

# Compilers and Formal Languages (3)

Email: christian.urban at kcl.ac.uk  
Office Hours: Thursdays 12 – 14  
Location: N7.07 (North Wing, Bush House)  
Slides & Progs: KEATS (also homework 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

# 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)

*matches*  $s$   $r$  if and only if  $s \in L(r)$

by Janusz Brzozowski (1964)

# The Derivative of a Rexp

$$\text{der } c \text{ (0)} \stackrel{\text{def}}{=} \mathbf{0}$$

$$\text{der } c \text{ (1)} \stackrel{\text{def}}{=} \mathbf{0}$$

$$\text{der } c \text{ (} d \text{)} \stackrel{\text{def}}{=} \text{if } c = d \text{ then } \mathbf{1} \text{ else } \mathbf{0}$$

$$\text{der } c \text{ (} r_1 + r_2 \text{)} \stackrel{\text{def}}{=} \text{der } c \text{ } r_1 + \text{der } c \text{ } r_2$$

$$\text{der } c \text{ (} r_1 \cdot r_2 \text{)} \stackrel{\text{def}}{=} \text{if nullable}(r_1) \\ \text{then } (\text{der } c \text{ } r_1) \cdot r_2 + \text{der } c \text{ } r_2 \\ \text{else } (\text{der } c \text{ } r_1) \cdot r_2$$

$$\text{der } c \text{ (} r^* \text{)} \stackrel{\text{def}}{=} (\text{der } c \text{ } r) \cdot (r^*)$$

$$\text{ders } [] \text{ } r \stackrel{\text{def}}{=} r$$

$$\text{ders } (c :: s) \text{ } r \stackrel{\text{def}}{=} \text{ders } s \text{ (der } c \text{ } r)$$

# Example

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

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

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  $r \stackrel{\text{def}}{=} ((a \cdot b) + b)^*$ , you can simplify as follows

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



# Proofs about Rexp

- $P$  holds for  $0$ ,  $1$  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  $r$ .

We proved

*nullable*( $r$ ) if and only if  $\epsilon \in L(r)$

by induction on the regular expression  $r$ .

We proved

*nullable*( $r$ ) if and only if  $\epsilon \in L(r)$

by induction on the regular expression  $r$ .

**Any Questions?**

# 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  $s$

# Correctness Proof for our Matcher

- We started from

$$s \in L(r)$$

$$\Leftrightarrow \square \in \text{Ders } s (L(r))$$

# Correctness Proof for our Matcher

- We started from

$$s \in L(r)$$

$$\Leftrightarrow [] \in \text{Ders } s (L(r))$$

- **if** we can show  $\text{Ders } s (L(r)) = L(\text{ders } s r)$  we have

$$\Leftrightarrow [] \in L(\text{ders } s r)$$

$$\Leftrightarrow \text{nullable}(\text{ders } s r)$$

$$\stackrel{\text{def}}{=} \text{matches } s r$$

We need to prove

$$L(\text{der } c r) = \text{Derc } (L(r))$$

also by induction on the regular expression  $r$ .

# (Basic) Regular Expressions

$r ::=$	$\mathbf{0}$	nothing
	$\mathbf{1}$	empty string / "" / []
	$c$	character
	$r_1 \cdot r_2$	sequence
	$r_1 + r_2$	alternative / choice
	$r^*$	star (zero or more)

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



# 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*!

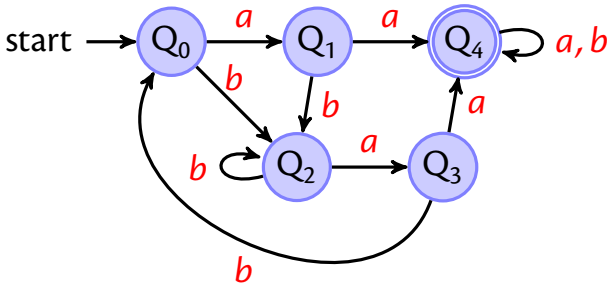
# Automata

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

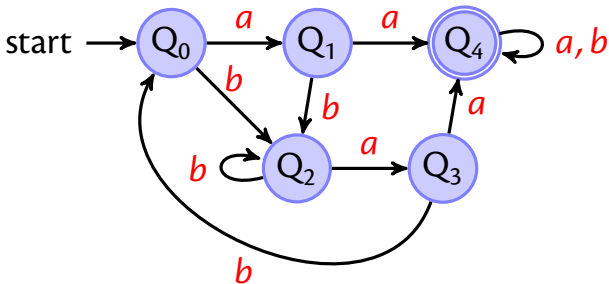
- an alphabet  $\Sigma$
- a set of states  $Q_s$
- one of these states is the start state  $Q_0$
- some states are accepting states  $F$ , and
- there is transition function  $\delta$

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

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



- 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  $\delta$  is the function

$$\begin{array}{lll}
 (Q_0, a) \rightarrow Q_1 & (Q_1, a) \rightarrow Q_4 & (Q_4, a) \rightarrow Q_4 \\
 (Q_0, b) \rightarrow Q_2 & (Q_1, b) \rightarrow Q_2 & (Q_4, b) \rightarrow Q_4 \quad \dots
 \end{array}$$

# Accepting a String

Given

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

you can define

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

# Accepting a String

Given

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

you can define

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

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**

$$\begin{aligned}(Q_1, a) &\rightarrow Q_2 \\ (Q_1, a) &\rightarrow Q_3 \quad \dots\end{aligned}$$

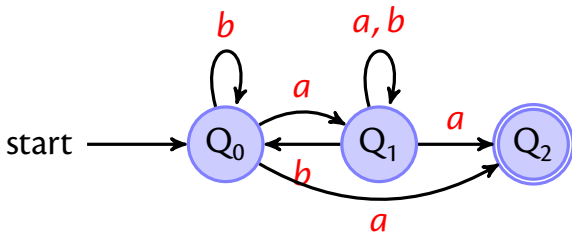
# Non-Deterministic Finite Automata

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**

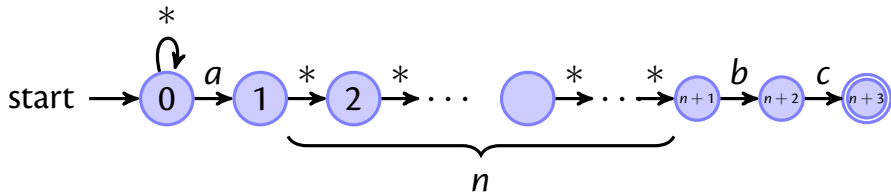
$$\begin{array}{l} (Q_1, a) \rightarrow Q_2 \quad \dots \quad (Q_1, a) \rightarrow \{Q_2, Q_3\} \\ (Q_1, a) \rightarrow Q_3 \end{array}$$

# An NFA Example



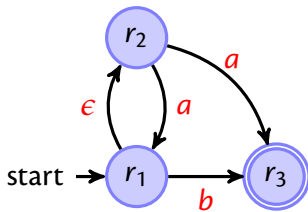
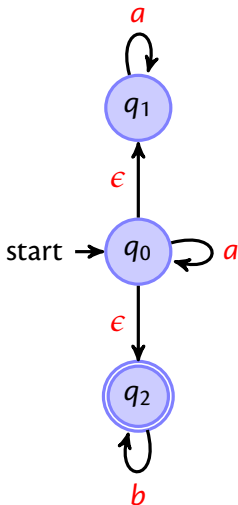
# Another Example

For the regular expression  $(.^*)a(.^{\{n\}})bc$



Note the star-transitions: accept any character.



# Two Epsilon NFA Examples



# Rexp to NFA

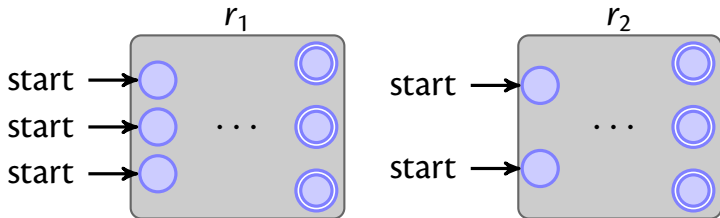
0 start → 

1 start → 

c start →  →   
*c*

# Case $r_1 \cdot r_2$

By recursion we are given two automata:

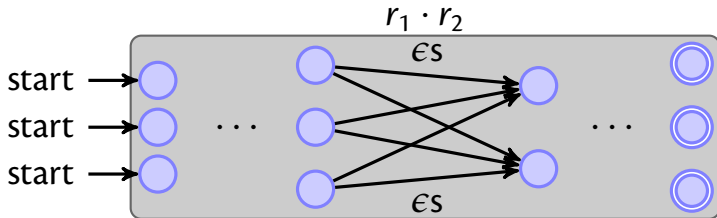


We need to (1) change the accepting nodes of the first automaton into non-accepting nodes, and (2) connect them via  $\epsilon$ -transitions to the starting state of the second automaton.



# Case $r_1 \cdot r_2$

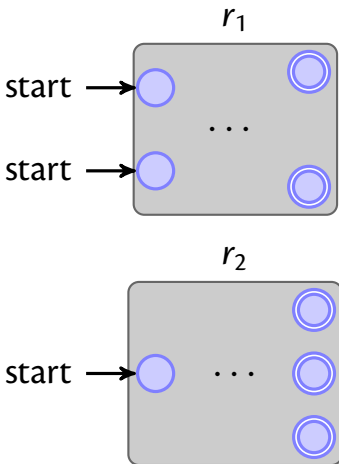
By recursion we are given two automata:



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

# Case $r_1 + r_2$

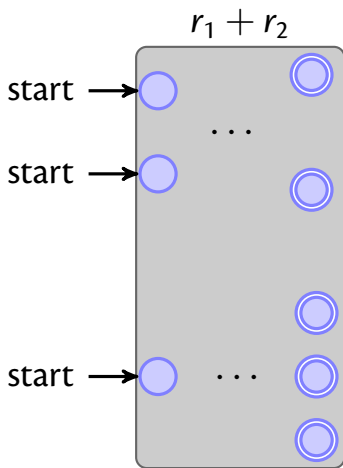
By recursion we are given two automata:



We can just put both automata together.

# Case $r_1 + r_2$

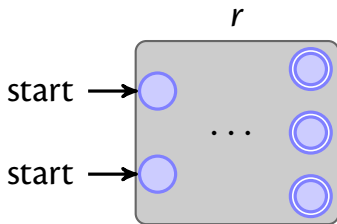
By recursion we are given two automata:



We can just put both automata together.

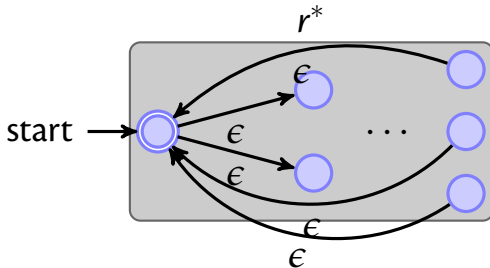
# Case $r^*$

By recursion we are given an automaton for  $r$ :



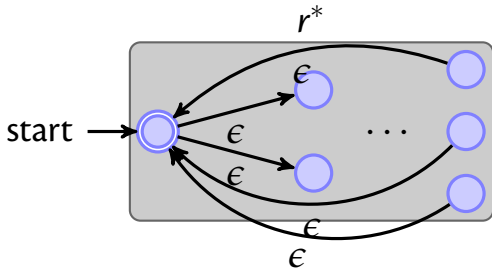
# Case $r^*$

By recursion we are given an automaton for  $r$ :



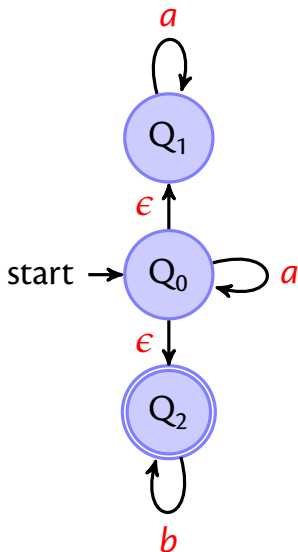
# Case $r^*$

By recursion we are given an automaton for  $r$ :



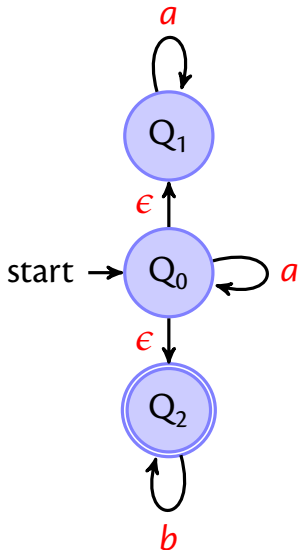
Why can't we just have an epsilon transition from the accepting states to the starting state?

# Subset Construction



nodes	$a$	$b$
$\{\}$		
$\{0\}$		
$\{1\}$		
$\{2\}$		
$\{0, 1\}$		
$\{0, 2\}$		
$\{1, 2\}$		
$\{0, 1, 2\}$		

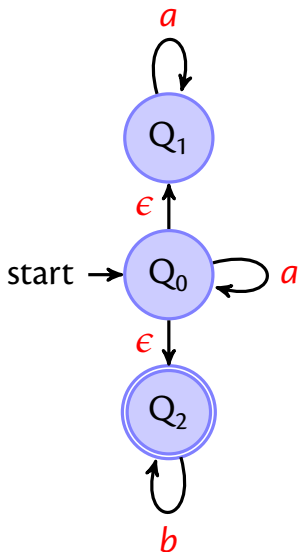
# Subset Construction



nodes	$a$	$b$
$\{\}$	$\{\}$	$\{\}$
$\{0\}$		
$\{1\}$		
$\{2\}$		
$\{0, 1\}$		
$\{0, 2\}$		
$\{1, 2\}$		
$\{0, 1, 2\}$		

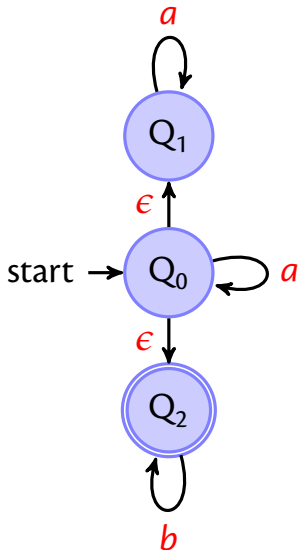


# Subset Construction



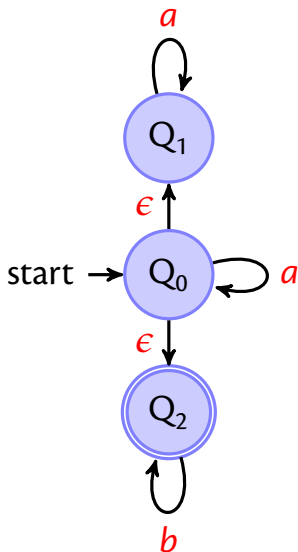
nodes	$a$	$b$
$\{\}$	$\{\}$	$\{\}$
$\{0\}$	$\{0, 1, 2\}$	$\{2\}$
$\{1\}$	$\{1\}$	$\{\}$
$\{2\}$	$\{\}$	$\{2\}$
$\{0, 1\}$		
$\{0, 2\}$		
$\{1, 2\}$		
$\{0, 1, 2\}$		

# Subset Construction



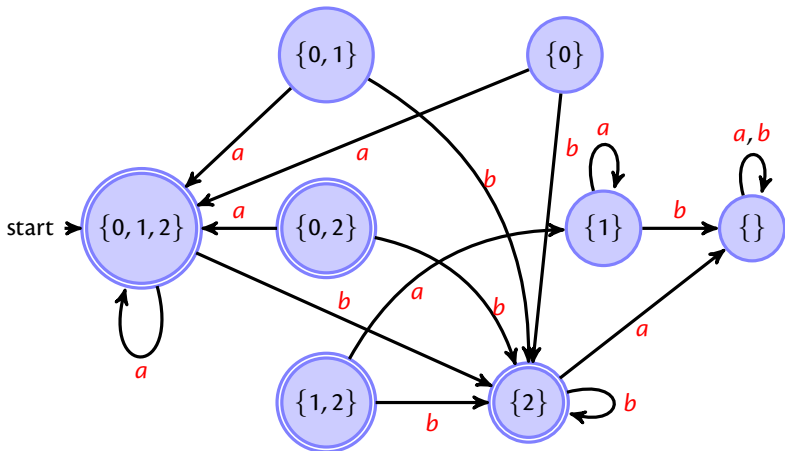
nodes	$a$	$b$
$\{\}$	$\{\}$	$\{\}$
$\{0\}$	$\{0, 1, 2\}$	$\{2\}$
$\{1\}$	$\{1\}$	$\{\}$
$\{2\}$	$\{\}$	$\{2\}$
$\{0, 1\}$	$\{0, 1, 2\}$	$\{2\}$
$\{0, 2\}$	$\{0, 1, 2\}$	$\{2\}$
$\{1, 2\}$	$\{1\}$	$\{2\}$
$\{0, 1, 2\}$	$\{0, 1, 2\}$	$\{2\}$

# Subset Construction



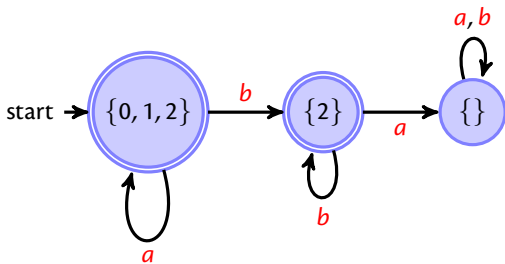
nodes	$a$	$b$
$\{\}$	$\{\}$	$\{\}$
$\{0\}$	$\{0, 1, 2\}$	$\{2\}$
$\{1\}$	$\{1\}$	$\{\}$
$\{2\}^*$	$\{\}$	$\{2\}$
$\{0, 1\}$	$\{0, 1, 2\}$	$\{2\}$
$\{0, 2\}^*$	$\{0, 1, 2\}$	$\{2\}$
$\{1, 2\}^*$	$\{1\}$	$\{2\}$
$s: \{0, 1, 2\}^*$	$\{0, 1, 2\}$	$\{2\}$

# The Result

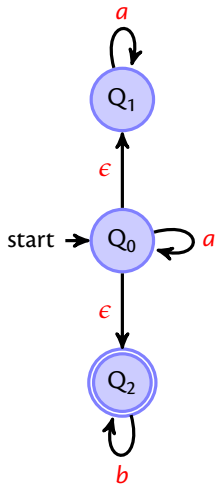


# Removing Dead States

DFA:



(original) NFA:

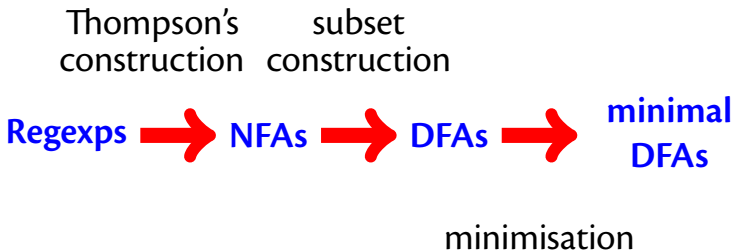


# Regexps and Automata

Thompson's construction      subset construction

Regexps  NFAs  DFAs

# Regexps and Automata



# DFA Minimisation

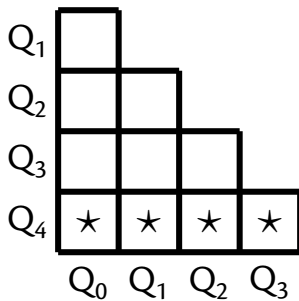
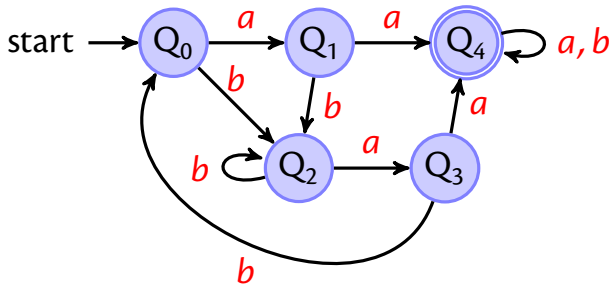
- 1 Take all pairs  $(q, p)$  with  $q \neq p$
- 2 Mark all pairs that accepting and non-accepting states
- 3 For all unmarked pairs  $(q, p)$  and all characters  $c$  test whether

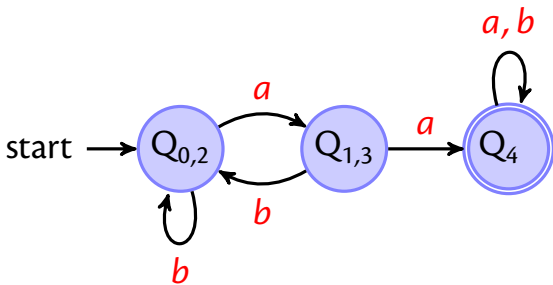
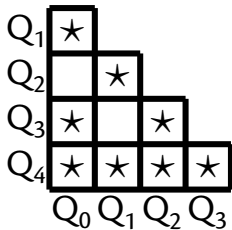
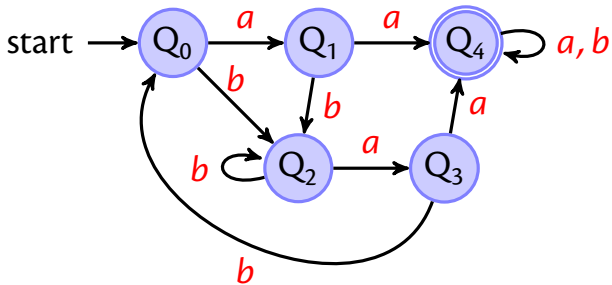
$$(\delta(q, c), \delta(p, c))$$

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.

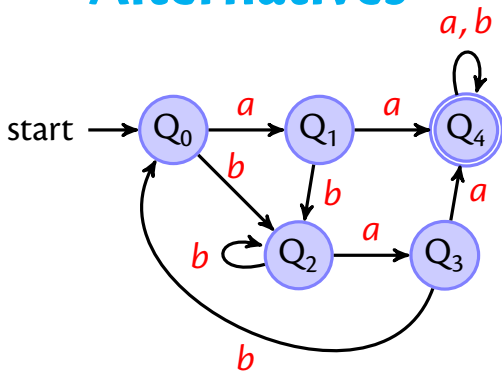






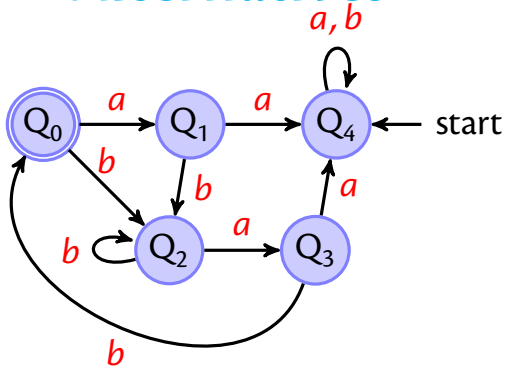
minimal automaton

# Alternatives



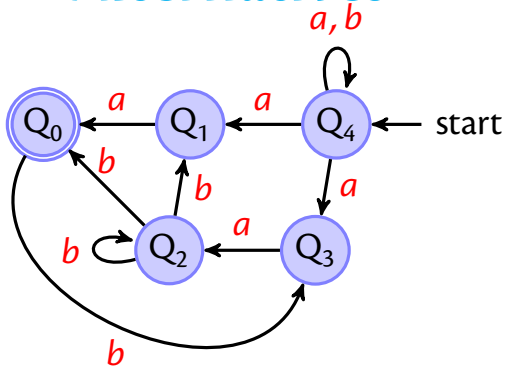
- exchange initial / accepting states

# Alternatives



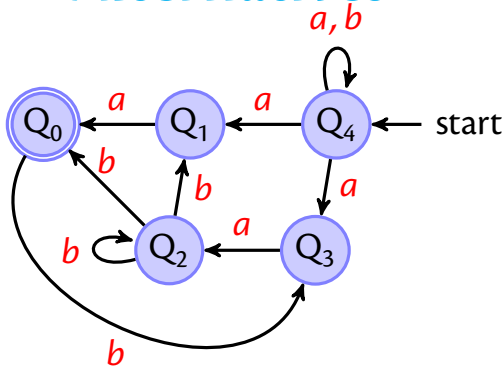
- exchange initial / accepting states
- reverse all edges

# Alternatives



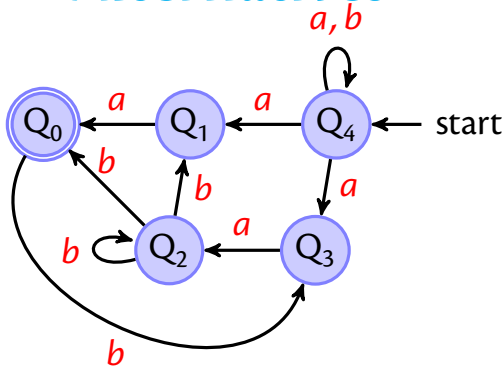
- exchange initial / accepting states
- reverse all edges
- subset construction  $\Rightarrow$  DFA

# Alternatives



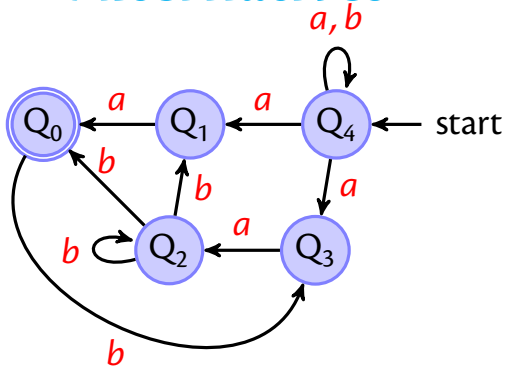
- exchange initial / accepting states
- reverse all edges
- subset construction  $\Rightarrow$  DFA
- remove dead states

# Alternatives



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

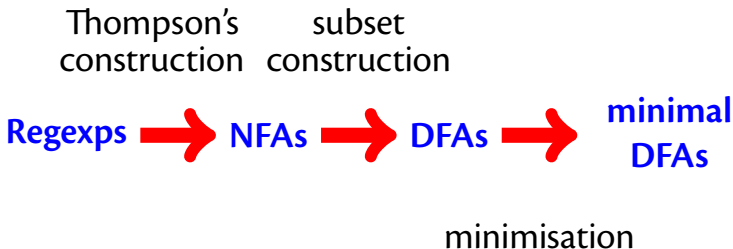
# Alternatives



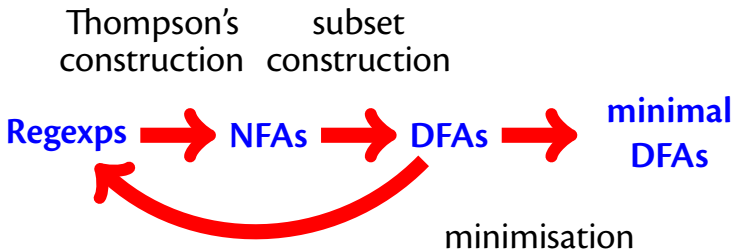
- exchange initial / accepting states
- reverse all edges
- subset construction  $\Rightarrow$  DFA
- remove dead states
- repeat once more  $\Rightarrow$  minimal DFA



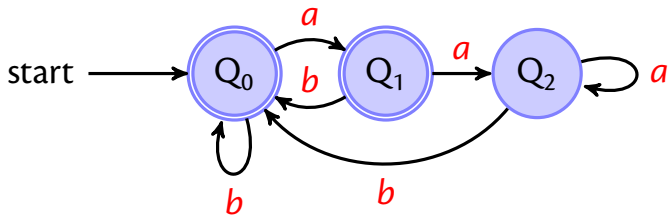
# Regexps and Automata

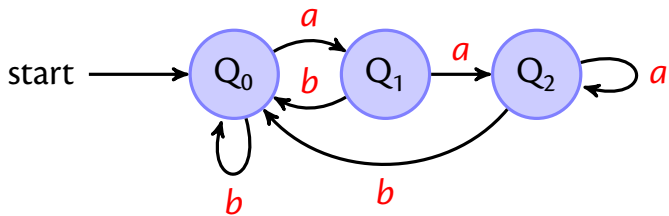


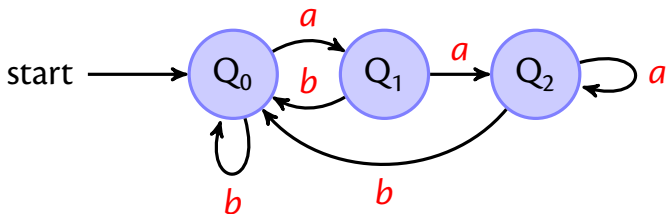
# Regexps and Automata



# DFA to Rexp





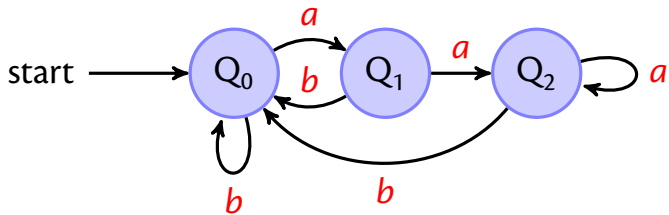


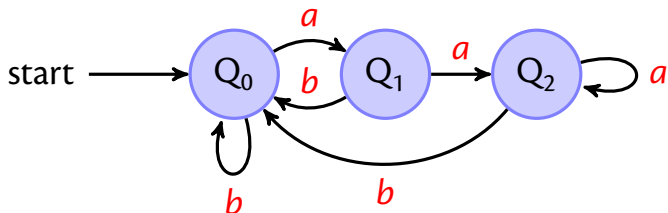
You know how to solve since school days, no?

$$Q_0 = 2Q_0 + 3Q_1 + 4Q_2$$

$$Q_1 = 2Q_0 + 3Q_1 + 1Q_2$$

$$Q_2 = 1Q_0 + 5Q_1 + 2Q_2$$

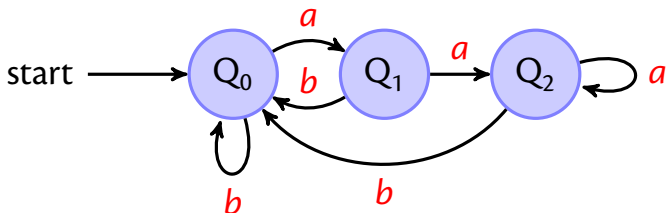




$$Q_0 = Q_0 b + Q_1 b + Q_2 b + 1$$

$$Q_1 = Q_0 a$$

$$Q_2 = Q_1 a + Q_2 a$$



$$Q_0 = Q_0 b + Q_1 b + Q_2 b + 1$$

$$Q_1 = Q_0 a$$

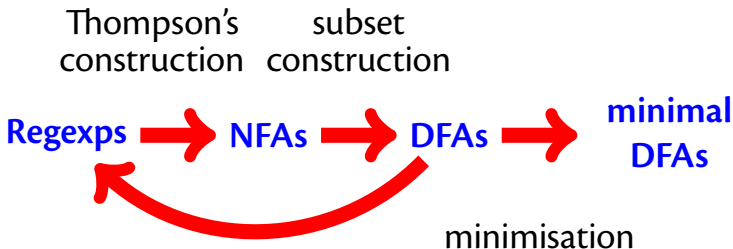
$$Q_2 = Q_1 a + Q_2 a$$

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

$$\text{rev}(\mathbf{0}) \stackrel{\text{def}}{=} \mathbf{0}$$

$$\text{rev}(\mathbf{1}) \stackrel{\text{def}}{=} \mathbf{1}$$

$$\text{rev}(c) \stackrel{\text{def}}{=} c$$

$$\text{rev}(r_1 + r_2) \stackrel{\text{def}}{=} \text{rev}(r_1) + \text{rev}(r_2)$$

$$\text{rev}(r_1 \cdot r_2) \stackrel{\text{def}}{=} \text{rev}(r_2) \cdot \text{rev}(r_1)$$

$$\text{rev}(r^*) \stackrel{\text{def}}{=} \text{rev}(r)^*$$

and the set

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

prove whether

$$L(\text{rev}(r)) = \text{Rev}(L(r))$$