

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/ProgInScalazed.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

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

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

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

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

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

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

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

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

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

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

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?

We need to prove

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

also by induction on the regular expression r .

Proofs about Rexp

- P holds for $\mathbf{0}$, $\mathbf{1}$ and \mathbf{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

$r ::=$	0	nothing
	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 of Regular Expr's

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

Negation of Regular Expr's

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

Used often for recognising comments:

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

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*!

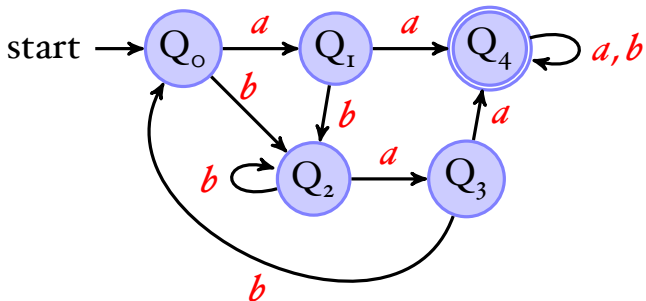
Automata

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

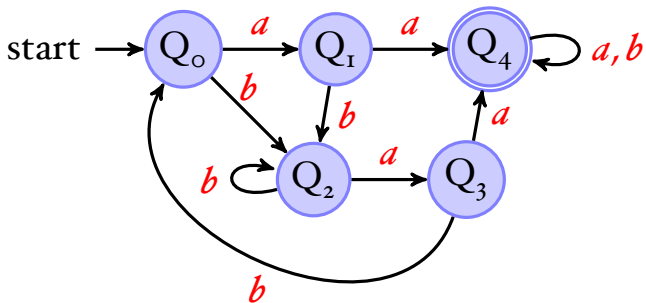
- an alphabet Σ
- a set of states Q_s
- one of these states is the start state Q_o
- some states are accepting states F , and
- there is transition function δ

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

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



- 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

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

Accepting a String

Given

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

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

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

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 ?

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

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**

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

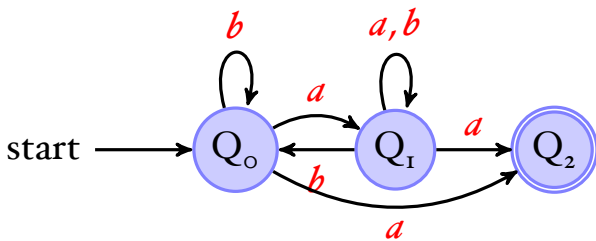
Non-Deterministic Finite Automata

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**

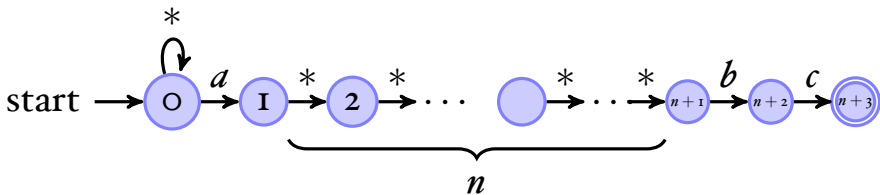
$$\begin{array}{l} (Q_1, a) \rightarrow Q_2 \\ (Q_1, a) \rightarrow Q_3 \end{array} \quad \dots \quad (Q_1, a) \rightarrow \{Q_2, Q_3\}$$

An NFA Example



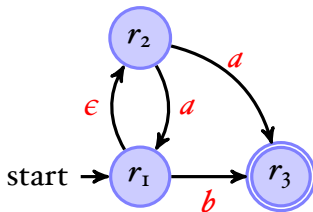
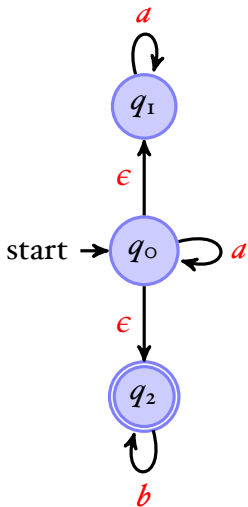
Another Example

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



Note the star-transitions: accept any character.

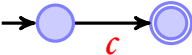
Two Epsilon NFA Examples



Rexp to NFA

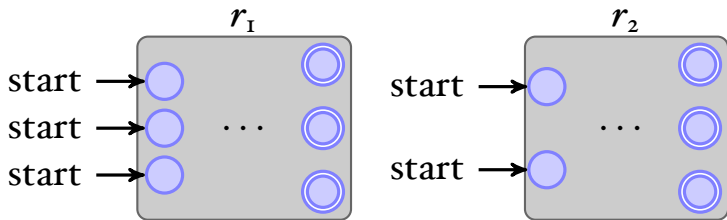
o start → 

I start → 

c start → 

Case $r_1 \cdot r_2$

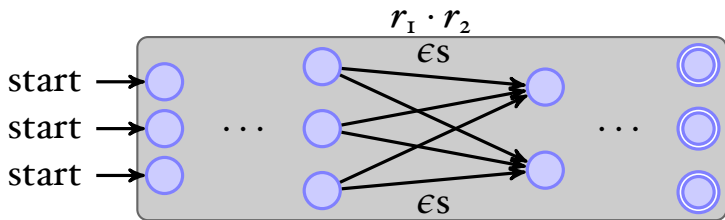
By recursion we are given two automata:



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 $r_1 \cdot r_2$

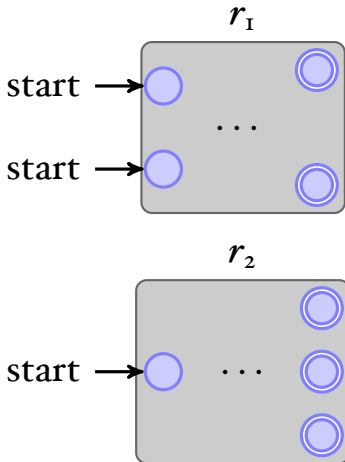
By recursion we are given two automata:



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 $r_1 + r_2$

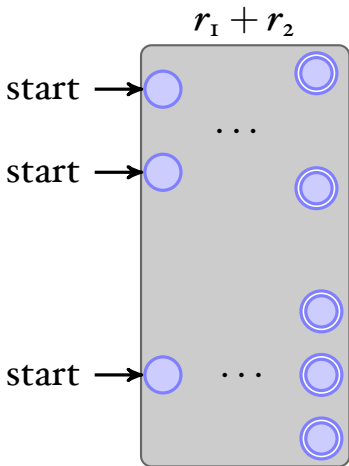
By recursion we are given two automata:



We can just put both automata together.

Case $r_1 + r_2$

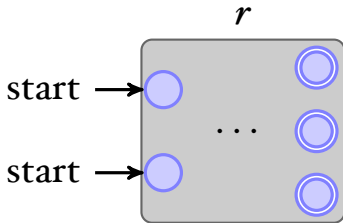
By recursion we are given two automata:



We can just put both automata together.

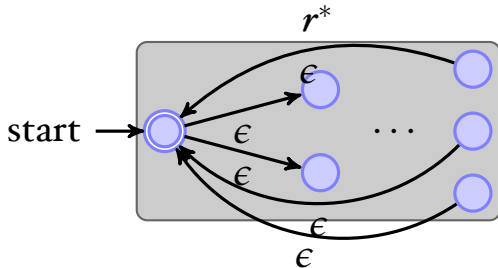
Case r^*

By recursion we are given an automaton for r :



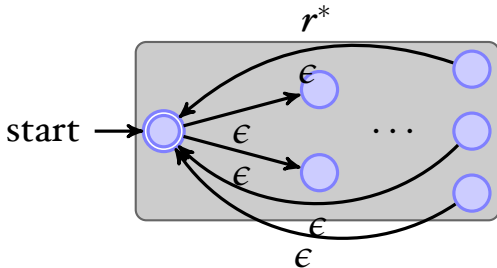
Case r^*

By recursion we are given an automaton for r :



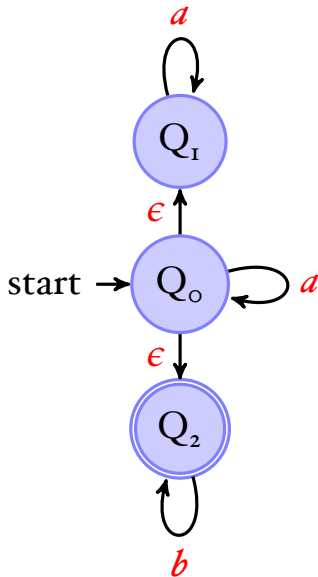
Case r^*

By recursion we are given an automaton for r :



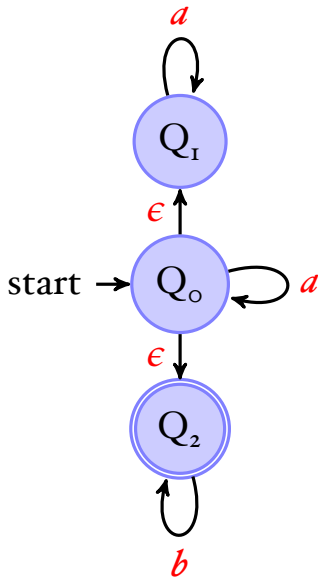
Why can't we just have an epsilon transition from the accepting states to the starting state?

Subset Construction



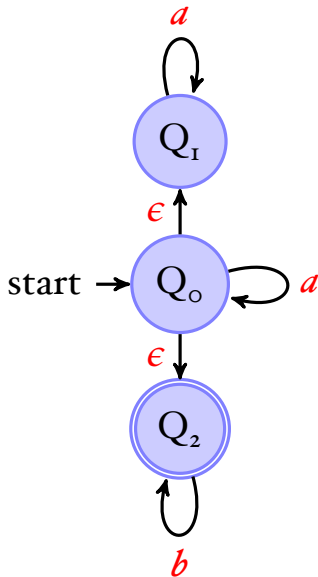
nodes	a	b
$\{\}$		
$\{0\}$		
$\{I\}$		
$\{2\}$		
$\{0, I\}$		
$\{0, 2\}$		
$\{I, 2\}$		
$\{0, I, 2\}$		

Subset Construction



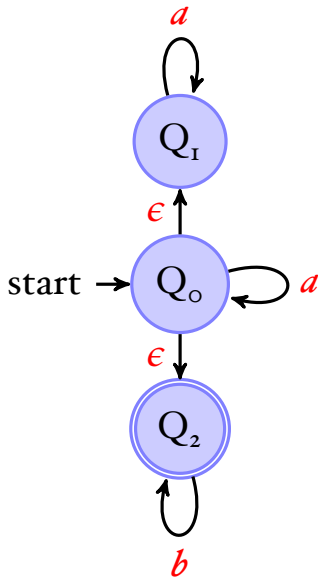
nodes	a	b
$\{\}$	$\{\}$	$\{\}$
$\{0\}$		
$\{I\}$		
$\{2\}$		
$\{0, I\}$		
$\{0, 2\}$		
$\{I, 2\}$		
$\{0, I, 2\}$		

Subset Construction



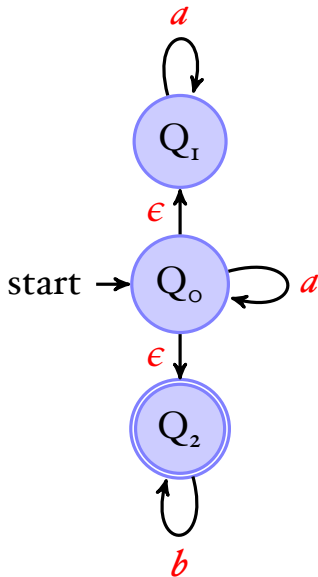
nodes	a	b
$\{\}$	$\{\}$	$\{\}$
$\{0\}$	$\{0, I, 2\}$	$\{2\}$
$\{I\}$	$\{I\}$	$\{\}$
$\{2\}$	$\{\}$	$\{2\}$
$\{0, I\}$		
$\{0, 2\}$		
$\{I, 2\}$		
$\{0, I, 2\}$		

Subset Construction



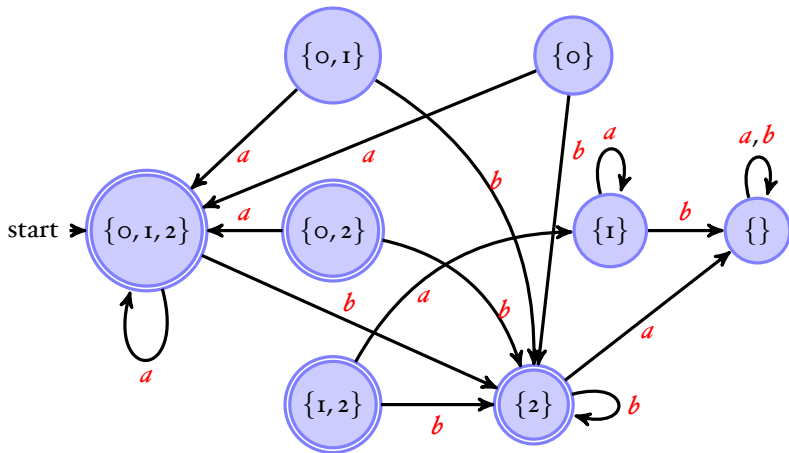
nodes	<i>a</i>	<i>b</i>
$\{\}$	$\{\}$	$\{\}$
$\{0\}$	$\{0, I, 2\}$	$\{2\}$
$\{I\}$	$\{I\}$	$\{\}$
$\{2\}$	$\{\}$	$\{2\}$
$\{0, I\}$	$\{0, I, 2\}$	$\{2\}$
$\{0, 2\}$	$\{0, I, 2\}$	$\{2\}$
$\{I, 2\}$	$\{I\}$	$\{2\}$
$\{0, I, 2\}$	$\{0, I, 2\}$	$\{2\}$

Subset Construction



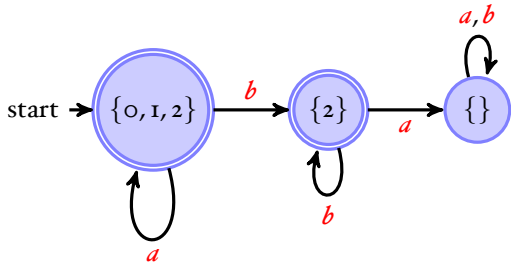
nodes	<i>a</i>	<i>b</i>
$\{\}$	$\{\}$	$\{\}$
$\{0\}$	$\{0, I, 2\}$	$\{2\}$
$\{I\}$	$\{I\}$	$\{\}$
$\{2\}^*$	$\{\}$	$\{2\}$
$\{0, I\}$	$\{0, I, 2\}$	$\{2\}$
$\{0, 2\}^*$	$\{0, I, 2\}$	$\{2\}$
$\{I, 2\}^*$	$\{I\}$	$\{2\}$
s: $\{0, I, 2\}^*$	$\{0, I, 2\}$	$\{2\}$

The Result

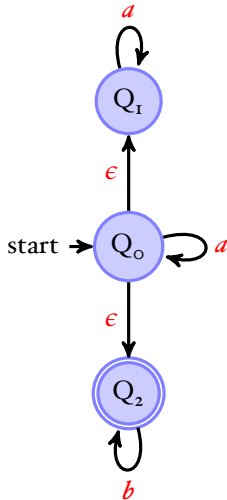


Removing Dead States

DFA:



(original) NFA:



Regexps and Automata

Thompson's construction subset construction

Regexps  **NFAs**  **DFAs**

Regexps and Automata

Thompson's construction subset construction



minimisation

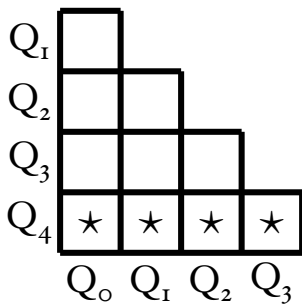
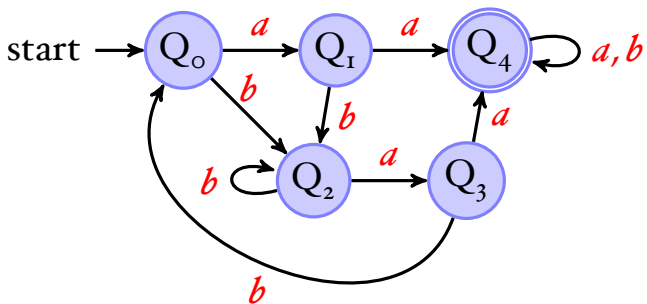
DFA Minimisation

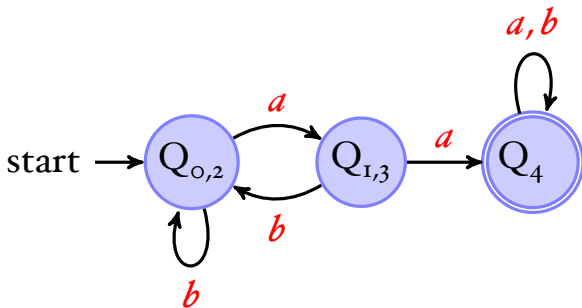
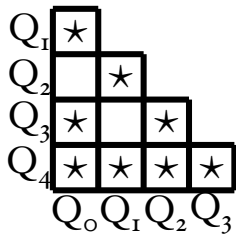
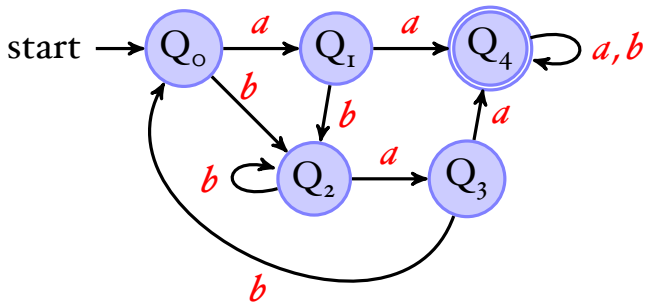
- 1 Take all pairs (q,p) with $q \neq p$
- 2 Mark all pairs that accepting and non-accepting states
- 3 For all unmarked pairs (q,p) and all characters c test whether

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

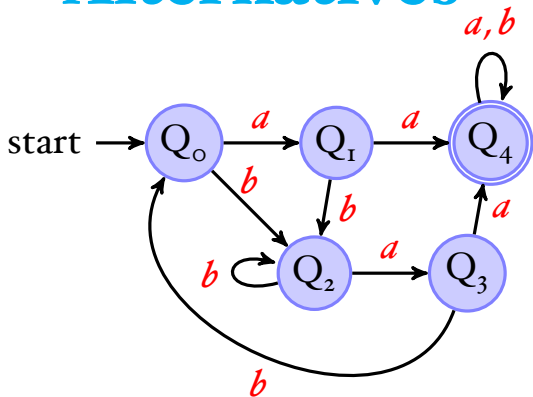
are marked. If yes in at least one case, then also mark (q,p) .

- 4 Repeat last step until no change.
- 5 All unmarked pairs can be merged.



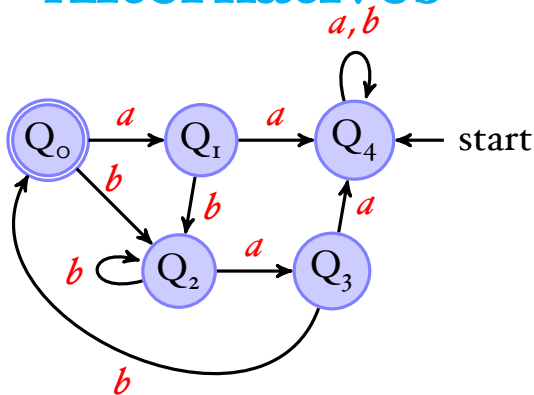


Alternatives



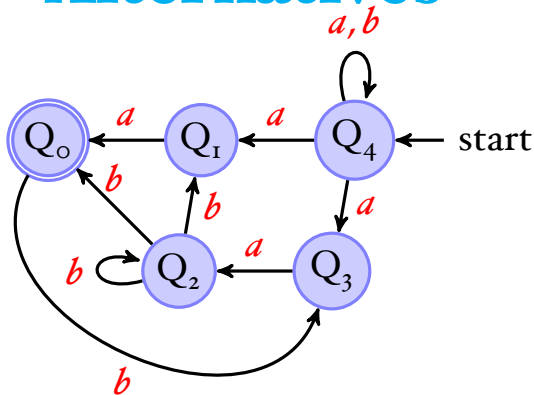
- exchange initial / accepting states

Alternatives



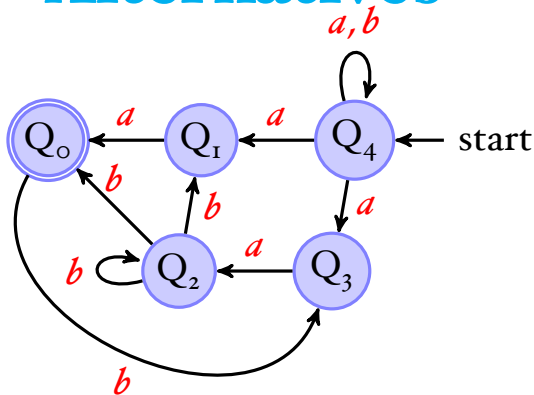
- exchange initial / accepting states
- reverse all edges

Alternatives



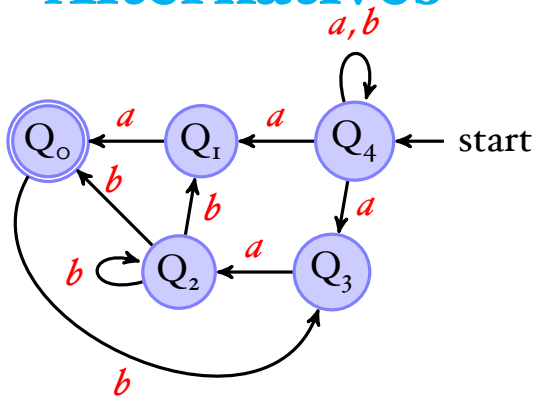
- exchange initial / accepting states
- reverse all edges
- subset construction \Rightarrow DFA

Alternatives



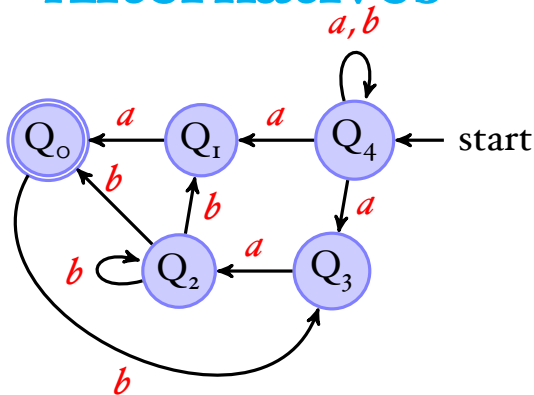
- exchange initial / accepting states
- reverse all edges
- subset construction \Rightarrow DFA
- remove dead states

Alternatives



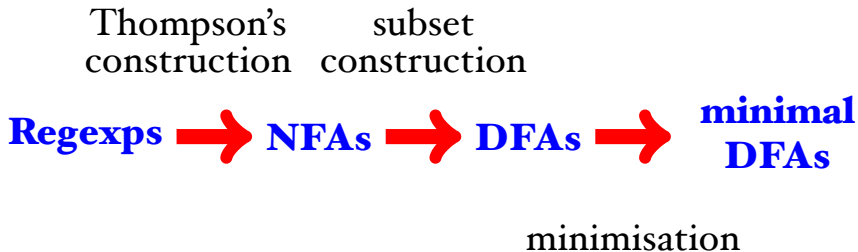
- exchange initial / accepting states
- reverse all edges
- subset construction \Rightarrow DFA
- remove dead states
- repeat once more

Alternatives

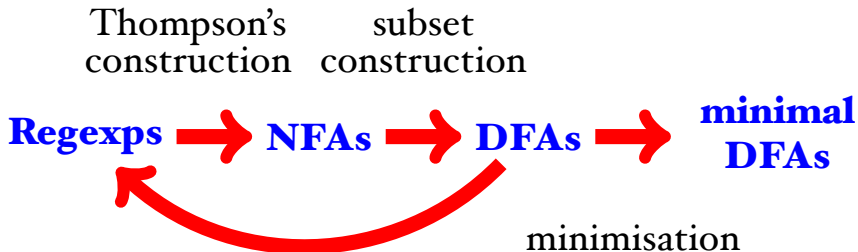


- exchange initial / accepting states
- reverse all edges
- subset construction \Rightarrow DFA
- remove dead states
- repeat once more \Rightarrow minimal DFA

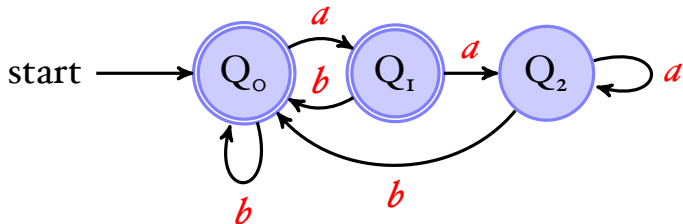
Regexps and Automata

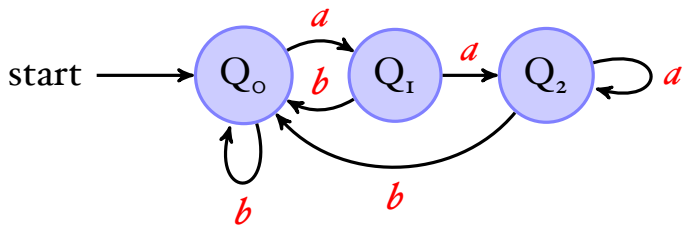


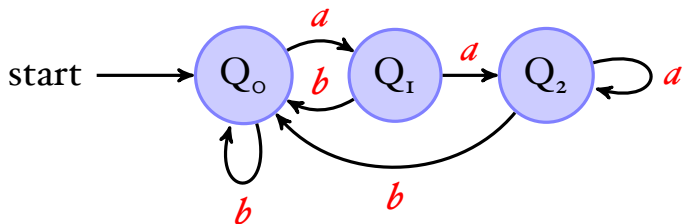
Regexps and Automata



DFA to Rexp





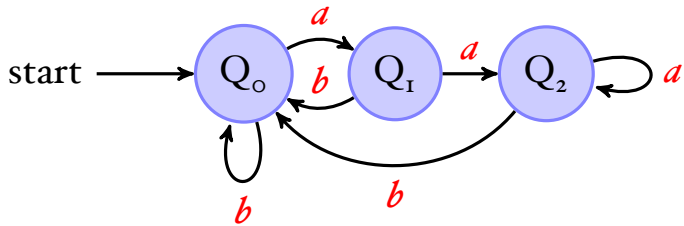


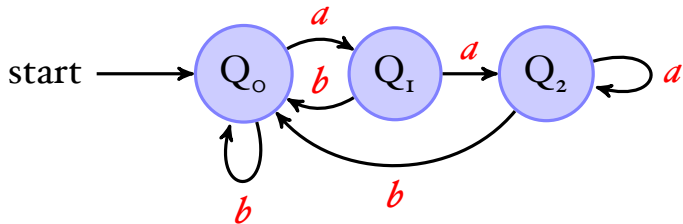
You know how to solve since school days, no?

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

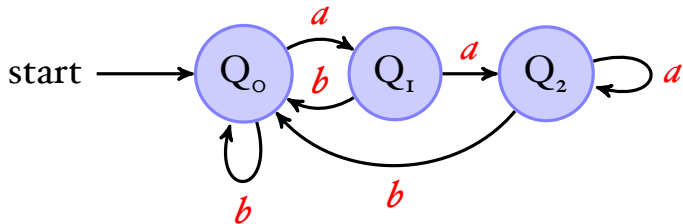
$$Q_I = 2Q_0 + 3Q_I + 1Q_2$$

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





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

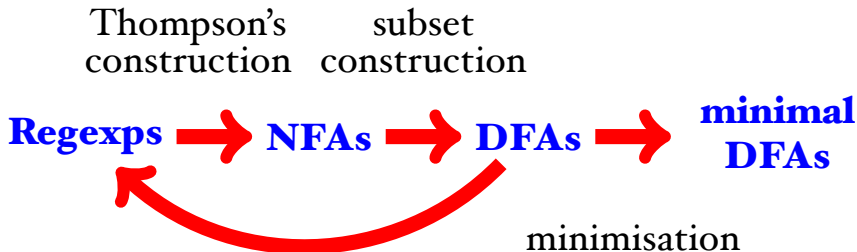


$$\begin{aligned}
 Q_0 &= \mathbf{1} + Q_0 b + Q_I b + Q_2 b \\
 Q_I &= Q_0 a \\
 Q_2 &= Q_I a + Q_2 a
 \end{aligned}$$

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

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

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

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

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

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

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

and the set

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

prove whether

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