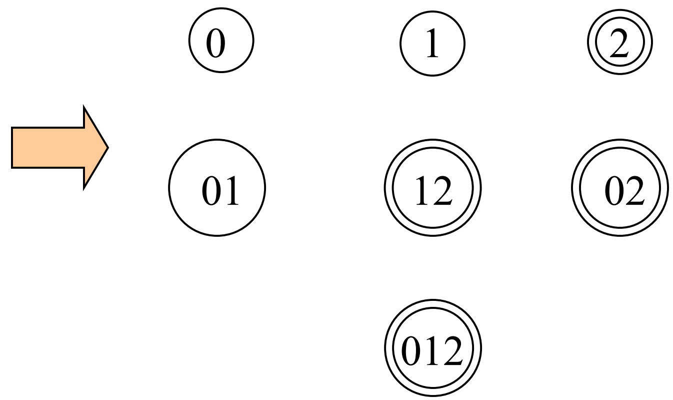
Transition function over subsets
δ'({0},ε)= {0,1,2},
δ'({0,1,2},a)={3,5,6,7,9},
δ'({0,1,2},b)= {4,5,6,7,9},
δ'({3,5,6,7,9},c)= {8,7,9},
δ'({4,5,6,7,9},c)= {8,7,9},
δ'({8,7,9},c)= {8,7,9}



An algorithm for subset construction

- Construction of DFSA $A' = <\Phi', \Sigma, \delta', q_0', F'>$ from NFSA $A = <\Phi, \Sigma, \delta, q_0, F>$
  - $\Phi' = \{B \mid B \subseteq \Phi\}$, if unconstrained can be $2^{|\Phi|}$
    with $|\Phi| = 33$ this could lead to an FSA with 2^{33} states
    (exceeds the range of integers in most programming languages)
  - Many of these states may be useless

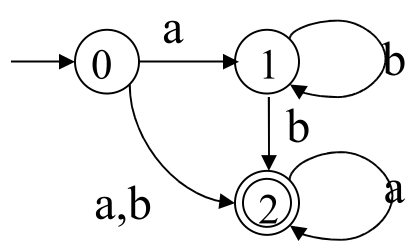


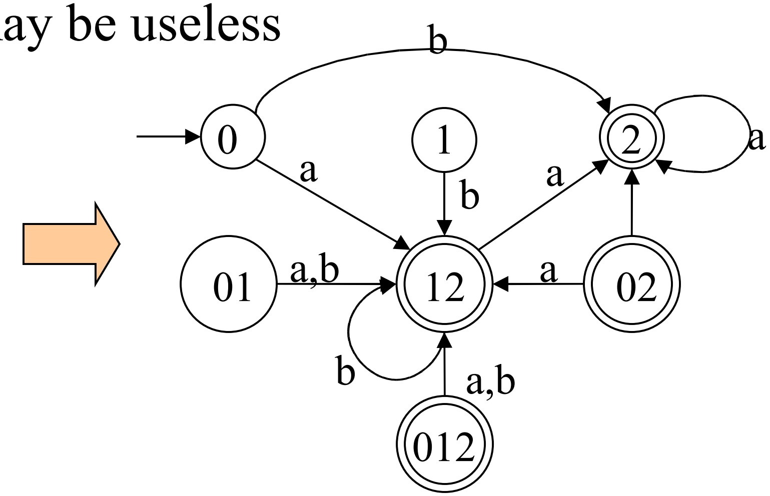


L= (a|b)a* ∪ ab+a*

An algorithm for subset construction

- Construction of DFSA $A' = \langle \Phi', \Sigma, \delta', q_0, F' \rangle$ from NFSA $A = \langle \Phi, \Sigma, \delta, q_0, F \rangle$
  - $\Phi' = \{B \mid B \subseteq \Phi\}$, if unconstrained can be $2^{|\Phi|}$
    with $|\Phi| = 33$ this could lead to an FSA with 2^{33} states
    (exceeds the range of integers in many programming languages)
  - Many of these states may be useless

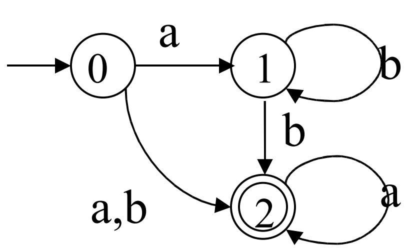


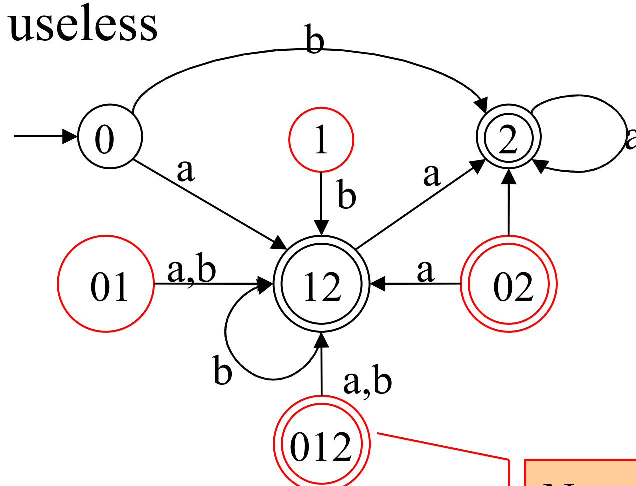


L= (a|b)a* ∪ ab+a*

An algorithm for subset construction

- Construction of DFSA $A'=<\Phi',\Sigma,\delta',q_0,F'>$ from NFSA $A=<\Phi,\Sigma,\delta,q_0,F>$
  - $\Phi'=\{B|\ B\subseteq\Phi\}$, if unconstrained can be $2^{|\Phi|}$
    with $|\Phi| = 33$ this could lead to an FSA with 2^{33} states
    (exceeds the range of integers in many programming languages)
  - Many of these states may be useless



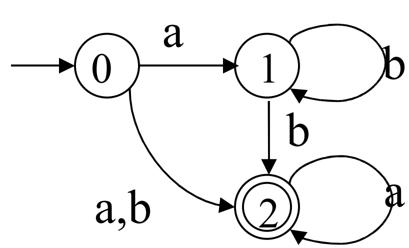


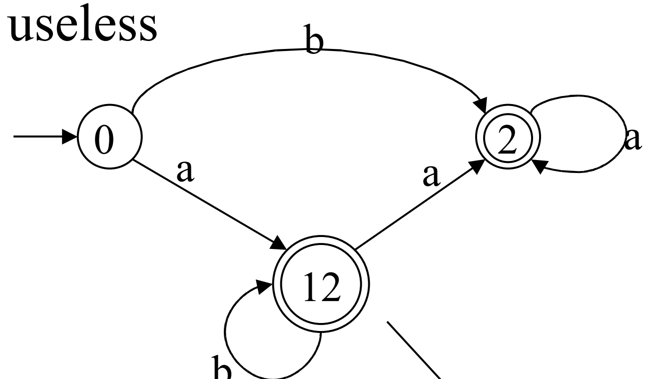
L= (a|b)a* ∪ ab+a*

No transition can ever enter these states

An algorithm for subset construction

- Construction of DFSA $A' = \langle \Phi', \Sigma, \delta', q_0, F' \rangle$ from NFSA $A = \langle \Phi, \Sigma, \delta, q_0, F \rangle$
  - $\Phi' = \{B \mid B \subseteq \Phi\}$, if unconstrained can be $2^{|\Phi|}$
    with $|\Phi| = 33$ this could lead to an FSA with 2^{33} states
    (exceeds the range of integers in many programming languages)
  - Many of these states may be useless



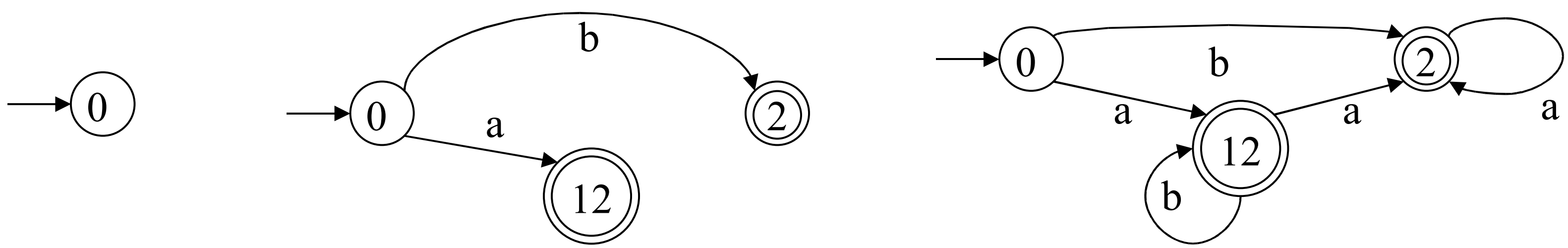


$L = (a|b)a^* \cup ab^+a^*$

Only consider states that can be traversed starting from q_0

An algorithm for subset construction

- Basic idea: we only need to consider states $B \subseteq \Phi$ that can ever be traversed by a string $w \in \Sigma^*$, starting from q_0'
- I.e., those $B \subseteq \Phi$ for which $B = \delta'(q_0,w)$, for some $w \in \Sigma^*$, with δ' the recursively constructed transition function for the target DFSA A'
- Consider all strings $w \in \Sigma^*$ in order of their length: ε, a,b, aa,ab,ba,bb, aaa,...



l=0 (ε)   l=1 (a,b)   l=2,3,4, … (aa, ab, ba, bb, aaa, aab, aba, …)

- Construction by increasing lengths of strings
- For each $a \in \Sigma$, construct transitions to known or new states according to δ
- New target states (A') are placed in a queue (FIFO)
- Termination: no states left on queue

An algorithm for subset construction

```
DETERMINIZE(Φ, Σ, δ, q0, F)
q0‘← q0
Φ’ ← {q0‘}
ENQUEUE(Queue, q0‘)
while Queue ≠ ∅
   S ← DEQUEUE(Queue)
   for a∈ Σ
     δ’(S,a) = ∪r∈ S δ(r,a)
     if  δ’(S,a) ∉  Φ’
        Φ’ ← Φ’ ∪ δ’(S,a)
        ENQUEUE(Queue, δ’(S,a))
        if  δ’(S,a) ∩ F ≠ ∅
           F‘ ← {δ’(S,a)}
        fi
     fi
return (Φ’,Σ, δ’, q0‘, F’)
```

Complexity

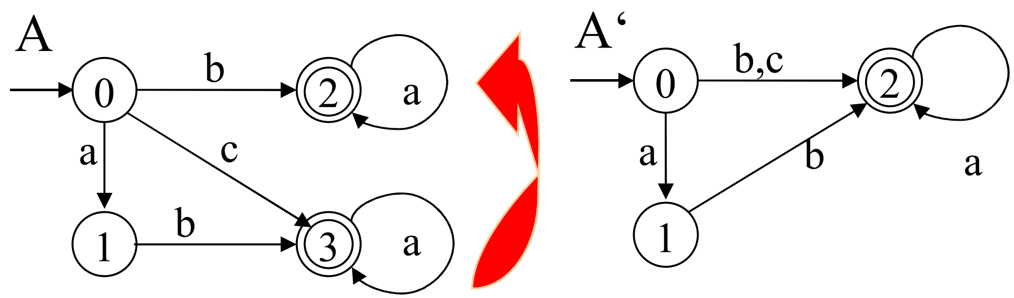
Maximal number of states placed in queue is $2^{|\Phi|}$
So, worst case runtime is exponential

- determinization is a costly operation,
- but results in an efficient FSA
  (linear in size of the input)
- avoids computation of isolated states

Actual run time depends on the shape of the NFSA

Minimization of FSA

- Can we transform a large automaton into a smaller one (provided a smaller one exists)?
- If A is a DFSA, is there an algorithm for constructing an equivalent minimal automaton A_{min} from A?



A is *equivalent* to A‘
i.e., L(A) = L(A‘)

A‘ is *smaller* than A
i.e., $|\Phi| > |\Phi'|$

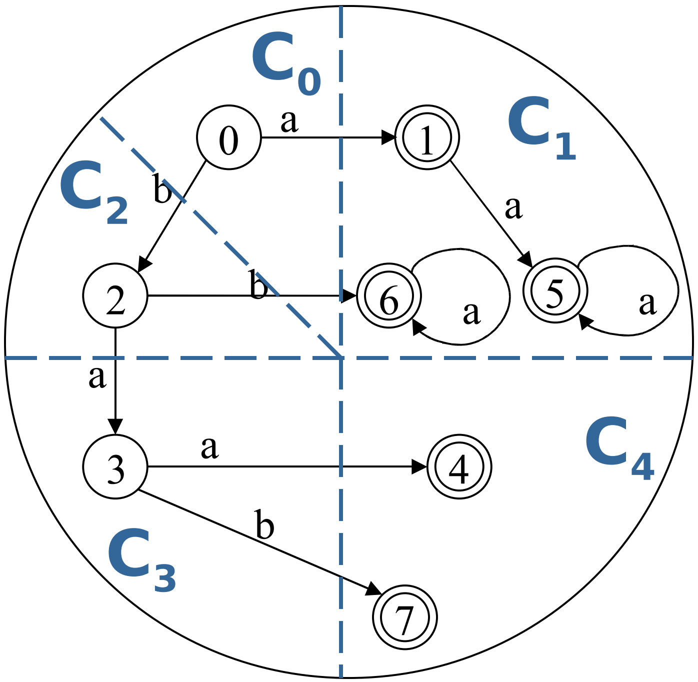
- A can be transformed to A‘:
  – States 2 and 3 in A “*do the same job*”: once A is in state 2 or 3, it *accepts the same suffix string*. Such states are called *equivalent.*
  – Thus, we can eliminate state 3 without changing the language of A, by *redirecting* all arcs leading to 3 to 2, instead.

Minimization of FSA

- A DFSA can be minimized
  if there are *pairs of states* $q,q' \in \Phi$ that are *equivalent*
- Two states q,q' are *equivalent* iff they accept the *same right language.*

- Right language of a state:
  - For $A=<\Phi,\Sigma, \delta, q_0,F>$ a DFSA, *the right language* $L^{\rightarrow}(q)$ *of a state* $q\in\Phi$ is the set of all strings accepted by A starting in state q:
    $L^{\rightarrow}(q) = \{w\in \Sigma^* \mid \delta^*(q,w) \in F\}$
  - Note: $L^{\rightarrow}(q_0) = L(A)$
- State equivalence:
  - For $A=<\Phi,\Sigma, \delta, q_0,F>$ a DFSA,
    if $q,q'\in\Phi$, *q and q' are equivalent* ($q \equiv q'$) iff $L^{\rightarrow}(q) = L^{\rightarrow}(q')$
  - $\equiv$ is an equivalence relation (i.e., reflexive, transitive and symmetric)
  - $\equiv$ *partitions* the set of states Φ into a number of *disjoint sets* $Q_1 .. Q_n$ *of equivalence classes* s.th. $\bigcup_{i=1..m} Q_i = \Phi$ and $q \equiv q'$ for all $q,q'\in Q_i$

Partitioning a state set into equivalence classes



All classes C_i consist of *equivalent states* $q_{j=i..n}$ that accept *identical right languages* $L^{\rightarrow}(q_j)$

Whenever two states q,q‘ belong to different classes, $L^{\rightarrow}(q) \neq L^{\rightarrow}(q')$

Equivalence classes on state set defined by ≡

Minimization: elimination of equivalent states

Minimization of a DFSA

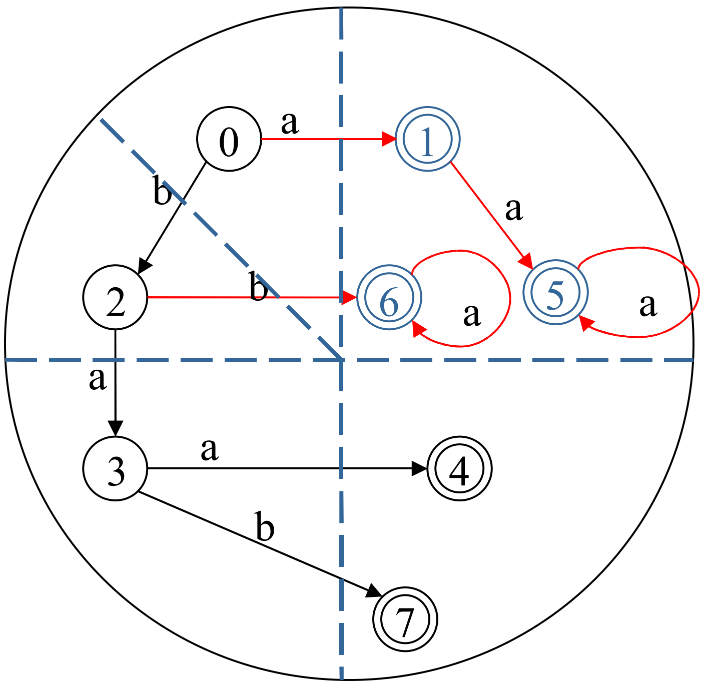
A DFSA $A=<\Phi,\Sigma,\delta,q_0,F>$ that contains *equivalent states q, q'*
can be transformed to a smaller, equivalent DFSA $A'=<\Phi',\Sigma,\delta',q_0,F'>$ where

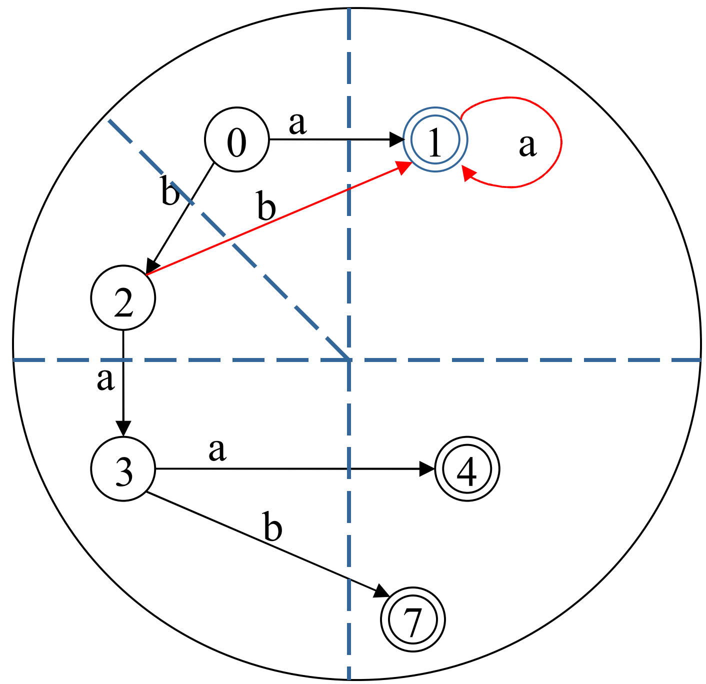
- $\Phi' = \Phi \backslash \{q'\}$, $F'=F\backslash\{q'\}$,
- δ' is like δ with all transitions to q' redirected to q

$\delta'(s,a) = q$ if $\delta(s,a) = q'$;
$\delta'(s,a) = \delta(s,a)$ otherwise

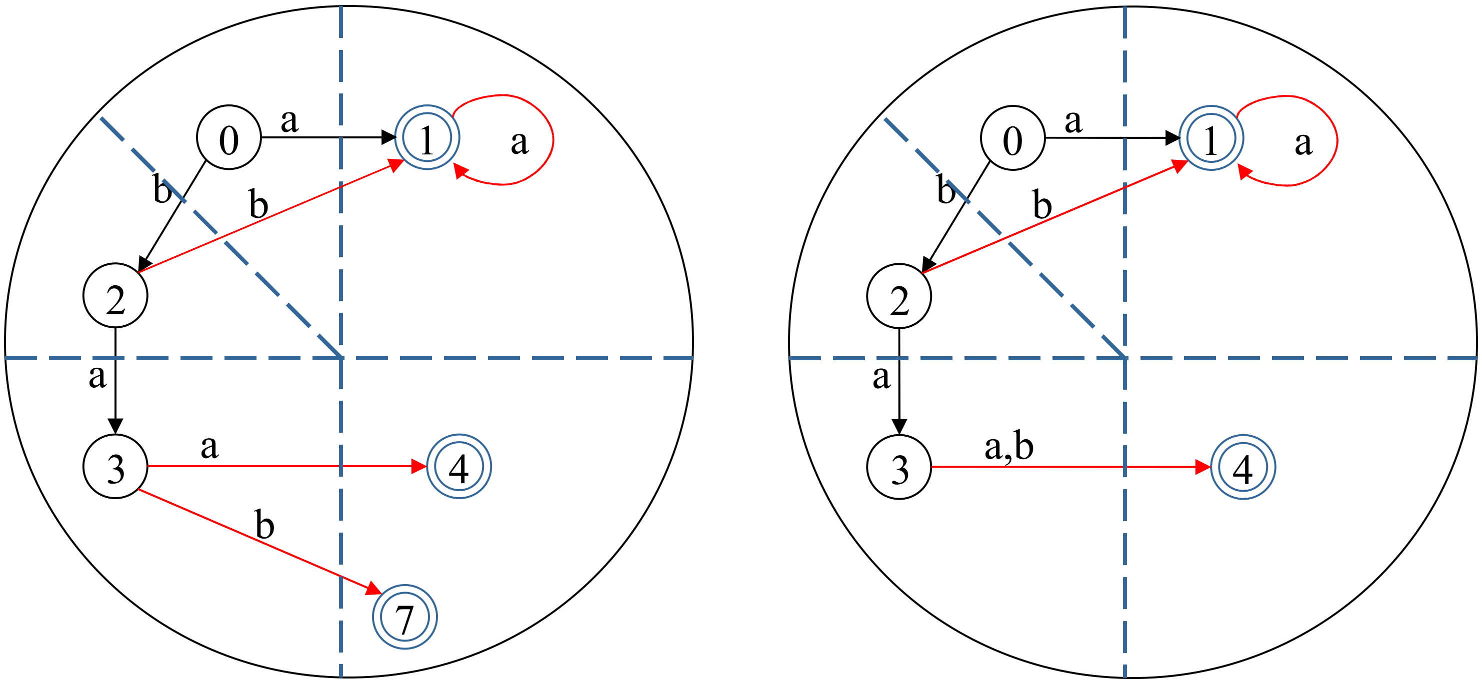
- Two-step algorithm
  - Determine all pairs of equivalent states q,q'
  - Apply DFSA reduction until no such pair q,q' is left in the automaton
- *Minimality*
  - The resulting FSA is the smallest DFSA (in size of Φ) that accepts L(A): we never merge different equivalence classes, so we obtain one state per class.
    - We cannot do any further reduction and still recognize L(A).
    - As long as we have >1 state per class, we can do further reduction steps.
- A DFSA $A=<\Phi,\Sigma,\delta,q_0,F>$ is *minimal* iff there is no pair of distinct but equivalent states $\in \Phi$, i.e. $\forall\, q, q' \in \Phi:\; q \equiv q' \Leftrightarrow q = q'$

Example





Example



Algorithm

```
MINIMIZE(Φ, Σ, δ, q0, F)
main
  EqClass[] ← PARTITION(A)
  q0 ← EqClass[q0]
  for <q,a,q'>∈ δ
      δ(q,a) ← min(EqClass[q'])
  for q∈ Φ
      if q ≠ min(EqClass[q])
        Φ ← Φ\{q}
        if q∈ F
           F ← F\{q}
```

MINIMIZE

- PARTITION(A):
  - determines all eqclasses of states in A
  - returns array EqClass[q] of eq. classes of q
- redirect all transitions <q,a,q'>∈ δ to point to min(EqClass[q'])
- remove all redundant states from Φ and F

Computing partitions: Naïve partitioning

```
NAIVE_PARTITION(Φ, Σ, δ, q0, F)
for each q∈ Φ
  EqClass[q] ← {q}
for each q∈ Φ
      for each q'∈ Φ
        if  EqClass[q] ≠ EqClass[q'] ∧ CHECKEQUIVALENCE(Aq,Aq') = True
            EqClass[q] ← EqClass[q] ∪ EqClass[q']
            EqClass[q'] ← EqClass[q]
```

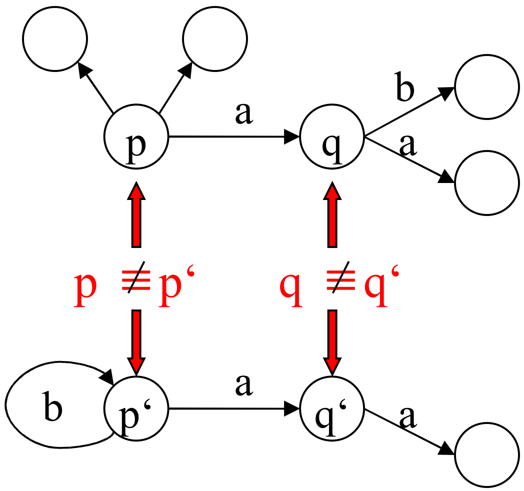
NAIVE_PARTITION

- array EqClass of pointers to disjoint sets for equivalence classes
- first loop: initializing EqClass by $\{q\}$, for each $q \in \Phi$
- second nested loop: if we find new equivalent states $q \equiv q'$, we merge the respective equivalence classes EqClasses and reset EqClass[q] to point to the new merged class

Runtime complexity: loops: $0(|\Phi|^2)$ CheckEquivalence: $0(|\Phi|^2 \cdot |\Sigma|) \Rightarrow 0(|\Phi|^4 \cdot |\Sigma|)$!

Computing partitions: Dynamic Programming

- Source of inefficiency: naive algorithm traverses the whole automaton to determine, for pairs q,q‘, whether they are equivalent
- Results of previous equivalence checks can be reused



If $q \not\equiv q'$, $L^{\rightarrow}(q) \neq L^{\rightarrow}(q')$,
therefore,
for all <p,p‘> s.th. $\delta^{-1}(p,a)=q$ and $\delta^{-1}(p',a)=q'$
for some $a \in \Sigma$, $p \not\equiv p'$.

- Thus, non-equivalence results can be propagated
  - Propagation from final/non-final pairs: $L^{\rightarrow}(q) \neq L^{\rightarrow}(q')$ if $q \in F \wedge q' \notin F$
  - Propagation from pairs <q,q’> where $\delta(q,a)$ is defined but $\delta(q',a)$ is not.

Propagation of non-equivalent states

```
LocalEquivalenceCheck(q,q‘)
if (q∈F and q‘∉F) or (q∉F and q‘∈F)
   return (False)
if ∃a∈Σ s.th. only one of δ(q,a), δ(q’,a)
    is defined
   return (False)
return (True)
```

```
PROPAGATE(q,q‘)
for a∈Σ
   for p∈δ^-1(q,a),
      for p’∈δ^-1(q’,a)
         if Equiv[min(p,p’),max(p,p’)]=1
            Equiv[min(p,p’),max(p,p’)] ← 0
            PROPAGATE(p,p‘)
```

Non-equivalence check for states <q,q‘>

– Only one of q, q’ is final

– For some $a \in \Sigma$, $\delta(q,a)$ is defined, $\delta(q',a)$ is not

Propagation (I): Table filling algorithm (Aho, Sethi, Ullman)

- represent equivalence relation as a table *Equiv*, cells filled with boolean values
- initialize all cells with 1;
  reset to 0 for non-equivalent states
- main loop: call of PROPAGATE for non-equivalent states from LocalEquivalenceCheck

Propagation of non-equivalent states

```
LocalEquivalenceCheck(q,q‘)
if (q∈F and q‘∉F) or (q∉F and q‘∈F)
    return (False)
if ∃a∈Σ s.th. only one of δ(q,a), δ(q’,a)
     is defined
    return (False)
return (True)
```

```
PROPAGATE(q,q‘)
for a∈Σ
    for p∈δ^-1(q,a),
      for p’∈δ^-1(q’,a)
        if Equiv[min(p,p’),max(p,p’)]=1
            Equiv[min(p,p’),max(p,p’)] ← 0
            PROPAGATE(p,p‘)
```

Runtime Complexity: $0(|\Phi|^2 \cdot |\Sigma|)$

- PROPAGATE is never called twice on a given pair of states (checks Equiv[q,q’]=1)

Space requirements: $0(|\Phi|^2)$ cells

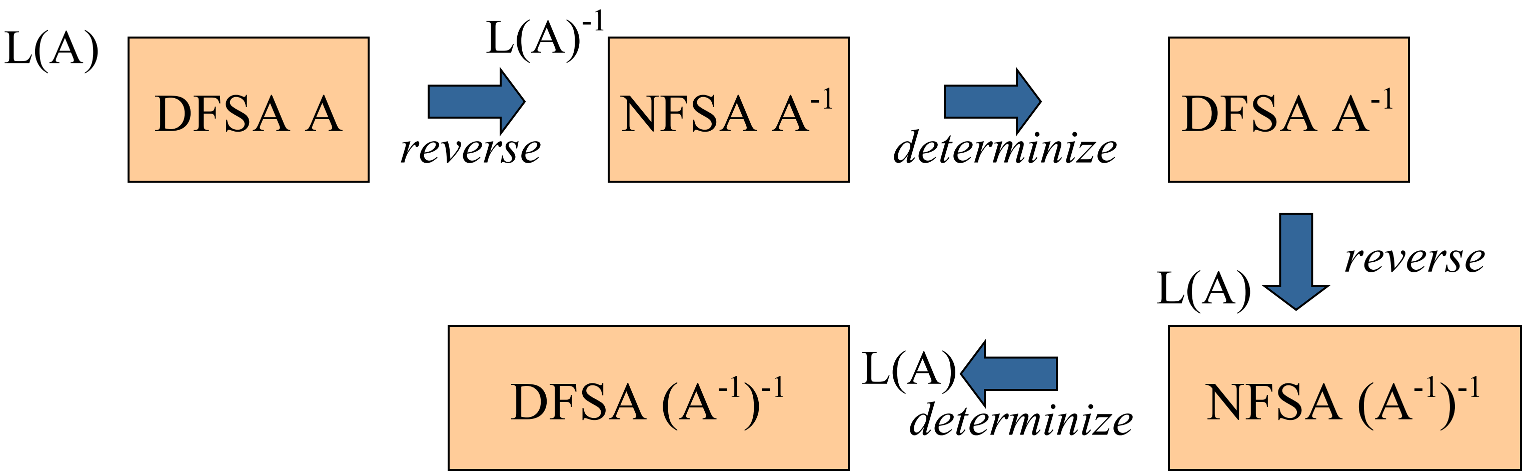
```
TableFillingPARTITION(Φ, Σ, δ, q_0, F)
for q,q‘∈Φ, q<q’
    Equiv[q,q’] ← 1
for q∈Φ
    for q‘∈Φ, q<q’
        if  Equiv[q,q’]=1 and
                LocalEquivalenceCheck(q,q’)=False
            Equiv[q,q’] ← 0
            PROPAGATE(q,q‘)
```

More optimizations

- Hopcroft and Ullman: space requirement $0(|\Phi|)$, by associating states with their equivalence classes
- Hopcroft: Runtime complexity of $0(|\Phi| \cdot \log|\Phi| \cdot |\Sigma|)$, by distinction of active/non-active blocks

Brzozowski‘s Algorithm

Minimization by reversal and determinization

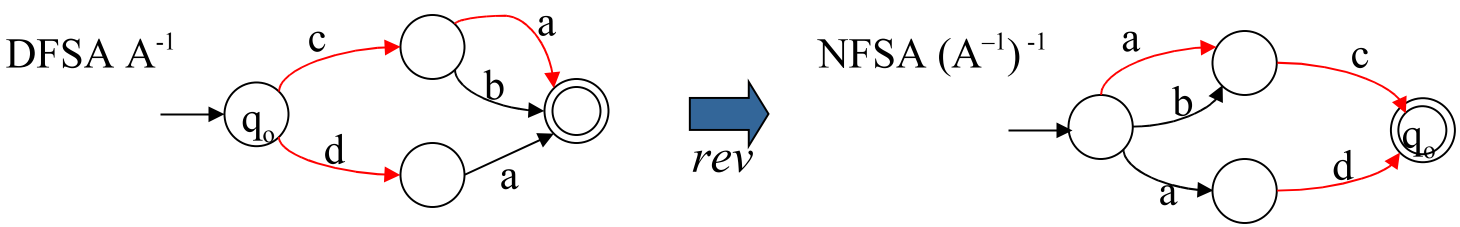


Reversal

- Final states of A^{-} : set of initial states of A
- Initial state of A^{-} : F of A
- $\delta^{-}(q,a) = \{p \in \Phi \mid \delta(p,a)=q\}$
- $L(A^{-1}) = L(A)^{-1}$

Why does it yield a minimal DFSA A‘?

DFSA A → *rev* → NFSA A^{-1} → *det* → DFSA A^{-1} → *rev* → NFSA $(A^{-1})^{-1}$ → *det* → DFSA $(A^{-1})^{-1}$



Consider the *right languages of states q, q‘ in NFSA $(A^{-1})^{-1}$:*

- *If* for all distinct states q, q‘ $L^{\rightarrow}(q) \neq L^{\rightarrow}(q')$, i.e. $L^{\rightarrow}(q) \cap L^{\rightarrow}(q') = \emptyset$, it holds that each pair of states q,q’ recognize *different right languages*, and thus, that the NFSA $(A^{-1})^{-1}$ satisfies the *minimality condition* for a DFSA.
- If there were states q,q’ in NFSA $(A^{-1})^{-1}$ s.th. $L^{\rightarrow}(q) \cap L^{\rightarrow}(q') \neq \emptyset$, there would be some string w that leads to two distinct states in DFSA A^{-1}. This contradicts the *determinicity* criterion of a DFSA.
- Determinization of NFSA $(A^{-1})^{-1}$ does not destroy the property of minimality

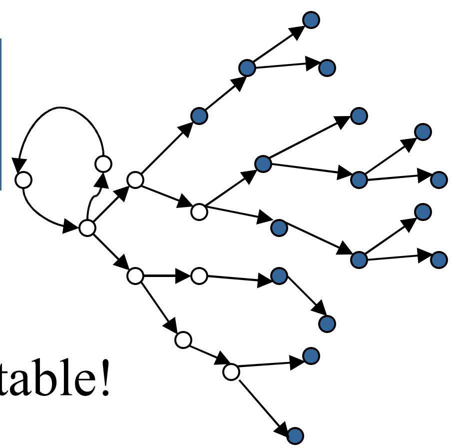
Applications of FSA: String Matching

- Exact, full string matching
  - Lexicon lookup: search for given word/string in a lexicon
  - Compile lexicon entries to FSA by union
  - Test input words for acceptance in lexicon-FSA

Word list

compile to FSA

transition table!



recognition/application/lookup of input word *w* in/to FSA $A_{lexicon:}$

$(q_0,w) \vdash^*_{Alexicon} (q_f, \varepsilon)$ is true, with q_0 initial state and $q_f \subseteq F$

traversal and recognition (acceptance)

Applications of FSA: String Matching

- Substring matching
  - Identify stop words in stream of text
  - Stem recognition: *small, smaller, smallest*
- Make use of full power of finite-state operations!
  - Regular expression with any-symbols for text search
    - ?∗ small(ε | er | est) ?∗
    - ?∗ (a | the | …) ?∗
  - Compilation to NFSA, convert to DFSA
  - Application by *composition* of FST with full text
    - $FSA_{\text{text stream}} \circ FST_{\text{small}}$: if defined, search term is substring of text

Application of FSA: Replacement

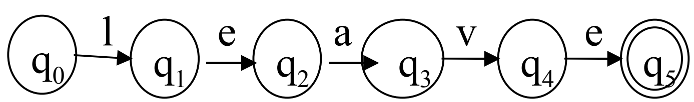
- (Sub)string replacement
  - Delete stop words in text
  - Stemming: reduce/replace inflected forms to stems: *smallest* → *small*
  - Morphology: map inflected forms to lemmas (and PoS-tags):
    good, better, best → good+Adj
  - Tokenization: insert token boundaries
  - …

⇒ Finite-state transducers (FST)

From Automata to Transducers

Automata

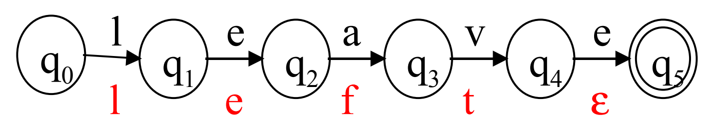
- recognition of an input string *w*



- define a *language*
- accept *strings,* with transitions defined for *symbols* ∈ Σ

Transducers

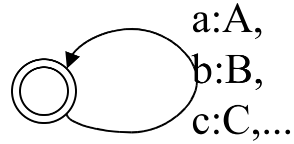
- recognition of an input string *w*
- *generation* of an output string *w'*



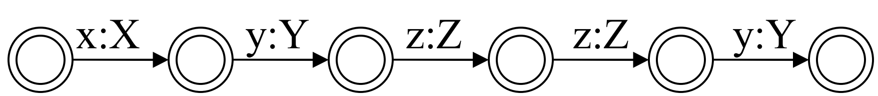
- define a *relation* between languages
- equivalent to FSA that accept *pairs of strings*, with transitions defined for pairs of symbols <*x,y*>
- operations: *replacement*
  - deletion <a, ε>, a ∈ Σ - {ε}
  - insertion <ε, a>, a ∈ Σ - {ε}
  - substitution <a, b>, a,b ∈ Σ, a ≠ b

Transducers and composition

- An FSTs encodes a relation between languages
- A relation may contain an infinite number of ordered pairs, e.g. translating lower case letters to upper case

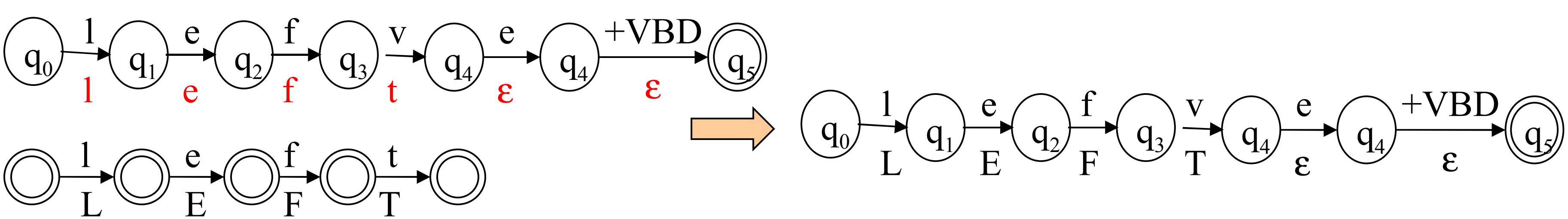


a lower/upper case transducer



a path through the lower/upper case transducer, for string xyzzy

- The application of a transducer to a string may also be viewed as *composition* of the FST with the (identity relation on the string)



Literature

- H.R. Lewis and C.H. Papadimitriou: Elements of the Theory of Computation. Prentice-Hall, New Jersey (Chapter 2).
- J. Hopcroft and J. Ullman: Introduction to Automata Theory, Languages, and Computation, Addison-Wesley, Massachusetts, (Chapter 2,3).
- B.H. Partee, A. ter Meulen and R.E. Wall: Mathematical Methods in Linguistics, Kluwer Academic Publishers, Dordrecht (Chapter 15.5,15.6, 17)
- D. Jurafsky and J.H. Martin: Speech and Language Processing. An introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition, Prentice-Hall, New Jersey (Chapter 2).
- C. Martin-Vide: Formal Grammars and Languages. In: R. Mitkov (ed): Oxford Handbook of Computational Linguistics, (Chapter 8).
- L. Karttunen: Finite-state Technology. In: R. Mitkov (ed): Oxford Handbook of Computational Linguistics, (Chapter 18).

Off-the-shelf finite-state tools

- Xerox finite-state tools
  - http://www.xrce.xerox.com/competencies/content-analysis/fst/ > Xerox Finite State Compiler (Demo)
  - XFST Tools (provided with Beesley and Karttunen: Finite-State Morphology, CSLI Publications)
- Geertjan van Noord's finite-state tools
  - http://odur.let.rug.nl/~vannoord/Fsa/
- FSA Utilities at John Hopkins
  - http://cs.jhu.edu/~jason/406/software.html
- AT&T FSM Library
  - http://www.research.att.com/sw/tools/fsm/

XEROX FINITE-STATE COMPILER

This page allows you to create a finite-state network from a regular expression and to apply the resulting network to strings. You can also try out some of our Examples.

› COMPILATION :

Type a regular expression in this area and submit it to the compiler by pressing the SUBMIT button. The compilation result will appear in a **new** browser window. Clear with RESET.

SUBMIT RESET.

```
(a|b)*c b+ (a|c) d
```

☑ Display the structure of the network (if it has not more than 50 states).

Regular expression

```
(a|b)*c b+ (a|c) d
 ;
```

Network

```
484 bytes. 5 states, 9 arcs, Circular.
Sigma: a b c d

s0:   a -> s0, b -> s0, c -> s1.
s1:   b -> s2.
s2:   a -> s3, b -> s2, c -> s3, d -> fs4.
s3:   d -> fs4.
fs4:  (no arcs)
```

Exercises

- Write a program for acceptance of a string by a DFSA.
  Then extend it to a finite-state transducer that can translate a surface form to lemma + POS, or between upper and lower case.
- Determinize the following NFSA by subset construction.
  A_1=<{p,q,r,s},{a,b},δ_1,p,{s}> where δ_1 is as follows:

δ_1	a	b
p	p,q	p
q	r	r
r	s	-
s	s	s

- Construct an NFSA with ε-transitions from the regular expression (a|b)ca*, according to the construction principles for union, concatenation and kleene star. Then transform the NFSA to a DFSA by subset construction.
- Find a minimal DFSA for the FSA A=<{A,..,E},{0,1}, δ_3,A,{C,E}> (using the table filling algorithm by propagation).

δ_3	0	1
A	B	D
B	B	C
C	D	E
D	D	E
E	C	-