Mathematics for Computing Science (CS2013)

Equivalence of FSAs and REs

Motivation

- The language of 0/1 strings with the same number of 0s and 1s
  - Cannot be recognised by any FSA
  - Cannot be expressed by any RE
- We mentioned that this was not a coincidence
  - FSAs and REs are equally expressive
  - This is formally established as a theorem (next slide)
- How interesting: proof are two algorithms
  - Algorithm 1: FSA specification as input and an equivalent RE as output
  - Algorithm 2: RE as input and an equivalent FSA as output

Kleene’s Theorem

- A language *L* is accepted by an FSA if, and only if, *L* is regular
  - Any language accepted by an FSA can be represented by an RE (if)
  - Any language represented by a RE has an FSA which accepts it (only if)

Non-Deterministic Finite State Automata (NDFSA)

A **Non-Deterministic Finite State Automaton** (**NDFSA**) A is a 5-tuple (Q, I, F, T, E) where:

- $Q = \{q_1, q_2, \ldots, q_n\}$ is the finite **set of states**
- $I \subseteq Q$, $I \neq \emptyset$, is the **set of initial states** (a nonempty subset of Q)
- $F \subseteq Q$, is the **set of final states** (a subset of Q)
- $T = \{a_1, a_2, \ldots, a_m\}$, is an **alphabet of symbols** a_i $1 \leq i \leq m$
- $E \subseteq Q \times (T \cup \{\lambda\}) \times Q$, is the set of edges (subset of Cartesian product)

Regular expressions

A **regular expression** over $T = \{a_1, a_2, \ldots, a_m\}$ is any string of the form

i. λ (lambda)
ii. $\varnothing$ (empty set)
iii. If $a_i \in T$ then $\boldsymbol{a_i}$ is a regular expression (symbol in boldface)
iv. If R and E are regular expressions then $(R+E)$ is a regular expression
v. If R and E are regular expressions then (RE) is a regular expression
vi. If R and E are regular expressions then (R^*) is a regular expression

Algorithm: from RE to NDFSA

- Let R be an RE. We will create A, an NDFSA accepting $S(R)$
  - A represented as a 5-tuple (Q, I, F, T, E) (T never changes below)

Base cases

- if $R = \lambda$ then $A = (\{q\}, \{q\}, \{q\}, T, \varnothing)$
- if $R = \varnothing$ then $A = (\{q\}, \{q\}, \varnothing, T, \varnothing)$
- if $R = \mathbf{a}$ then $A = (\{p, q\}, \{p\}, \{q\}, T, \{(p, a, q)\})$

Algorithm: from RE to NDFSA (cont'd)

General cases

- if $R = E_1 + E_2$ then obtain (recursively)

$$A_1 = (Q_1, \{i_1\}, \{f_1\}, T, E_1) \text{ such that } S(E_1) = L(A_1)$$

$$A_2 = (Q_2, \{i_2\}, \{f_2\}, T, E_2) \text{ such that } S(E_2) = L(A_2)$$

and combine A_1 with A_1:

$$A = (Q_1 \cup Q_2 \cup \{i, f\}, \{i\}, \{f\}, T, E_1 \cup E_2 \cup \{(i,\lambda,i_1),(i,\lambda,i_2),(f_1,\lambda,f),(f_2,\lambda,f)\})$$

- if $R = E_1 E_2$ then obtain A_1 and A_2 as above and combine them:

$$A = (Q_1 \cup Q_2, \{i_1\}, \{f_2\}, T, E_1 \cup E_2 \cup \{(f_1,\lambda,i_2)\})$$

- if $R = E_1^*$ then obtain A_1 as above and use it as follows:

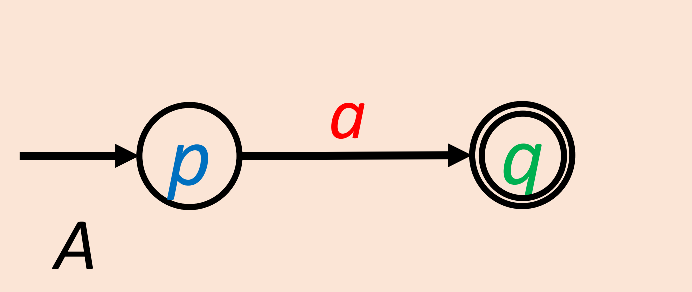
$$A = (Q_1 \cup \{i, f\}, \{i\}, \{f\}, T, E_1 \cup \{(i,\lambda,i_1),(i,\lambda,f),(f_1,\lambda,f),(f_1,\lambda,i_1)\})$$

Algorithm: from RE to NDFSA (explained)

- States created on demand, and combined accordingly
- Transitions/edges capture effects of operators in REs
- FSA created with a single initial state and a single final state
  - Cheap trick: λ-transitions to combine FSAs (all's fair…)

Base cases

- if $R = \lambda$ then $A = (\{q\}, \{q\}, \{q\}, T, \emptyset)$ ($I = F$, that is, initial state is final)
- if $R = \emptyset$ then $A = (\{q\}, \{q\}, \emptyset, T, \emptyset)$ ($F = \emptyset$, that is, no final states!)
- if $R = \mathbf{a}$ then $A = (\{p, q\}, \{p\}, \{q\}, T, \{(p, a, q)\})$ (that is, the FSA below)



Algorithm: from RE to NDFSA (explained; cont'd)

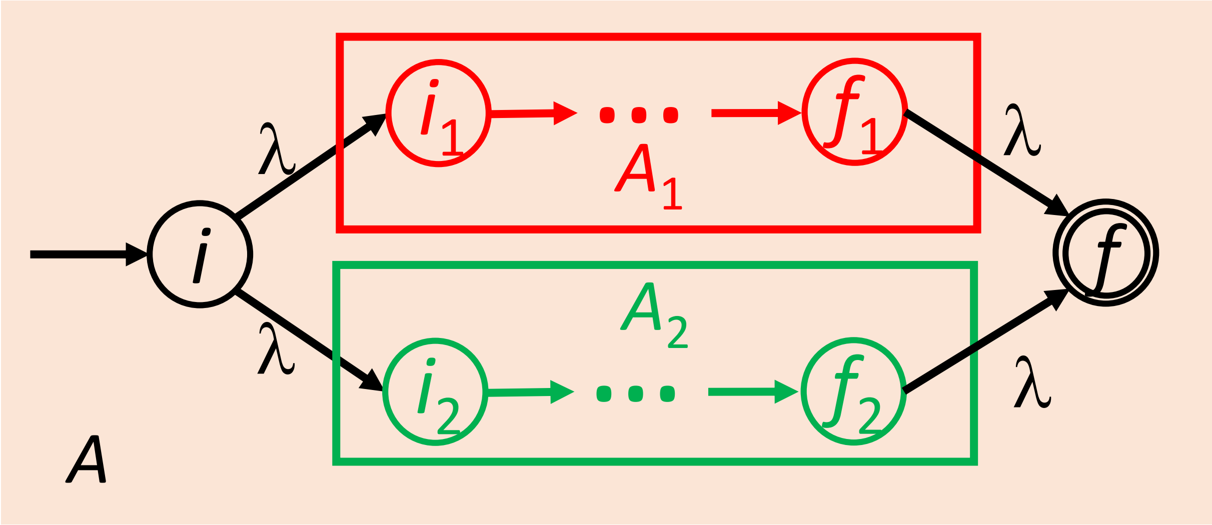
- if $R = E_1 + E_2$ then obtain (recursively)

$$A_1 = (Q_1, \{i_1\}, \{f_1\}, T, E_1) \text{ such that } S(E_1) = L(A_1)$$

$$A_2 = (Q_2, \{i_2\}, \{f_2\}, T, E_2) \text{ such that } S(E_2) = L(A_2)$$

and combine A_1 with A_1:

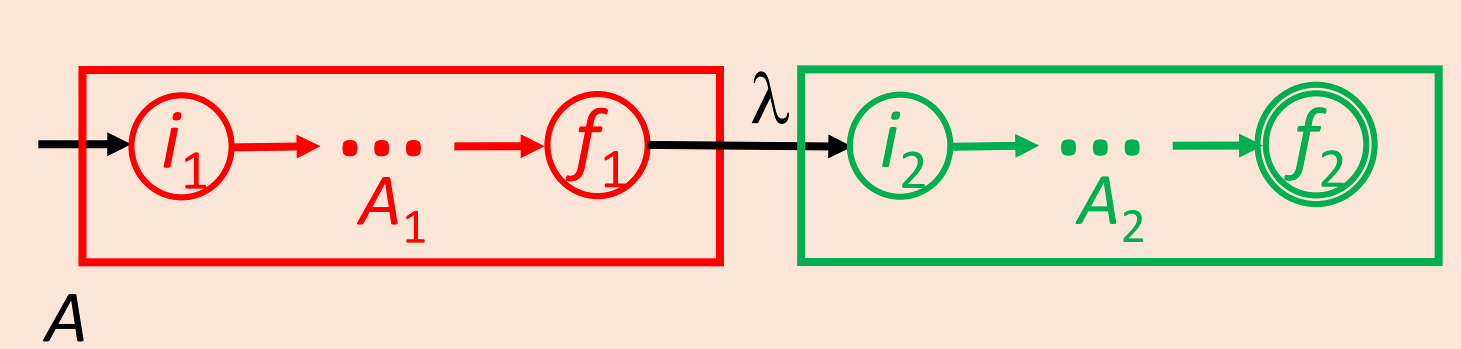
$$A = (Q_1 \cup Q_2 \cup \{i, f\}, \{i\}, \{f\}, T, E_1 \cup E_2 \cup \{(i,\lambda,i_1),(i,\lambda,i_2),(f_1,\lambda,f),(f_2,\lambda,f)\})$$



Algorithm: from RE to NDFSA (explained; cont'd)

- if $R = E_1E_2$ then obtain A_1 and A_2 as above and combine them:

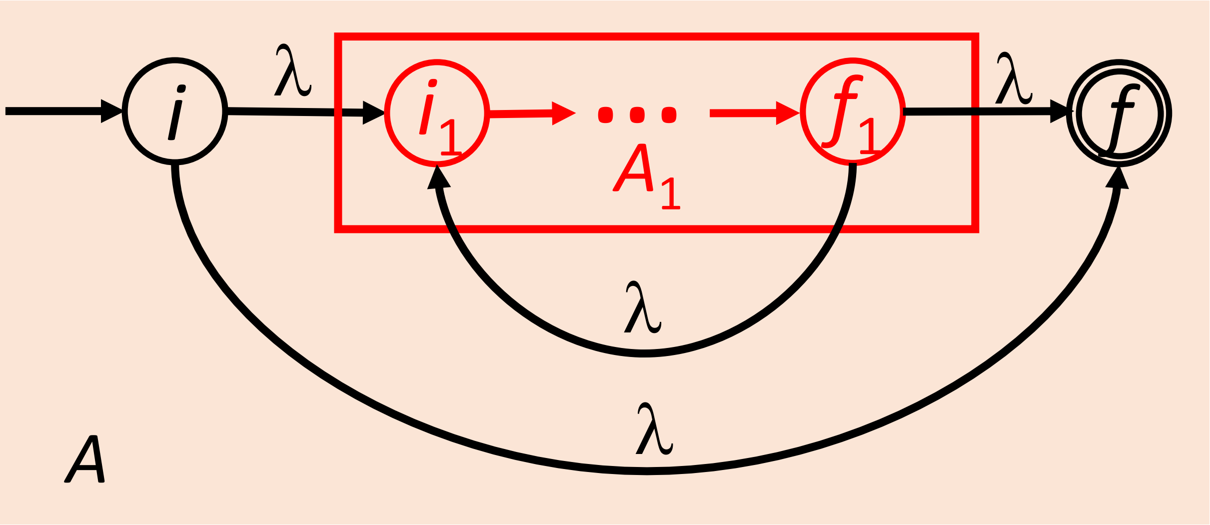
$$A = (Q_1 \cup Q_2, \{i_1\}, \{f_2\}, T, E_1 \cup E_2 \cup \{(f_1,\lambda,i_2)\})$$



Algorithm: from RE to NDFSA (explained; cont'd)

- if $R = E_1^*$ then obtain A_1 as above and use it as follows:

$$A = (Q_1 \cup \{i, f\}, \{i\}, \{f\}, T, E_1 \cup \{(i,\lambda,i_1),(i,\lambda,f),(f_1,\lambda,f),(f_1,\lambda,i_1)\})$$



Algorithm: from RE to NDFSA – where are we?

- All three base cases (simple REs) addressed
- All other kinds of more complex REs addressed
- Any RE combines these cases a finite number of times
- We nailed it – the RE to FSA direction is proven!

Example – from RE to NDFSA

Using our algorithm, find FSA for (**b**+**ab**)(**b**+**ab**)*, $T = \{a, b\}$

- Decompose RE to its basic cases
- Let's use numbers to label states (and not q_n)

1. RE (**b**+**ab**) decomposed (recursively) into sub-REs, that is,

1.1. **b**

1.2. **ab**

1.2.1 **a** is $(\{1,2\},\{1\},\{2\},T,\{(1,a,2)\})$

1.2.2 **b** is $(\{3,4\},\{3\},\{4\},T,\{(3,b,4)\})$

1.2. **ab** is $(\{1,2,3,4\},\{1\},\{4\},\mathrm{T},\{(1,a,2),(2,\lambda,3),(3,b,4)\})$

1.1 **b** is $(\{5,6\},\{5\},\{6\},T,\{(5,b,6)\})$

1 is

$(\{1,\ldots,6,7,8\},\{7\},\{8\},T,\{(7,\lambda,1),(7,\lambda,5),(1,a,2),(2,\lambda,3),(3,b,4),(5,b,6), (4,\lambda,8),(6,\lambda,8)\})$

Example – from RE to NDFSA (cont'd)

2. RE (**b**+**ab**)* decomposed (recursively) into sub-REs, that is,

2.1. (**b**+**ab**) (same as 1 of previous slide, but different states labels)

({9,…,16},{15},{16},*T*,
{(15,λ,9),(15,λ,13),(9,a,10),
(10,λ,11),(11,b,12),(13,b,14),
(12,λ,16),(14,λ,16)})

2. We add new initial and final states and the λ-transitions:

({9,…,16,17,18},{17},{18},*T*,
{(17,λ,15),(17,λ,18),(15,λ,9),(15,λ,13),
(9,a,10),(10,λ,11),(11,b,12),(13,b,14),
(12,λ,16),(14,λ,16),(16,λ,18), (16,λ,15)})

Example – from RE to NDFSA (cont'd)

We put all together:

- A = ({1,2,...,18},{7},{18},T,

  {(7,λ,1),(7,λ,5),(1,a,2),(2,λ,3),(3,b,4),(5,b,6),(4,λ,8),(6,λ,8),

  (8,λ,17),

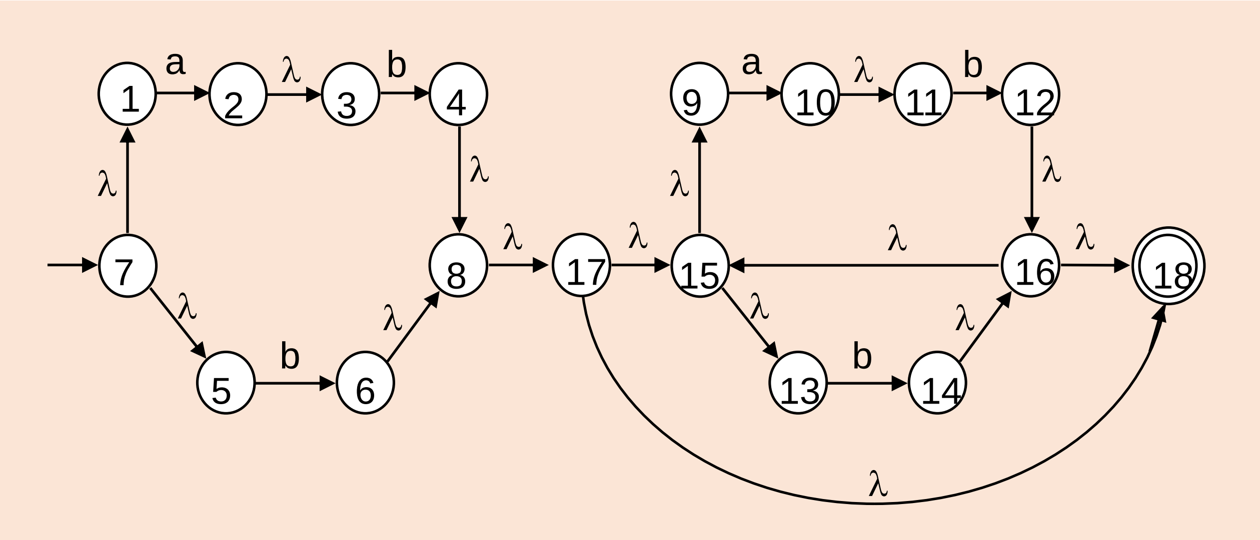
  (17,λ,15),(17,λ,18),(15,λ,9),(15,λ,13),

  (9,a,10),(10,λ,11),(11,b,12),

  (13,b,14),(12,λ,16),(14,λ,16),(16,λ,18),(16,λ,15)})

Example – from RE to NDFSA (cont'd)

Graphically:



- Non-deterministic…
  - We can create (automatically!) a deterministic version of it
  - We skipped the proof/algorithm…

Where are we?

- We’ve seen how REs can be used to obtain an NDFSA that accepts the same language
- We now need to do the reverse, that is, given an NDFSA specification, we obtain (automatically) an RE which denotes the same language

Non-Deterministic Finite State Automata (NDFSA)

A **Non-Deterministic Finite State Automaton** (**NDFSA**) A is a 5-tuple (Q, I, F, T, E) where:

- $Q = \{q_1, q_2, \ldots, q_n\}$ is the finite **set of states**
- $I \subseteq Q$, $I \neq \emptyset$, is the **set of initial states** (a nonempty subset of Q)
- $F \subseteq Q$, is the **set of final states** (a subset of Q)
- $T = \{a_1, a_2, \ldots, a_m\}$, is an **alphabet of symbols** a_i $1 \leq i \leq m$
- $E \subseteq Q \times (T \cup \{\lambda\}) \times Q$, is the set of edges (subset of Cartesian product)

Algorithm: from FSA to RE

convertFSAtoRE(A)

input: FSA A

output: RE R such that $S(R) = L(A)$

1. $A' \leftarrow$ *uniqueInitialState*(A) % create equivalent A' with 1 initial state
2. $A'' \leftarrow$ *uniqueFinalState*(A') % create equivalent A'' with 1 final state
3. **return** *convertFSA*(A'') % convert A'' onto RE

Algorithm: from FSA to RE (cont'd)

uniqueInitialState(A)

input A of the form (Q, I, F, T, E)

output: A' of the form (Q', I', F, T, E')

1. $Q' \leftarrow Q \cup \{i\}$ % create "fresh" state as new initial state
2. $E' \leftarrow E$ % initialise edges with previous ones
3. **for all** $q \in I$ **do**
4. $\quad E' \leftarrow E' \cup \{(i, \lambda, q)\}$ % connect new initial state to old initial states
5. $I' \leftarrow \{i\}$ % new set of initial states is just $\{i\}$
6. **return** A'

Algorithm: from FSA to RE (cont'd)

uniqueFinalState(A)

input A of the form (Q, I, F, T, E)

output: A' of the form (Q', I, F', T, E')

1. $Q' \leftarrow Q \cup \{f\}$ % create "fresh" state as new initial state
2. $E' \leftarrow E$ % initialise edges with previous ones
3. **for all** $q \in F$ **do**
4. $\quad E' \leftarrow E' \cup \{(q, \lambda, f)\}$ % connect old final states to new final state
5. $F' \leftarrow \{f\}$ % new set of final states is just $\{f\}$
6. **return** A'

A short detour: regular FSAs

- Conversion is easier if we use a "trick"
  - An FSA with regular expression on edges/transitions
  - This is not cheating – it's a bit like a temporary data structure
- A **regular finite state automaton** (**RFSA**) is a FSA where edge labels are regular expressions
  - An edge labelled with a regular expression R means that we can move along that edge on input of any string in $S(R)$ (the language represented by R)

Algorithm: unique-initial-and-final-states FSA to RE

- Let A be an FSA with unique initial and final states
  - We went through steps 1 and 2
  - We need to define step 3 – $convertFSA(A'')$
- In a number of steps, A can be converted to an RFSA with just one edge, whose label is the required regular expression
- We show the whole process in the next slide and how it is used

Algorithm: unique FSA to RE (cont'd)

convertFSA(A)

input: FSA A

output: RE R

while $(Q - \{i, f\}) \neq \emptyset$ % while there are states in Q different from the unique new initial and final states $\{i, f\}$

  for all $p \in Q$ % p stands for any state (a "variable")

    $PPEs \leftarrow \{(p, R_i, p) \mid (p, R_i, p) \in E\}$ % get edges from p to itself

    if $|PPEs| > 1$ **then** $E \leftarrow (E - PPEs) \cup \{(p, R_1+R_2+\ldots+R_n, p)\}$ % if more than 1 p-p edge replace them with single equivalent one

  for all $p, q \in Q$, $p \neq q$

    $PQEs \leftarrow \{(p, R_i, q) \mid (p, R_i, q) \in E\}$ % get edges from p to q

    if $|PQEs| > 1$ **then** $E \leftarrow (E - PQEs) \cup \{(p, R_1+R_2+\ldots+R_n, q)\}$ % if more than 1 p-q edge replace them with single equivalent one

  select $s \in (Q - \{i, f\})$ % select a state different from new initial and final states

  for all $p, q \in Q$, $p \neq q$, $p \neq s$, $q \neq s$ % for all pairs p and q, different from each other and different from chosen s

    if $(p, R_1, s) \in E$ **and** $(s, R_2, q) \in E$ **then** % if p connected to s and s connected to q

      if $(s, R_3, s) \in E$ **then** $E \leftarrow E \cup \{(p, R_1R_3{}^*R_2, q)\}$ % create this edge if s connected to itself

        else $E \leftarrow E \cup \{(p, R_1R_2, q)\}$ % create this edge if s not connected to itself

  $SEs \leftarrow \{(t, R, s) \mid (t, R, s) \in E\} \cup \{(s, R, t) \mid (s, R, t) \in E\}$ % get all edges to and from s

  $E \leftarrow E - SEs$ % remove these edges

  for all $p \in Q$ **if** there is no path(i, p) **then** $Q \leftarrow Q - \{p\}$ % remove all states which cannot be reached from i

  for all $(p, R, q) \in E$ **if** $p \notin Q$ **or** $q \notin Q$ **then** $E \leftarrow E - \{(p, R, q)\}$ % remove all edges whose states (either of them) no longer exist

return R, where $E = \{(i, R, f)\}$ % final set of edges has exactly one edge from i to f

Algorithm: unique FSA to RE (cont'd)

- It's a lot to take in, but don't panic
  - We will illustrate how the algorithm work with examples
- It works by examining the input FSA and **altering** it:
  - Edges are created and added to it
  - Old edges and states no longer necessary are removed
- At the end of the while loop
  - There will be just two states *i* and *f*
  - One single edge (*i*, *R*, *f*)
- The label *R* of the remaining edge is the regular expression we want
- Let's examine some parts of the algorithm to get a feel for it

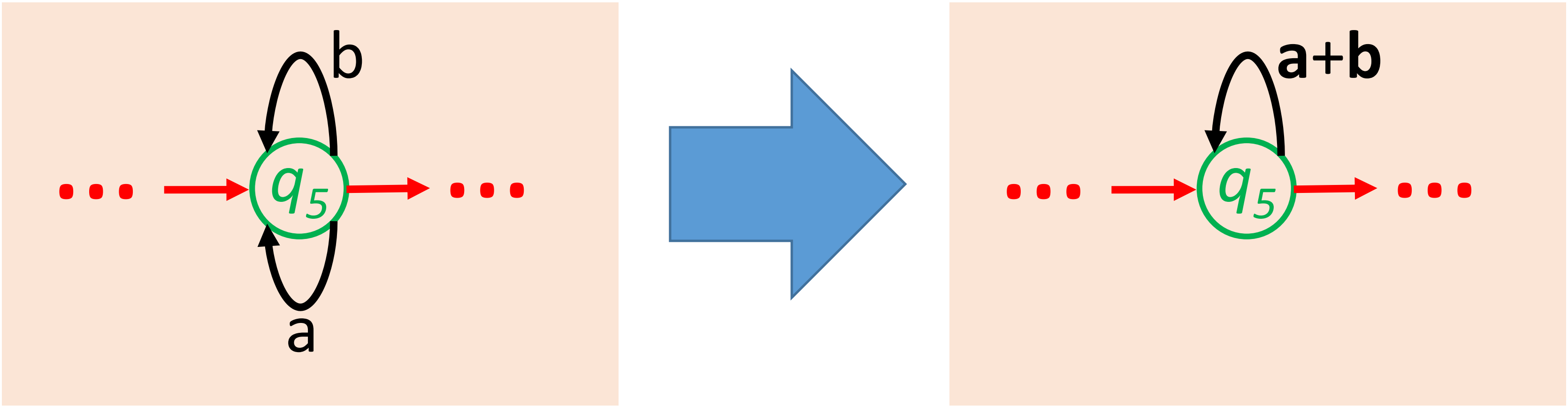
Algorithm: unique FSA to RE (cont'd)

while $(Q - \{i, f\}) \neq \emptyset$

for all $p \in Q$

$PPEs \leftarrow \{(p, R_i, p) \mid (p, R_i, p) \in E\}$

if $|PPEs| > 1$ **then** $E \leftarrow (E - PPEs) \cup \{(p, R_1+R_2+\ldots+R_n, p)\}$



Algorithm: unique FSA to RE (cont'd)

convertFSA(A)

input: FSA A

output: RE R

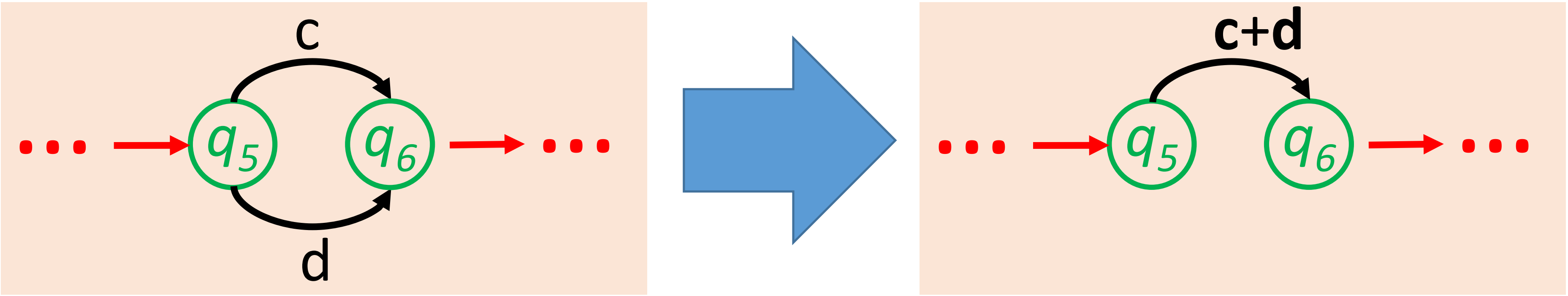
while $(Q - \{i, f\}) \neq \emptyset$

…

for all $p, q \in Q$, $p \neq q$

$PQEs \leftarrow \{(p, R_i, q) \mid (p, R_i, q) \in E\}$

if $|PQEs| > 1$ **then** $E \leftarrow (E - PQEs) \cup \{(p, R_1+R_2+\ldots+R_n, q)\}$



Algorithm: unique FSA to RE (cont'd)

convertFSA(A)

input: FSA A

output: RE R

while $(Q - \{i, f\}) \neq \emptyset$

…

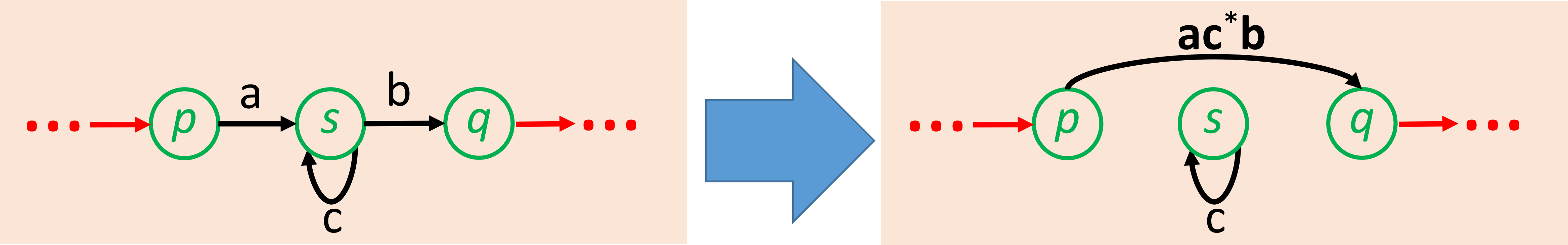
select $s \in (Q - \{i, f\})$

for all $p, q \in Q$, $p \neq q$, $p \neq s$, $q \neq s$

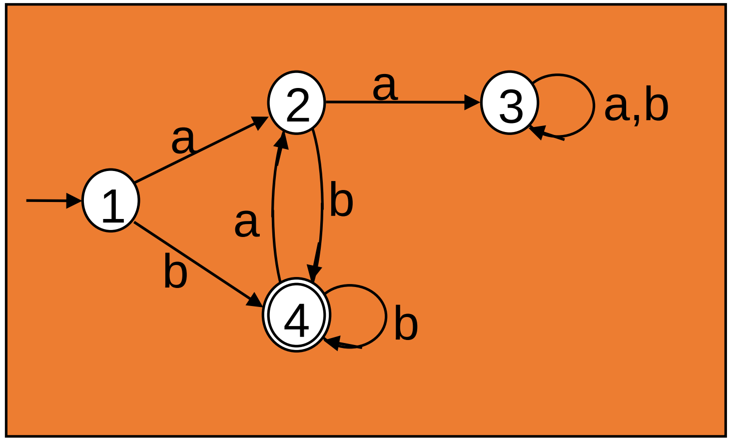
if $(p, R_1, s) \in E$ **and** $(s, R_2, q) \in E$ **then**

if $(s, R_3, s) \in E$ **then** $E \leftarrow E \cup \{(p, R_1R_3{}^*R_2, q)\}$ % create this edge if s connected to itself

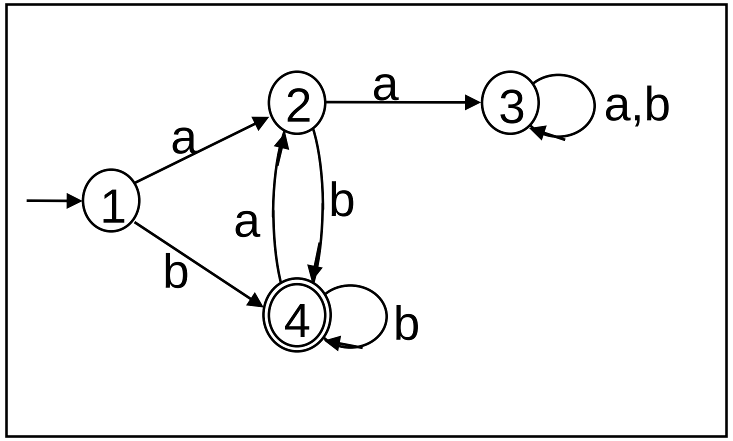
else $E \leftarrow E \cup \{(p, R_1R_2, q)\}$ % create this edge if s not connected to itself



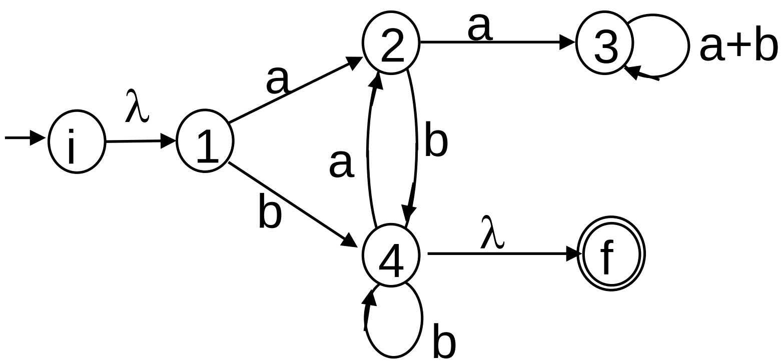
Complete example: from FSA to RE



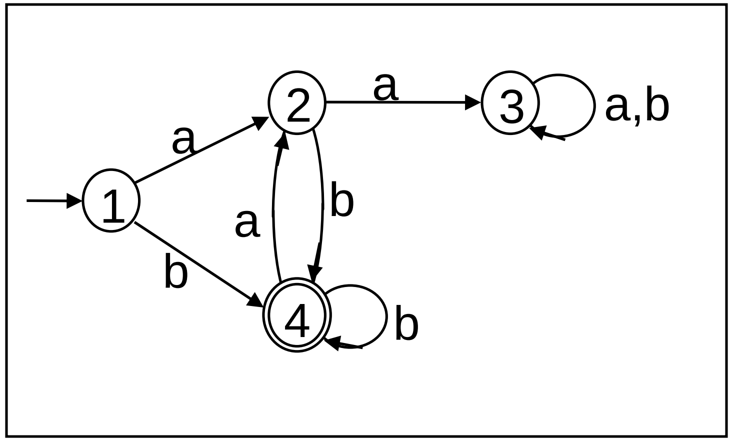
Complete example: from FSA to RE (cont'd)



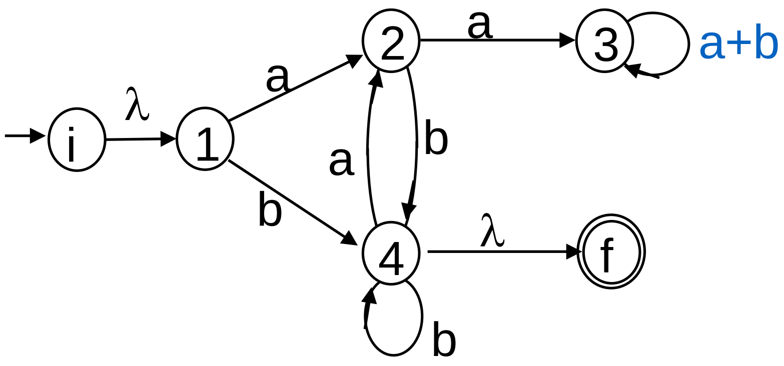
create unique initial and final states; add a+b loop



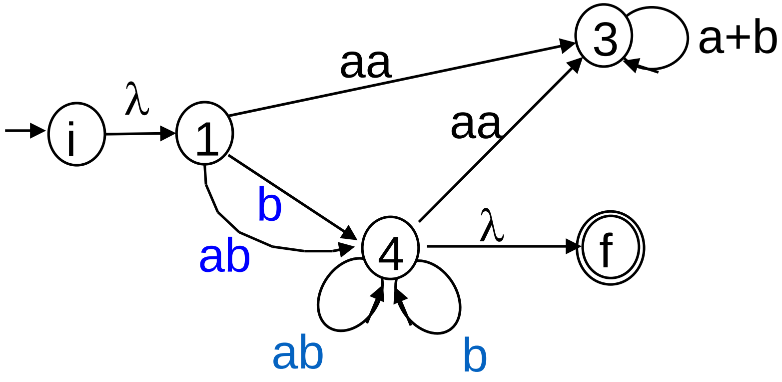
Complete example: from FSA to RE (cont'd)



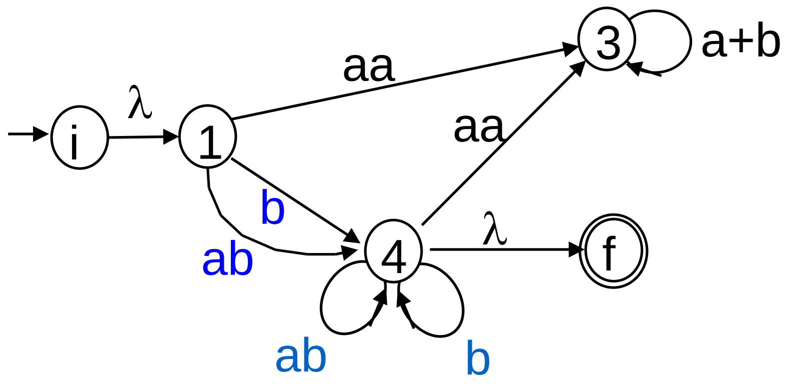
create unique initial and final states; add a+b loop



remove state 2 - edges are 1-3, 1-4, 4-3, 4-4

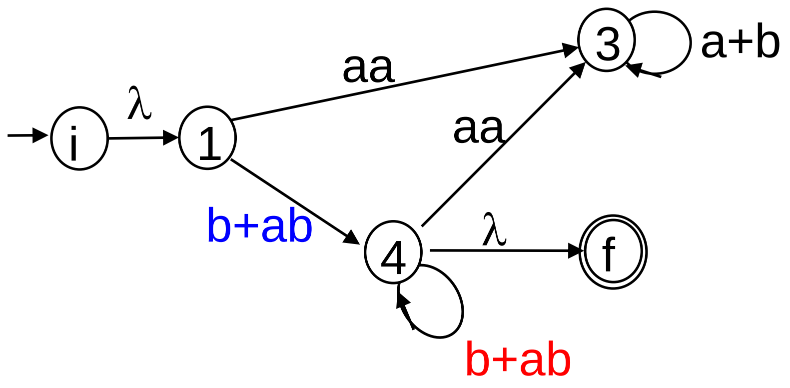


Complete example: from FSA to RE (cont'd)

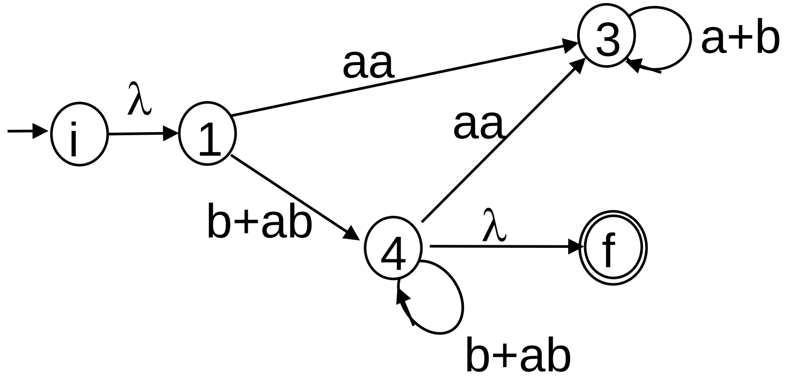


Complete example: from FSA to RE (cont'd)

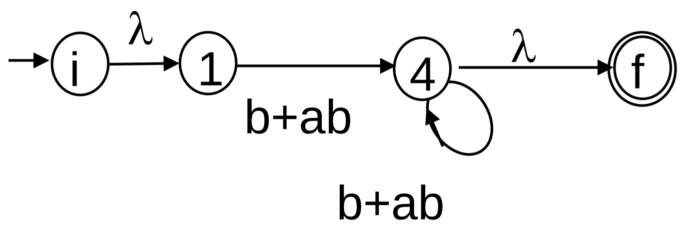
combine: b+ab, b+ab



Complete example: from FSA to RE (cont'd)

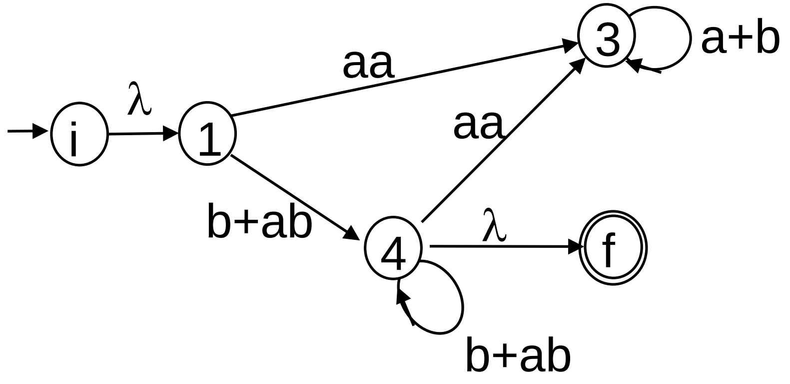


remove state 3 - no edges

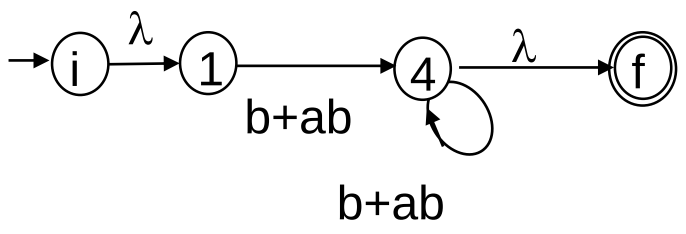


Complete example: from FSA to RE (cont'd)

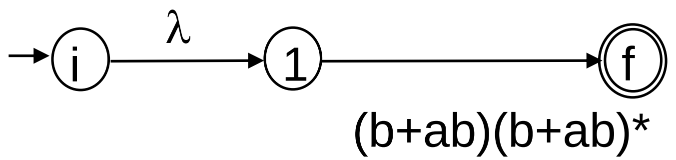
remove edge pairs



remove state 3 - no edges

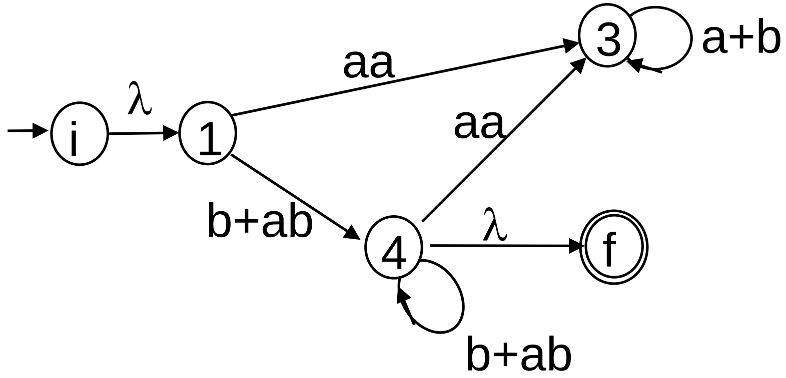


remove state 4 - edge is 1-f

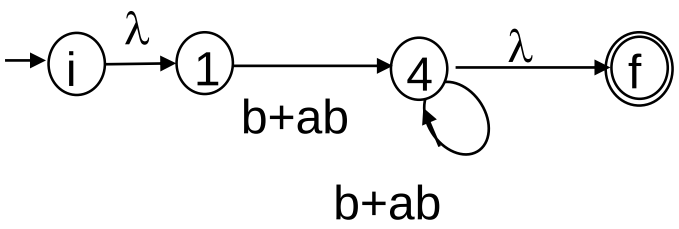


Complete example: from FSA to RE (cont’d)

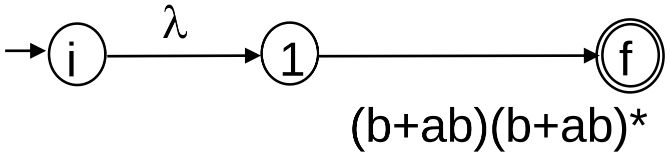
remove edge pairs



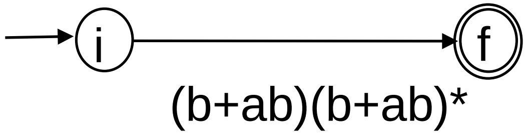
remove state 3 - no edges



remove state 4 - edge is 1-f



remove state 1 - edge is i-f



expression is: (**b**+**ab**)(**b**+**ab**)*

Is this a proper proof/algorithm?

- The first half (RE to FSA) is not problematic
  - Proof follows the (recursive) definition of syntax of REs
  - Wrinkle: A may lack a final state (the case where $L = \{\}$)
- The second half (FSA to RE):
  - Algorithm not fully specified (e.g., “select a state s”)
  - Does the order in which states are selected not matter?
  - Is the resulting RE always equivalent to the initial FSA?
- These wrinkles can be ironed out

Summary

- Finite state automata and regular expressions have same expressivity
- Proof:
  - Given a regular expression we can find an equivalent FSA
  - Given an FSA we can find an equivalent regular expression
- You should now be able to use the algorithm

Further reading

- Chapter 1 of "Introduction to the theory of computation", by Michael Sipser
- Chapter 3 of "An Introduction to Formal Languages and Automata", by Peter Linz (PDF available on-line)