Java II
Finite Automata I

Bernd Kiefer
Bernd.Kiefer@dfki.de

Deutsches Forschungszentrum für künstliche Intelligenz

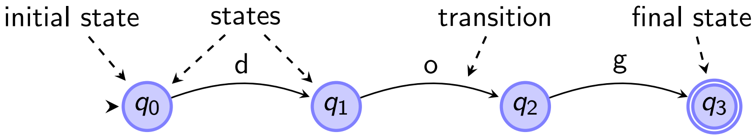
January 18, 2011

Processing Regular Expressions

- We already learned about Java's regular expression functionality
- Now we get to know the machinery behind
  - `Pattern` and
  - `Matcher` classes
- Compiling a regular expression into a `Pattern` object produces a *Finite Automaton*
- This automaton is then used to perform the matching tasks
- We will see how to construct a finite automaton that *recognizes* an input string, i.e., tries to find a full match

Definition: Finite Automaton

- A finite automaton (FA) is a tuple $A = < Q, \Sigma, \delta, q_0, F >$
  - Q a finite non-empty set of states
  - Σ a finite alphabet of input letters
  - δ a (total) transition function $Q \times \Sigma \longrightarrow Q$
  - $q_0 \in Q$ the initial state
  - $F \subseteq Q$ the set of final (accepting) states
- Transition graphs (diagrams):



Finite Automata: Matching

- A finite automaton *accepts* a given input string s if there is a sequence of states $p_1, p_2, \ldots, p_{|s|} \in Q$ such that
  1. $p_1 = q_0$, the start state
  2. $\delta(p_i, s_i) = p_{i+1}$, where s_i is the i-th character in s
  3. $p_{|s|} \in F$, i.e., a final state
- A string is successfully *matched* if we have found the appropriate sequence of states
- Imagine the string on an input tape with a pointer that is advanced when using a δ transition
- The set of strings accepted by an automaton is the accepted *language*, analogous to regular expressions

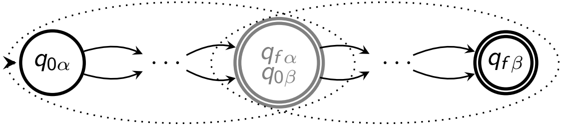
(Non)deterministic Automata

- in the definition of automata, δ was a total function $\Rightarrow$ given an input string, the path through the automaton is uniquely determined
- those automata are therefore called *deterministic*
- for *nondeterministic FA*, δ is a *transition relation*
- $\delta : Q \times \Sigma \cup \{\epsilon\} \longrightarrow \mathcal{P}(Q)$, where $\mathcal{P}(Q)$ is the powerset of Q
- allows transitions from one state into several states with the same input symbol
- need not be total
- can have transitions labeled ϵ (not in Σ), which represents the empty string

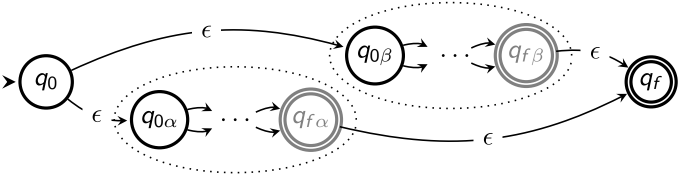
RegExps $\longrightarrow$ Automata

Construct nondeterminstic automata from regular expressions

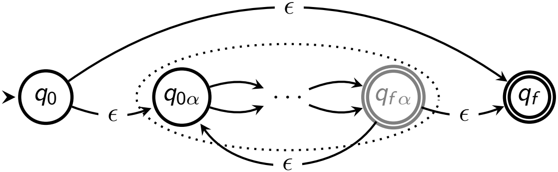
$(\alpha\beta)$



$(\alpha|\beta)$



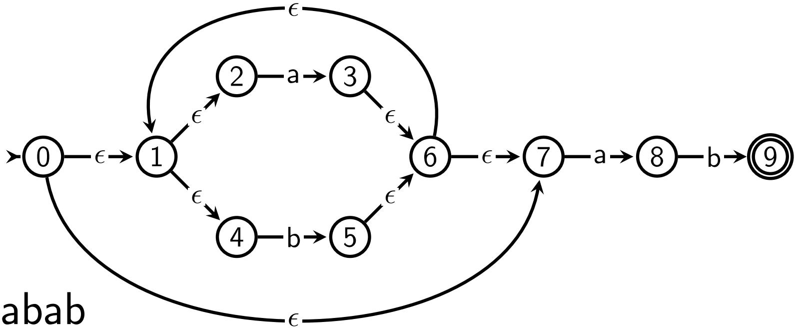
$(\alpha)*$



NFA vs. DFA

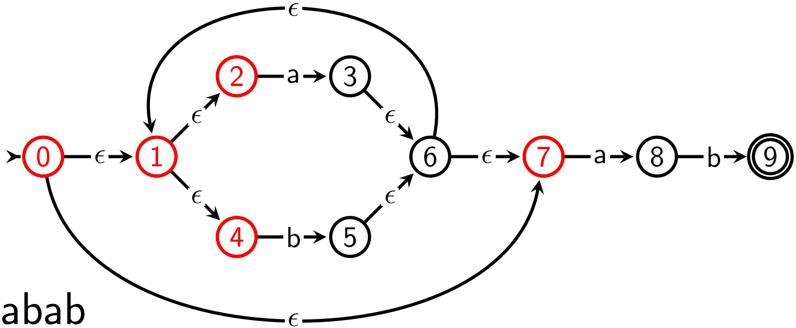
- Traversing a DFA is easy given the input string: the path is uniquely determined
- In contrast, traversing an NFA requires keeping track of a set of (current) states, starting with the set $\{q_o\}$
- Processing the next input symbol means taking all possible outgoing transitions from this set and collecting the new set
- From every NFA, an equivalent DFA (one which does accept the same language), can be computed
- Basic Idea: track the subsets that can be reached for every possible input

Traversing an NFA



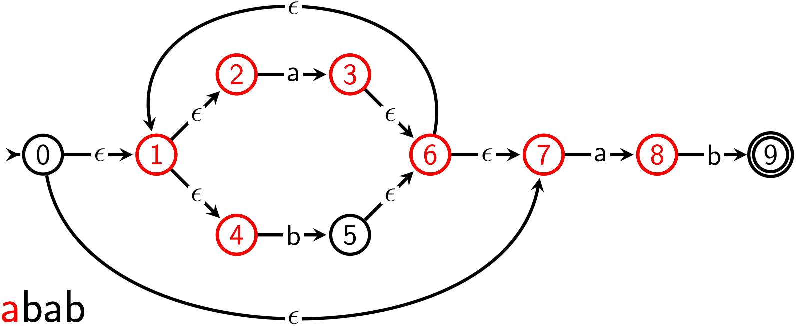
abab

Traversing an NFA



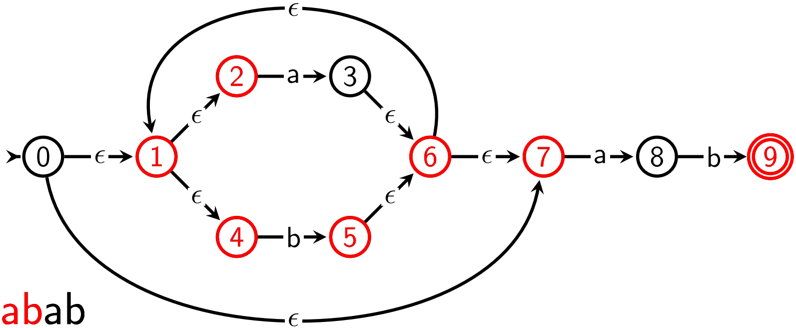
abab

Traversing an NFA



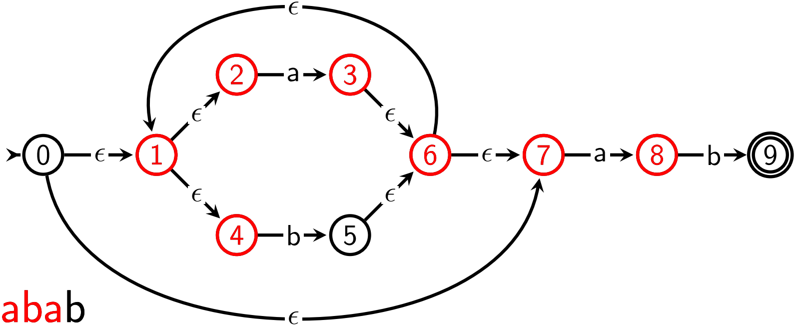
abab

Traversing an NFA



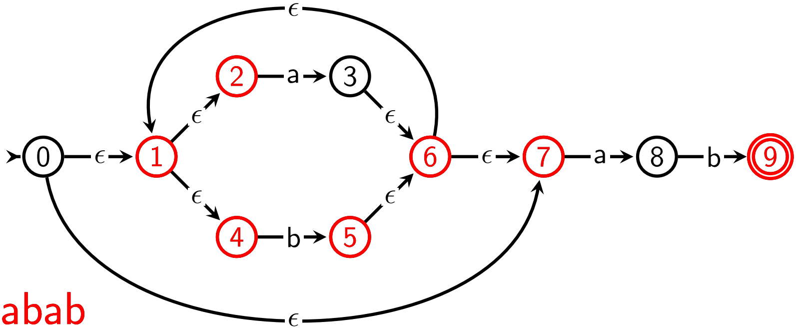
abab

Traversing an NFA



ababab

Traversing an NFA



abab

NFA $\longrightarrow$ DFA: Subset Construction

- Simulate "in parallel" all possible moves the automaton can make
- The states of the resulting DFA will represent sets of states of the NFA, i.e., elements of $\mathcal{P}(Q)$
- We use two operations on states/state-sets of the NFA

ϵ-*closure*(T)	Set of states reachable from any state s in T on ϵ-transitions
move(T, a)	Set of states to which there is a transition from one state in T on input symbol a

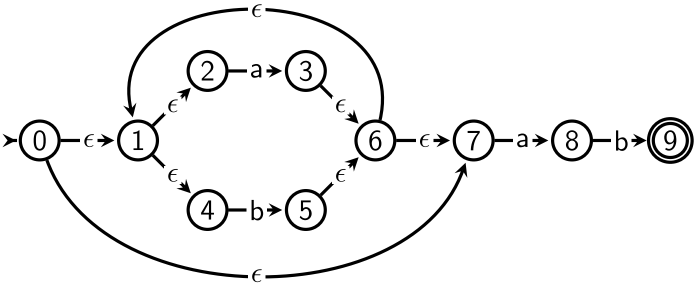
- The final states of the DFA are those where the corresponding NFA subset contains a final state

Algorithm: Subset Construction

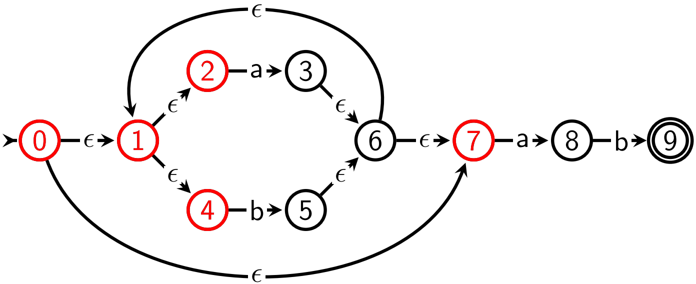
proc $SubsetConstruction(s_0) \equiv$
  $DFAStates = \epsilon\text{-}closure(\{s_0\})$
  while there is an unmarked state T in $DFAStates$ **do**
    mark T
    for each input symbol a **do**
      $U := \epsilon\text{-}closure(move(T, a))$
      $DFADelta[T, a] := U$
      if $U \notin DFAStates$ **then** add U as unmarked to $DFAStates$

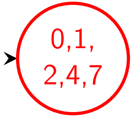
proc $\epsilon\text{-}closure(T) \equiv$
  $\epsilon\text{-}closure := T$; $to\_check := T$
  while to_check not empty **do**
    get some state t from to_check
    for each state u with edge labeled ϵ from t to u
      if $u \notin \epsilon\text{-}closure$ **then** add u to $\epsilon\text{-}closure$ and to_check

Example: Subset construction

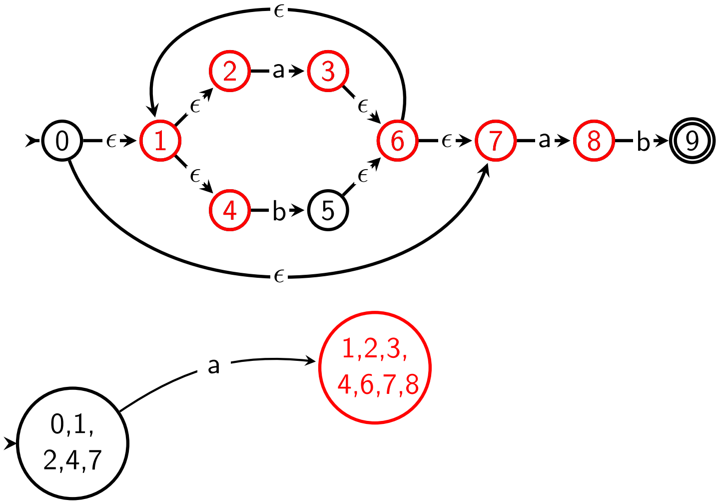


Example: Subset construction

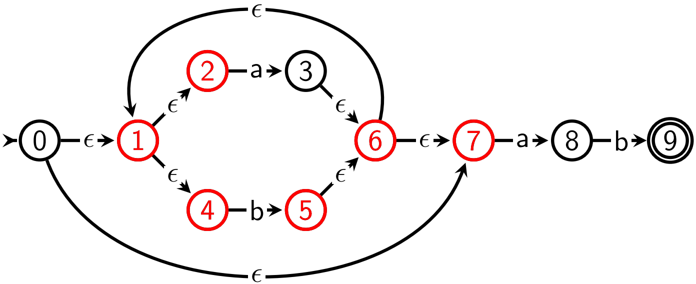


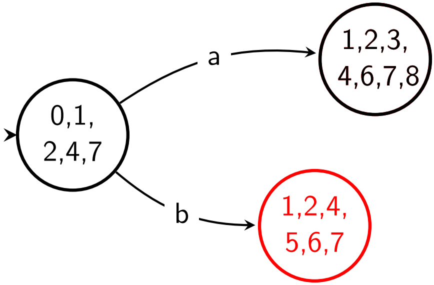


Example: Subset construction

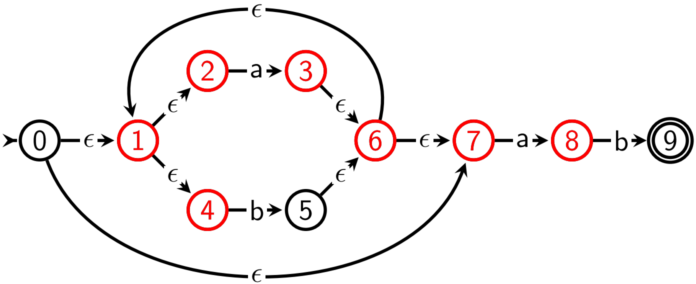


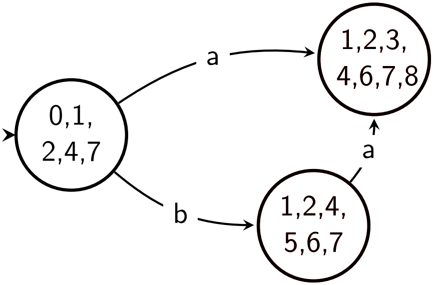
Example: Subset construction



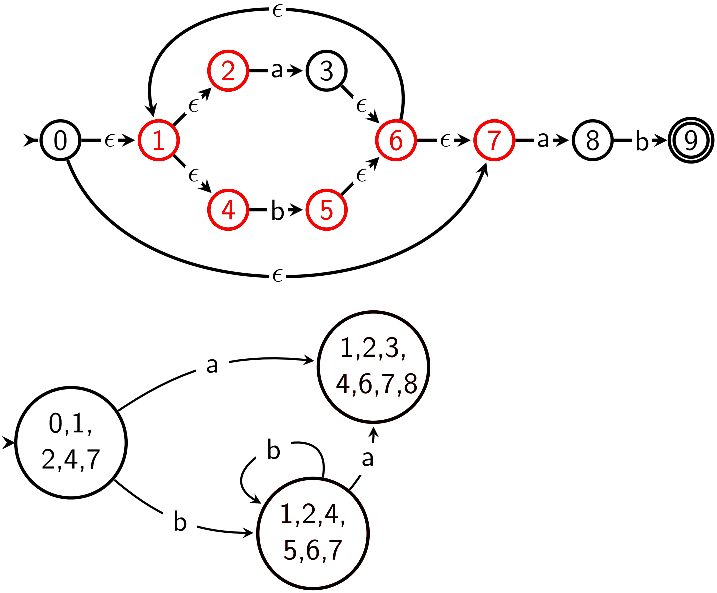


Example: Subset construction

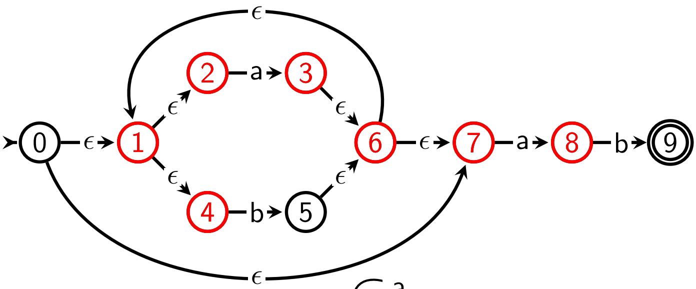


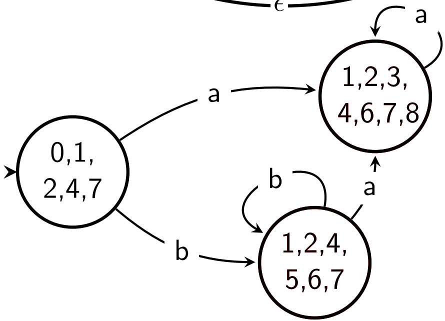


Example: Subset construction

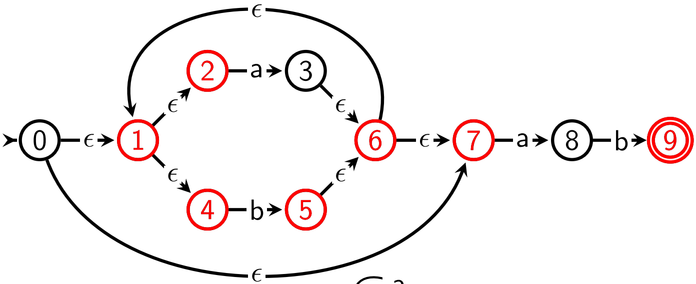


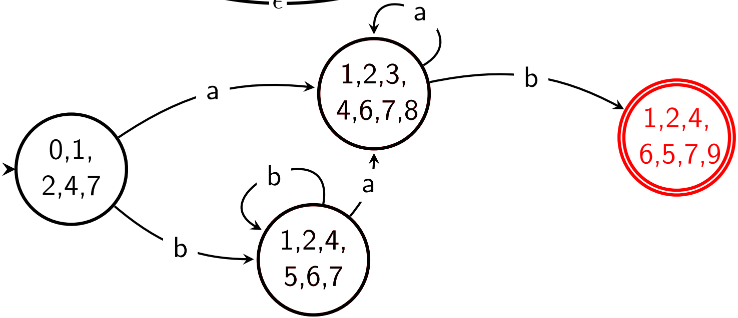
Example: Subset construction



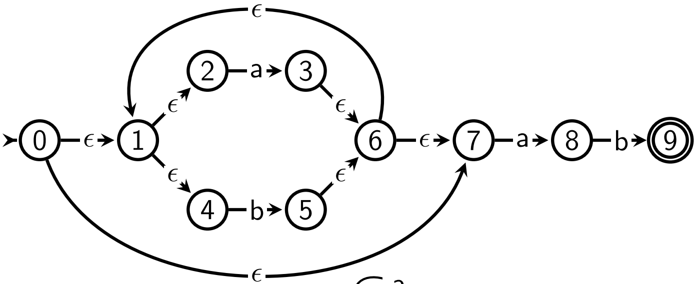


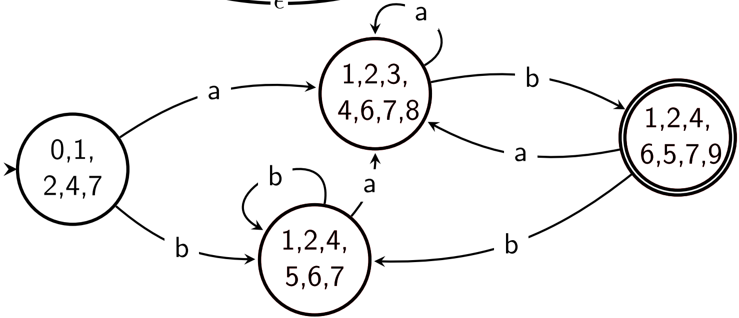
Example: Subset construction





Example: Subset construction





Time/Space Considerations

- DFA traversal is linear to the length of input string x
- NFA needs $\mathcal{O}(n)$ space (states+transitions), where n is the length of the regular expression
- NFA traversal may need time $n \times |x|$, so why use NFAs?

Time/Space Considerations

- DFA traversal is linear to the length of input string x
- NFA needs $\mathcal{O}(n)$ space (states+transitions), where n is the length of the regular expression
- NFA traversal may need time $n \times |x|$, so why use NFAs?
- There are DFA that have at least 2^n states!

Time/Space Considerations

- DFA traversal is linear to the length of input string x
- NFA needs $\mathcal{O}(n)$ space (states+transitions), where n is the length of the regular expression
- NFA traversal may need time $n \times |x|$, so why use NFAs?
- There are DFA that have at least 2^n states!
- Solution 1: "Lazy" construction of the DFA: construct DFA states on the fly up to a certain amount and cache them
- Solution 2: Try to minimize the DFA:
  There is a unique (modulo state names) minimal automaton for a regular language!

Automata Minimization

- Take any state q of the deterministic automaton to minimize and assume it to be the (single) start state
- We call the language that this automaton accepts the *right language* of q
- The language of each state consists of suffixes of the overall accepted language
- If two states accept the same language, they are equivalent and can be merged
- To minimize the automaton, merge all equivalent nodes
- This is implemented by first partitioning the original set of states into *equivalence classes*, i.e., sets of equivalent states
- Finally, each equivalence class is replaced by a single state, merging transitions accordingly