From FAs to Regular Expressions - I

L.EIC, 2nd Year

João M. P. Cardoso
Email: jmpc@acm.org

U.PORTO
FEUP FACULDADE DE ENGENHARIA UNIVERSIDADE DO PORTO

DEI DEPARTAMENTO DE ENGENHARIA INFORMÁTICA

Conversion from FAs to Regular Expressions

- Given an FA (Finite Automaton), how to generate an equivalent regular expression (RE)?
- We will focus on two methods:
  - State Elimination Method
  - Construction of Paths (Transitive Closure Method)
- Both algorithms work with Finite Automata (FA) as input, i.e., DFAs, NFAs, and ε-NFAs

State Elimination Method

State Elimination Method

- Maintains an extended finite automaton (FA)
  - start with the original FA where transitions represent the symbol possibilities for each transition to occur and are represented as regular expressions, rather than alphabet symbols
  - FA is transformed by eliminating states and reflecting the elimination in transitions and on their regular expressions
  - also known as generalized nondeterministic finite automaton (GNFA)
- Results in shorter regular expressions than the Construction of Paths technique (to be presented)

Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0

Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **Eliminate each** non initial and non final **state** of the FA **and substitute it by the transitions to/from that state**
3. Resultant FA:

   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.
   RE(q0→qf).(RE(qf→qf))*

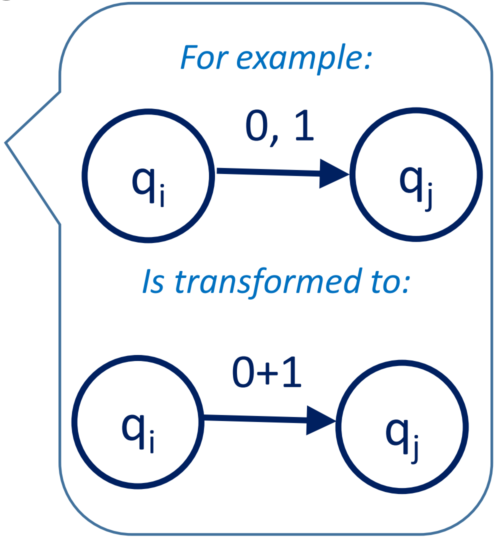
   3.2. One state: RE = ((RE(q0→q0))*

When the DFA includes $n \geq 2$ final states:
(a) Transform it in a ε-NFA with one final state (see generalized nondeterministic finite automaton (GNFA));
(b) Consider n FAs (each one with a final state) and determine an RE for each FA. The resultant RE is the union of the REs of each of the FAs.

Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **Eliminate each** non initial and non final **state** of the FA **and substitute it by the transitions to/from that state**
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.
   RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*



Algorithm based on State Elimination

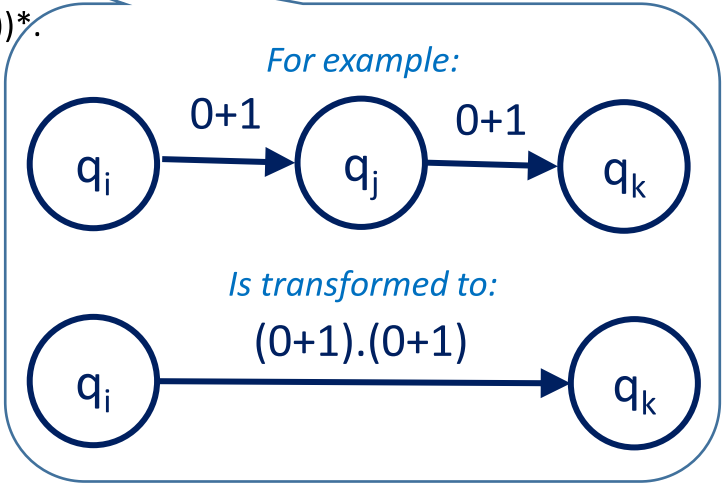
Input: FA with a single final state qf and an initial state q0

Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **Eliminate each** non initial and non final **state** of the FA **and substitute it by the transitions to/from that state**
3. Resultant FA:

   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.
   RE(q0→qf).(RE(qf→qf))*

   3.2. One state: RE = ((RE(q0→q0))*



Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0

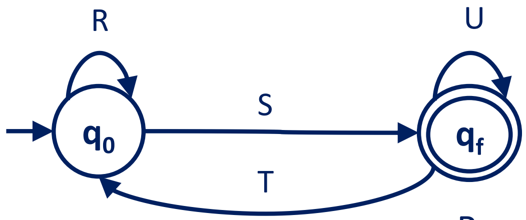
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **Eliminate each** non initial and non final **state** of the FA **and substitute it by the transitions to/from that state**
3. Resultant FA:

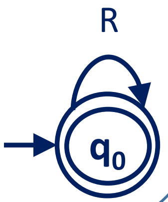
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.
   RE(q0→qf).(RE(qf→qf))*

   3.2. One state: RE = ((RE(q0→q0))*

If qf≠q0, we obtain an FA with 2 states:
(R+SU*T)*SU*



If not, we obtain an FA with a single state:
R*



Algorithm based on State Elimination

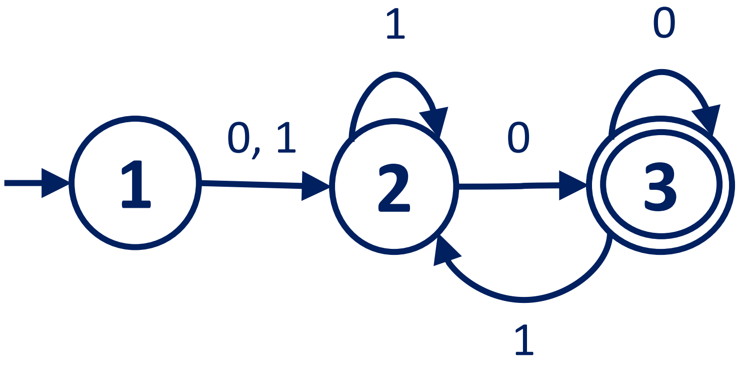
Input: FA with a single final state qf and an initial state q0

Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **Eliminate each** non initial and non final **state** of the FA **and substitute it by the transitions to/from that state**
3. Resultant FA:

   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.
        RE(q0→qf).(RE(qf→qf))*

   3.2. One state: RE = ((RE(q0→q0))*



Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist (i.e., if it is ∅);
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj)+
         RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

A possible pseudo-code representing an implementation of the algorithm

Note: For FAs with more than one final state, multiple FAs are considered, each one with one of the final states (and the other final states marked as non-final). This conversion is applied to each one of the FAs and the resulting RE is the union of the individual REs. With a generalized FA (an FA where include a start and a final state connected to the original FA via ε), step 3 is not needed and we just need to apply the conversion to a single FA and RE = RE(q0→qf).

Let’s show show the algorithm works step-by-step by using an example FA:

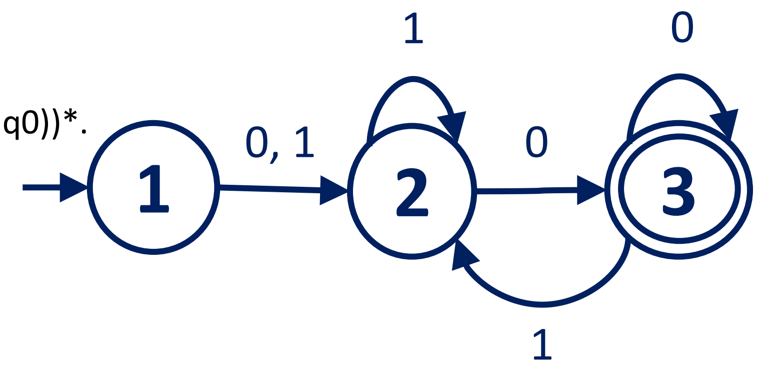
Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0

Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj)+
         RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.
      RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

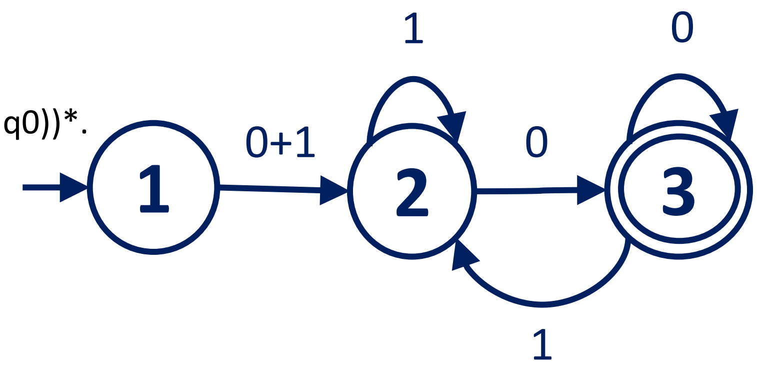
Input FA:



Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*



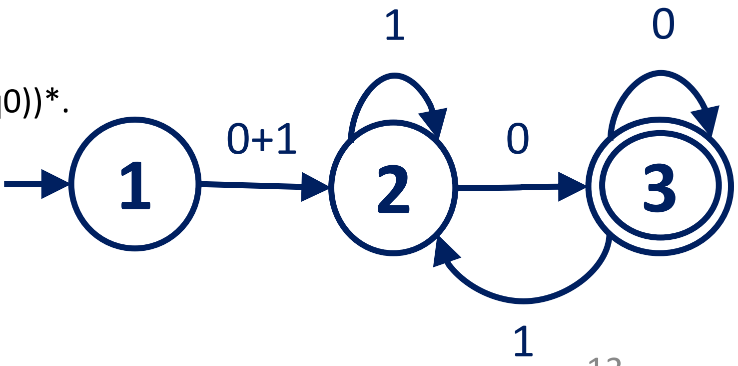
Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0

Output: a regular expression RE representing the language of the FA

q: 

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | $q \neq q0 \wedge q \neq qf$
   - 2.1. Get S (sources), the set of all states from which the state q can be reached
   - 2.2. Get D (destinations), the set of all states that can be reached from q
   - 2.3. Remove q from FA but keep info about transitions to/from q;
   - **2.4. foreach** pair (si, dj) | $si \in S \wedge dj \in D$ **do**
     - 2.4.1. Add a transition si → dj if it does not exist;
     - 2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   - 3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf).(RE(qf→qf))*
   - 3.2. One state: RE = ((RE(q0→q0))*



Algorithm based on State Elimination

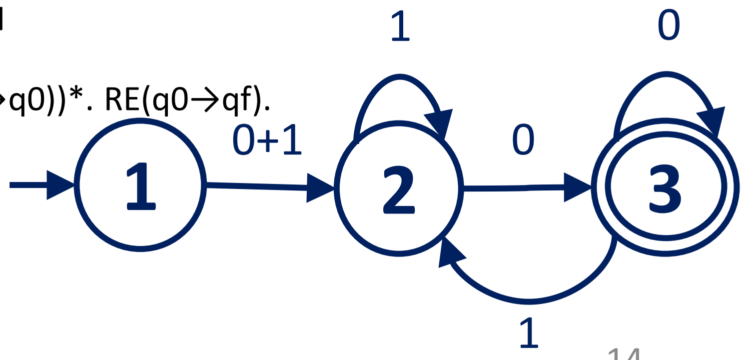
Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf). (RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

q: 2

S = {**1**, **3**}

D = {}



Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0

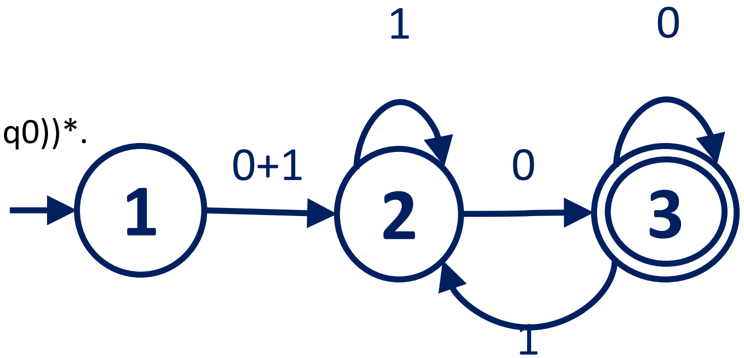
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   - 2.1. Get S (sources), the set of all states from which the state q can be reached
   - 2.2. Get D (destinations), the set of all states that can be reached from q
   - 2.3. Remove q from FA but keep info about transitions to/from q;
   - **2.4. foreach** pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
     - 2.4.1. Add a transition si → dj if it does not exist;
     - 2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   - 3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf).(RE(qf→qf))*
   - 3.2. One state: RE = ((RE(q0→q0))*

q: **2**

S = {**1**, **3**}

D = {**3**}



Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

Regular expression representing the transition between 2 states (i, j).

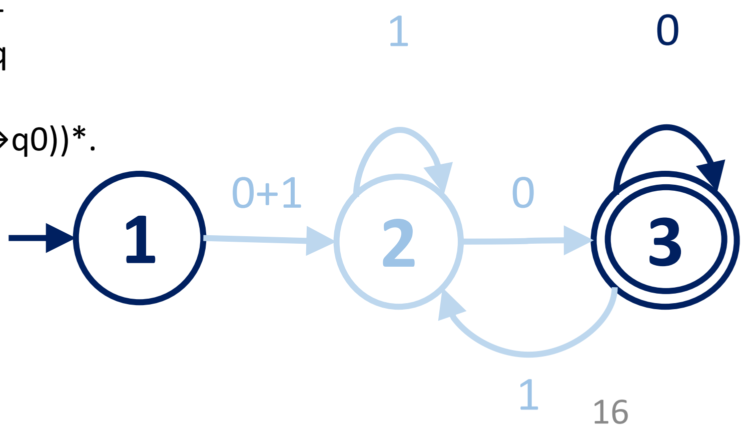
q: 2

S = {**1, 3**}

D = {**3**}

RE(1,2) = 0+1
RE(2,2) = 1
RE(2,3) = 0
RE(3,2) = 1

RE(i,j) = RE(i→j)



Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

q: **2**

S = {**1, 3**}

D = {**3**}

RE(1,2) = 0+1
RE(2,2) = 1
RE(2,3) = 0
RE(3,2) = 1

pairs(si, dj) = {(1,3), (3,3)}

pair(1, 3) = 1→3

Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

q: 2

S = {**1**, **3**}

D = {**3**}

RE(1,2) = 0+1
RE(2,2) = 1
RE(2,3) = 0
RE(3,2) = 1

pairs(si, dj) = {(1,3), (3,3)}

pair(1, 3) = 1→3

Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) +
         RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.
      RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

q: 2

S = {**1**, **3**}

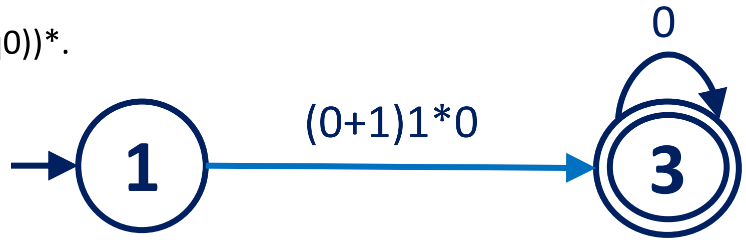
D = {**3**}

RE(1,2) = 0+1
RE(2,2) = 1
RE(2,3) = 0
RE(3,2) = 1

pairs(si, dj) = {(1,3), (3,3)}

pair(1, 3) = 1→3

RE(1,2).(RE(2,2))*.RE(2,3);



Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) +
         RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.
      RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

q: **2**

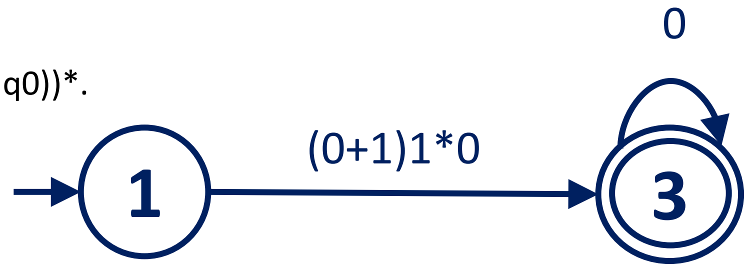
S = {**1, 3**}

D = {**3**}

RE(1,2) = 0+1
RE(2,2) = 1
RE(2,3) = 0
RE(3,2) = 1

pairs(si, dj) = {(1,3), (3,3)}

pair(3, 3) = 3→3



Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | $q \neq q0 \wedge q \neq qf$
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | $si \in S \wedge dj \in D$ **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) +
         RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*.
      RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

q: **2**

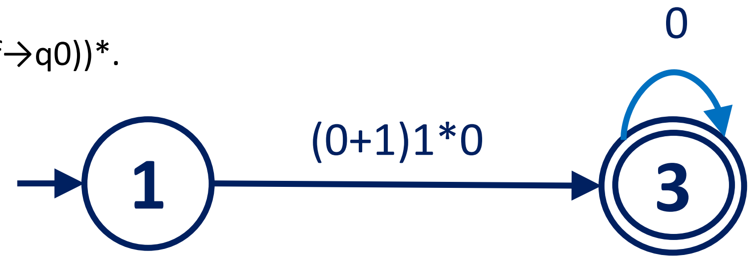
S = {**1**, **3**}

D = {**3**}

RE(1,2) = 0+1
RE(2,2) = 1
RE(2,3) = 0
RE(3,2) = 1

pairs(si, dj) = {(1,3), (3,3)}

pair(3, 3) = 3→3



Algorithm based on State Elimination

Input: FA with a single final state qf and an initial state q0

Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

q: 2

S = {**1**, **3**}

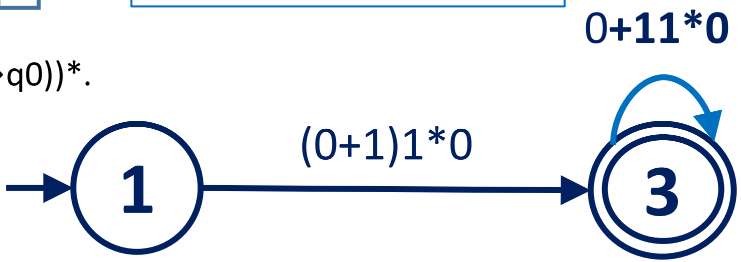
D = {**3**}

RE(1,2) = 0+1
RE(2,2) = 1
RE(2,3) = 0
RE(3,2) = 1

pairs(si, dj) = {(1,3), (3,3)}

pair(3, 3) = 3→3

RE(3,2).(RE(2,2))*.RE(2,3)



Algorithm based on State Elimination

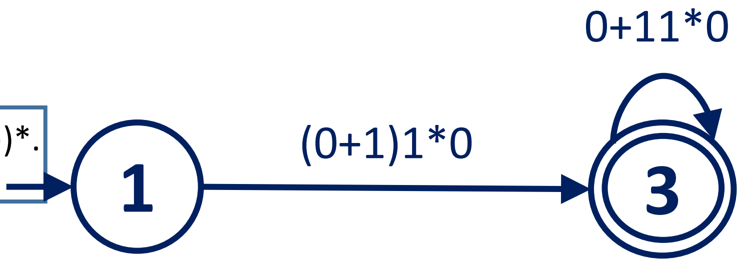
Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

q:

S = {}

D = {}



RE(1,1) = ε

RE(3,1) = ∅

RE=((RE(1,1)+RE(1,3).(RE(3,3))*.RE(3,1))*.RE(1,3).(RE(3,3))*

Algorithm based on State Elimination

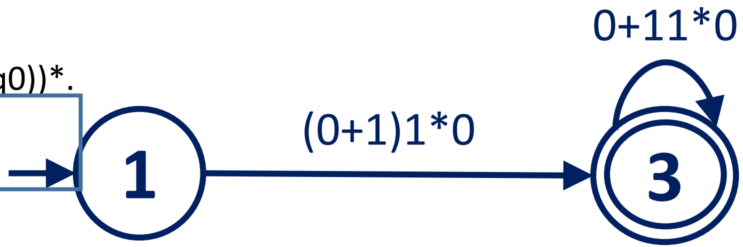
Input: FA with a single final state qf and an initial state q0
Output: a regular expression RE representing the language of the FA

1. Change transitions with multiple symbols to transitions with unions of those symbols;
2. **foreach** state q of the FA | q ≠ q0 ∧ q ≠ qf
   2.1. Get S (sources), the set of all states from which the state q can be reached
   2.2. Get D (destinations), the set of all states that can be reached from q
   2.3. Remove q from FA but keep info about transitions to/from q;
   2.4. foreach pair (si, dj) | si ∈ S ∧ dj ∈ D **do**
      2.4.1. Add a transition si → dj if it does not exist;
      2.4.2. Add a regular expression to transition si→dj: RE(si→dj) + RE(si→q).(RE(q→q))*.RE(q→dj); // passing through q
3. Resultant FA:
   3.1. Two states: RE = ((RE(q0→q0)+RE(q0→qf).(RE(qf→qf))*.RE(qf→q0))*. RE(q0→qf).(RE(qf→qf))*
   3.2. One state: RE = ((RE(q0→q0))*

q:

S = {}

D = {}



Output RE:

RE=(ε+(0+1)1*0.(0+11*0)*.∅)*.(0+1)1*0.(0+11*0)*=(0+1)1*0.(0+11*0)*

State Elimination: in the presence of more than one final state

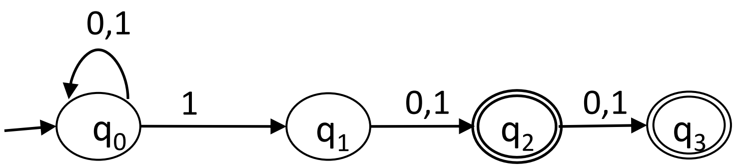
- And when we have more than one final state?
- Possible method:
  - Consider one FA per final state (mark all the other states as non-final) and then make the union of the regular expressions obtained for each FA
- Other method:
  - Mark the final states of the FA as non-final, connect them via ε transitions to a new final state
  - When there are input transitions to the initial state, we can also mark the initial as non-initial and connect a new initial state via an ε transition to the old initial state (this way we always obtain an FA with two states and the final regular expression in the only transition between the two)
  - The resultant FA is also known as a special form of the generalized nondeterministic finite automaton (GNFA)
  - This method avoids lines 3 and 4 in the pseudo-code presented in the beginning

State Elimination Method

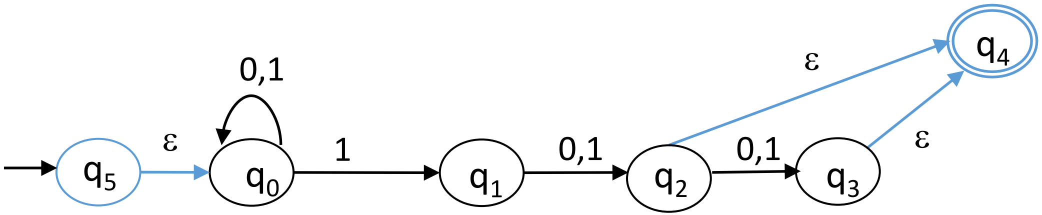
- We can always use the generalized nondeterministic finite automaton (GNFA)
- The use of GNFA usually helps

State Elimination: in the presence of more than one final state (example using the GNFA)

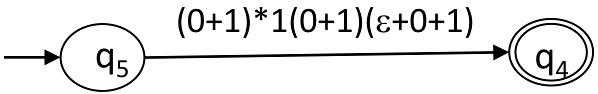
- NFA:



- GNFA:



Eliminating:
q1, then q0, then q3, and, finally, q2:



RE = (0+1)*1(0+1)(ε+0+1)

Ordering of the elimination of states

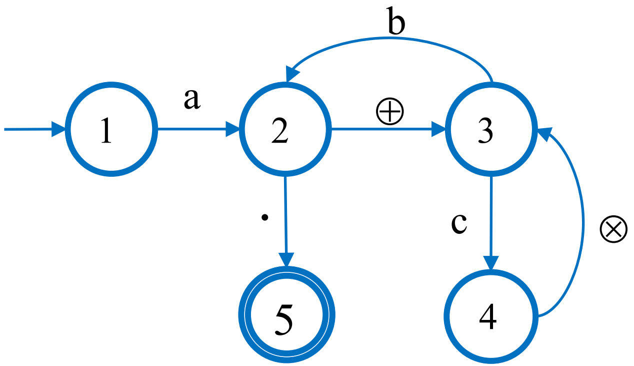
Impacts the length of the final regular expression

State Elimination: elimination ordering

- The order we consider the elimination of states influences the length (complexity) of the regular expression (RE)
  - One can use a heuristic to provide an ordering of the states to be eliminated
    - Example of heuristic: sort those states by the number of input/output transitions from/to other states in the FA
  - There are heuristics that for each state eliminated recalculate and select the next state to eliminate

State Elimination: elimination ordering

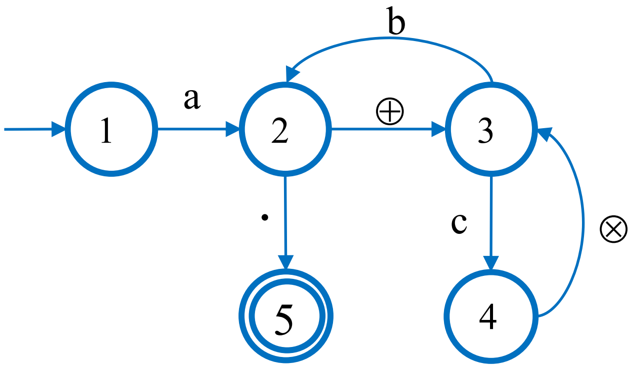
- Example of heuristic to select the elimination ordering:
  - the number of input/output transitions from/to other states in the FA
  - For FA A results in the following state elimination ordering: 4 $\rightarrow$ 3 $\rightarrow$ 2 or 4 $\rightarrow$ 2 $\rightarrow$ 3



State	#in/out transitions from/to other states
2	4
3	4
4	2

State Elimination: elimination ordering

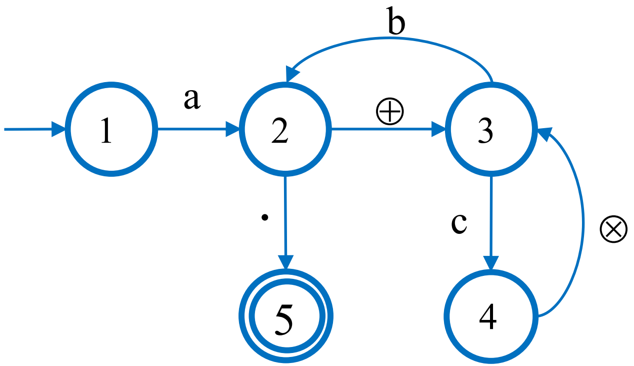
- Example of heuristic to select the elimination ordering:
  - the number of input/output transitions from/to other states in the FA
- For the FA below results in the following state elimination ordering: $4 \rightarrow 3 \rightarrow 2$ or $4 \rightarrow 2 \rightarrow 3$
  - The recalculation for each state eliminated would result in: $4 \rightarrow 3 \rightarrow 2$



State	#in/out transitions from/to other states	
	start	after eliminating 4
2	4	4
3	4	2
4	2	-

State Elimination: elimination ordering

- Use the DFA to show the impact of the elimination order in the final regular expression
- Does the ordering provided by the heuristic produce a more compact regular expression?



Summary

- We presented the State elimination technique to convert Finite Automata to Regular Expressions
- The size of the regular expressions obtained using the state elimination technique depends on the order the states are eliminated (there are methods for selecting the states to eliminate)
- The conversion of the FA to a generalized nondeterministic finite automaton (GNFA), with a start state with only a single transition to another state, and with a single final state, may help in the conversion

Further Reading

- State Elimination (known as *reduction procedure*):
  - J. A. Brzozowski & E. J. McCluskey (1963): Signal flow graph techniques for sequential circuit state diagrams. In IEEE Transactions on Computers C-12(2), pp. 67–76
- Heuristics for selecting the elimination ordering of states:
  - M. Delgado & J. Morais (2004): Approximation to the Smallest Regular Expression for a Given Regular Language. In Proceedings of the 9th Conference on Implementation and Application of Automata, LNCS 3317, Springer, Kingston, Ontario, Canada, pp. 312–314.
  - Yo-Sub Han and Derick Wood (2007): Obtaining shorter regular expressions from finite-state automata. *Theor. Comput. Sci.* 370, 1-3 (February 2007), pp. 110-120.
  - Yo-Sub Han (2013): State Elimination Heuristics for Short Regular Expressions, Fundamenta Informaticae, v.128 n.4, (October 2013) pp. 445-462.

Further Reading (cont.)

- Construction of Paths (Kleene’s transitive closure method)
  - Robert McNaughton & Hisao Yamada (1960): Regular expressions and state graphs for automata. IRE Transactions on Electronic Computers EC-9(1), pp. 39–47
- Brzozowski Algebraic method (based on Arden’s lemma)
  - D. N. Arden (1961): Delayed-Logic and Finite-State Machines. In T. Mott, editor: Proceedings of the 1st and 2nd Annual Symposium on Switching Theory and Logical Design, American Institute of Electrical Engineers, New York, Detroit, Michigan, USA, pp. 133–151.
  - Janusz A. Brzozowski, “Derivatives of regular expressions”, J. ACM,11(4) pp. 481-494, 1964.
  - J. H. Conway (1971): Regular Algebra and Finite Machines. Chapman and Hall.