Compilers and Formal Languages (3)

Email: christian.urban at kcl.ac.uk

Office: N7.07 (North Wing, Bush House)

Slides: KEATS (also homework and course-work is there)

Scala Book, Exams

- https://nms.kcl.ac.uk/christian.urban/ProgInScala2ed.pdf
- homework (written exam 80%)
- coursework (20%)

- short survey at KEATS; to be answered until Sunday

Regular Expressions

In programming languages they are often used to recognise:

- symbols, digits
- identifiers
- numbers (non-leading zeros)
- keywords
- comments

`http://www.regexper.com`

Last Week

Last week I showed you a regular expression matcher that works provably correct in all cases (we only started with the proving part though)

$$\mathit{matches}\ s\ r \quad \text{if and only if} \quad s \in L(r)$$

by Janusz Brzozowski (1964)

The Derivative of a Rexp

$$
\begin{array}{lcl}
der\, c\, (\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
der\, c\, (\mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{0} \\
der\, c\, (d) & \stackrel{\text{def}}{=} & \text{if } c = d \text{ then } \mathbf{1} \text{ else } \mathbf{0} \\
der\, c\, (r_1 + r_2) & \stackrel{\text{def}}{=} & der\, c\, r_1 + der\, c\, r_2 \\
der\, c\, (r_1 \cdot r_2) & \stackrel{\text{def}}{=} & \text{if } nullable(r_1) \\
 & & \text{then } (der\, c\, r_1) \cdot r_2 + der\, c\, r_2 \\
 & & \text{else } (der\, c\, r_1) \cdot r_2 \\
der\, c\, (r^*) & \stackrel{\text{def}}{=} & (der\, c\, r) \cdot (r^*) \\
 & & \\
ders\, []\, r & \stackrel{\text{def}}{=} & r \\
ders\, (c :: s)\, r & \stackrel{\text{def}}{=} & ders\, s\, (der\, c\, r)
\end{array}
$$

Example

Given $r \stackrel{\text{def}}{=} ((a \cdot b) + b)^*$ what is

$$
\begin{array}{lcl}
der\, a\, ((a \cdot b) + b)^* & \Rightarrow & der\, a\, \underline{((a \cdot b) + b)^*} \\
& = & (der\, a\, (\underline{(a \cdot b) + b})) \cdot r \\
& = & ((der\, a\, (\underline{a \cdot b})) + (der\, a\, b)) \cdot r \\
& = & (((der\, a\, \underline{a}) \cdot b) + (der\, a\, b)) \cdot r \\
& = & ((\mathbf{1} \cdot b) + (der\, a\, \underline{b})) \cdot r \\
& = & ((\mathbf{1} \cdot b) + \mathbf{0}) \cdot r
\end{array}
$$

Input: string abc and regular expression r

1. $der\, a\, r$
2. $der\, b\, (der\, a\, r)$
3. $der\, c\, (der\, b\, (der\, a\, r))$

Input: string abc and regular expression r

1. $der\, a\, r$
2. $der\, b\, (der\, a\, r)$
3. $der\, c\, (der\, b\, (der\, a\, r))$

4. finally check whether the last regular expression can match the empty string

Simplification

Given $r \stackrel{\text{def}}{=} ((a \cdot b) + b)^*$ what is

$$\begin{array}{rcl} ((\mathbf{1} \cdot b) + \mathbf{0}) \cdot r & \Rightarrow & (\underline{(\mathbf{1} \cdot b)} + \mathbf{0}) \cdot r \\ & = & (\underline{b + \mathbf{0}}) \cdot r \\ & = & b \cdot r \end{array}$$

We proved

$$\mathit{nullable}(r) \text{ if and only if } [] \in L(r)$$

by induction on the regular expression r.

We proved

$$nullable(r) \text{ if and only if } [] \in L(r)$$

by induction on the regular expression r.

Any Questions?

We need to prove

$$L(der\, c\, r) = Der\, c\, (L(r))$$

also by induction on the regular expression r.

Proofs about Rexps

- P holds for $\mathbf{0}$, $\mathbf{1}$ and c
- P holds for $r_1 + r_2$ under the assumption that P already holds for r_1 and r_2.
- P holds for $r_1 \cdot r_2$ under the assumption that P already holds for r_1 and r_2.
- P holds for r^* under the assumption that P already holds for r.

Proofs about Natural Numbers and Strings

- P holds for 0 and
- P holds for $n + 1$ under the assumption that P already holds for n

- P holds for $[]$ and
- P holds for $c :: s$ under the assumption that P already holds for s

Regular Expressions

$$
\begin{array}{rcll}
r & ::= & \mathbf{0} & \text{nothing} \\
 & | & \mathbf{1} & \text{empty string / ”” / []} \\
 & | & c & \text{character} \\
 & | & r_1 \cdot r_2 & \text{sequence} \\
 & | & r_1 + r_2 & \text{alternative / choice} \\
 & | & r^* & \text{star (zero or more)}
\end{array}
$$

How about ranges $[a\text{-}z]$, r^+ and $\sim r$? Do they increase the set of languages we can recognise?

Negation of Regular Expr's

- $\sim r$ (everything that r cannot recognise)
- $L(\sim r) \stackrel{\text{def}}{=} UNIV - L(r)$
- $nullable(\sim r) \stackrel{\text{def}}{=} not\,(nullable(r))$
- $der\,c\,(\sim r) \stackrel{\text{def}}{=} \sim (der\,c\,r)$

Negation of Regular Expr's

- $\sim r$ (everything that r cannot recognise)
- $L(\sim r) \stackrel{\text{def}}{=} UNIV - L(r)$
- $nullable(\sim r) \stackrel{\text{def}}{=} not\,(nullable(r))$
- $der\,c\,(\sim r) \stackrel{\text{def}}{=} \sim (der\,c\,r)$

Used often for recognising comments:

$$/ \cdot * \cdot (\sim ([a\text{-}z]^* \cdot * \cdot / \cdot [a\text{-}z]^*)) \cdot * \cdot /$$

Negation

Assume you have an alphabet consisting of the letters *a*, *b* and *c* only. Find a (basic!) regular expression that matches all strings *except* *ab* and *ac*!

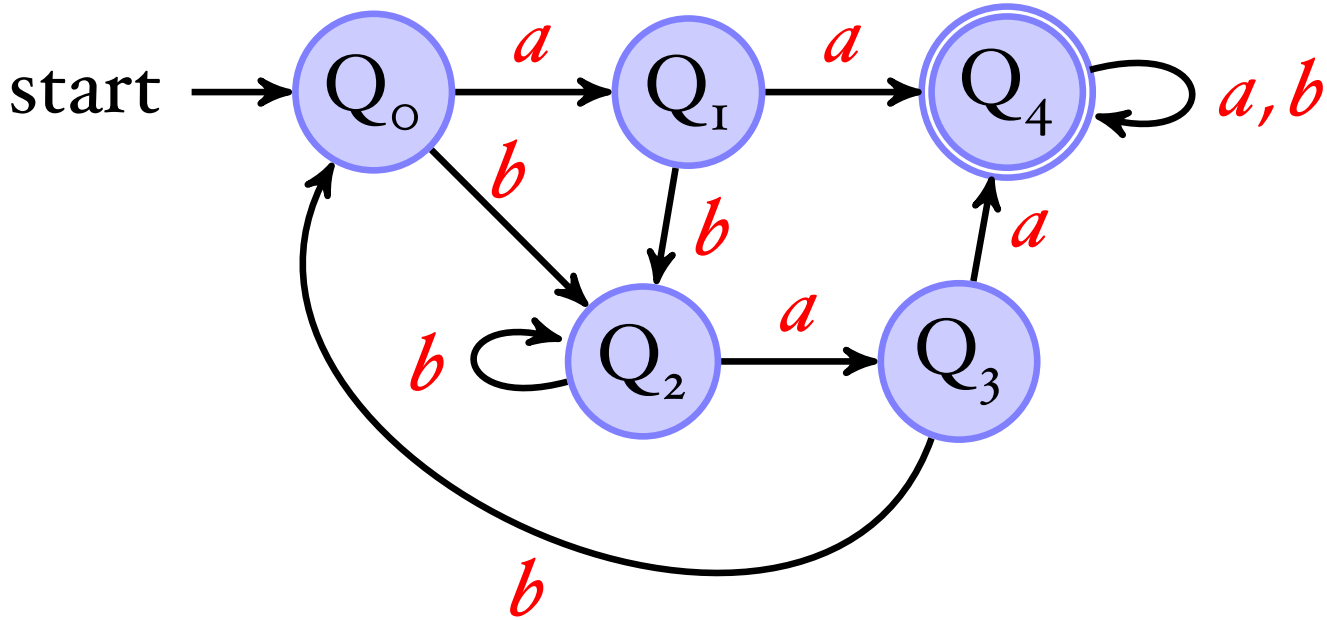
Automata

A **deterministic finite automaton**, DFA, consists of:

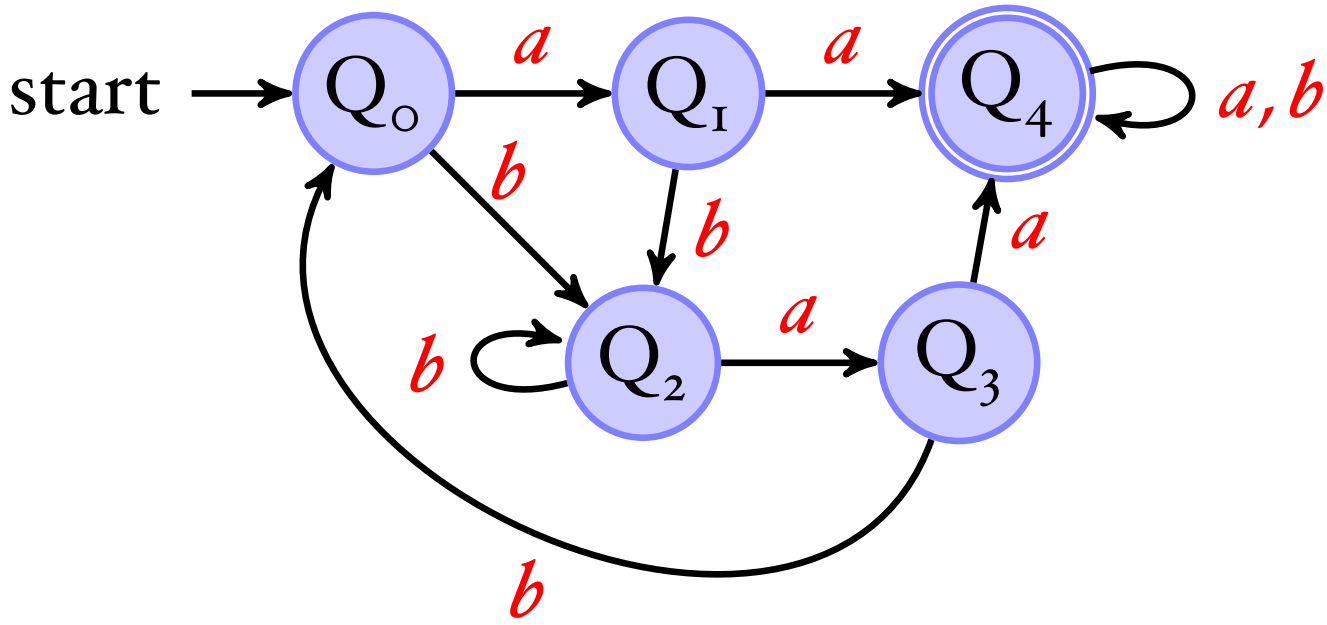
- an alphabet Σ
- a set of states $\mathcal{Q}$
- one of these states is the start state Q_o
- some states are accepting states F, and
- there is transition function δ

which takes a state as argument and a character and produces a new state; this function might not be everywhere defined $\Rightarrow$ partial function

$$A(\Sigma, Q, Q_o, F, \delta)$$



- the start state can be an accepting state
- it is possible that there is no accepting state
- all states might be accepting (but this does not necessarily mean all strings are accepted)



for this automaton δ is the function

$$(Q_0, a) \to Q_1 \quad (Q_1, a) \to Q_4 \quad (Q_4, a) \to Q_4$$
$$(Q_0, b) \to Q_2 \quad (Q_1, b) \to Q_2 \quad (Q_4, b) \to Q_4 \quad \ldots$$

Accepting a String

Given

$$A(\Sigma, Q, Q_0, F, \delta)$$

you can define

$$\widehat{\delta}(q, []) \stackrel{\text{def}}{=} q$$
$$\widehat{\delta}(q, c :: s) \stackrel{\text{def}}{=} \widehat{\delta}(\delta(q, c), s)$$

Accepting a String

Given

$$A(\Sigma, Q, Q_0, F, \delta)$$

you can define

$$\widehat{\delta}(q, []) \stackrel{\text{def}}{=} q$$
$$\widehat{\delta}(q, c :: s) \stackrel{\text{def}}{=} \widehat{\delta}(\delta(q, c), s)$$

Whether a string s is accepted by A?

$$\widehat{\delta}(Q_0, s) \in F$$

Regular Languages

A **language** is a set of strings.

A **regular expression** specifies a language.

A language is **regular** iff there exists a regular expression that recognises all its strings.

Regular Languages

A **language** is a set of strings.

A **regular expression** specifies a language.

A language is **regular** iff there exists a regular expression that recognises all its strings.

not all languages are regular, e.g. $a^n b^n$ is not

Regular Languages (2)

A language is **regular** iff there exists a regular expression that recognises all its strings.

or **equivalently**

A language is **regular** iff there exists a deterministic finite automaton that recognises all its strings.

Non-Deterministic Finite Automata

A non-deterministic finite automaton (NFA) consists again of:

- a finite set of states
- some these states are the start states
- some states are accepting states, and
- there is transition relation

$$(Q_1, a) \rightarrow Q_2$$
$$(Q_1, a) \rightarrow Q_3 \quad \ldots$$

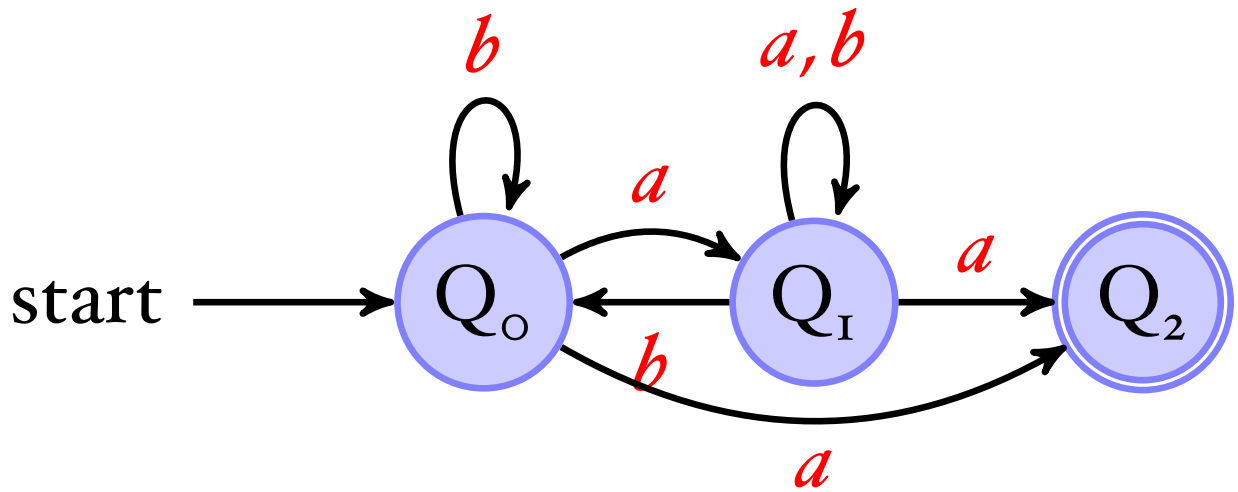
Non-Deterministic Finite Automata

A non-deterministic finite automaton (NFA) consists again of:

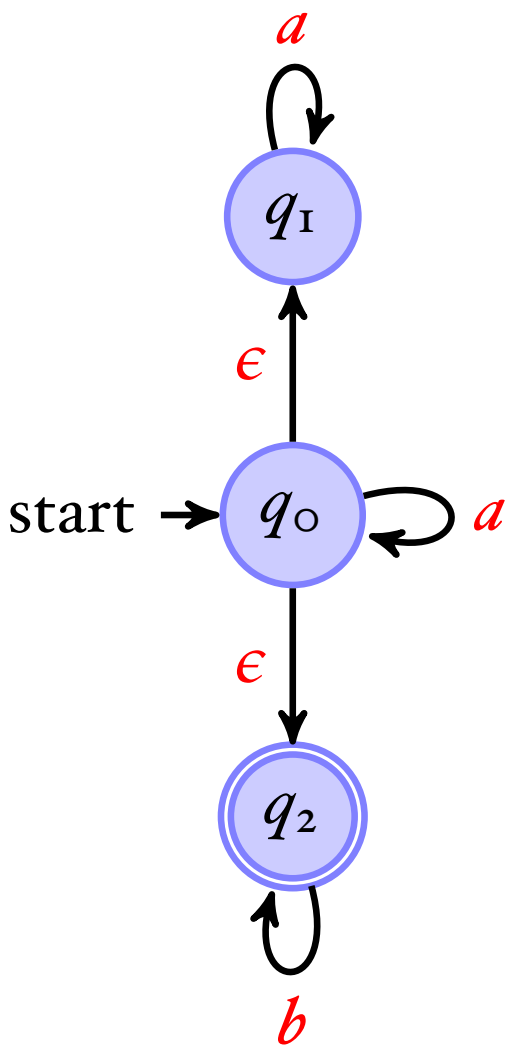
- a finite set of states
- some these states are the start states
- some states are accepting states, and
- there is transition relation

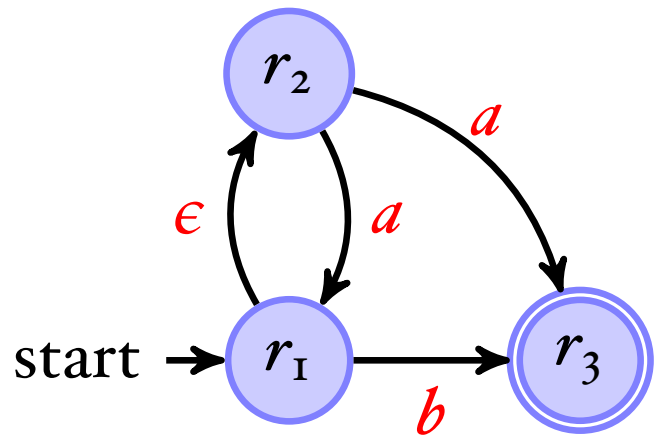
$$(Q_1, a) \to Q_2 \quad (Q_1, a) \to Q_3 \quad \ldots \quad (Q_1, a) \to \{Q_2, Q_3\}$$

An NFA Example



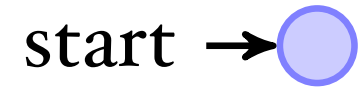
Two Epsilon NFA Examples



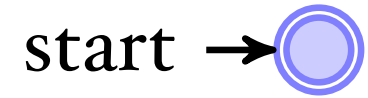


Rexp to NFA

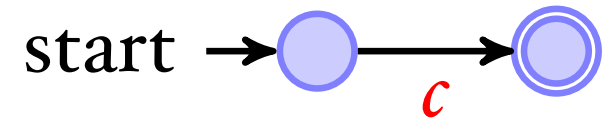
$\mathbf{0}$



$\mathbf{1}$

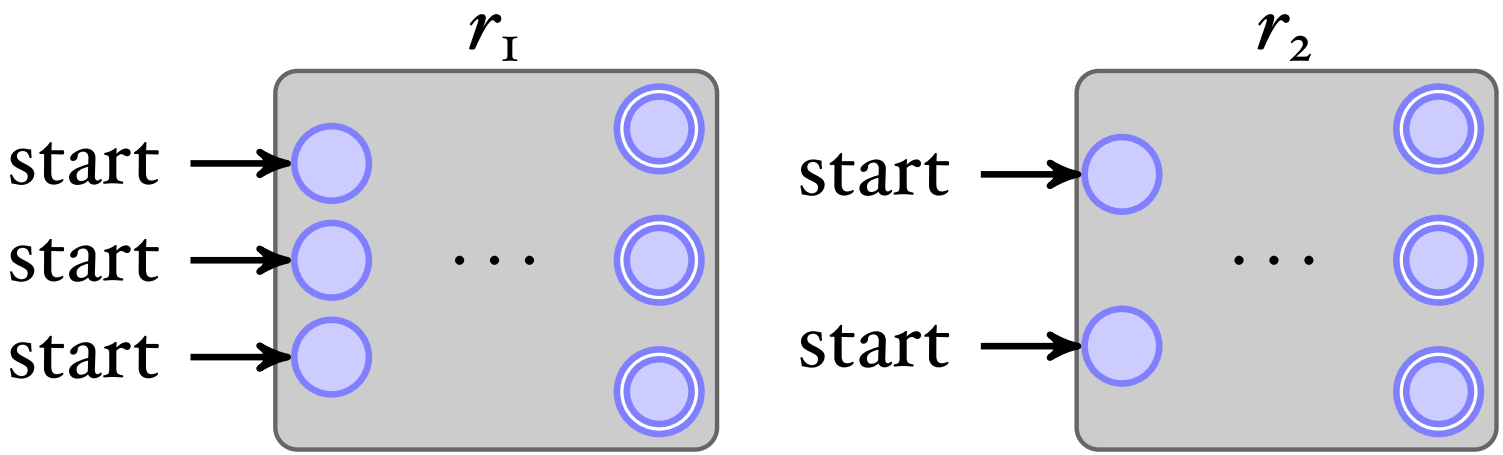


c



Case $r_1 \cdot r_2$

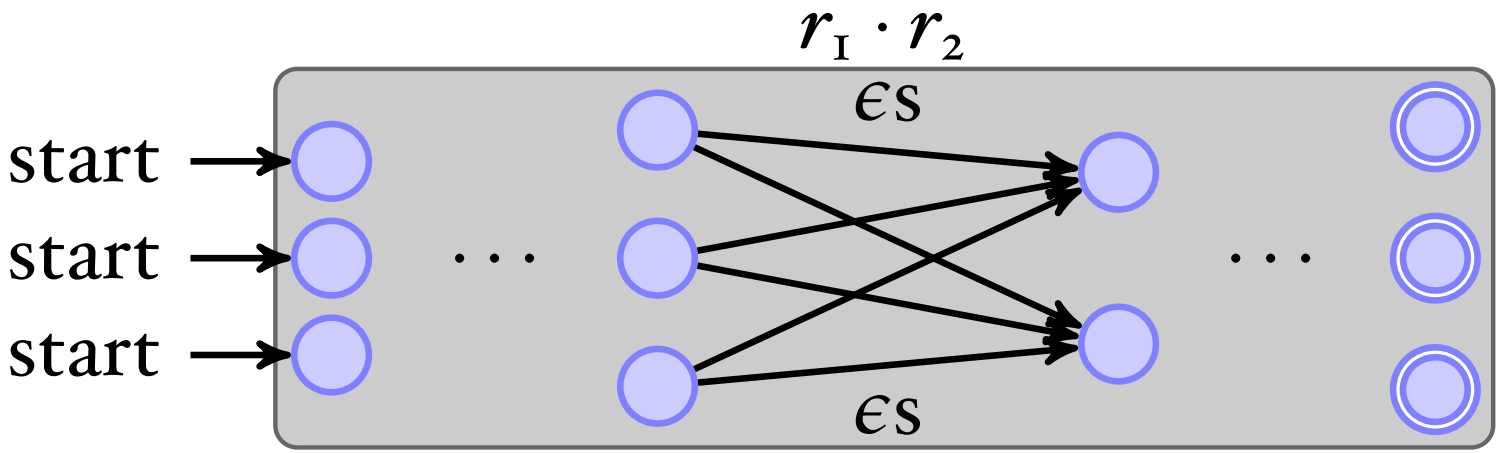
By recursion we are given two automata:



We need to (1) change the accepting nodes of the first automaton into non-accepting nodes, and (2) connect them via ϵ-transitions to the starting state of the second automaton.

Case $r_1 \cdot r_2$

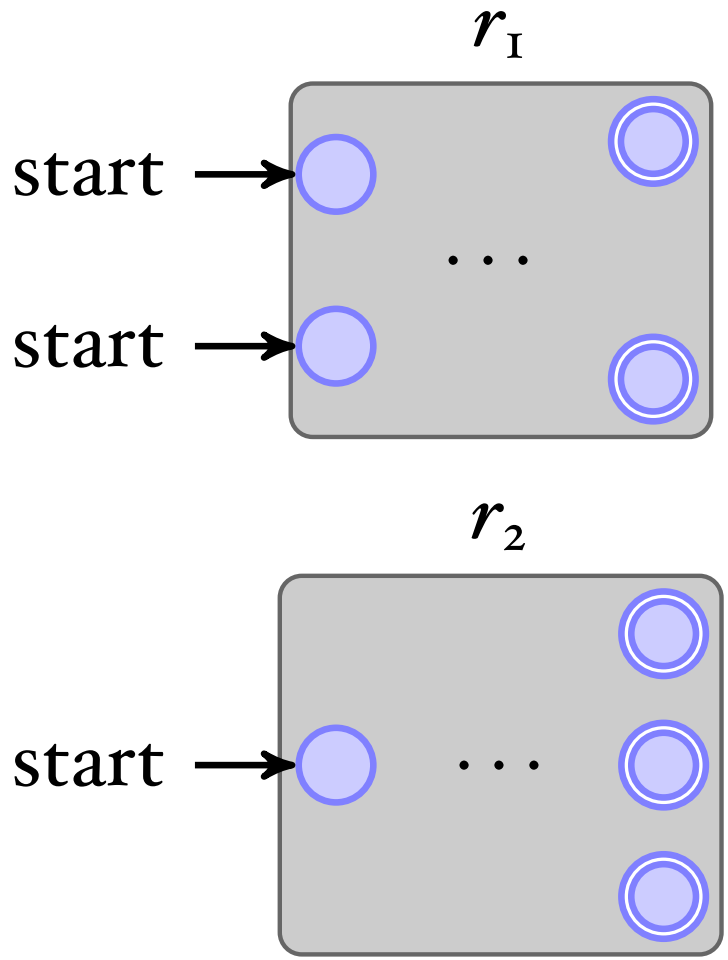
By recursion we are given two automata:



We need to (1) change the accepting nodes of the first automaton into non-accepting nodes, and (2) connect them via ϵ-transitions to the starting state of the second automaton.

Case $r_1 + r_2$

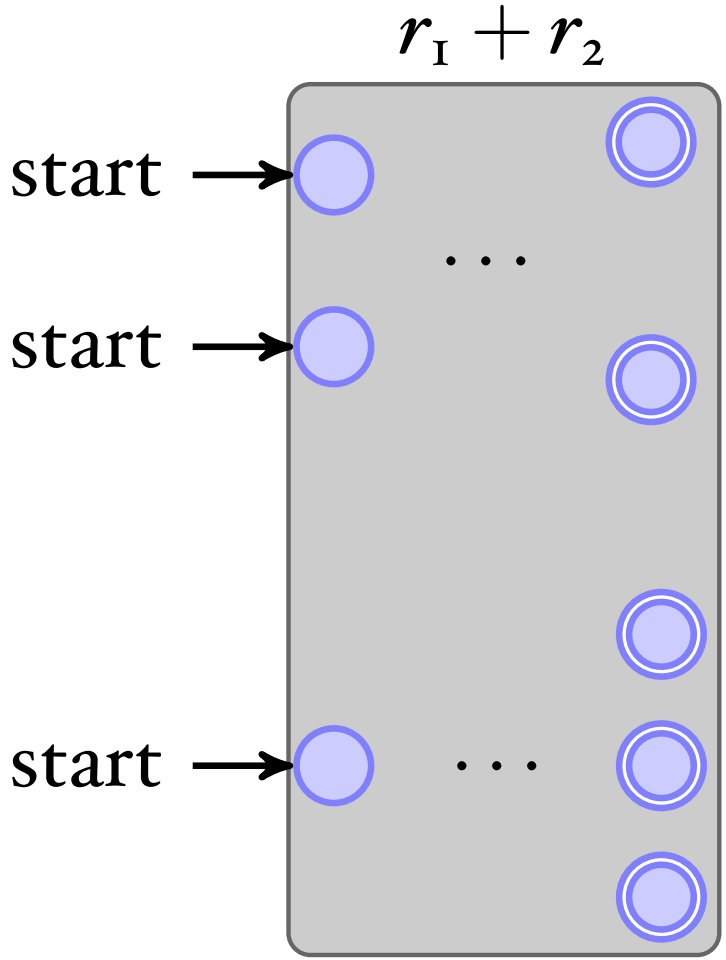
By recursion we are given two automata:



We can just put both automata together.

Case $r_1 + r_2$

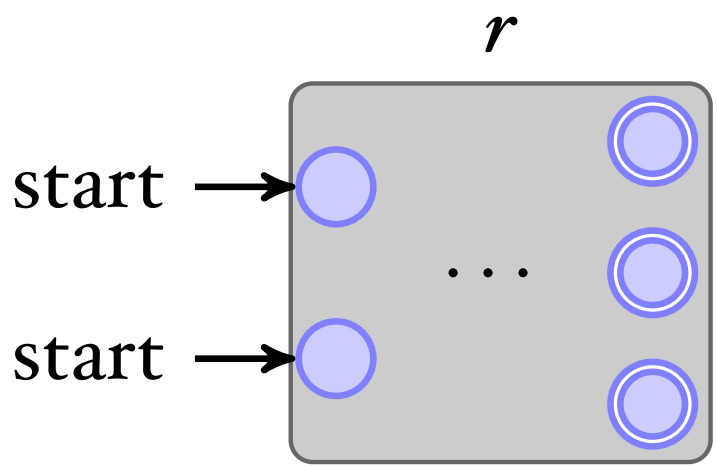
By recursion we are given two automata:



We can just put both automata together.

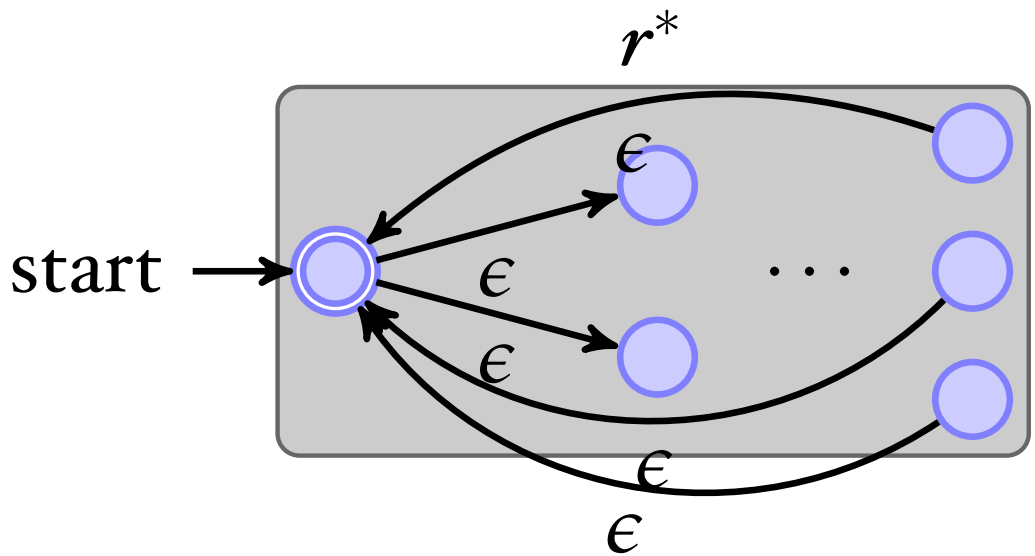
Case r^*

By recursion we are given an automaton for r:



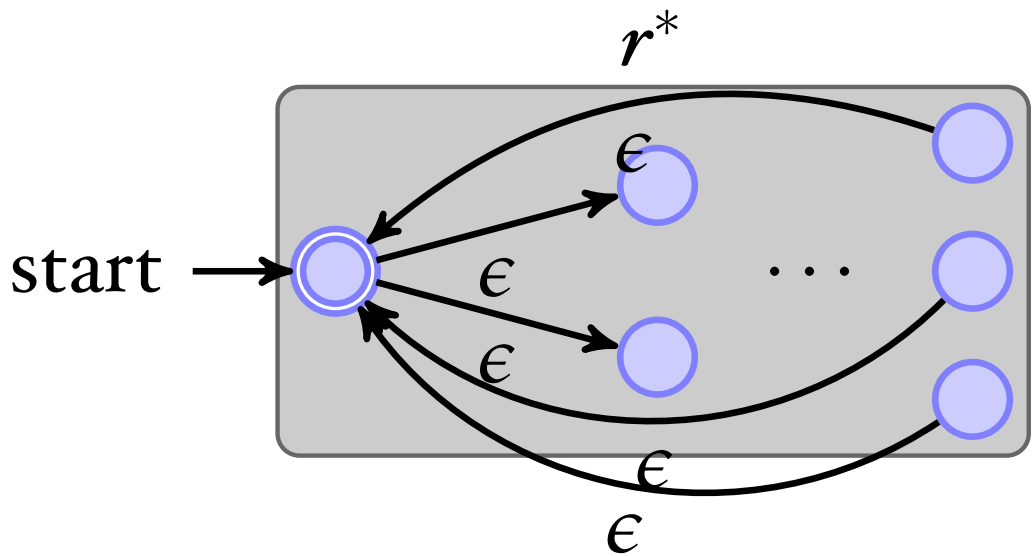
Case r^*

By recursion we are given an automaton for r:



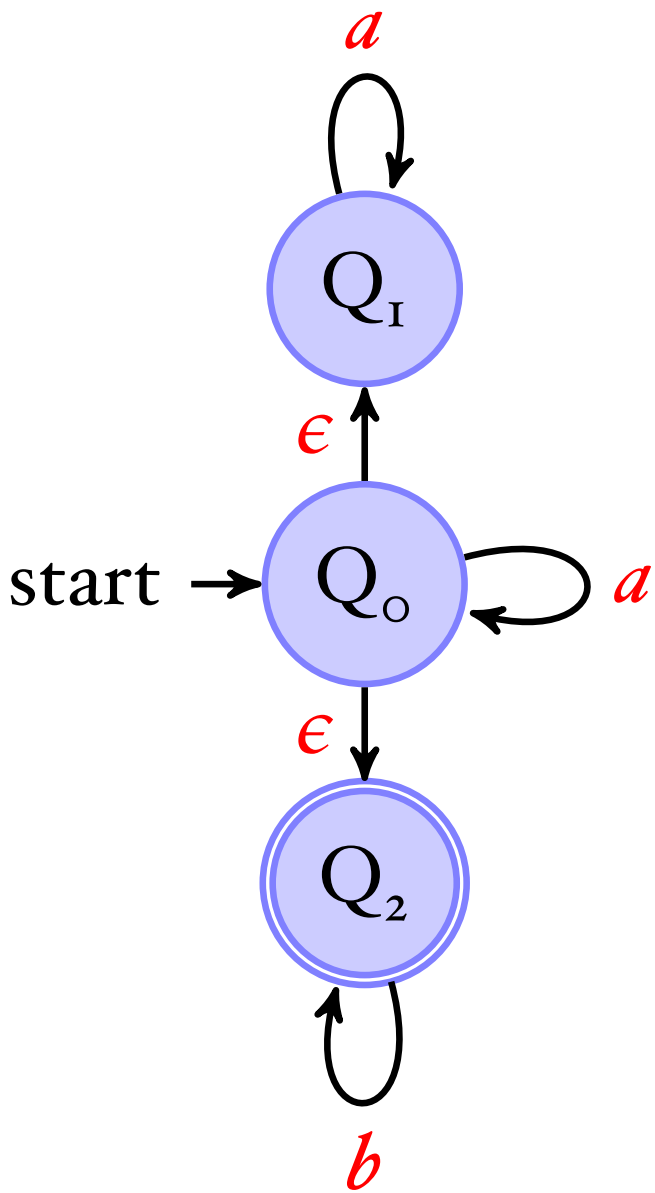
Case r^*

By recursion we are given an automaton for r:



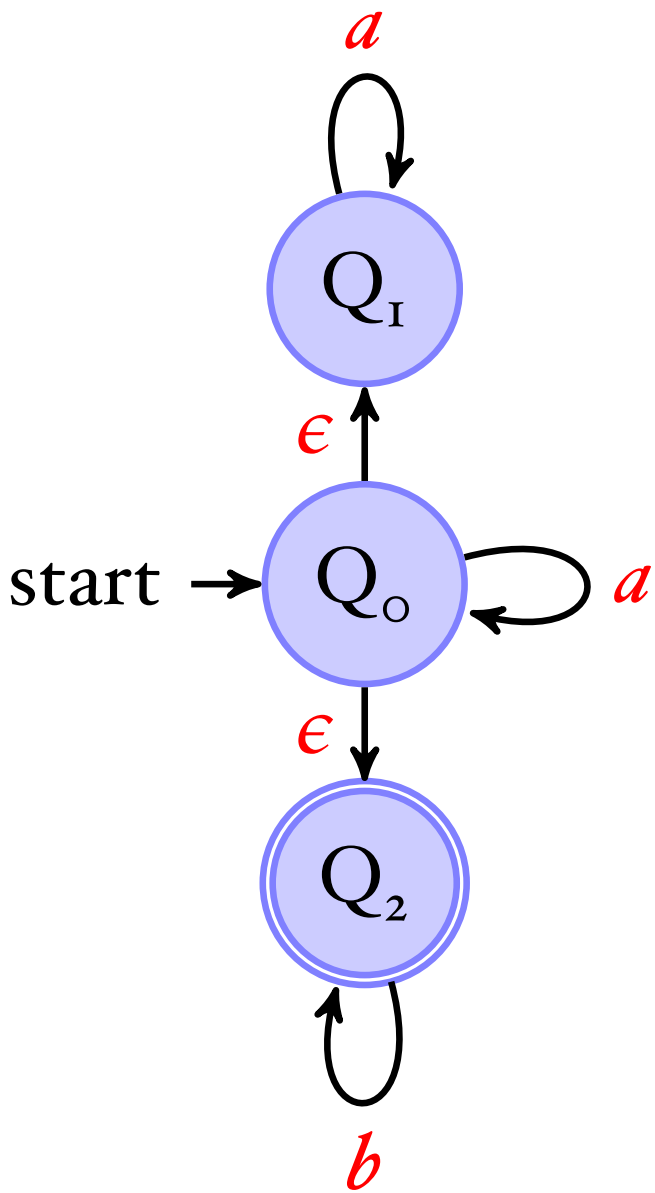
Why can't we just have an epsilon transition from the accepting states to the starting state?

Subset Construction



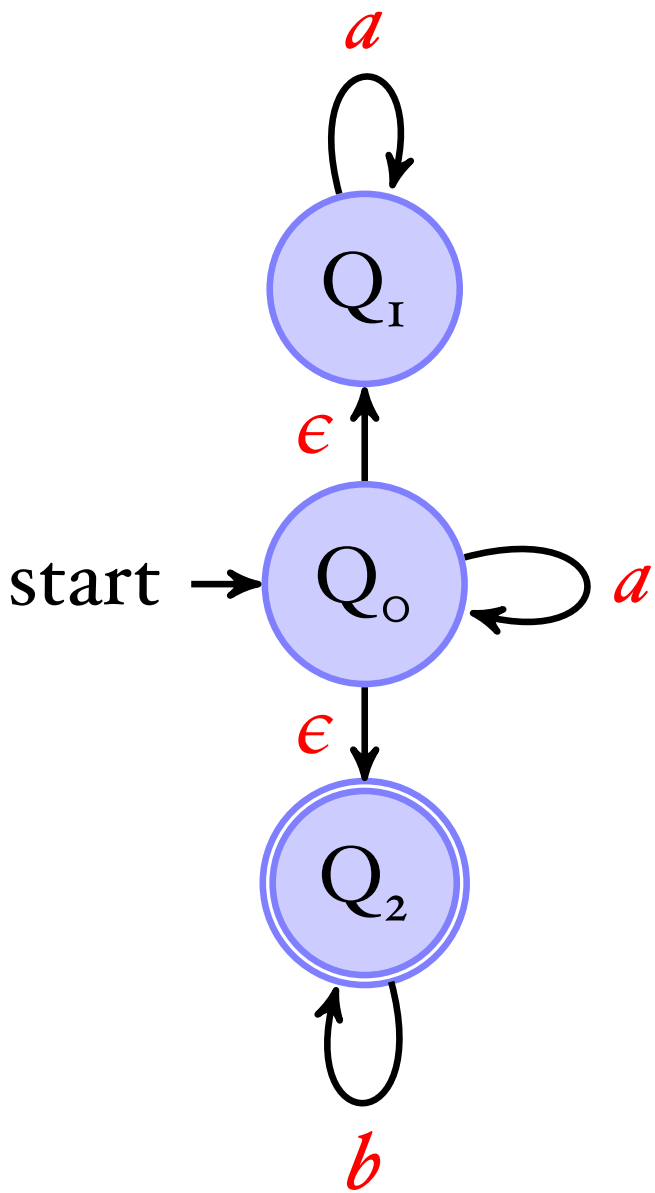
nodes	*a*	*b*
{}		
{0}		
{1}		
{2}		
{0, 1}		
{0, 2}		
{1, 2}		
{0, 1, 2}		

Subset Construction



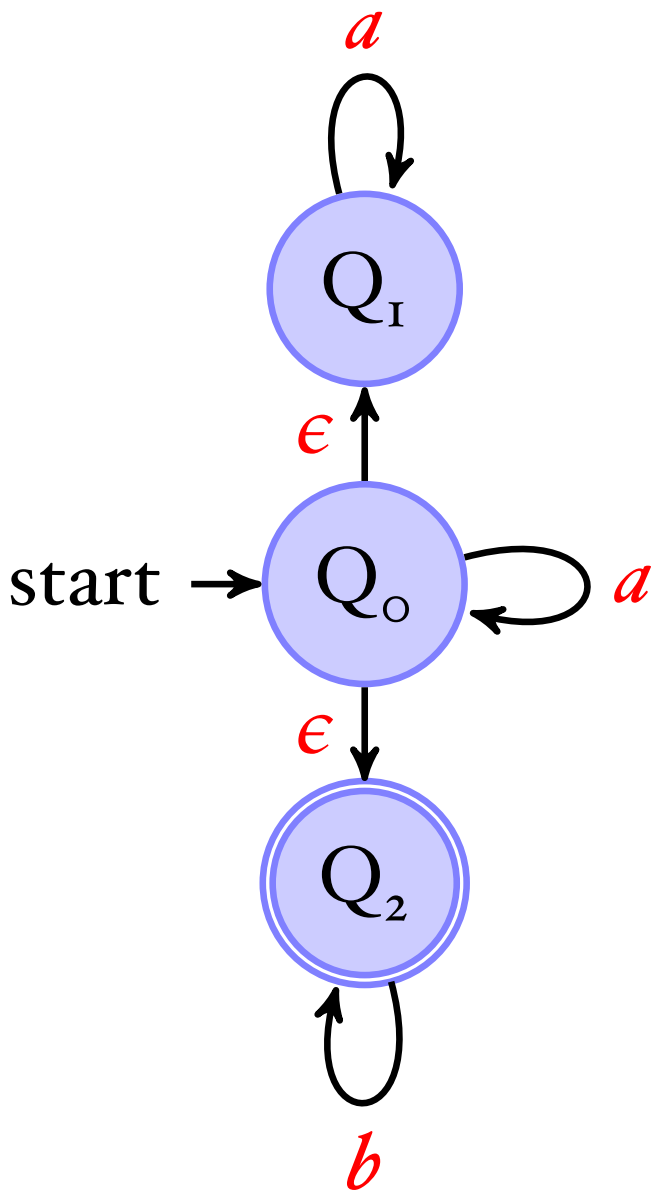
nodes	*a*	*b*
{}	{}	{}
{0}		
{1}		
{2}		
{0, 1}		
{0, 2}		
{1, 2}		
{0, 1, 2}		

Subset Construction



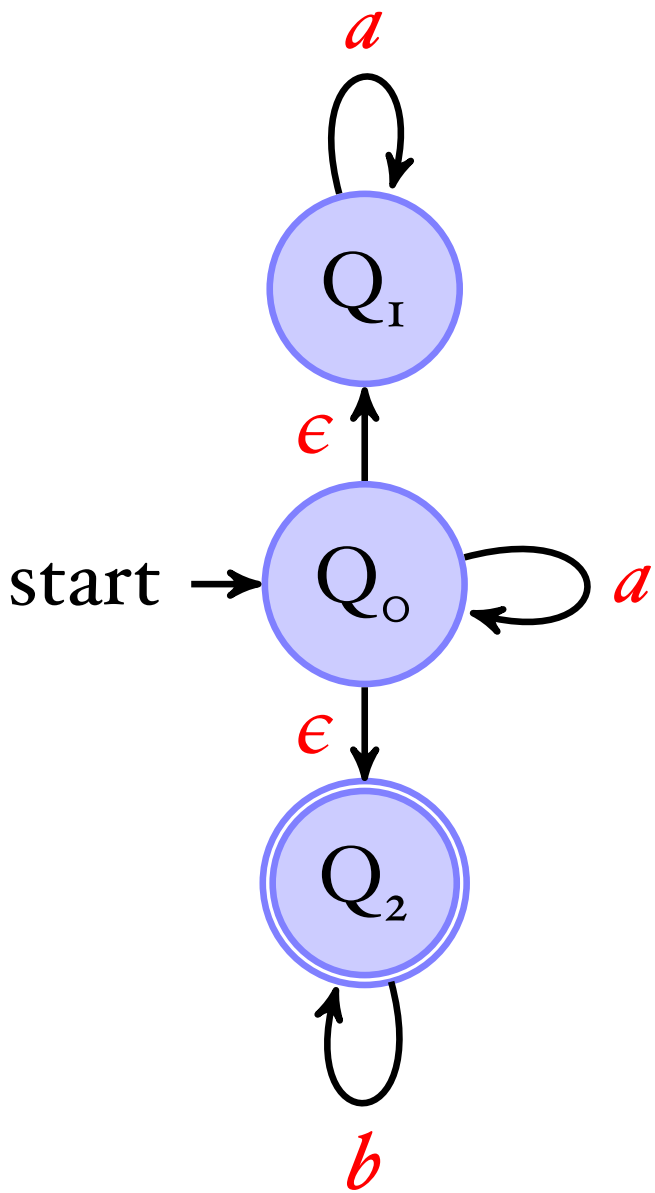
nodes	*a*	*b*
{}	{}	{}
{0}	{0, 1, 2}	{2}
{1}	{1}	{}
{2}	{}	{2}
{0, 1}		
{0, 2}		
{1, 2}		
{0, 1, 2}		

Subset Construction



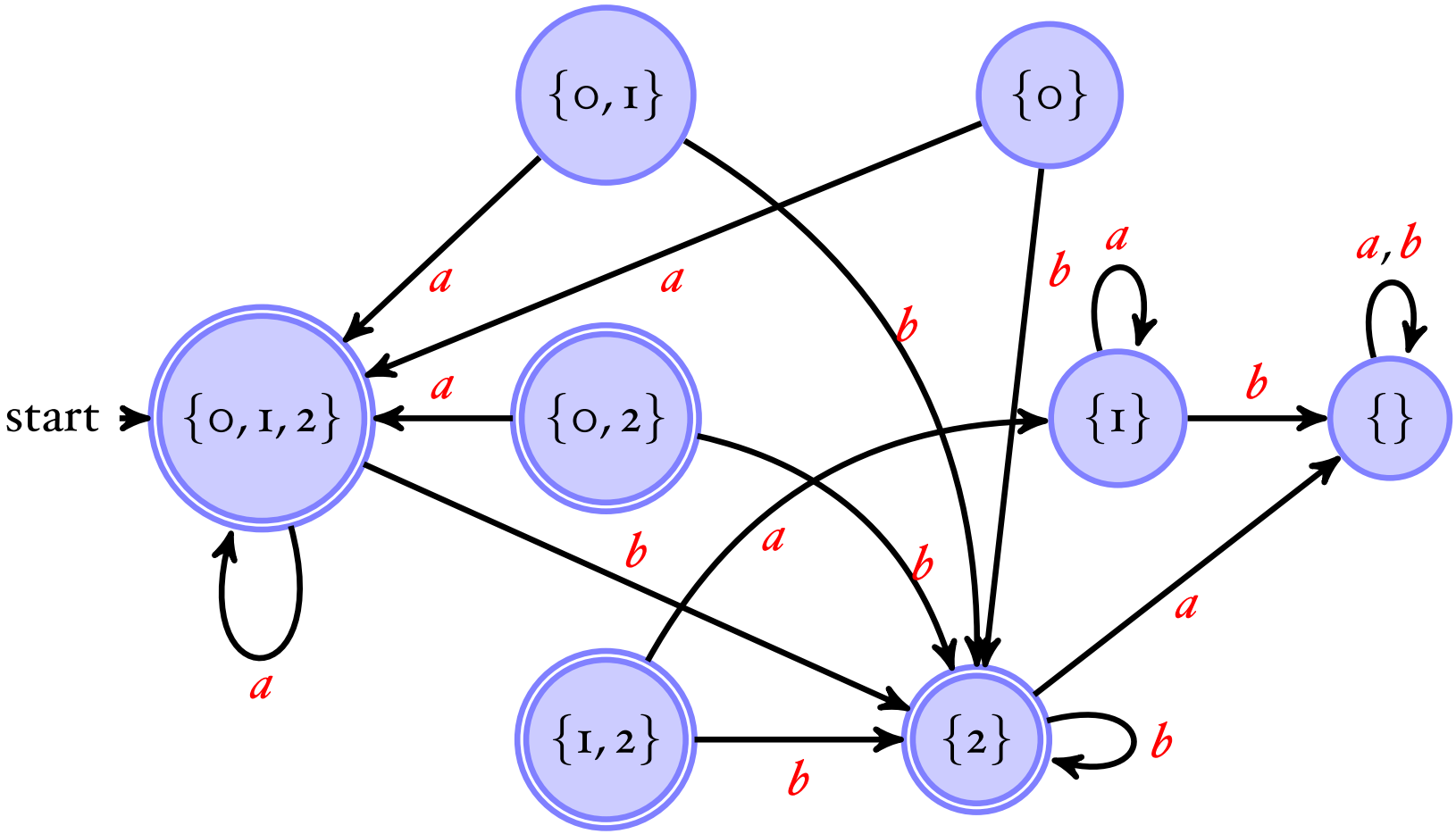
nodes	*a*	*b*
{}	{}	{}
{0}	{0, 1, 2}	{2}
{1}	{1}	{}
{2}	{}	{2}
{0, 1}	{0, 1, 2}	{2}
{0, 2}	{0, 1, 2}	{2}
{1, 2}	{1}	{2}
{0, 1, 2}	{0, 1, 2}	{2}

Subset Construction



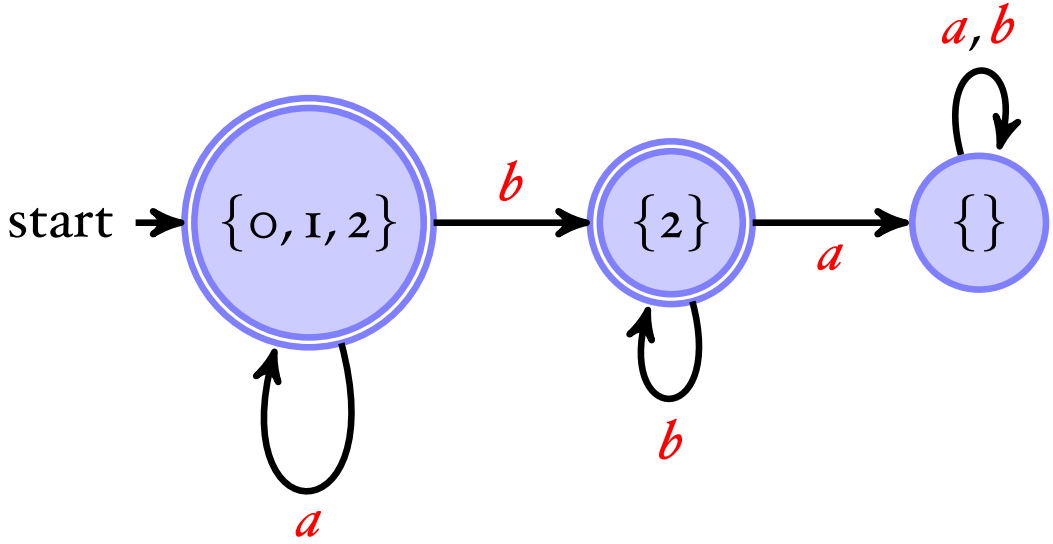
nodes		*a*	*b*
{}		{}	{}
{0}		{0, 1, 2}	{2}
{1}		{1}	{}
{2}	*	{}	{2}
{0, 1}		{0, 1, 2}	{2}
{0, 2}	*	{0, 1, 2}	{2}
{1, 2}	*	{1}	{2}
s: {0, 1, 2}	*	{0, 1, 2}	{2}

The Result

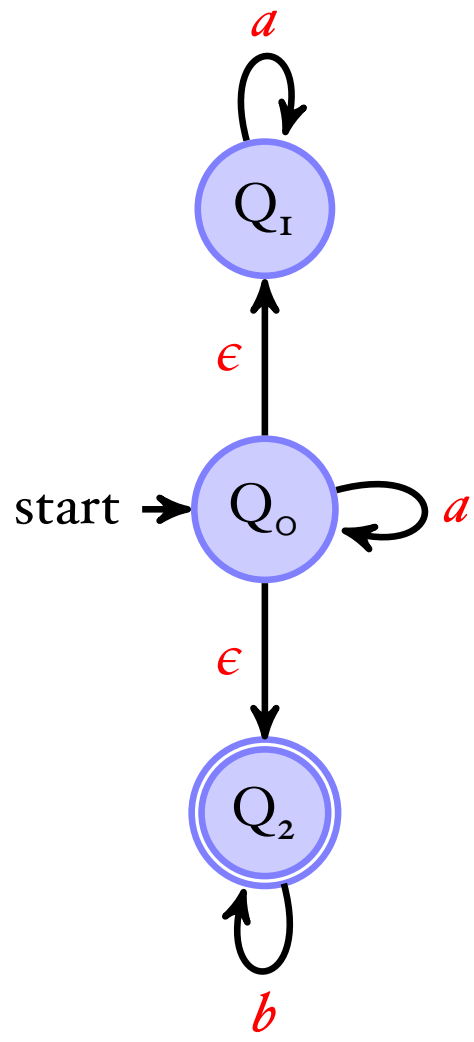


Removing Dead States

DFA:



(original) NFA:

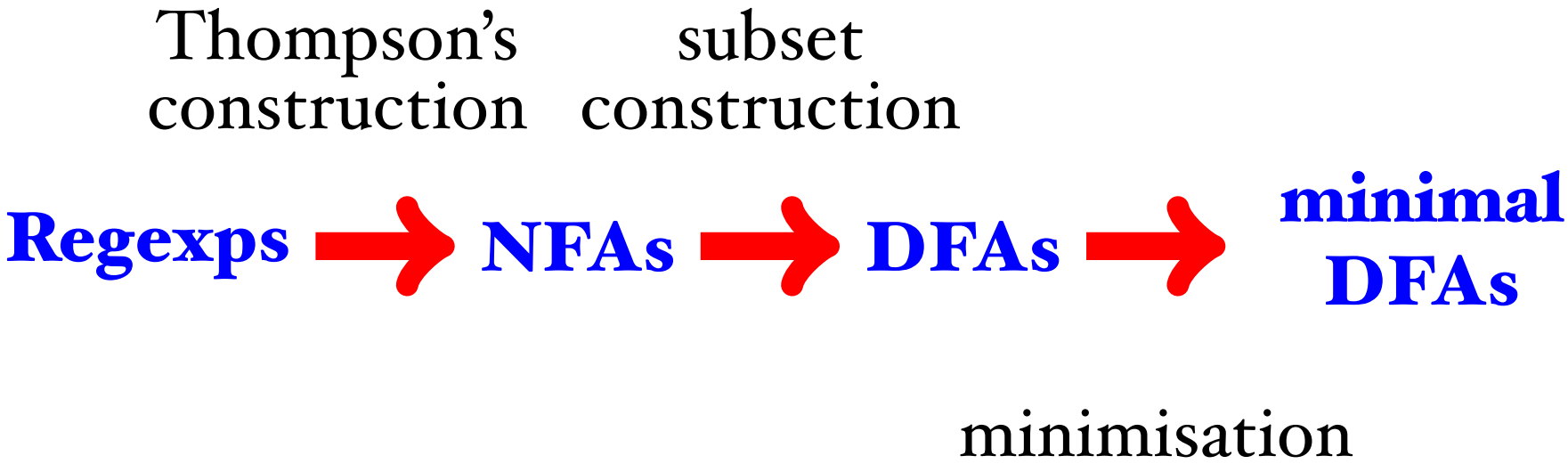


Regexps and Automata

Thompson's construction

subset construction

Regexps and Automata

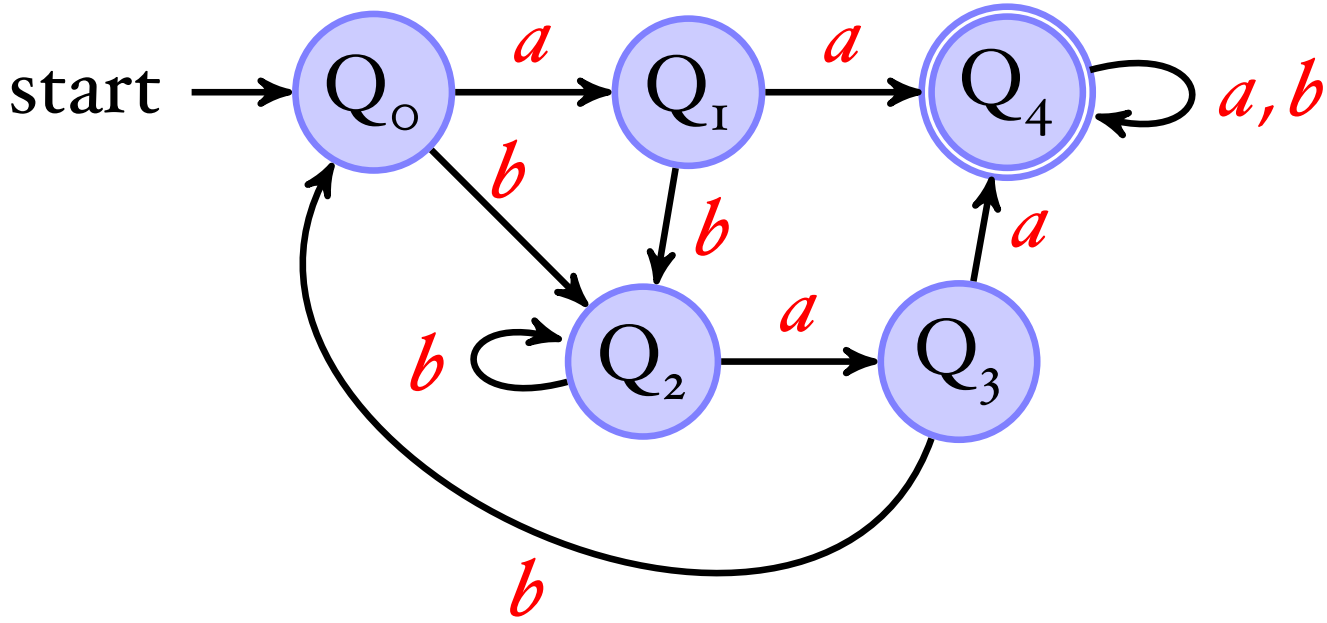


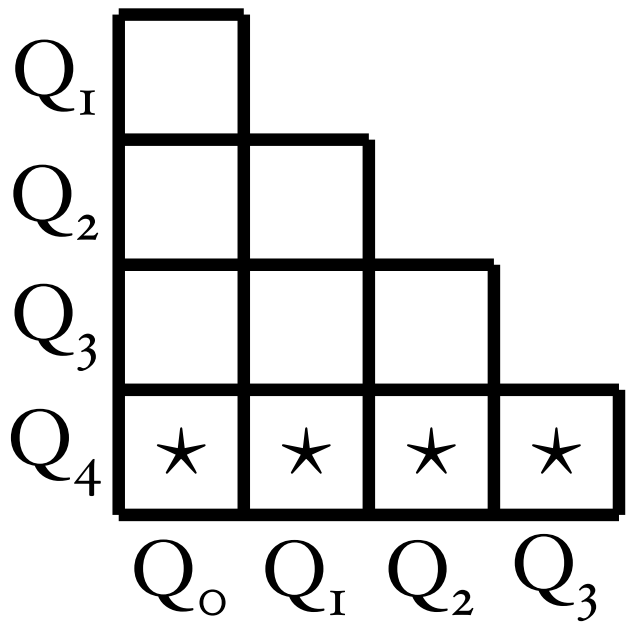
DFA Minimisation

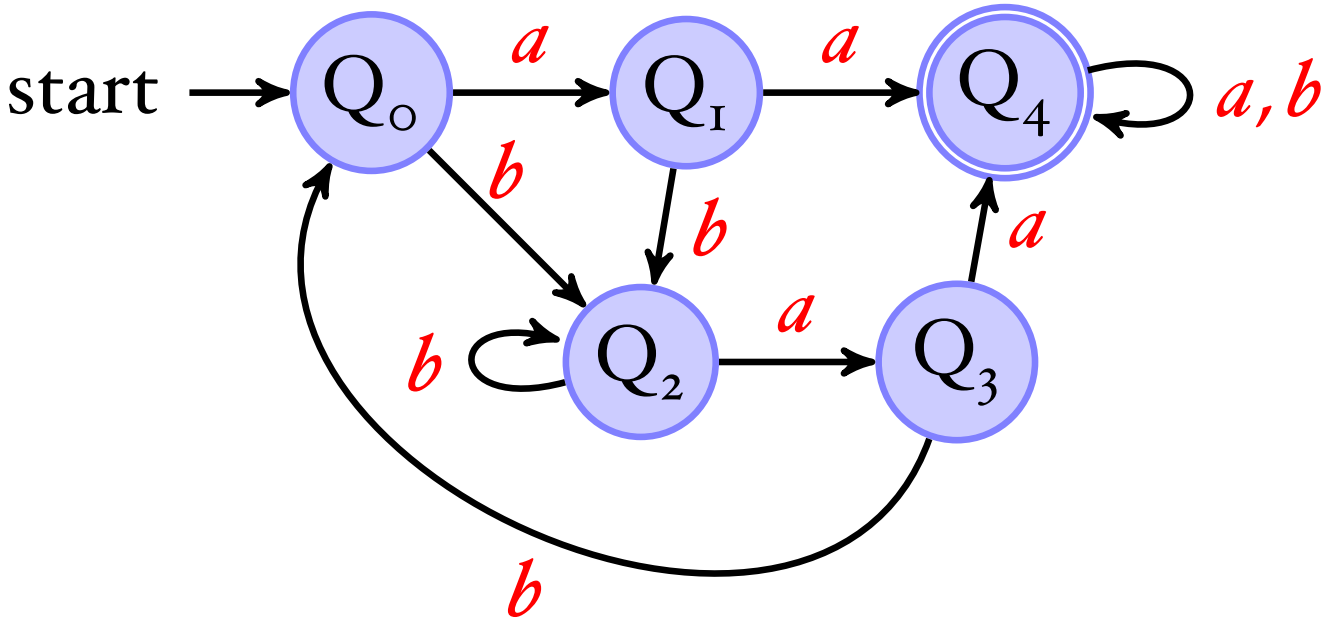
1. Take all pairs (q,p) with $q \neq p$
2. Mark all pairs that accepting and non-accepting states
3. For all unmarked pairs (q,p) and all characters c test whether

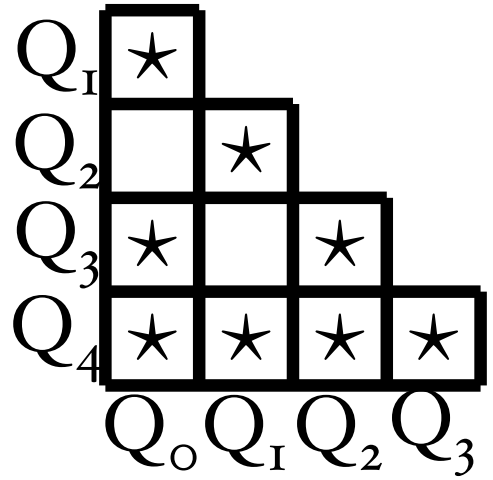
$$(\delta(q,c),\delta(p,c))$$

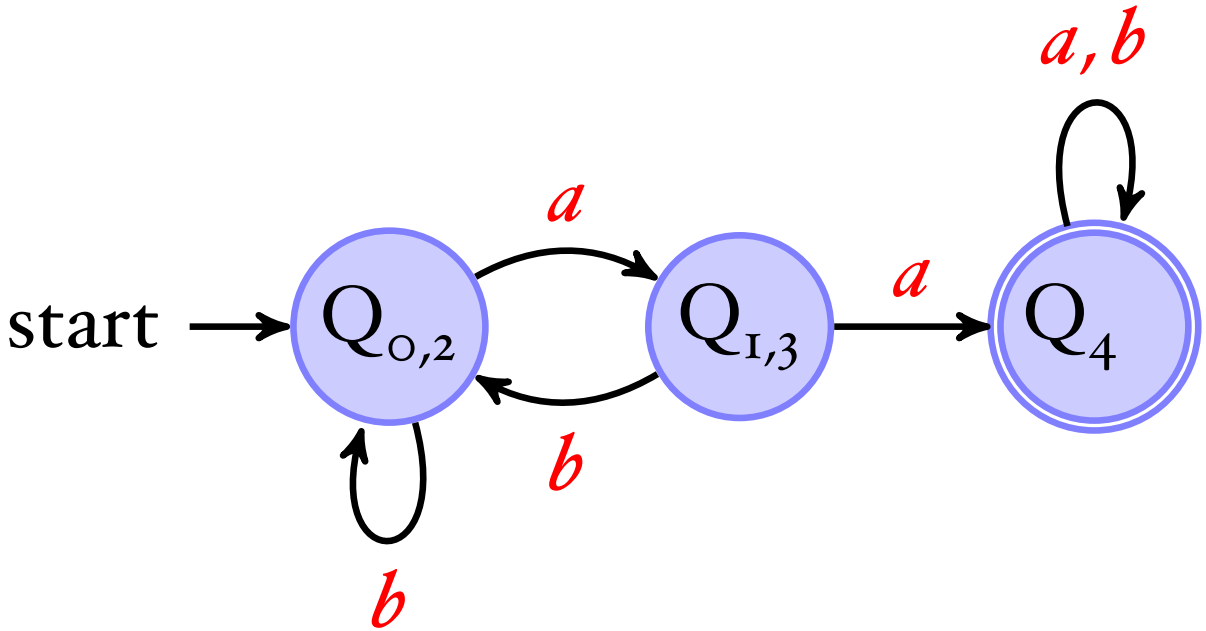
   are marked. If yes in at least one case, then also mark (q,p).
4. Repeat last step until no change.
5. All unmarked pairs can be merged.


start
Q_0
Q_1
Q_4
Q_2
Q_3
a
a
a, b
b
b
a
b
a
b

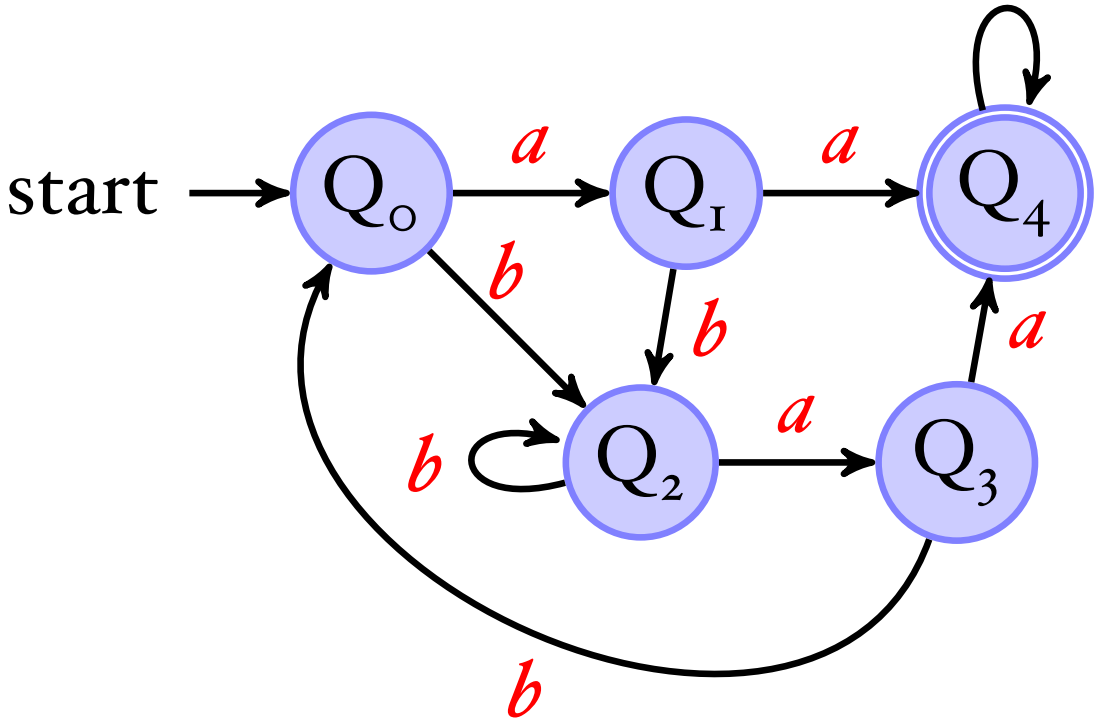

Q_1
Q_2
Q_3
Q_4
Q_0 Q_1 Q_2 Q_3


start
Q_0
Q_1
Q_4
Q_2
Q_3
a
a
a, b
b
b
a
b
a
b

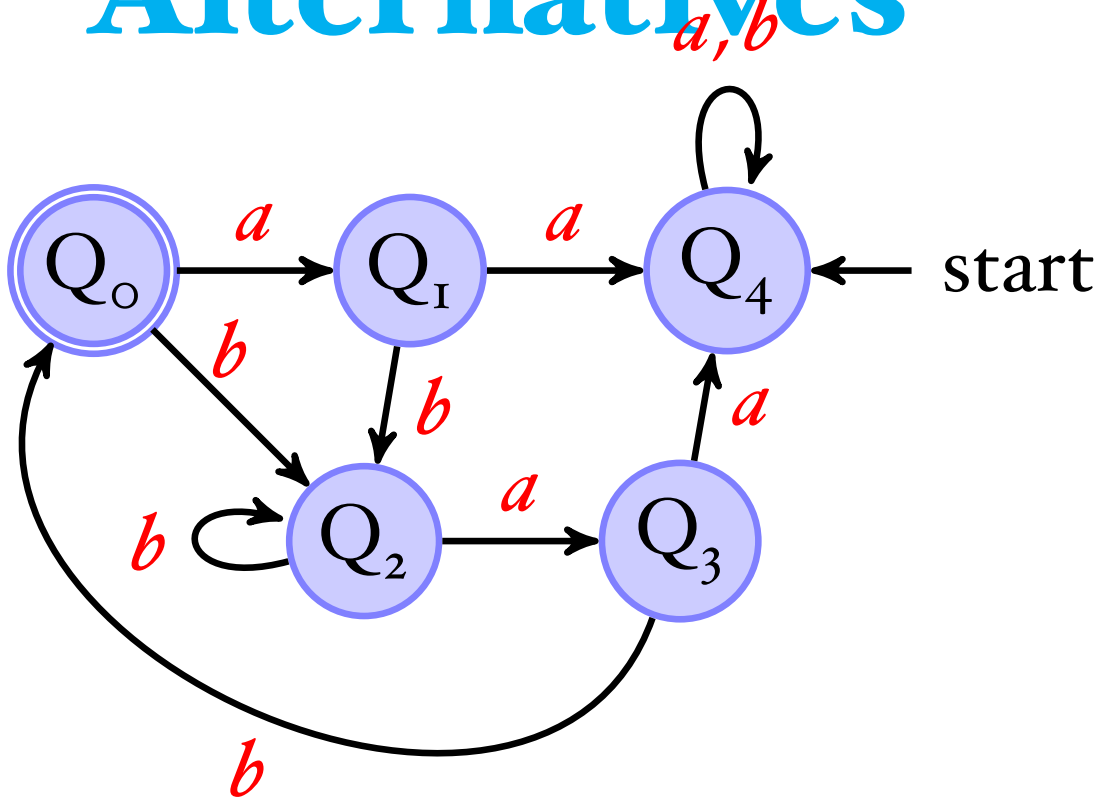

Q_1
Q_2
Q_3
Q_4
Q_0 Q_1 Q_2 Q_3


start
$Q_{0,2}$
$Q_{1,3}$
Q_4
a
a
a, b
b
b

Alternatives a, b

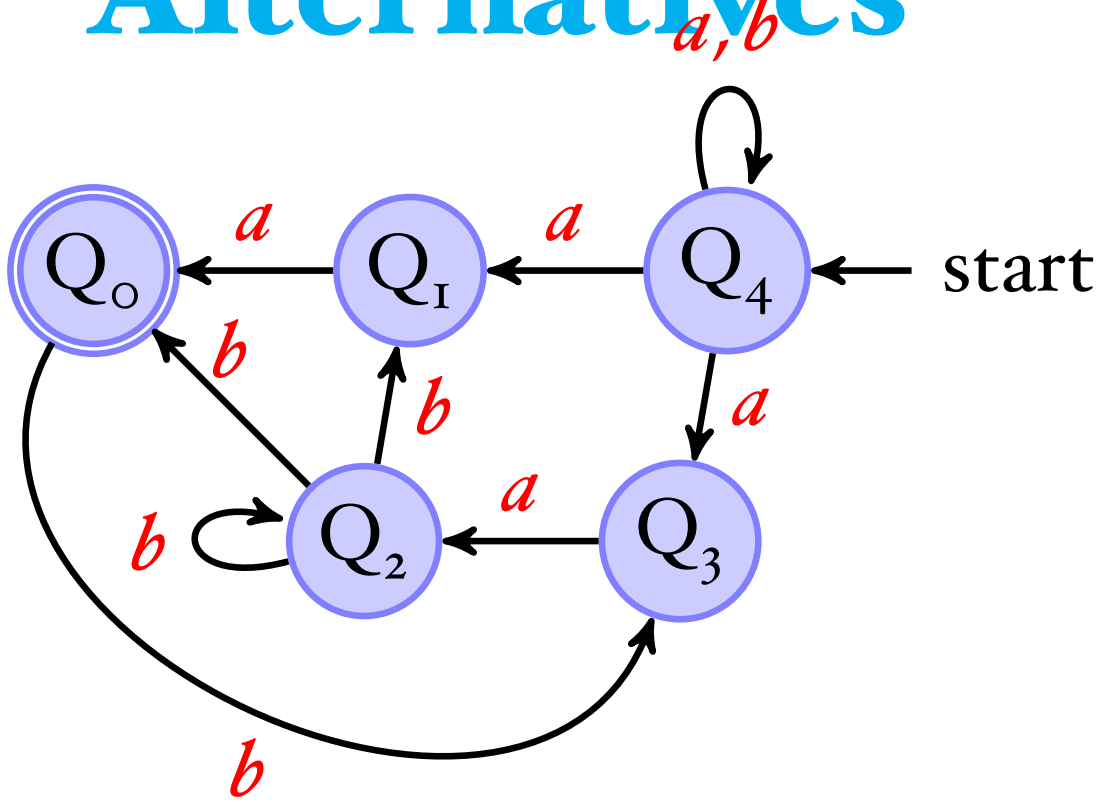


Alternatives



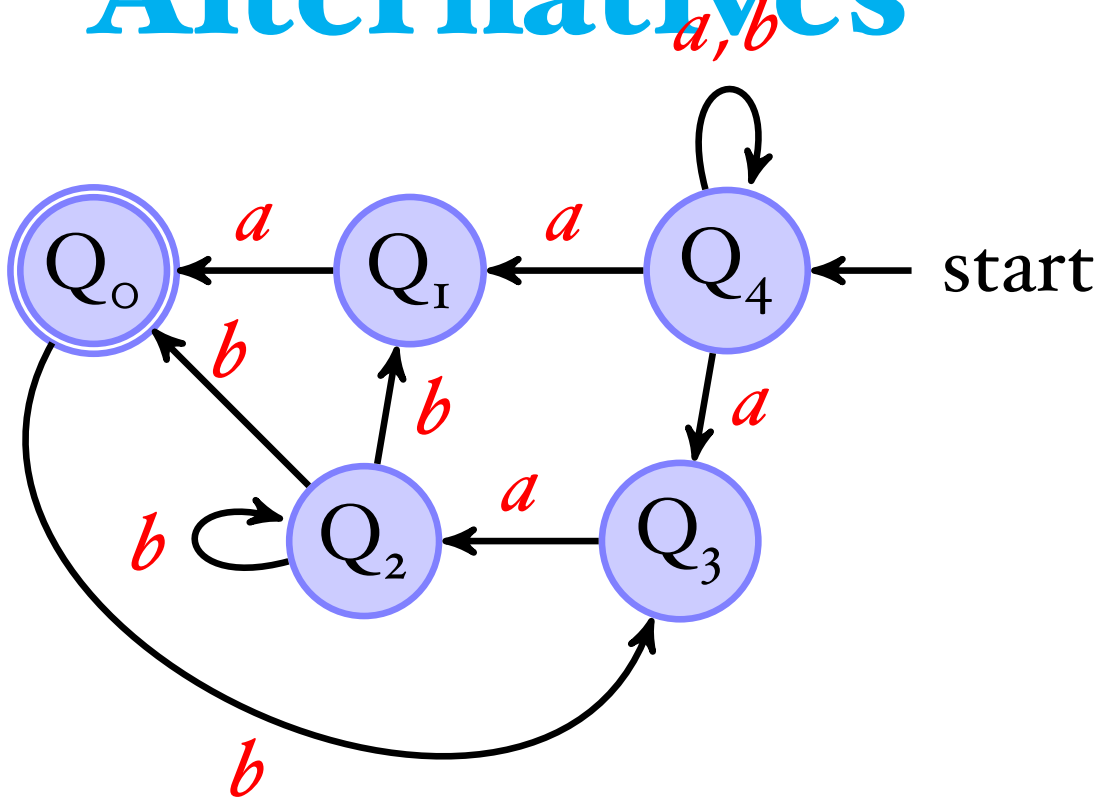
- exchange initial / accepting states

Alternatives



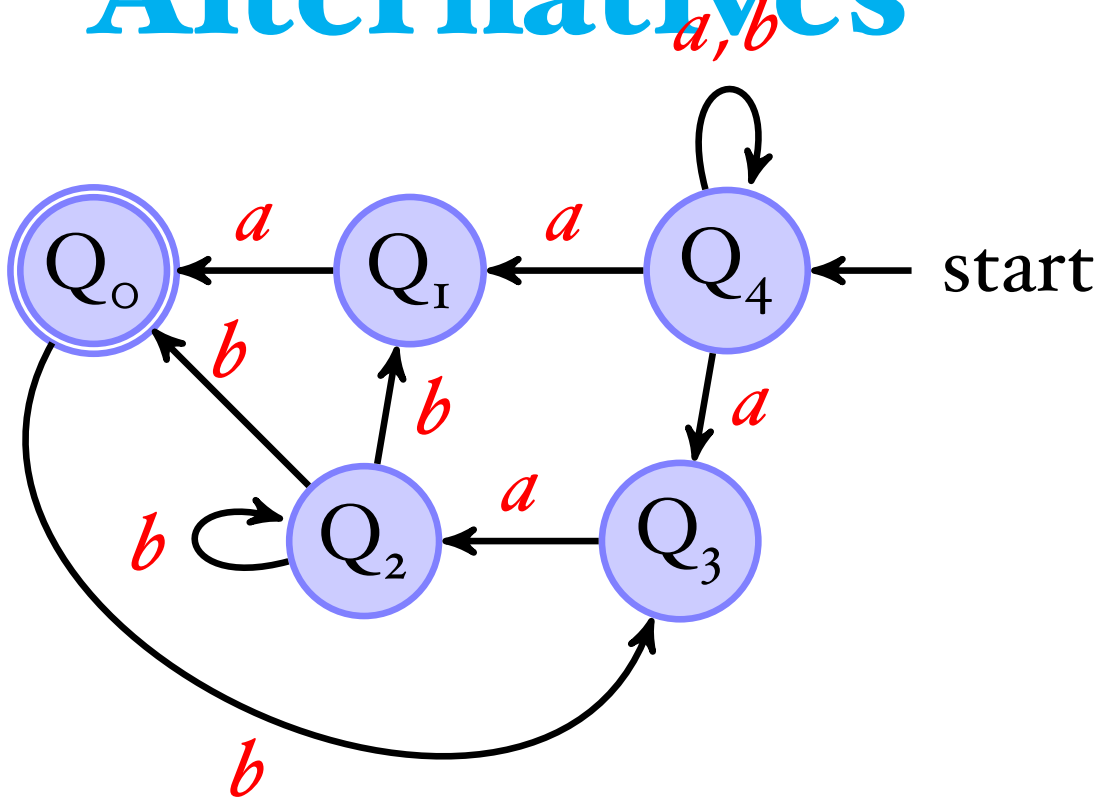
- exchange initial / accepting states
- reverse all edges

Alternatives



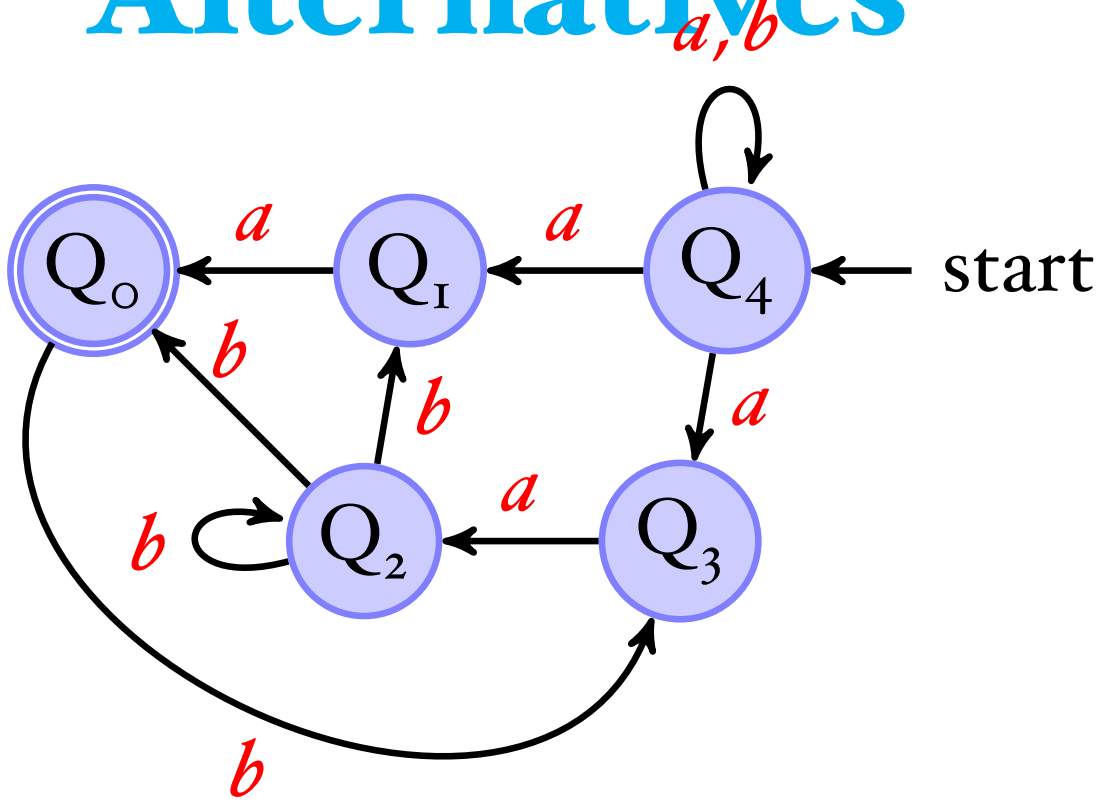
- exchange initial / accepting states
- reverse all edges
- subset construction $\Rightarrow$ DFA

Alternatives



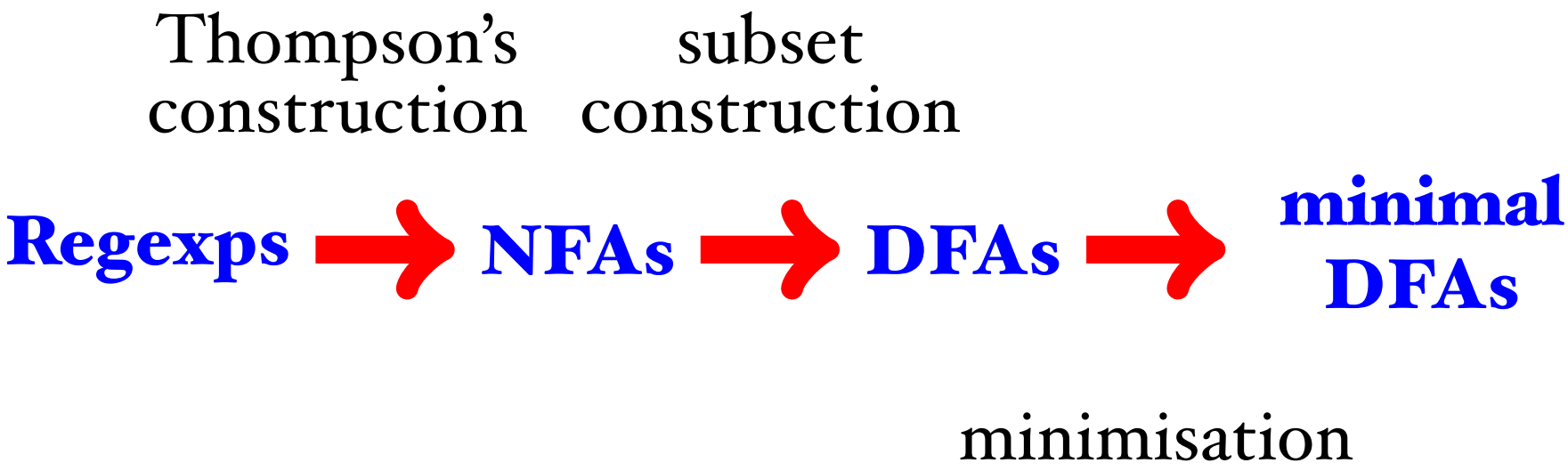
- exchange initial / accepting states
- reverse all edges
- subset construction $\Rightarrow$ DFA
- remove dead states

Alternatives

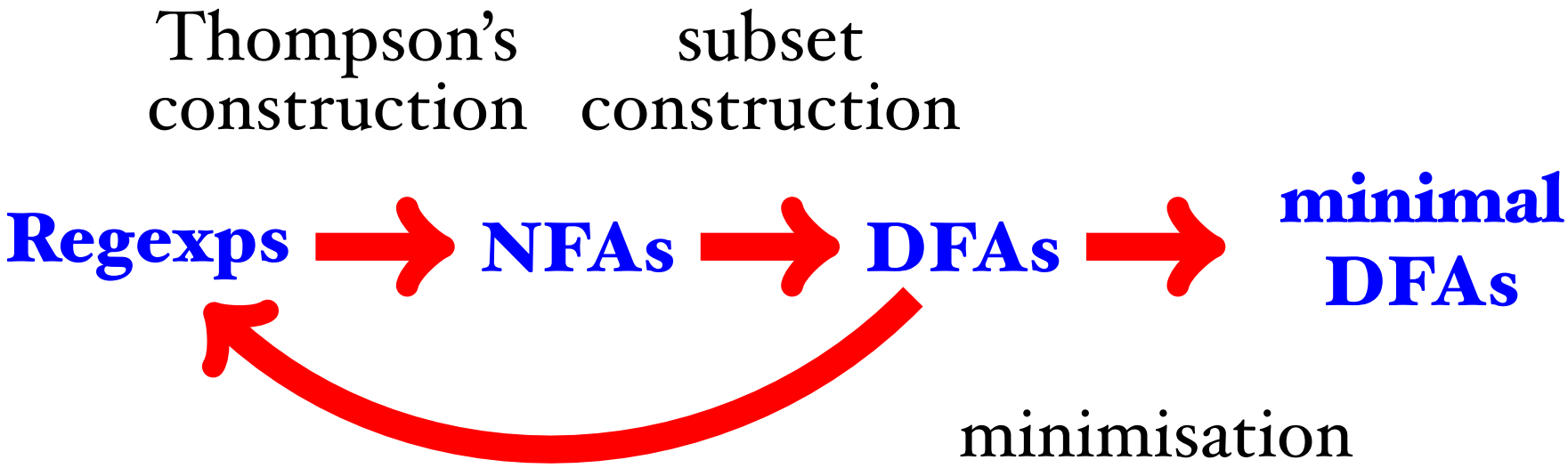


- exchange initial / accepting states
- reverse all edges
- subset construction ⇒ DFA
- remove dead states
- repeat once more ⇒ minimal DFA

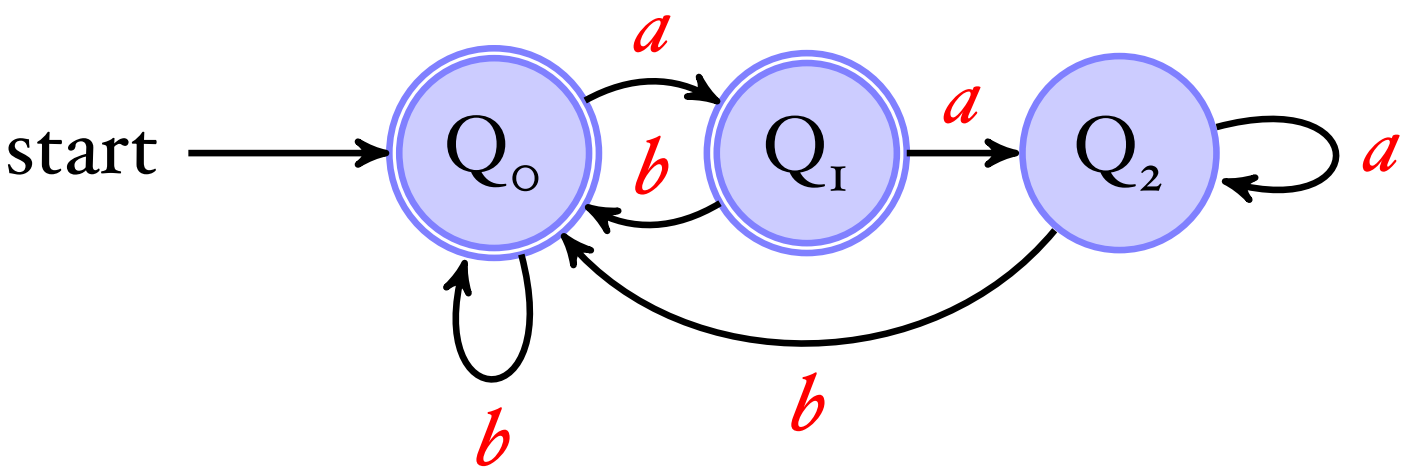
Regexps and Automata

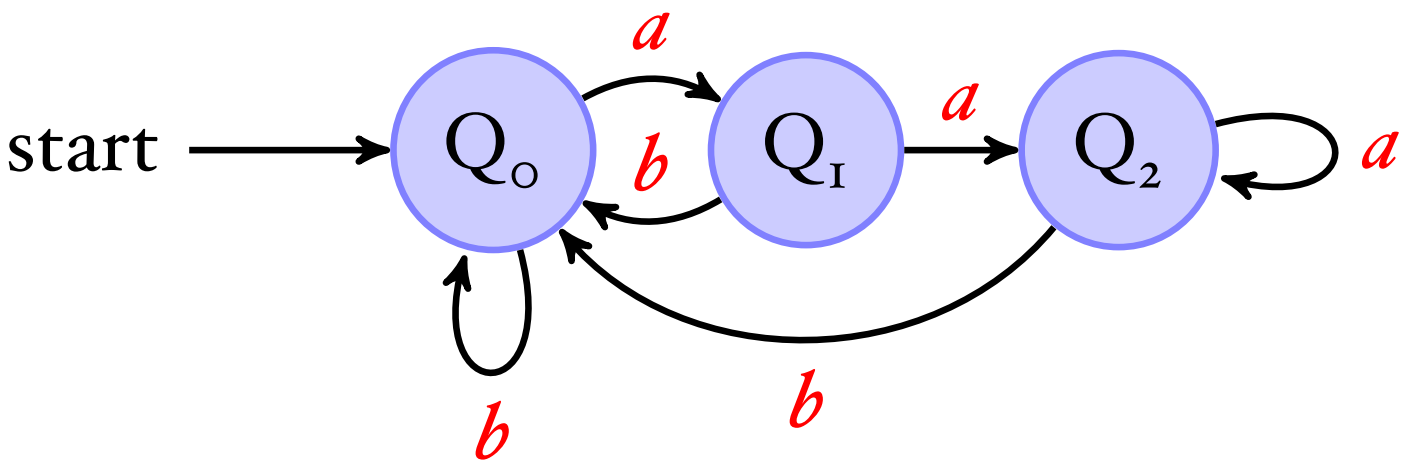


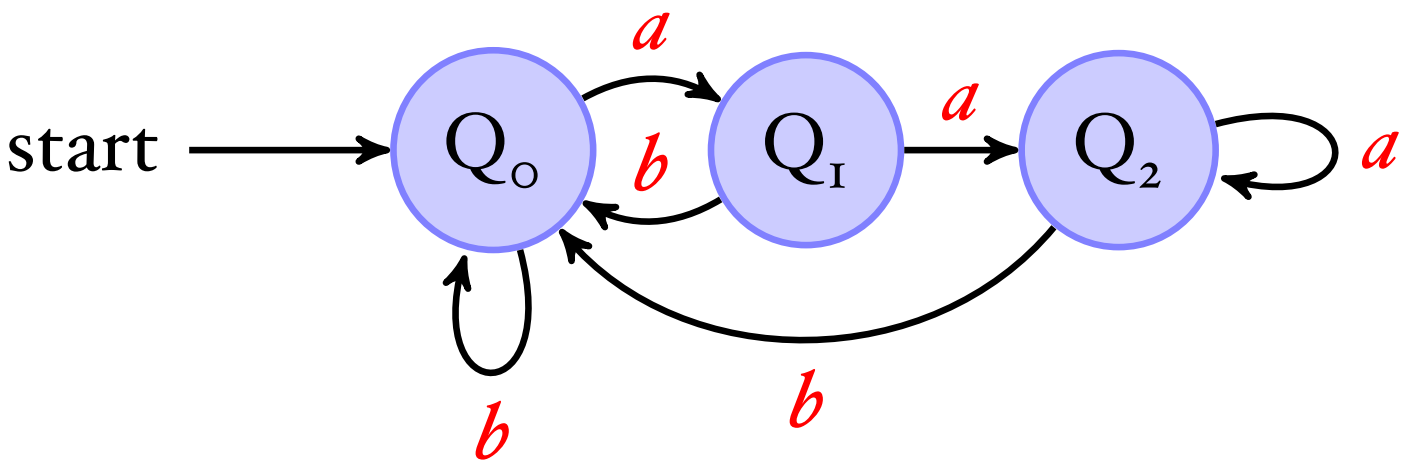
Regexps and Automata



DFA to Rexp

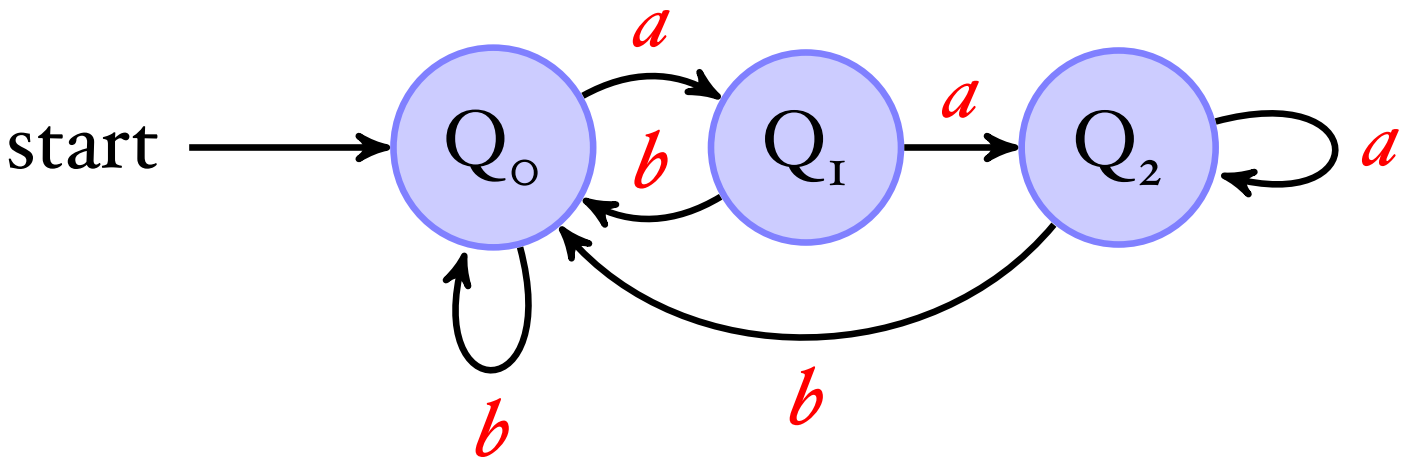


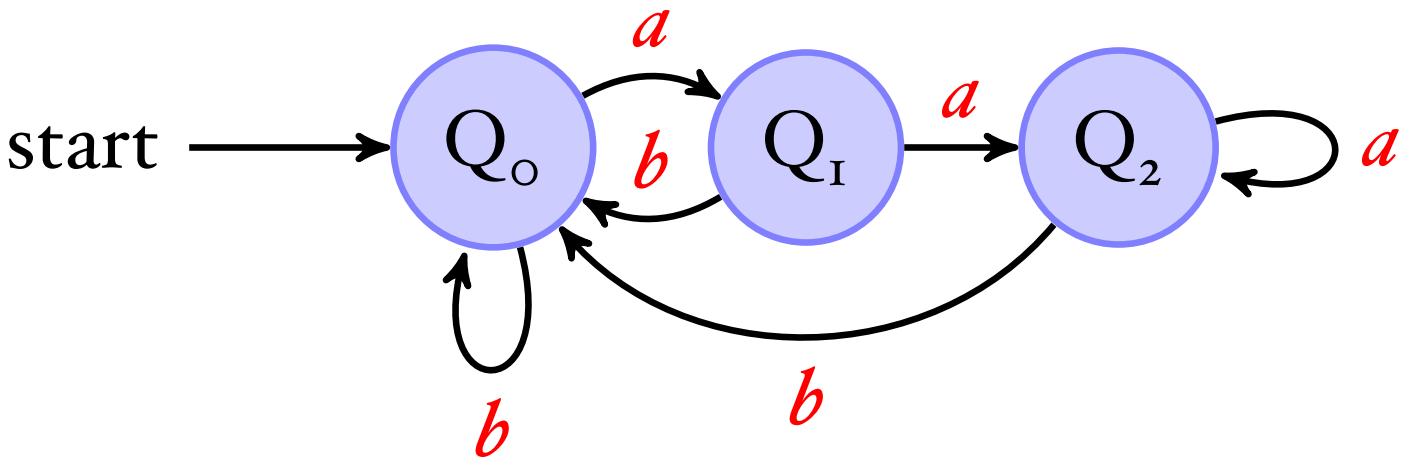

start
Q_0
Q_1
Q_2
a
b
a
a
b
b



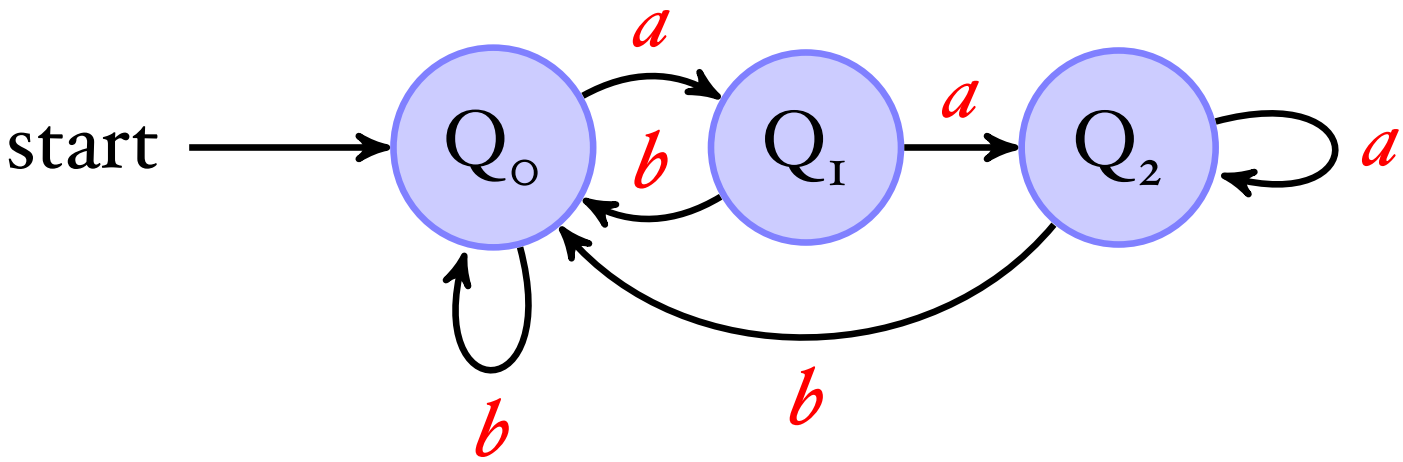
You know how to solve since school days, no?

$$
\begin{aligned}
Q_0 &= 2Q_0 + 3Q_1 + 4Q_2 \\
Q_1 &= 2Q_0 + 3Q_1 + 1Q_2 \\
Q_2 &= 1Q_0 + 5Q_1 + 2Q_2
\end{aligned}
$$


start
Q_0
Q_1
Q_2
a
b
a
a
b
b



$$
\begin{aligned}
Q_0 &= \mathbf{1} + Q_0\, b + Q_1\, b + Q_2\, b \\
Q_1 &= Q_0\, a \\
Q_2 &= Q_1\, a + Q_2\, a
\end{aligned}
$$

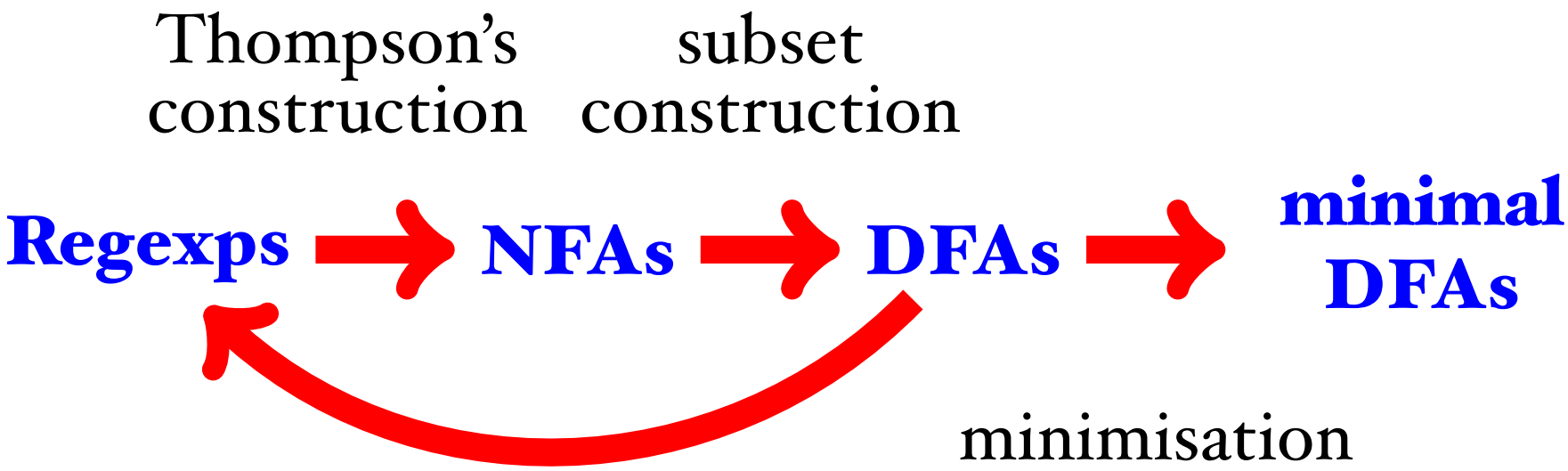


$$
\begin{aligned}
Q_0 &= \mathbf{1} + Q_0 b + Q_1 b + Q_2 b \\
Q_1 &= Q_0 a \\
Q_2 &= Q_1 a + Q_2 a
\end{aligned}
$$

Arden's Lemma:

$$\text{If } q = qr + s \text{ then } q = sr^*$$

Regexps and Automata



Regular Languages (3)

A language is **regular** iff there exists a regular expression that recognises all its strings.

or **equivalently**

A language is **regular** iff there exists a deterministic finite automaton that recognises all its strings.

Regular Languages (3)

A language is **regular** iff there exists a regular expression that recognises all its strings.

or **equivalently**

A language is **regular** iff there exists a deterministic finite automaton that recognises all its strings.

Why is every finite set of strings a regular language?

Given the function

$$\begin{array}{rcl} rev(\mathbf{0}) & \stackrel{\text{def}}{=} & \mathbf{0} \\ rev(\mathbf{1}) & \stackrel{\text{def}}{=} & \mathbf{1} \\ rev(c) & \stackrel{\text{def}}{=} & c \\ rev(r_1 + r_2) & \stackrel{\text{def}}{=} & rev(r_1) + rev(r_2) \\ rev(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & rev(r_2) \cdot rev(r_1) \\ rev(r^*) & \stackrel{\text{def}}{=} & rev(r)^* \end{array}$$

and the set

$$Rev\, A \stackrel{\text{def}}{=} \{s^{-1} \mid s \in A\}$$

prove whether

$$L(rev(r)) = Rev(L(r))$$