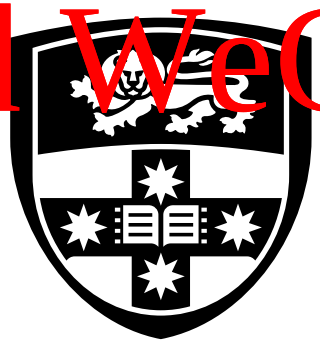
COMP2022: Formal Languages and Logic

2018, Semester 2, Week 6

Joseph Godbehere

6th September, 2018



THE UNIVERSITY OF SYDNEY

Outline

- **Regular Expressions**

- Equivalence of FA and Regular Expressions

- Proving if a language is, *or is not*, regular

Outline

- Regular Expressions

- **Equivalence of FA and Regular Expressions**

- Proving if a language is, *or is not*, regular

Equivalence of RegEx and FA

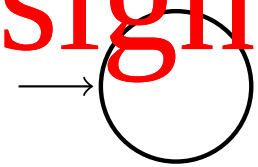
Theorem:
A language is regular if and only if a regular expression describes it.

Proof
Show the equivalence of RegEx and FA (RegEx ⇔ FA)

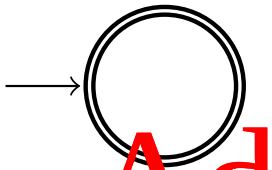
1. RegEx ⇒ FA:
   Show that for each RegEx, there exists an NFA which recognises the same language
2. FA ⇒ RegEx:
   Show that for each NFA, there exists a RegEx which recognises the same language

From RegEx to FA: atomic cases

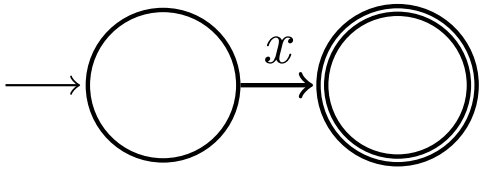
Automaton for $\emptyset$



Automaton for ε



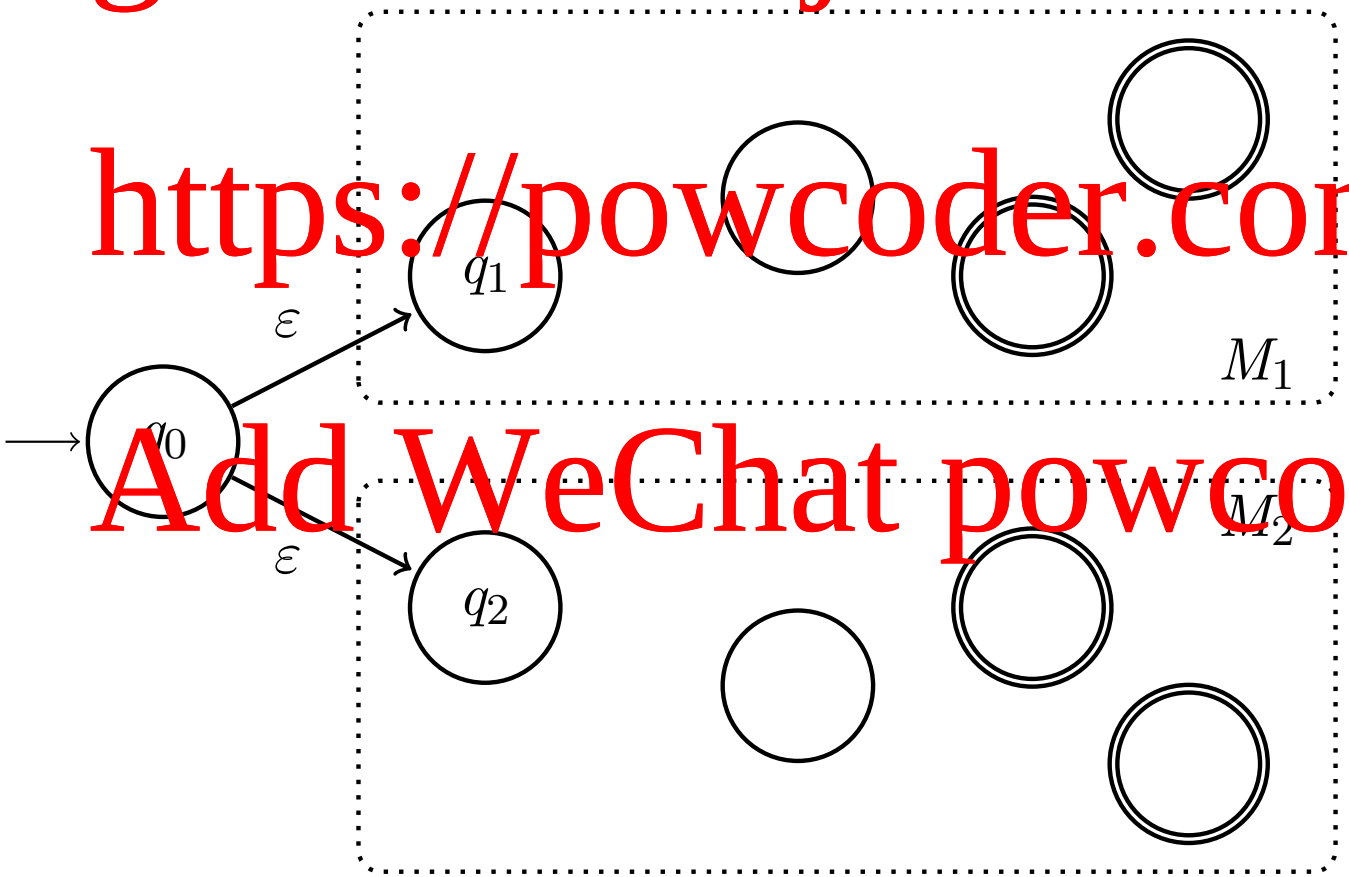
Automaton for $x \in \Sigma$



NFA for regular operations: Union

Let M_1 and M_2 be automata recognising $L(R_1)$ and $L(R_2)$

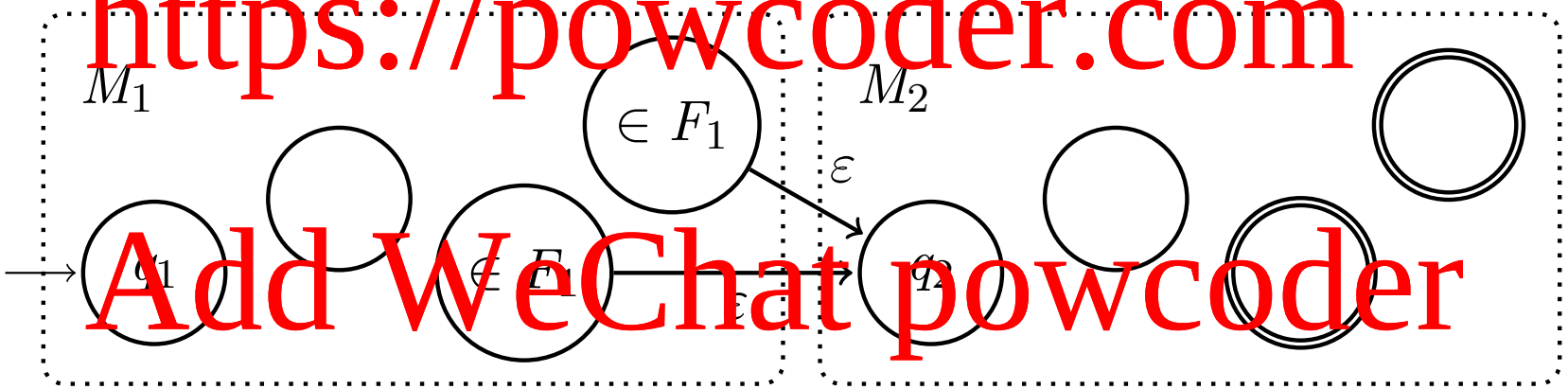
Then an automaton M for $R_1 \mid R_2$ is:



NFA for regular operations: Concatenation

Let M_1 and M_2 be automata recognising $L(R_1)$ and $L(R_2)$

Then an automaton M for $R_1 \circ R_2$ is:

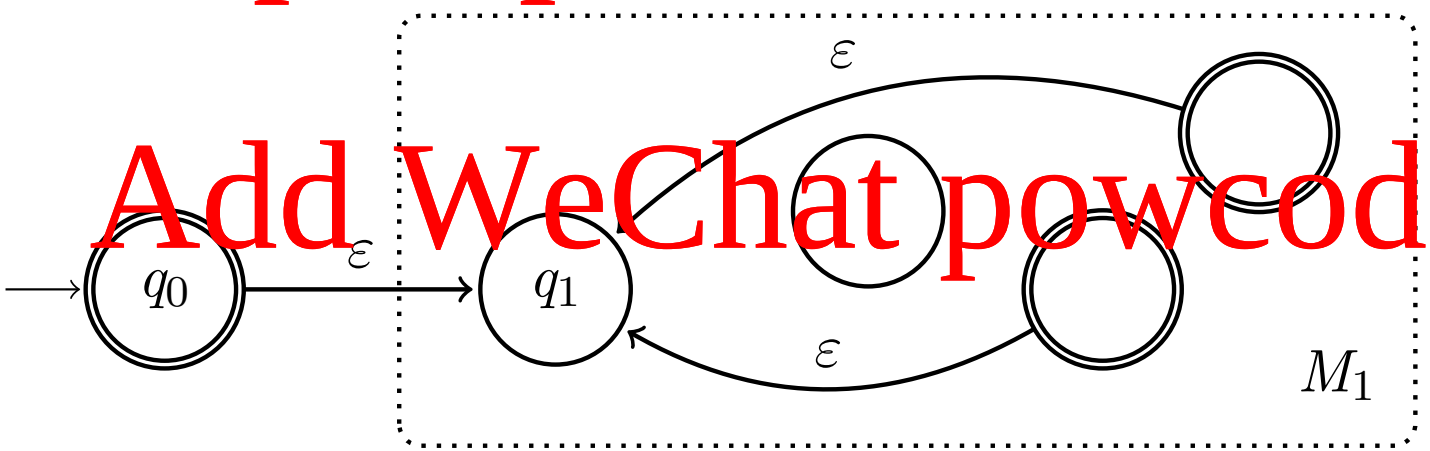


Reminder: the accept states of M_1 are *not* accept states in M

NFA for regular operations: Star closure

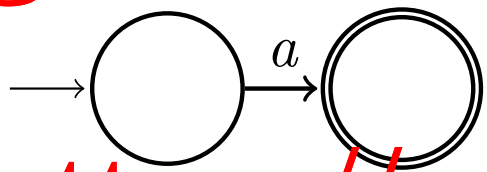
Let M_1 be an automaton recognising $L(R_1)$

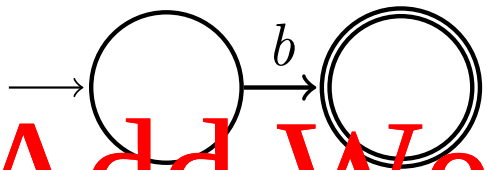
Then an automaton M for $R_1^\star$ is:

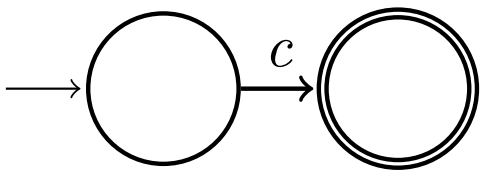


Example: $a \mid bc^\star$: from RegEx to NFA

1. Construct automata for the atomic regular expressions a, b, c

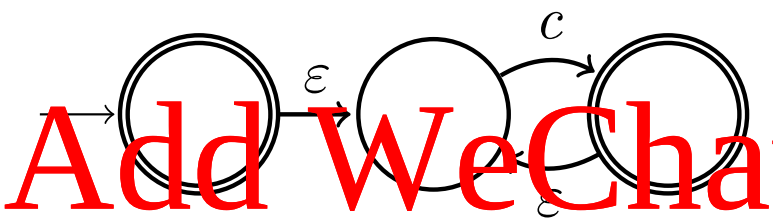






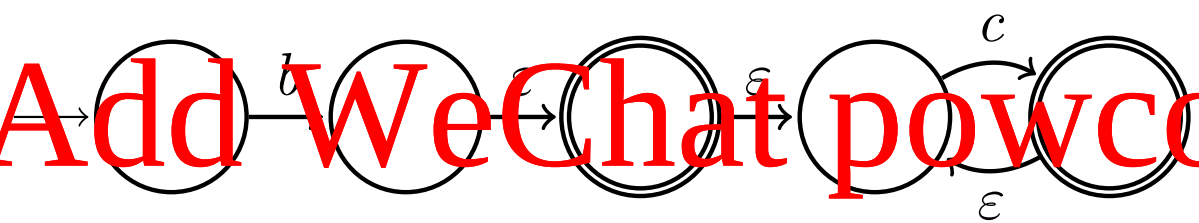
Example: $a \mid bc^\star$: from RegEx to NFA

1. Construct automata for the atomic regular expressions a, b, c
2. Use the Star Closure operation to find an automaton for $c^\star$



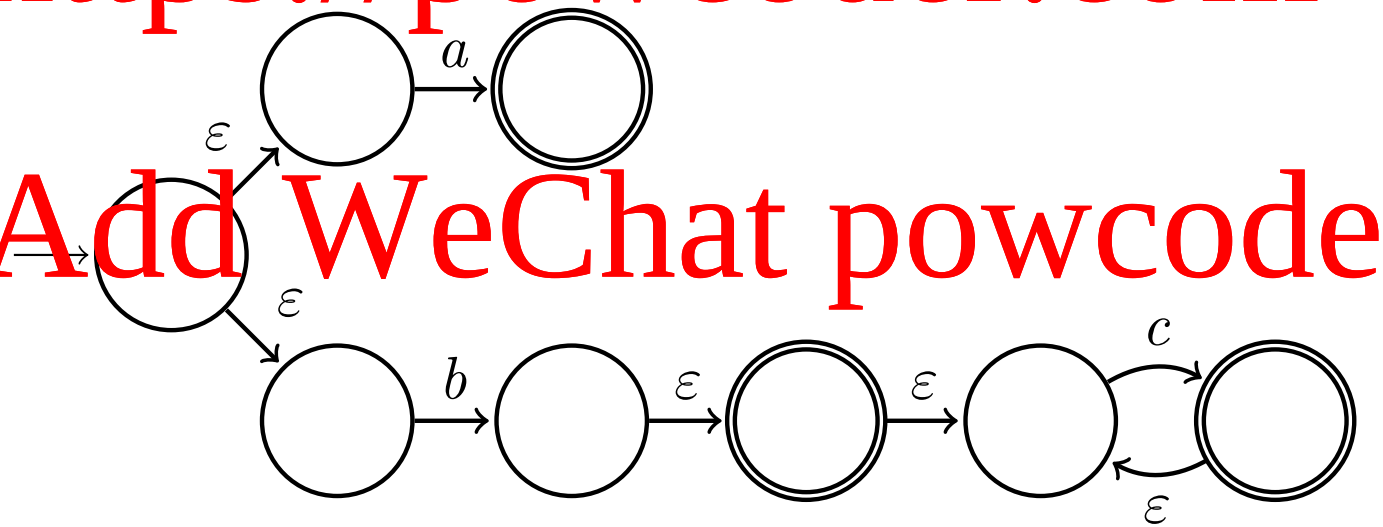
Example: $a \mid bc^\star$: from RegEx to NFA

1. Construct automata for the atomic regular expressions a, b, c
2. Use the Star Closure operation to find an automaton for $c^\star$
3. Use the Concatenation operation to find an automaton for $bc^\star$



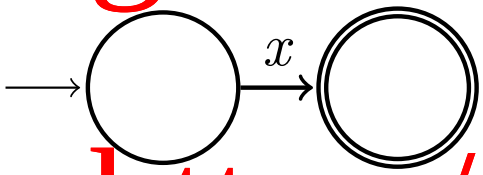
Example: $a \mid bc^\star$: from RegEx to NFA

1. Construct automata for the atomic regular expressions a, b, c
2. Use the Star Closure operation to find an automaton for $c^\star$
3. Use the Concatenation operation to find an automaton for $bc^\star$
4. Use the Union operation to find an automaton for $a|bc^\star$

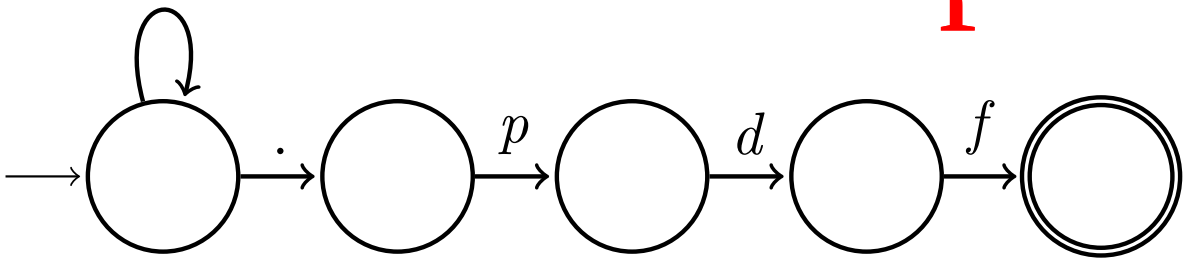


From NFA to RegEx: Simple examples

This automaton gives the RegEx x

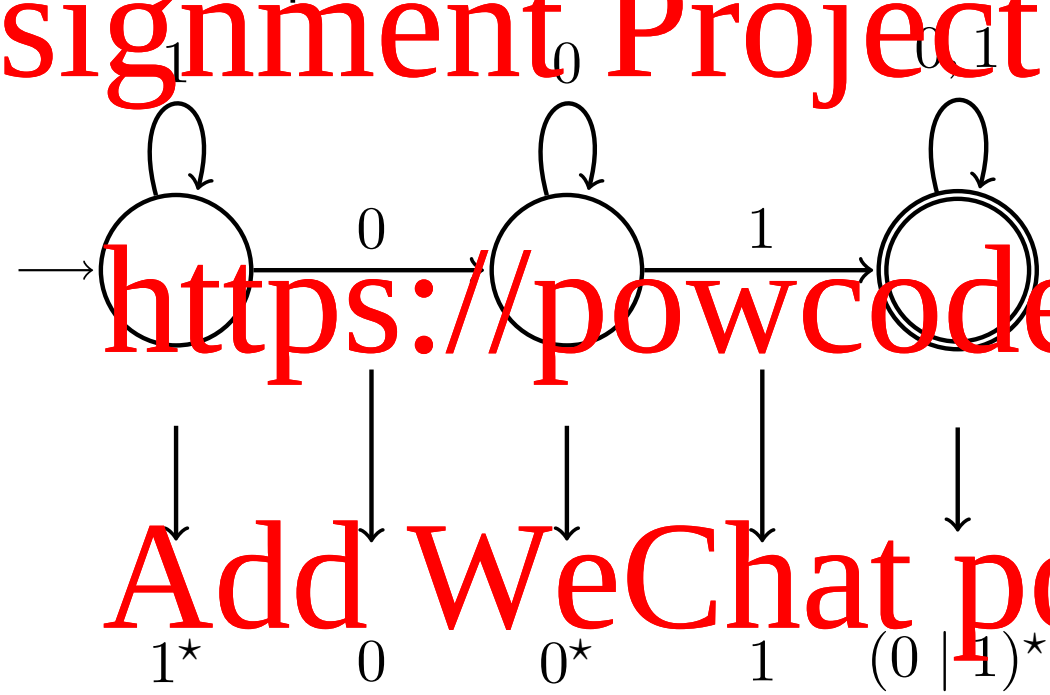


This one gives $(a \mid b \mid ... \mid Y \mid Z)^\star.pdf$



From NFA to RegEx: Simple examples

A more complex one:



$1^\star 00^\star 1(0 \mid 1)^\star$

Concept

Algorithm to convert an NFA to a RegEx:

1. Convert the NFA to a Generalised NFA (GNFA)
   - Only one start state, no incoming transitions
   - Only one accept state, no outgoing transitions
   - Transitions described by RegEx

2. Progressively *eliminate* all states between the start and accept state

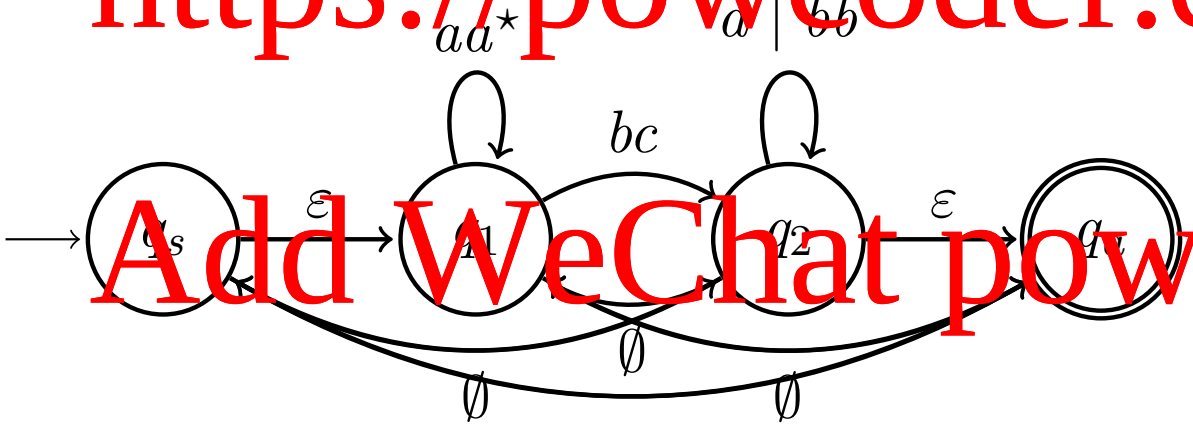
3. The transition between the start and the accept state is now a regular expression describing L

Generalised NFA (GNFA)

The start state q_s is non-accepting and has no incoming transitions

The only accept state q_a has no outgoing transitions, and $q_s \neq q_a$

There is exactly one transition between each ordered pair of states, labelled with a RegEx.



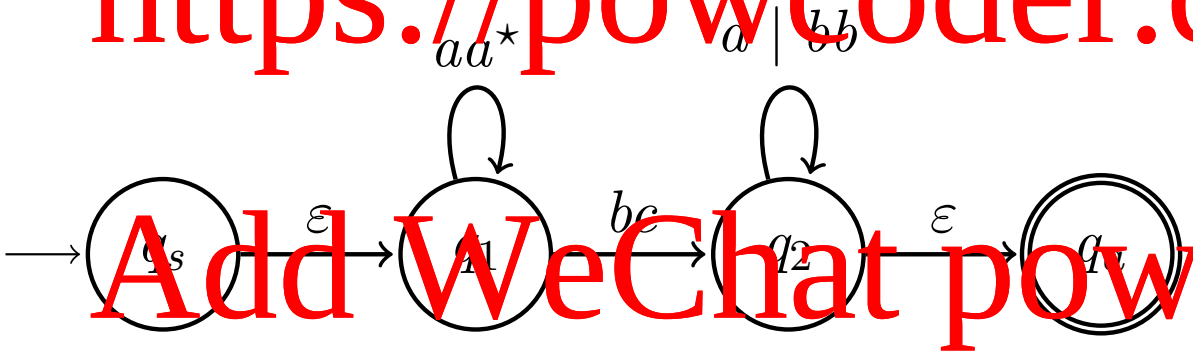
$\{abc, abca, abcbb, aaaaaabc\} \subseteq L$

Generalised NFA (GNFA)

The start state q_s is non-accepting and has no incoming transitions

The only accept state q_a has no outgoing transitions, and $q_s \neq q_a$

There is exactly one transition between each ordered pair of states, labelled with a RegEx.



We do not show the $\emptyset$ transitions (why not?)

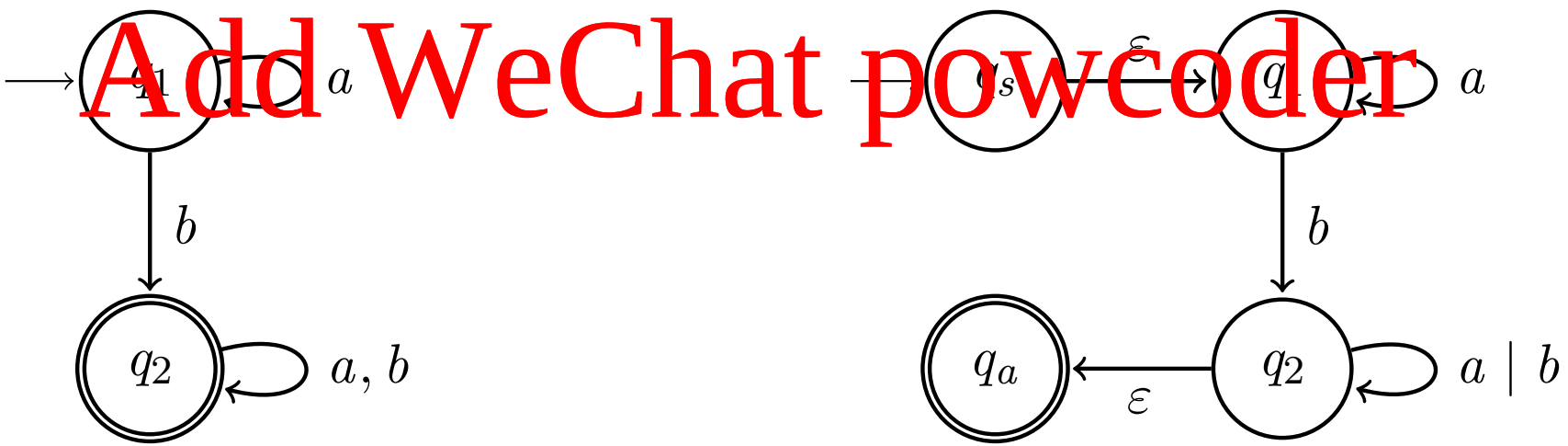
$\{abc, abca, abcbb, aaaaaabc\} \subseteq L$

Converting an NFA to a GNFA

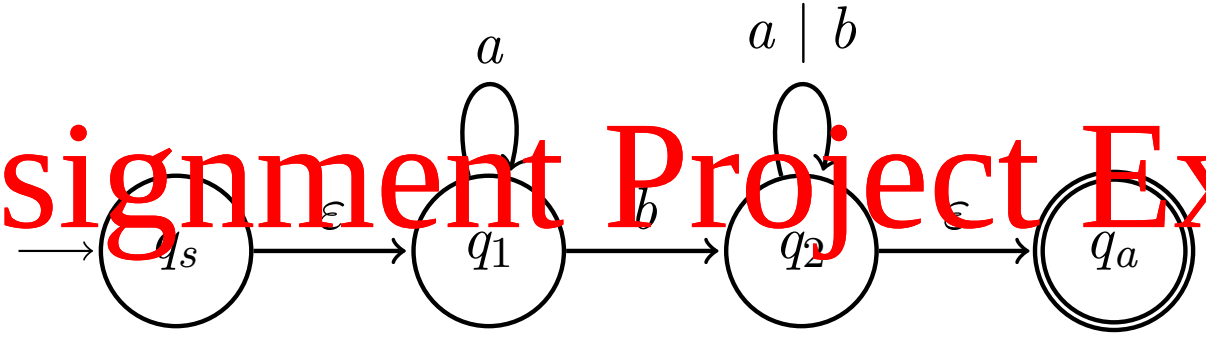
Create a new non-accepting start state q_s, with a ε-transition to the original start state.

Create a new accept state q_a with ε-transitions from the original accept states, which are no longer accepting.

Label the transitions between every ordered pair (q_i, q_j) of states from the NFA as the union of the atomic RegEx describing each transition from q_i to q_j in the NFA.



Simple example: eliminate state q_1



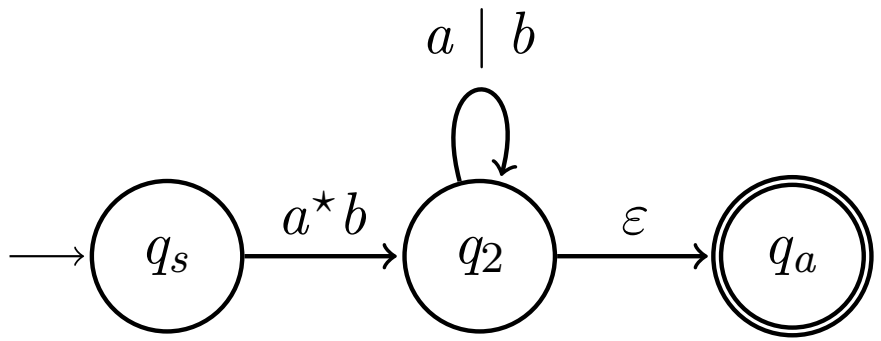
There is only one way to pass through q_1:

$q_s \to q_1$ ε

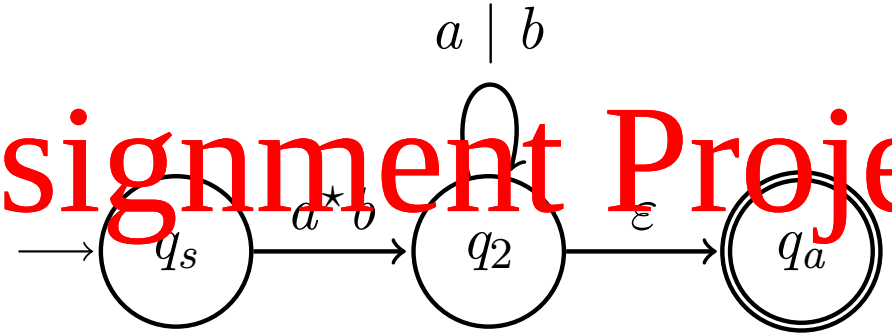
$q_1 \to q_1$ any number of times $a^\star$

$q_1 \to q_2$ b

We set the transition $q_s \to q_2$ to the union of the new RegEx and it's old value ($\emptyset$). This is $\varepsilon a^\star b \mid \emptyset$, which simplifies to $a^\star b$



Simple example: eliminate state q_2

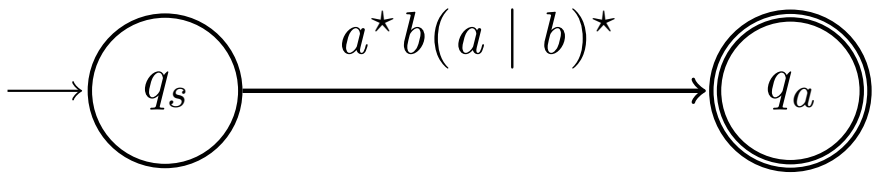


There is only one way to pass through q_2:

$q_s \to q_2$ $\quad a^\star b$

$q_2 \to q_2$ any number of times $\quad (a|b)^\star$

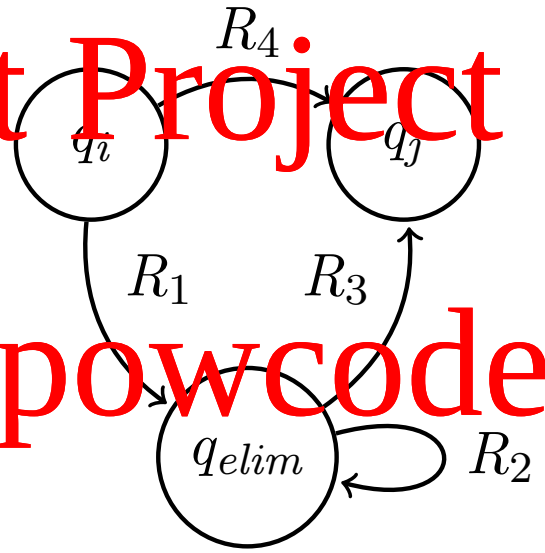
$q_2 \to q_a$ $\quad \varepsilon$



The language of the original automaton is $a^\star b(a \mid b)^\star$

General method to eliminate state q_{elim}

Consider each pair (q_i, q_j) where $q_i, q_j \in Q \setminus \{q_{elim}\}$



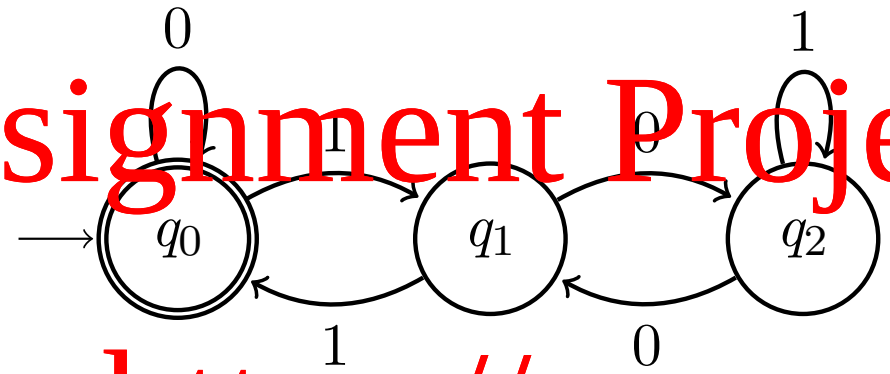
Replace the transition from q_i to q_j with the expression

$$((R_1)(R_2)^\star(R_3) \mid (R_4))$$

Note:

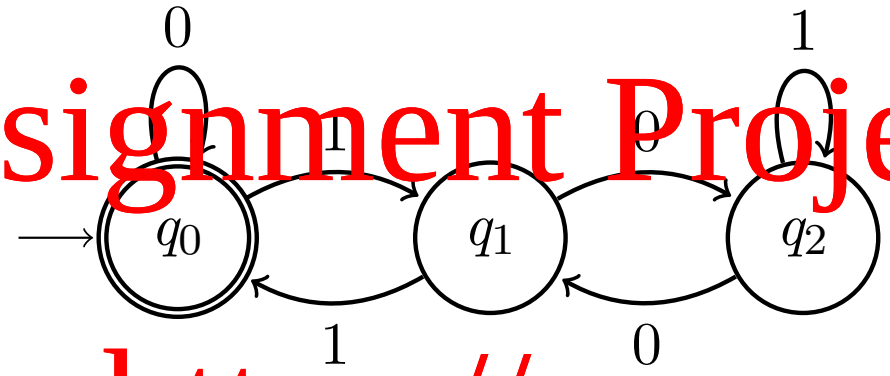
- Possibly $q_i = q_j$
- Recall that pairs are ordered, so we also consider (q_j, q_i)
- If there is no transition R_x, then $R_x = \emptyset$

Example 2: divisible by 3

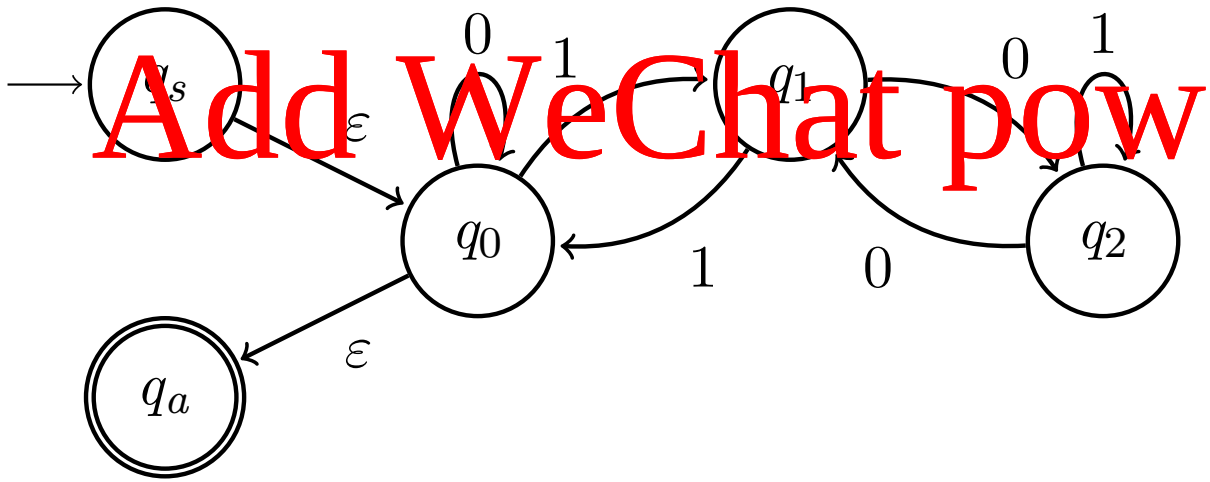


Convert to GNFA:

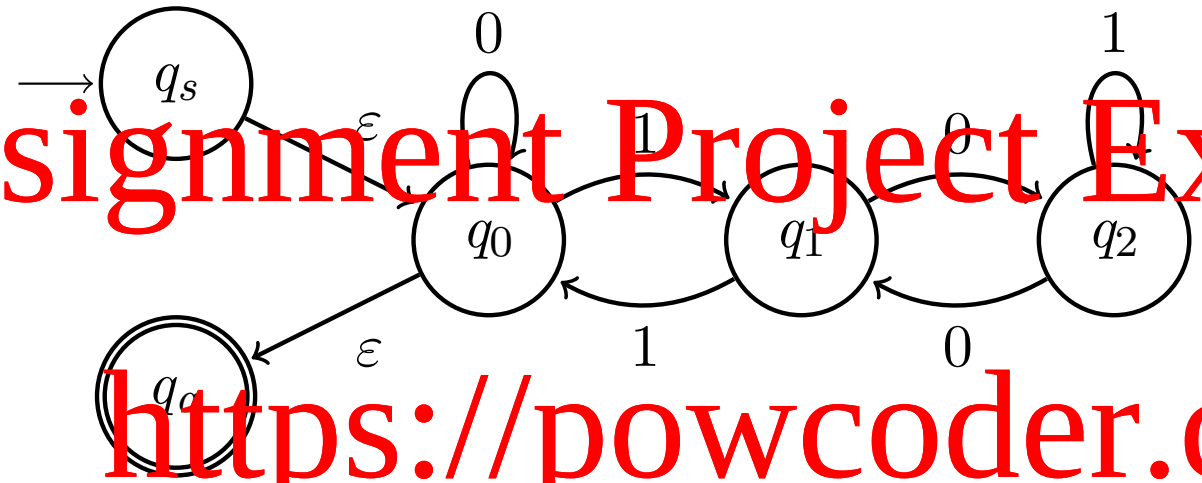
Example 2: divisible by 3



Convert to GNFA:



Example 2: divisible by 3

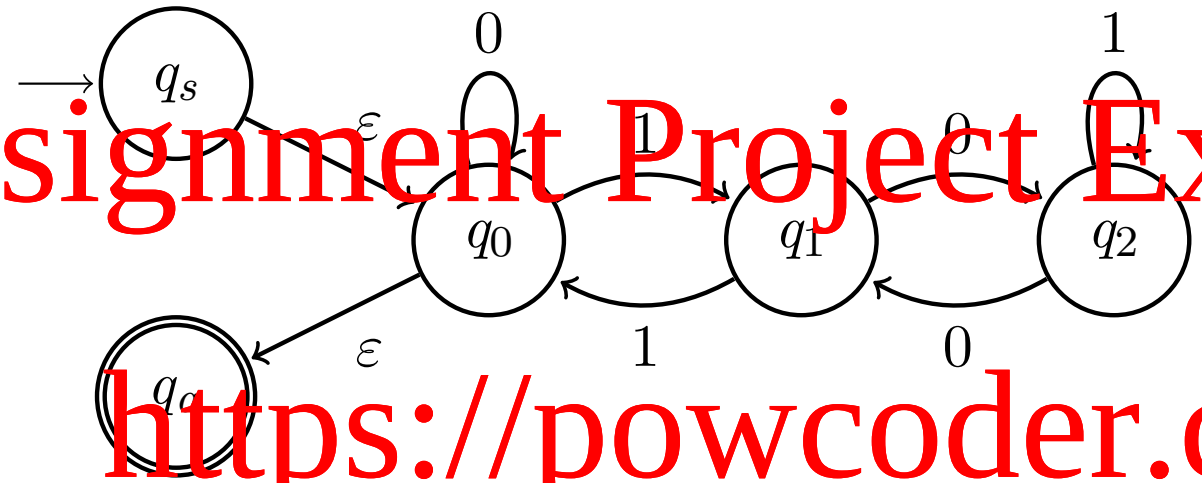


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- ▶
- ▶
- ▶
- ▶

Example 2: divisible by 3

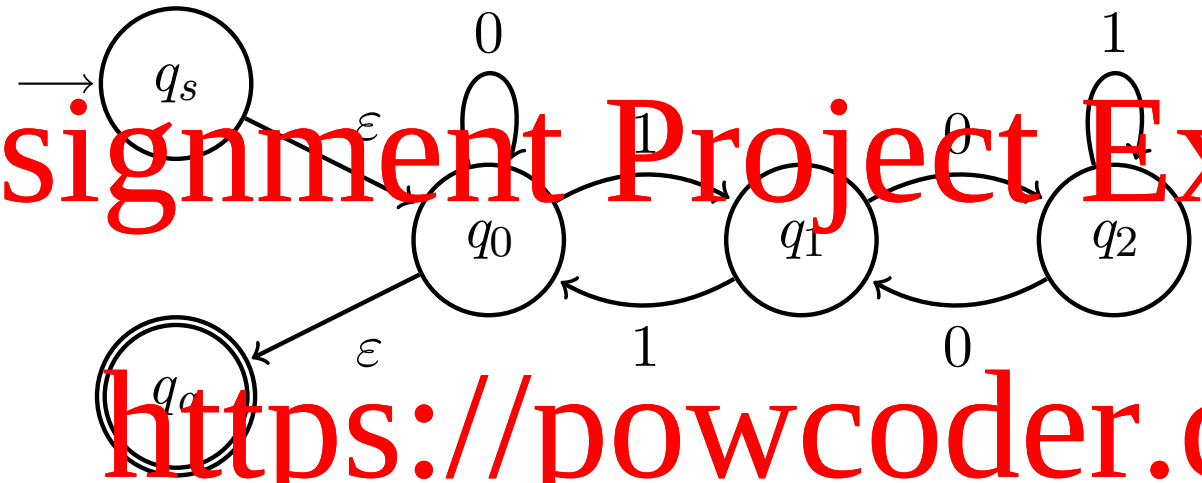


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- (q_s, q_1) :
- (q_s, q_a) :
- (q_1, q_1) :
- (q_1, q_a) :

Example 2: divisible by 3

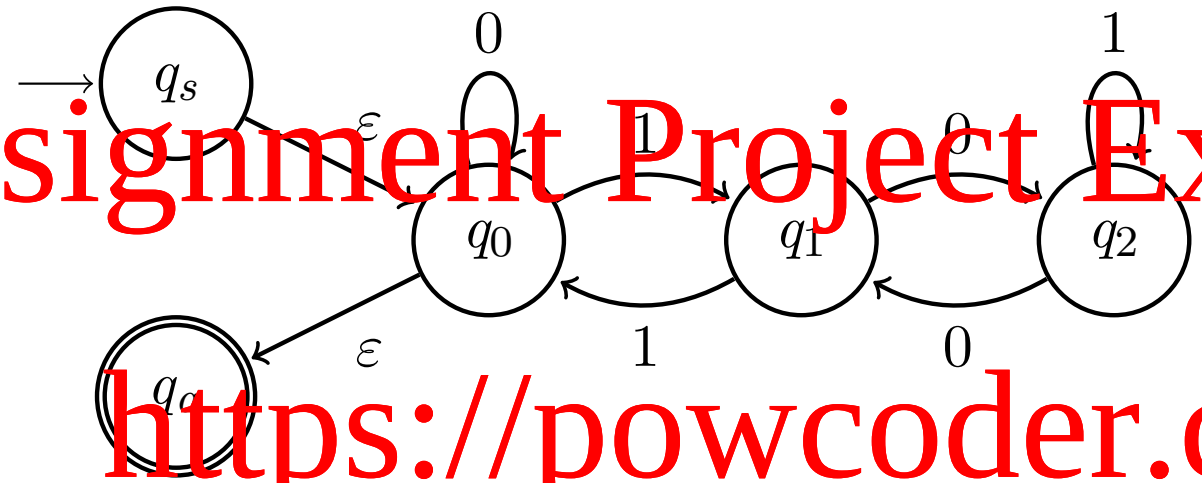


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon,$
- $(q_s, q_a) :$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

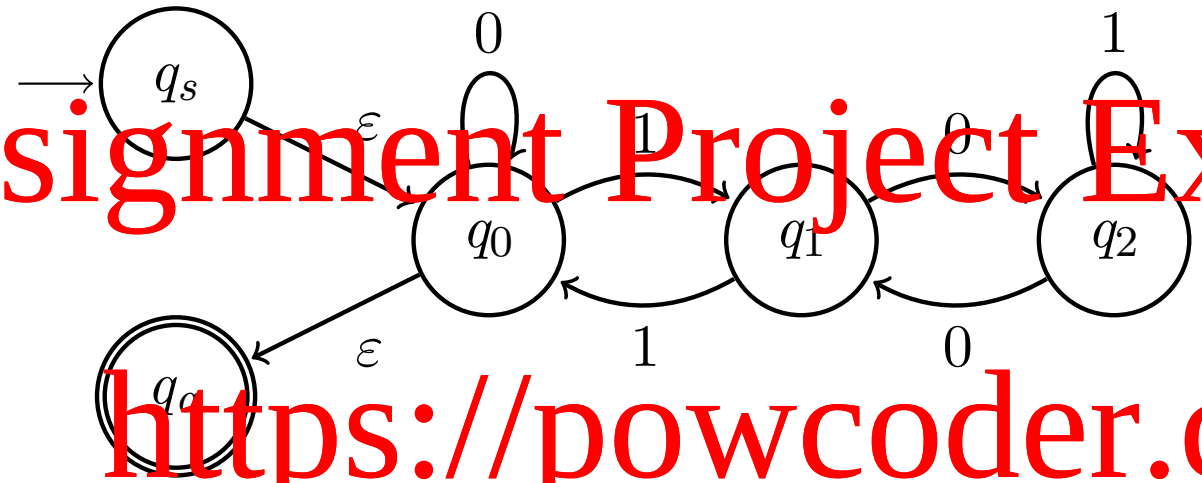


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0,$
- $(q_s, q_a) :$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

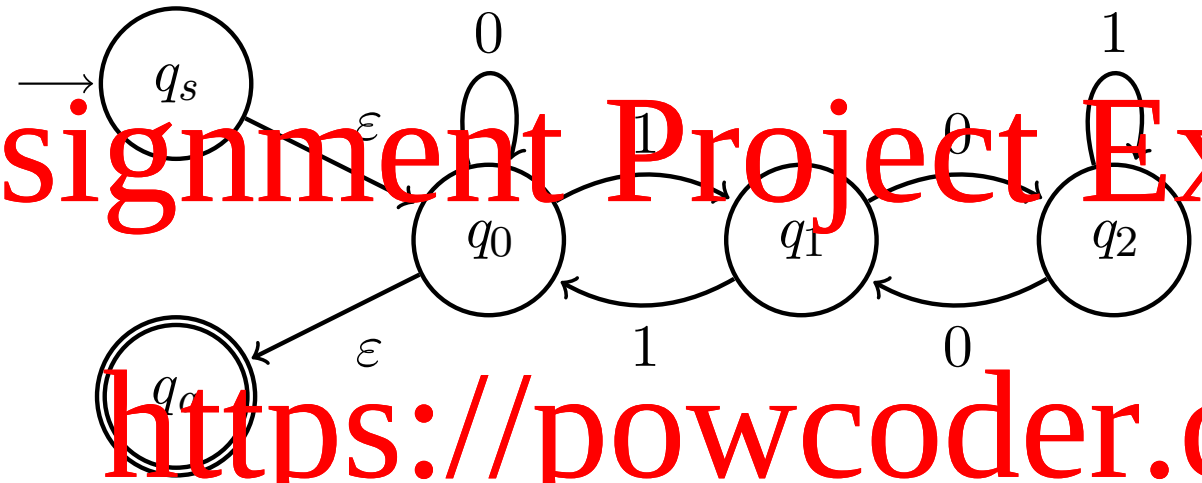


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1,$
- $(q_s, q_a) :$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

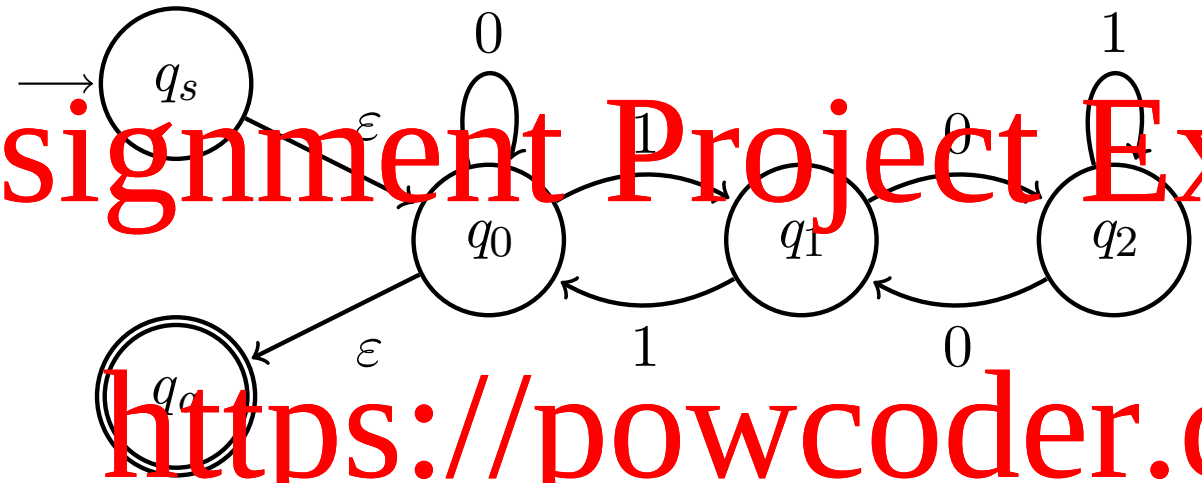


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset$
- $(q_s, q_a) :$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

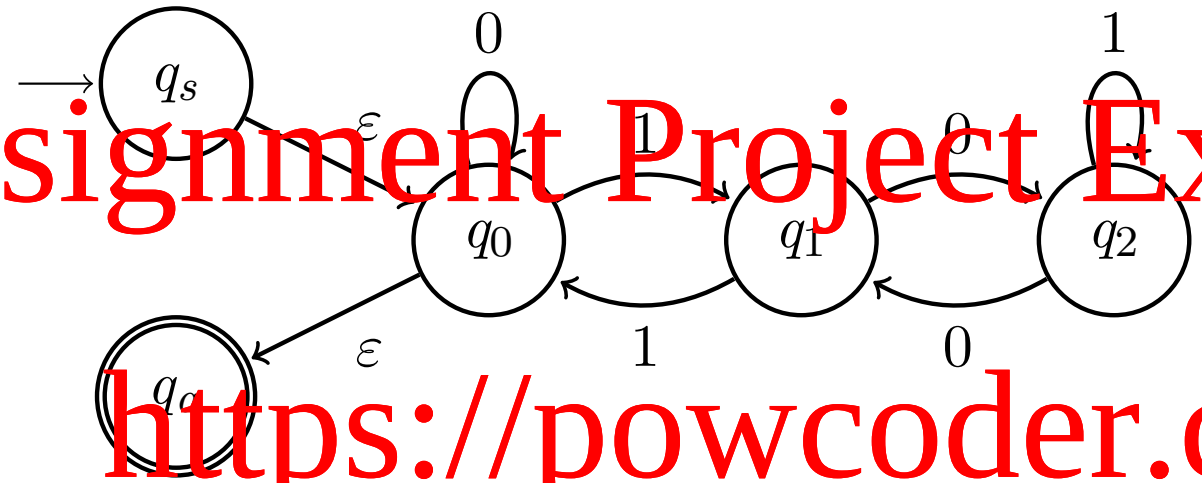


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset$
- $(q_s, q_a) :$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

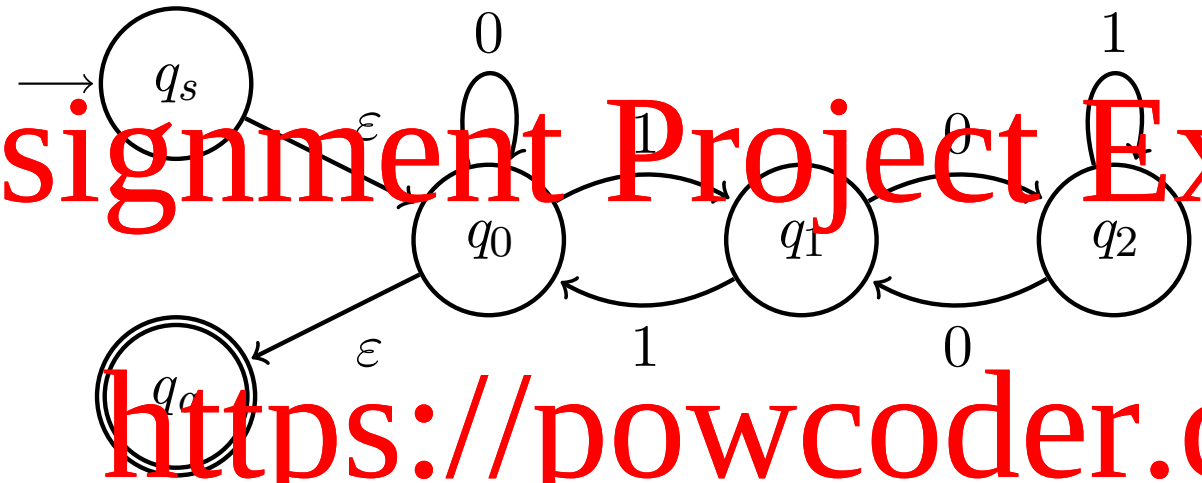


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset = 0^\star 1$
- $(q_s, q_a) :$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

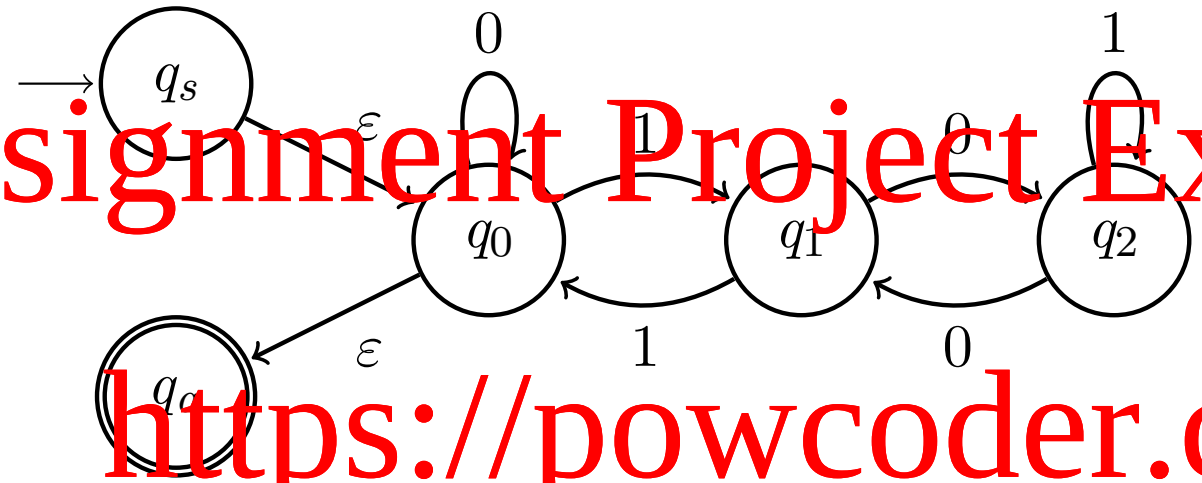


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset = 0^\star 1$
- $(q_s, q_a) : R_1 = \varepsilon,$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

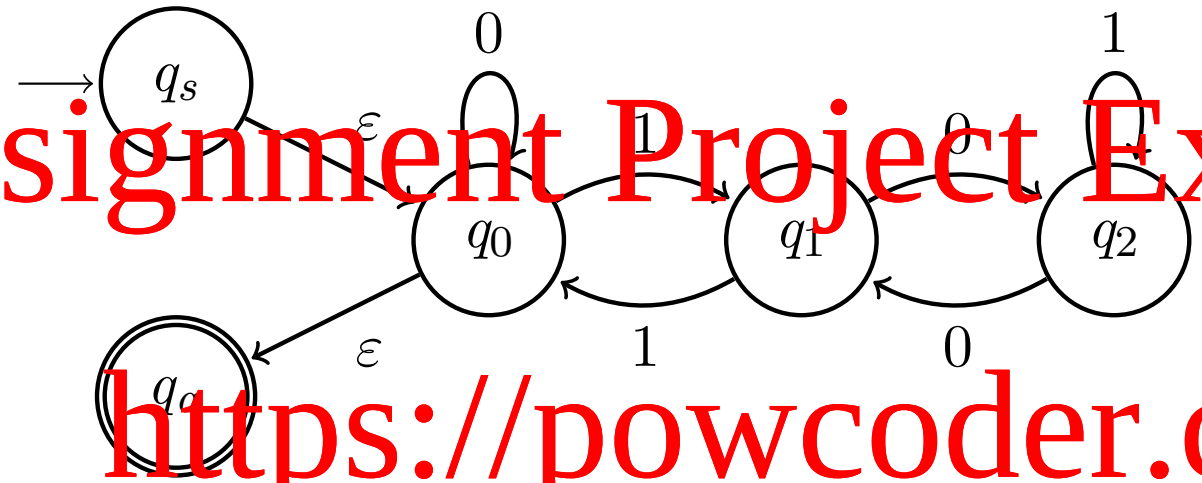


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset = 0^\star 1$
- $(q_s, q_a) : R_1 = \varepsilon, R_2 = 0,$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

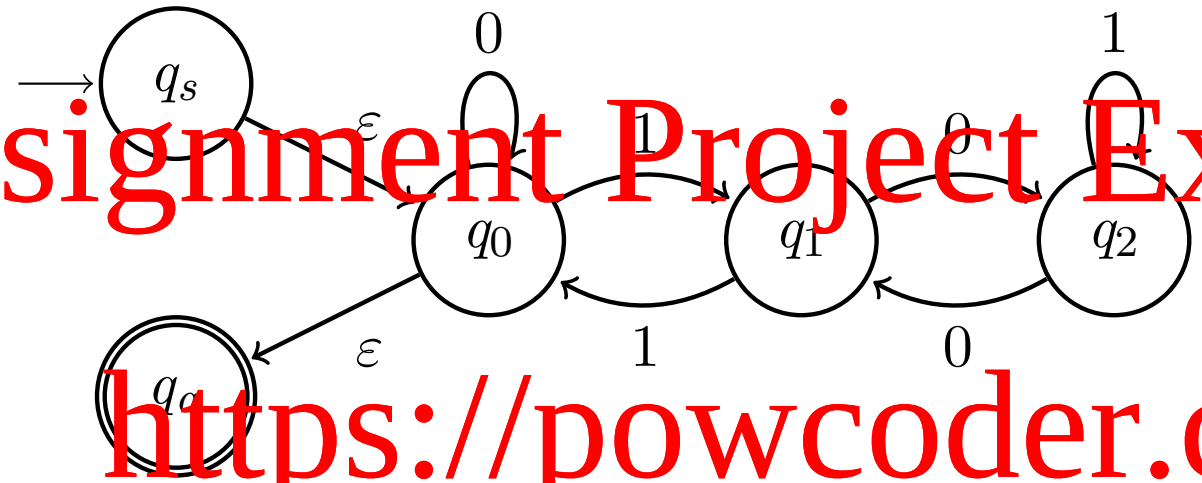


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset = 0^\star 1$
- $(q_s, q_a) : R_1 = \varepsilon, R_2 = 0, R_3 = \varepsilon,$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

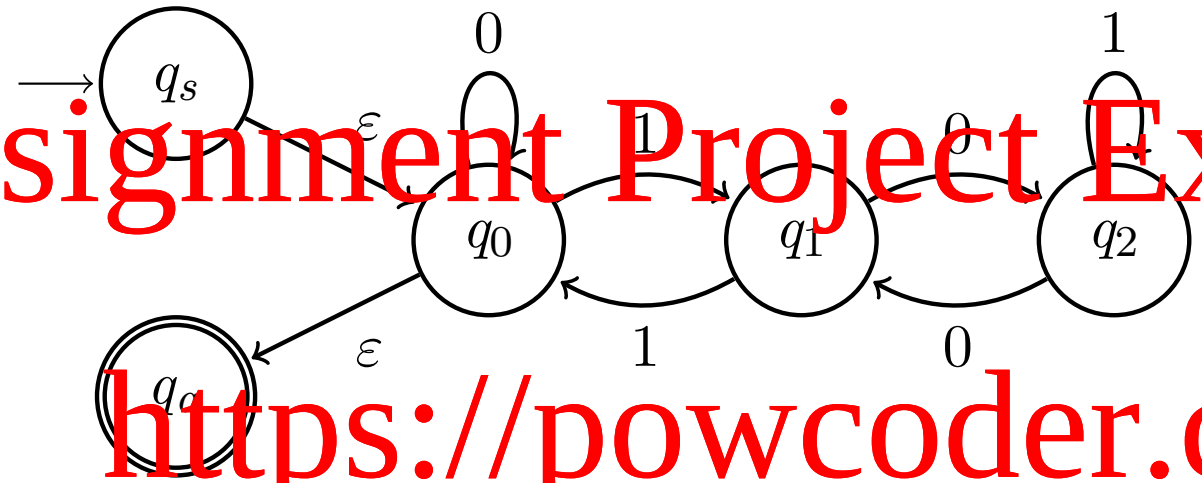


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset = 0^\star 1$
- $(q_s, q_a) : R_1 = \varepsilon, R_2 = 0, R_3 = \varepsilon, R_4 = \emptyset$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

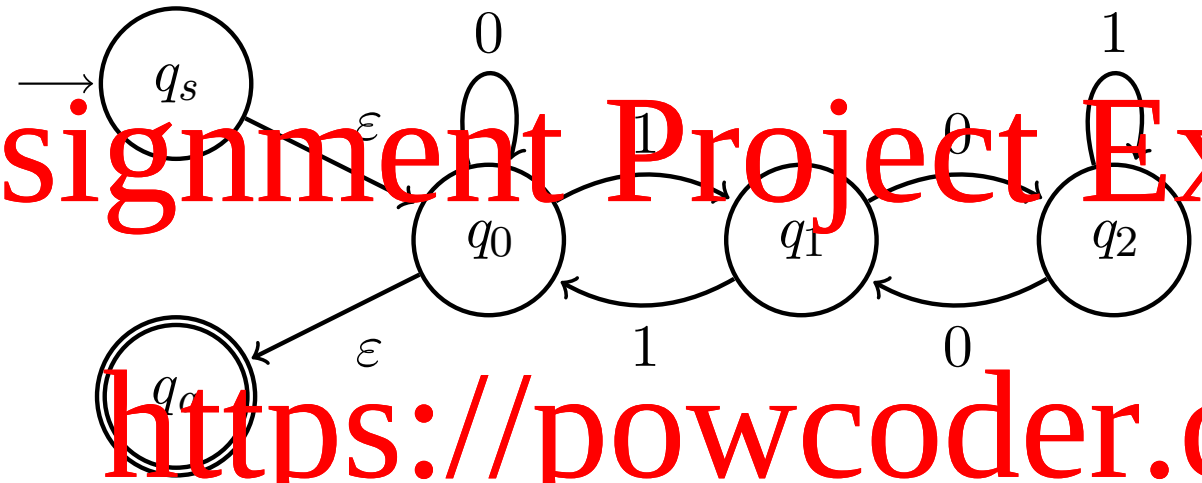


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset = 0^\star 1$
- $(q_s, q_a) : R_1 = \varepsilon, R_2 = 0, R_3 = \varepsilon, R_4 = \emptyset \Rightarrow \varepsilon 0^\star \varepsilon \mid \emptyset$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

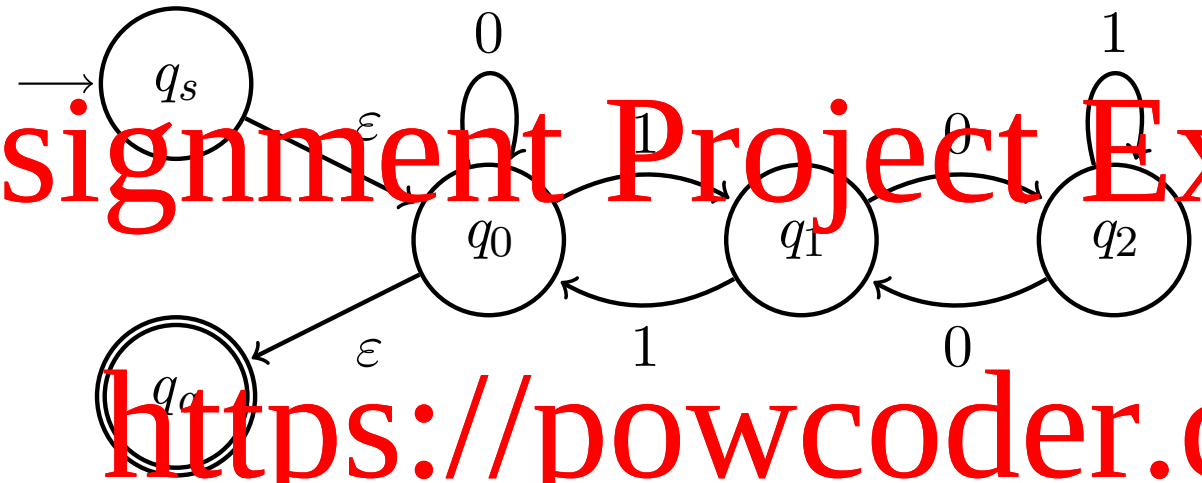


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset = 0^\star 1$
- $(q_s, q_a) : R_1 = \varepsilon, R_2 = 0, R_3 = \varepsilon, R_4 = \emptyset \Rightarrow \varepsilon 0^\star \varepsilon \mid \emptyset = 0^\star$
- $(q_1, q_1) :$
- $(q_1, q_a) :$

Example 2: divisible by 3

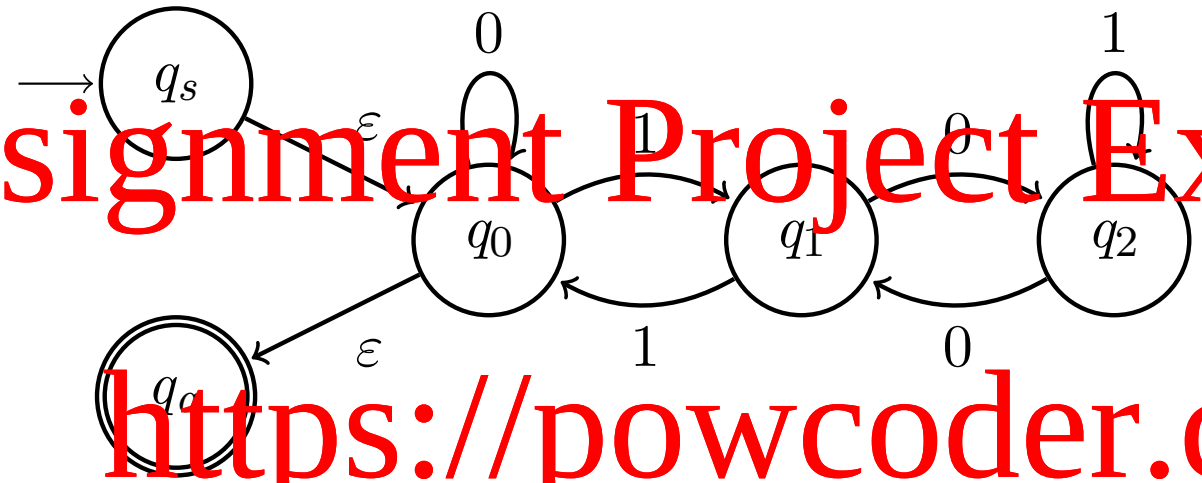


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset = 0^\star 1$
- $(q_s, q_a) : R_1 = \varepsilon, R_2 = 0, R_3 = \varepsilon, R_4 = \emptyset \Rightarrow \varepsilon 0^\star \varepsilon \mid \emptyset = 0^\star$
- $(q_1, q_1) : R_1 = 1, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow 10^\star 1 \mid \emptyset = 10^\star 1$
- $(q_1, q_a) :$

Example 2: divisible by 3

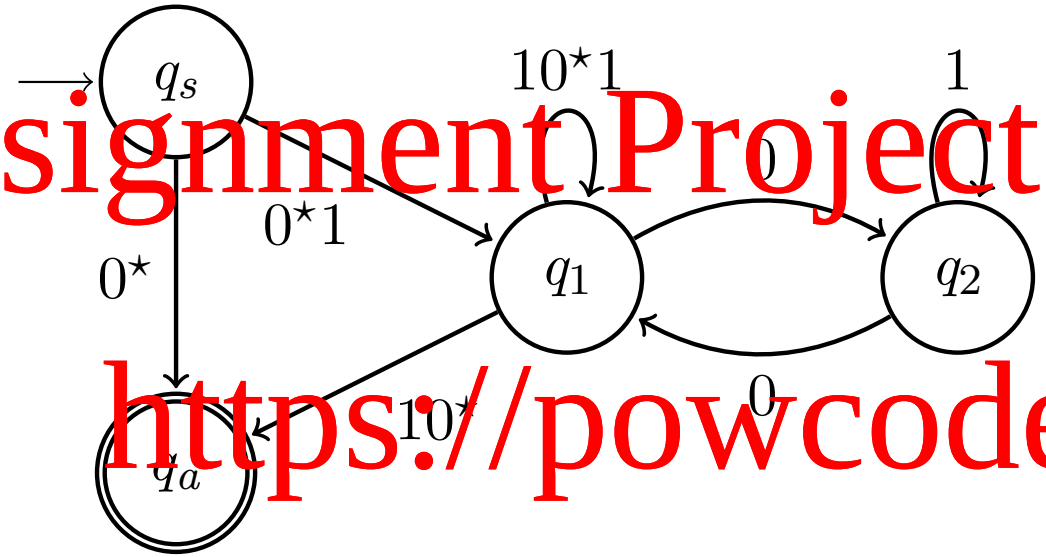


We can eliminate the states in any order. Eliminate q_0.

All pairs of states with transitions which are not $\emptyset$ through q_0:

- $(q_s, q_1) : R_1 = \varepsilon, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow \varepsilon 0^\star 1 \mid \emptyset = 0^\star 1$
- $(q_s, q_a) : R_1 = \varepsilon, R_2 = 0, R_3 = \varepsilon, R_4 = \emptyset \Rightarrow \varepsilon 0^\star \varepsilon \mid \emptyset = 0^\star$
- $(q_1, q_1) : R_1 = 1, R_2 = 0, R_3 = 1, R_4 = \emptyset \Rightarrow 10^\star 1 \mid \emptyset = 10^\star 1$
- $(q_1, q_a) : R_1 = 1, R_2 = 0, R_3 = \varepsilon, R_4 = \emptyset \Rightarrow 10^\star \varepsilon \mid \emptyset = 10^\star$

Example 2: divisible by 3

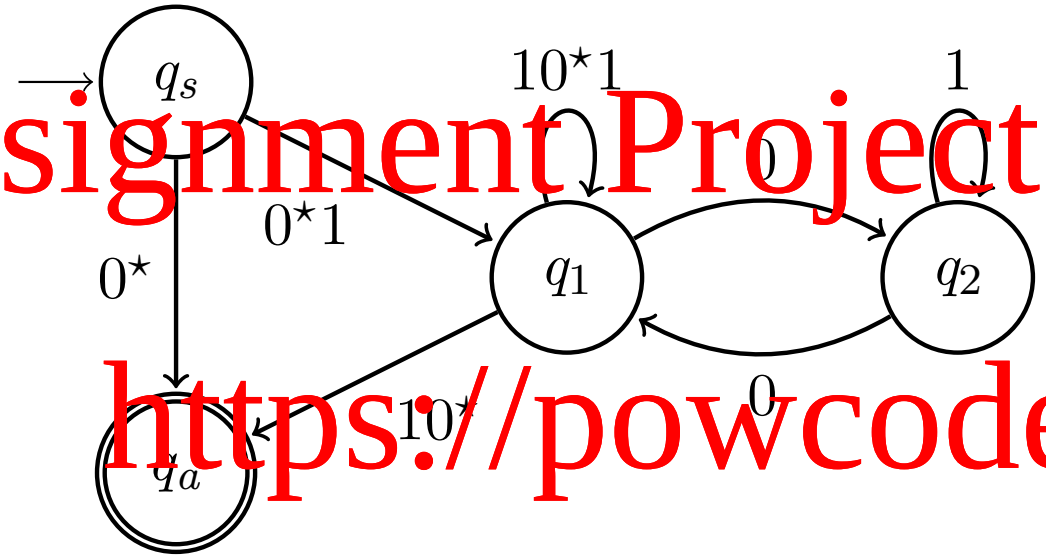


We can eliminate the states in any order. Eliminate q_2.

All pairs of states with transitions which are not $\emptyset$ through q_2:

- ▶

Example 2: divisible by 3

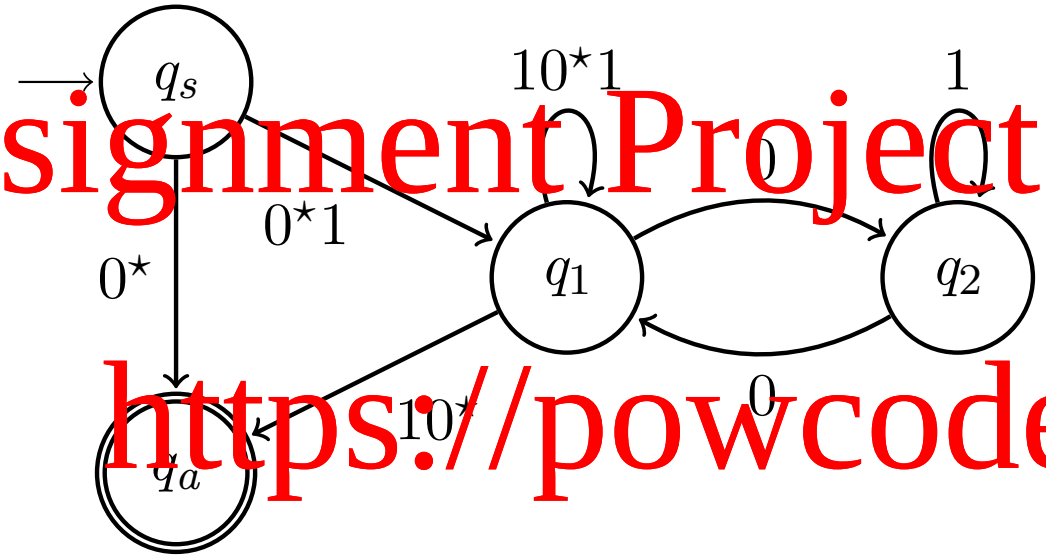


We can eliminate the states in any order. Eliminate q_2.

All pairs of states with transitions which are not $\emptyset$ through q_2:

- (q_1, q_1) :

Example 2: divisible by 3

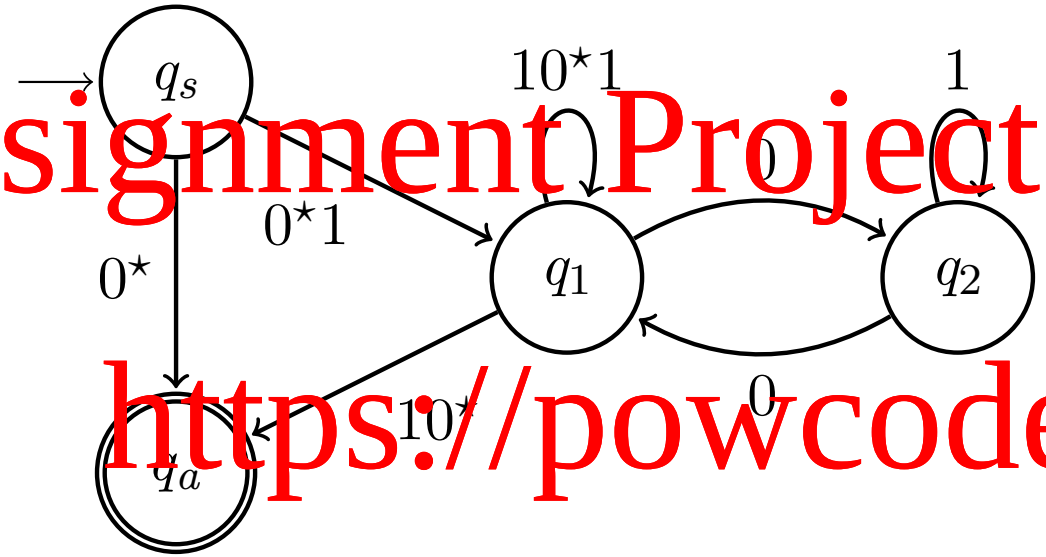


We can eliminate the states in any order. Eliminate q_2.

All pairs of states with transitions which are not $\emptyset$ through q_2:

- $(q_1, q_1) : R_1 = 0,$

Example 2: divisible by 3

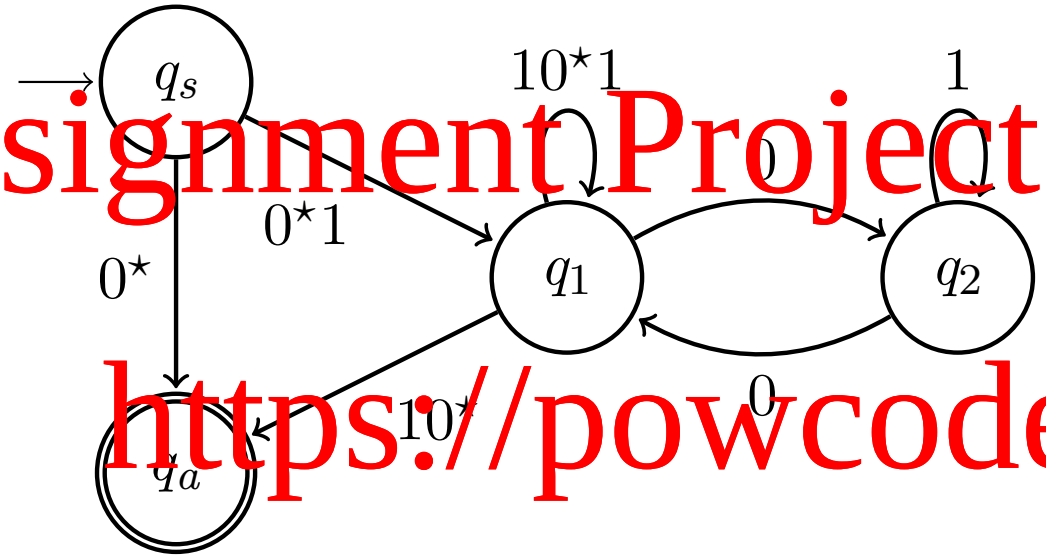


We can eliminate the states in any order. Eliminate q_2.

All pairs of states with transitions which are not $\emptyset$ through q_2:

- $(q_1, q_1) : R_1 = 0, R_2 = 1,$

Example 2: divisible by 3

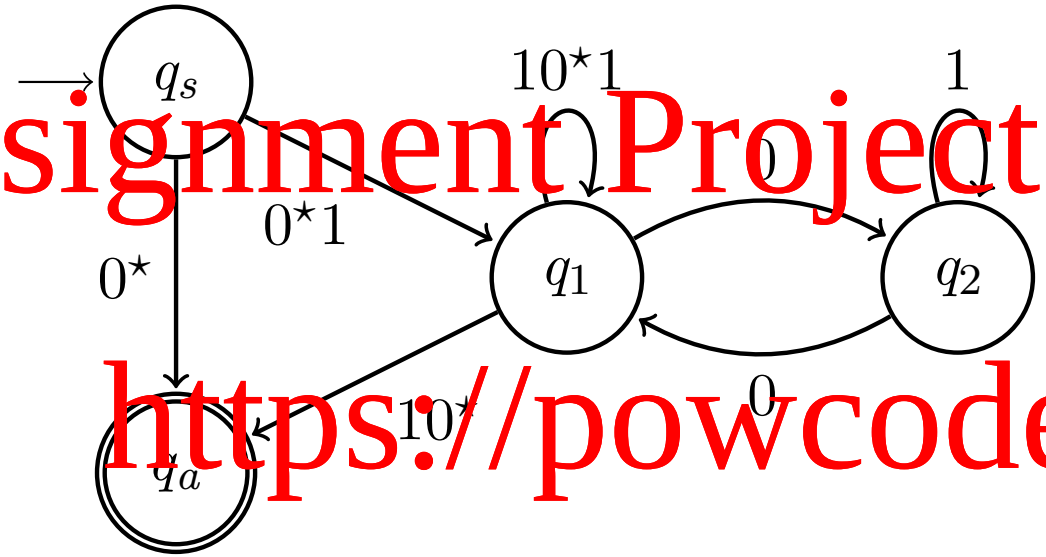


We can eliminate the states in any order. Eliminate q_2.

All pairs of states with transitions which are not $\emptyset$ through q_2:

- $(q_1, q_1) : R_1 = 0, R_2 = 1, R_3 = 0,$

Example 2: divisible by 3

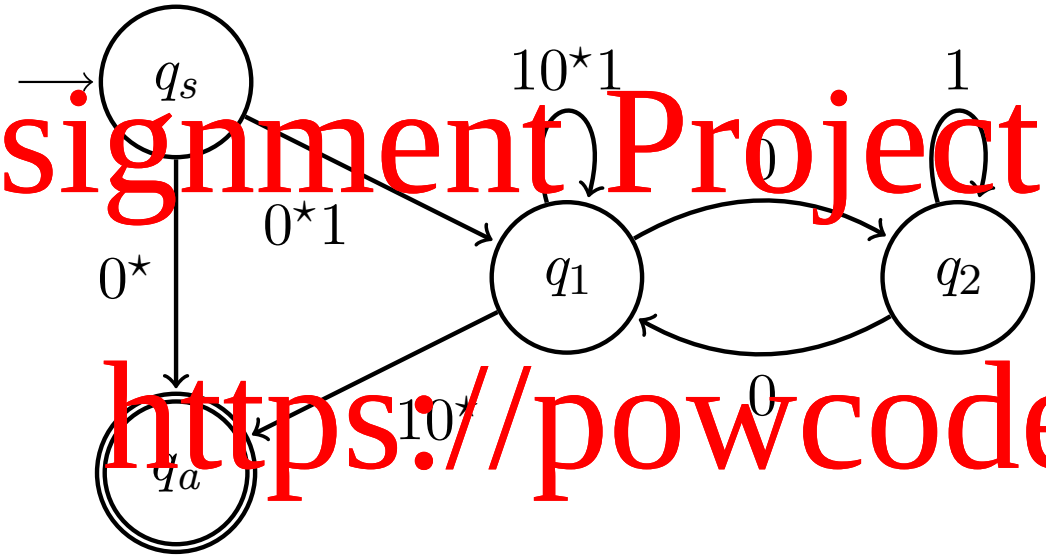


We can eliminate the states in any order. Eliminate q_2.

All pairs of states with transitions which are not $\emptyset$ through q_2:

- $(q_1, q_1) : R_1 = 0, R_2 = 1, R_3 = 0, R_4 = 10^\star1$

Example 2: divisible by 3



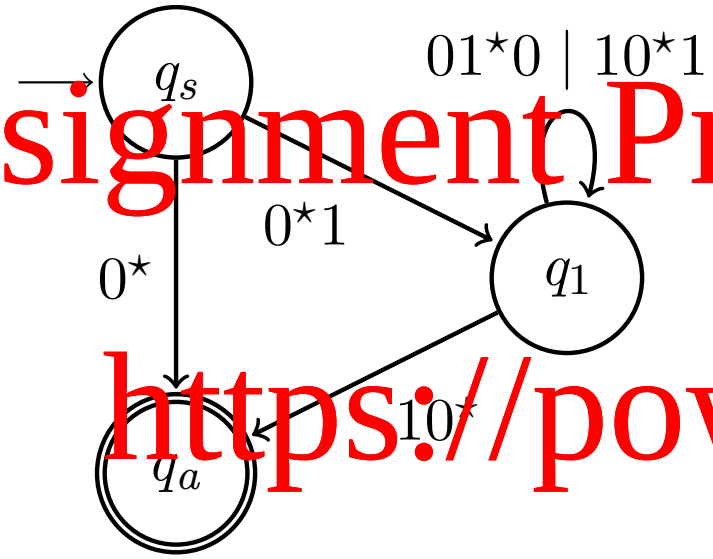
We can eliminate the states in any order. Eliminate q_2.

All pairs of states with transitions which are not $\emptyset$ through q_2:

- $(q_1, q_1) : R_1 = 0, R_2 = 1, R_3 = 0, R_4 = 10^\star1 \Rightarrow 01^\star0 \mid 10^\star1$

Much easier!

Example 2: divisible by 3

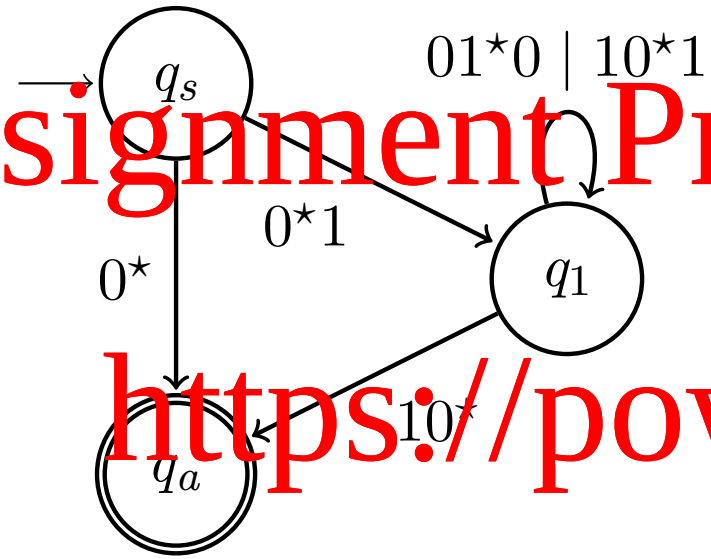


There is only q_1 left to remove.

All pairs of states with transitions which are not $\emptyset$ through q_1:

- ▶

Example 2: divisible by 3

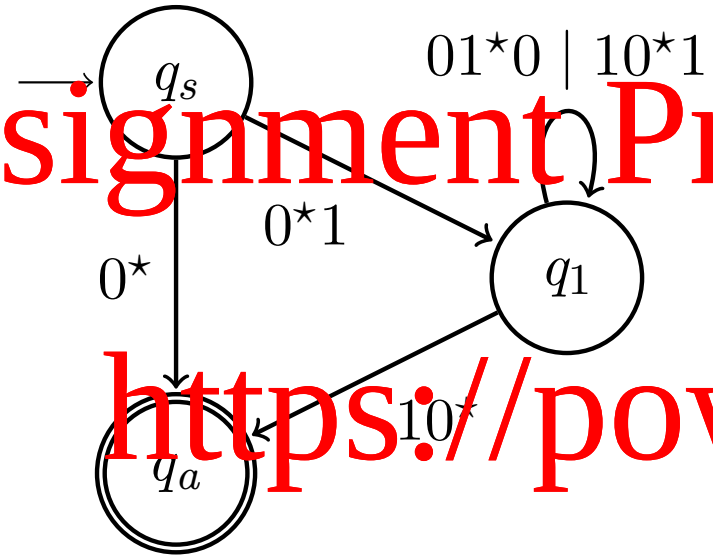


There is only q_1 left to remove.

All pairs of states with transitions which are not $\emptyset$ through q_1:

- (q_s, q_a) :

Example 2: divisible by 3

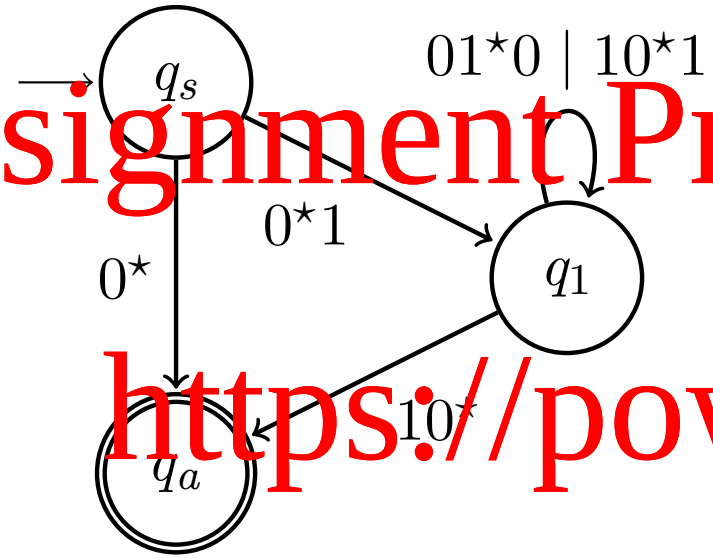


There is only q_1 left to remove.

All pairs of states with transitions which are not $\emptyset$ through q_1:

- $(q_s, q_a) : R_1 = 0^\star1,$

Example 2: divisible by 3

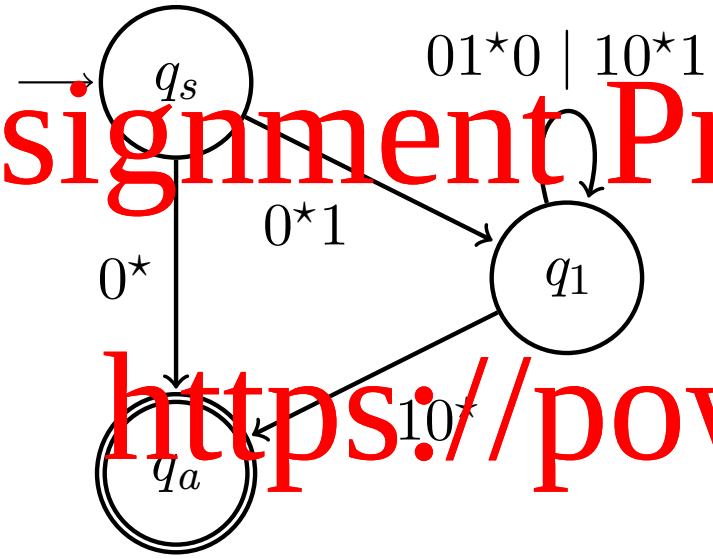


There is only q_1 left to remove.

All pairs of states with transitions which are not $\emptyset$ through q_1:

- $(q_s, q_a) : R_1 = 0^*1, R_2 = 01^*0 \mid 10^*1,$

Example 2: divisible by 3

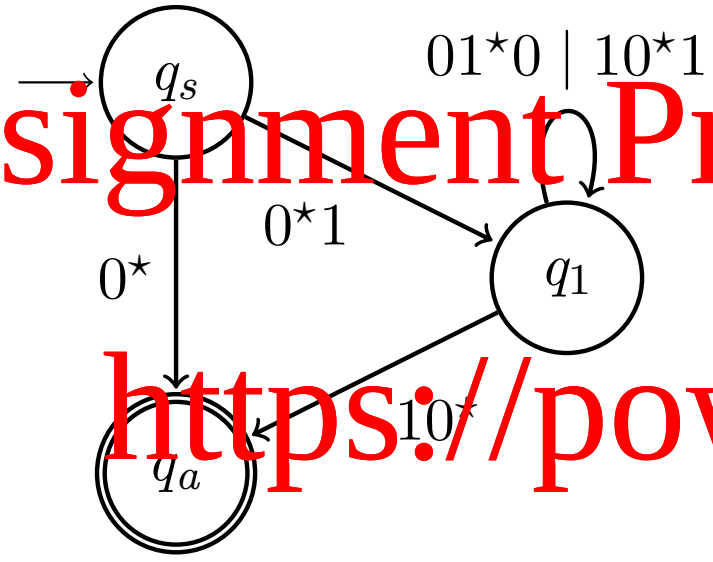


There is only q_1 left to remove.

All pairs of states with transitions which are not $\emptyset$ through q_1:

- $(q_s, q_a) : R_1 = 0^\star1, R_2 = 01^\star0 \mid 10^\star1, R_3 = 10^\star,$

Example 2: divisible by 3

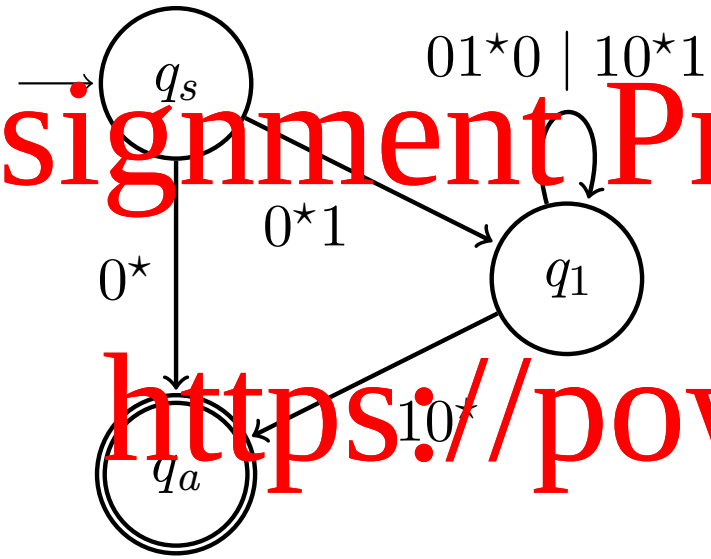


There is only q_1 left to remove.

All pairs of states with transitions which are not $\emptyset$ through q_1:

- $(q_s, q_a) : R_1 = 0^\star1, R_2 = 01^\star0 \mid 10^\star1, R_3 = 10^\star, R_4 = 0^\star$

Example 2: divisible by 3

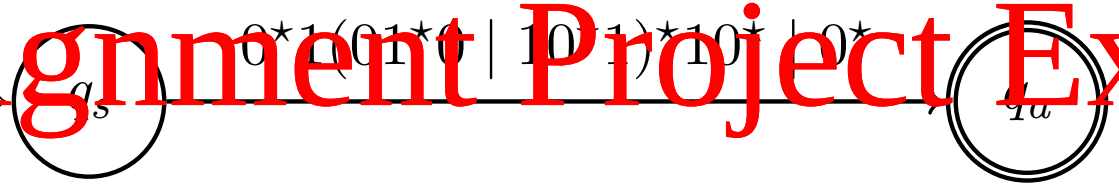


There is only q_1 left to remove.

All pairs of states with transitions which are not $\emptyset$ through q_1:

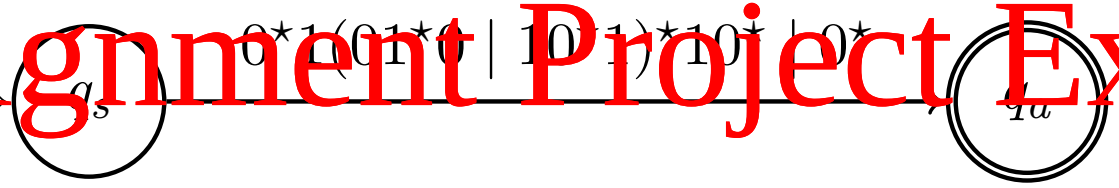
- $(q_s, q_a) : R_1 = 0^\star1, R_2 = 01^\star0 \mid 10^\star1, R_3 = 10^\star, R_4 = 0^\star$
  so $(R_1)(R_2)^\star(R_3) \mid (R_4) = 0^\star1(01^\star0 \mid 10^\star1)^\star10^\star \mid 0^\star$

Example 2: divisible by 3



So, $0^\star 1(01^\star 0 \mid 10^\star 1)^\star 10^\star \mid 0^\star$ is a RegEx which recognises binary strings which are divisible by 3.

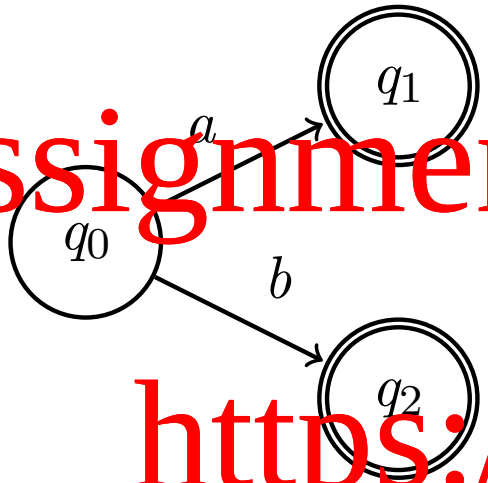
Example 2: divisible by 3



So, $0^\star 1(01^\star 0 \mid 10^\star 1)^\star 10^\star \mid 0^\star$ is a RegEx which recognises binary strings which are divisible by 3.

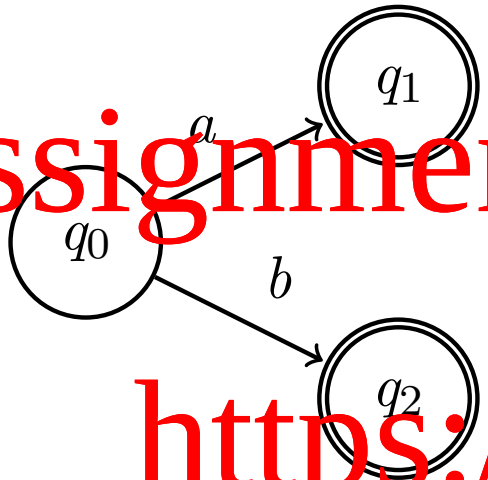
Eliminating states in a different order can result in a different, but equivalent RegEx.
e.g. eliminating q_2, q_1, then q_0 yields: $(1(01^\star 0)^\star 1|0)^\star$

Example 3: Multiple accept states

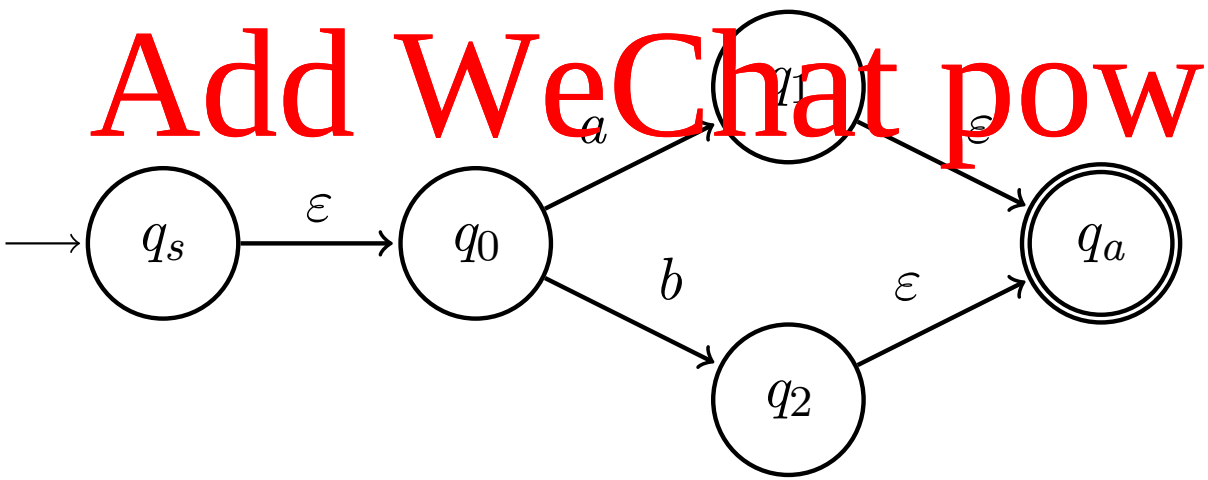


As GNFA:

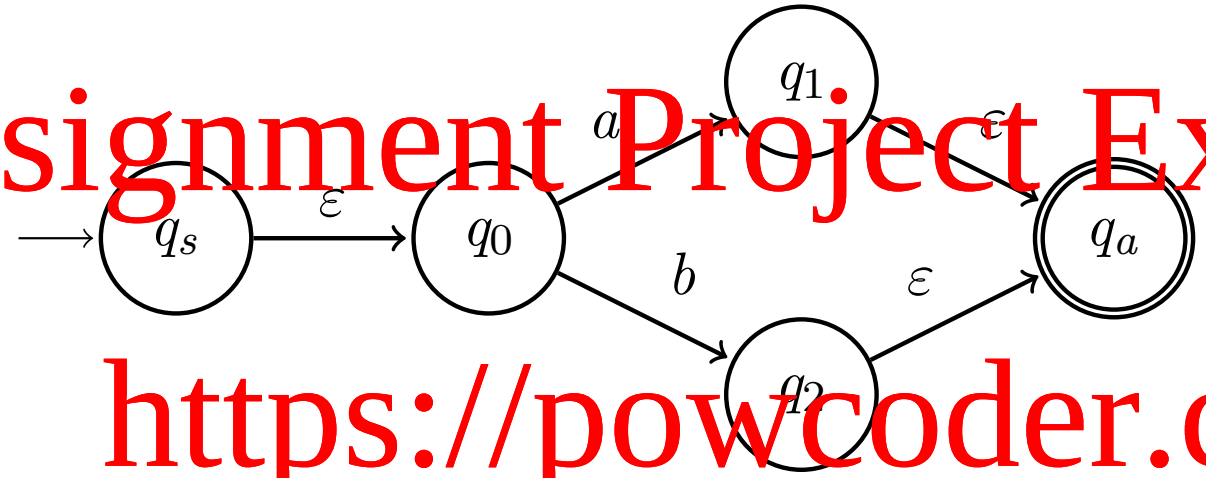
Example 3: Multiple accept states



As GNFA:

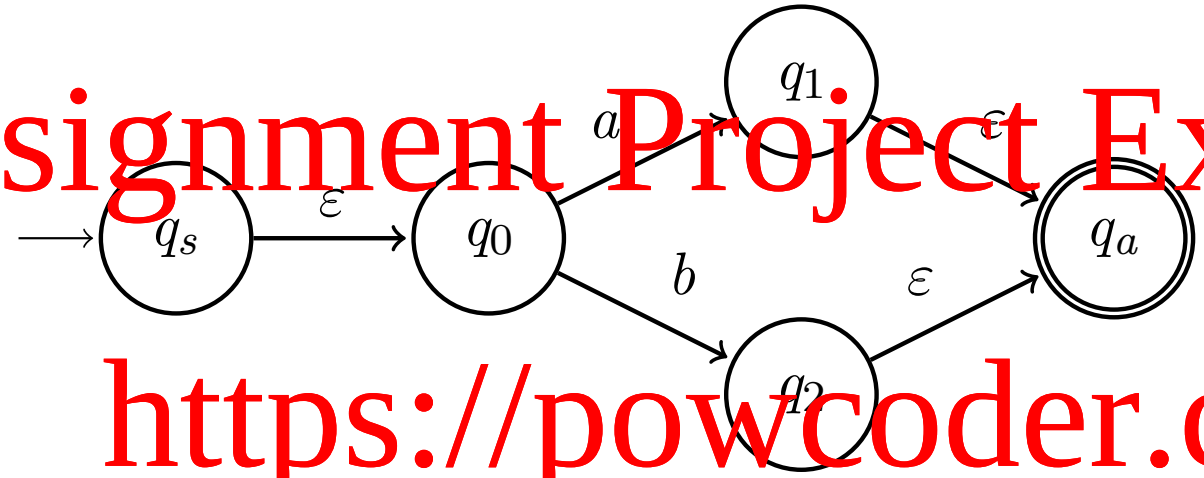


Example 3: Multiple accept states

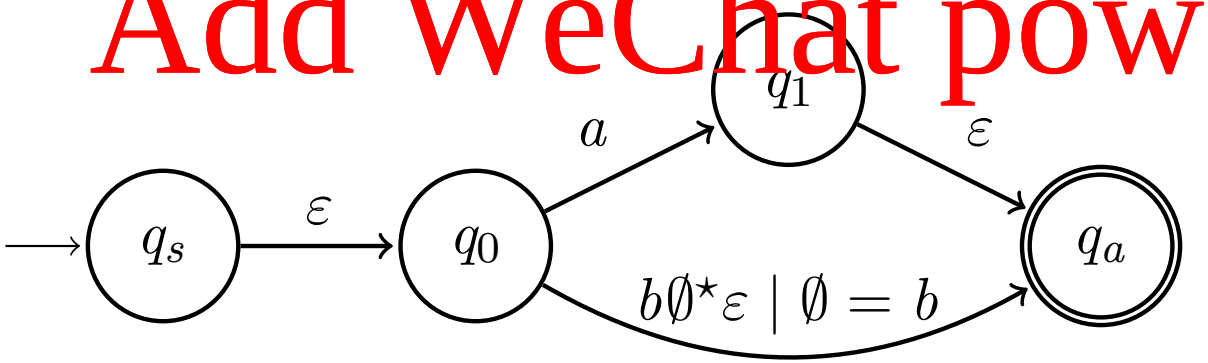


Eliminate q_2:

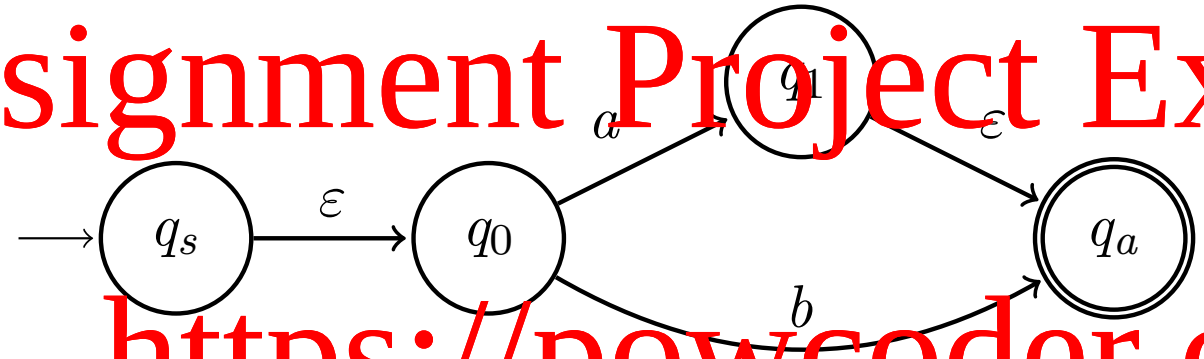
Example 3: Multiple accept states



Eliminate q_2:

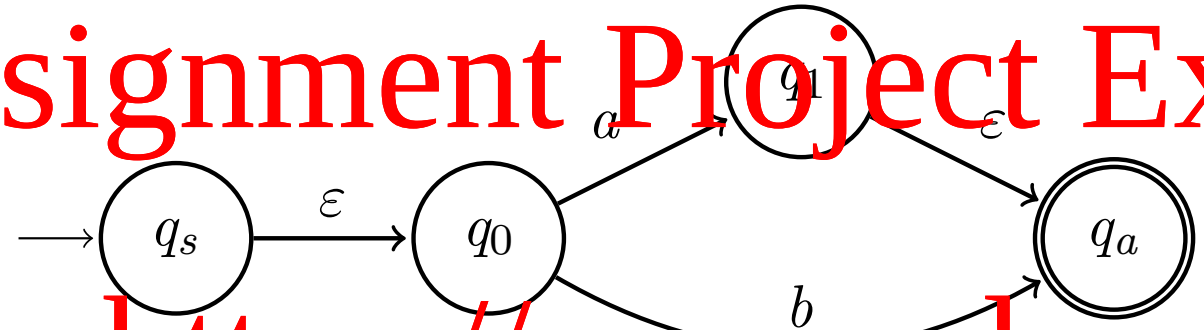


Example 3: Multiple accept states

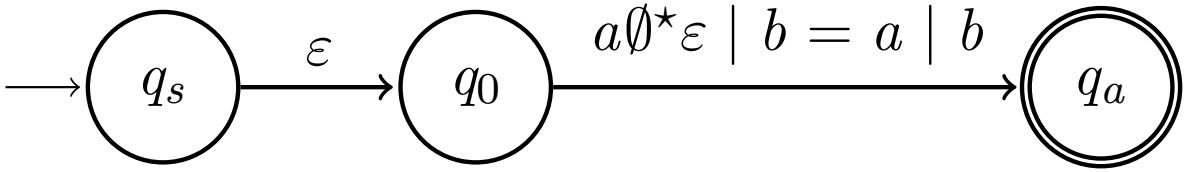


Eliminate q_1:

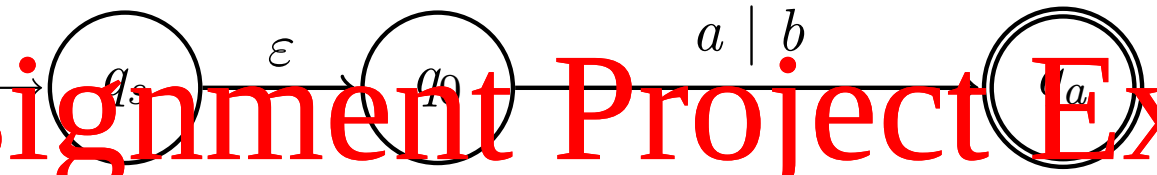
Example 3: Multiple accept states



Eliminate q_1:



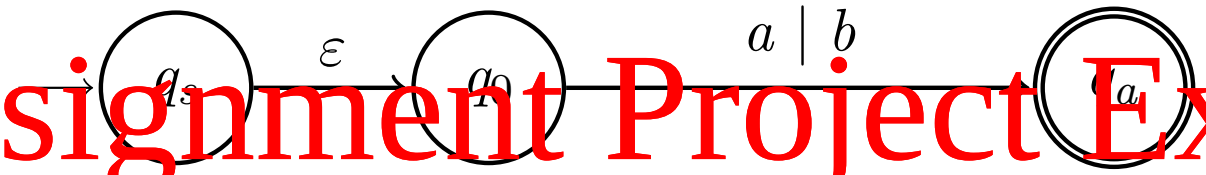
Example 3: Multiple accept states



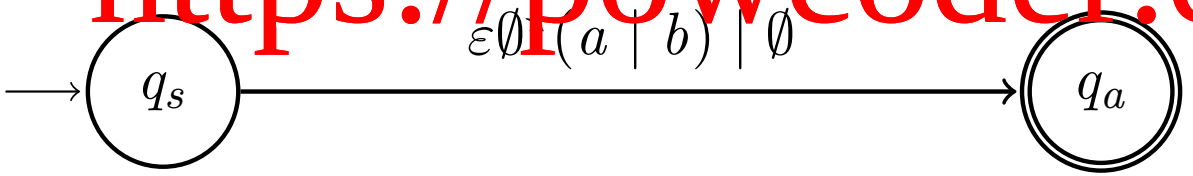
Eliminate q_0:

Simplify, to get the expected:

Example 3: Multiple accept states

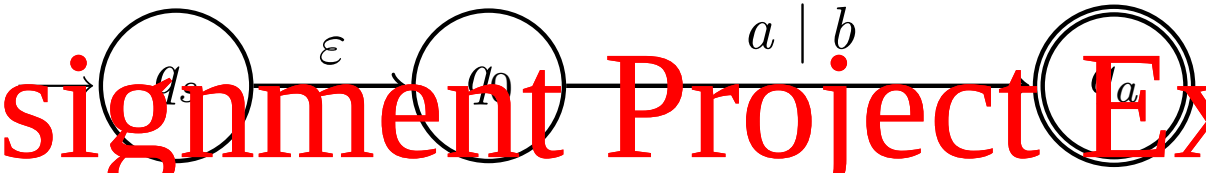


Eliminate q_0:

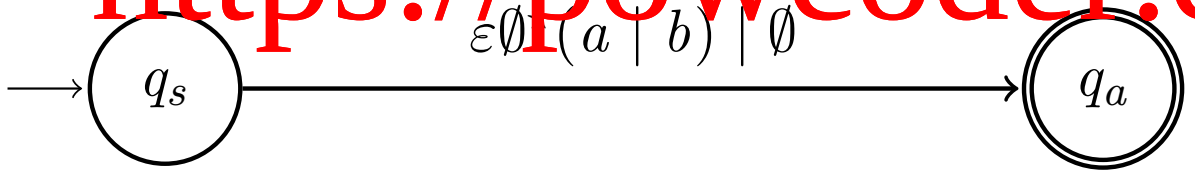


Simplify, to get the expected:

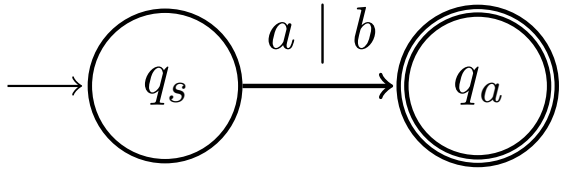
Example 3: Multiple accept states



Eliminate q_0:



Simplify, to get the expected:



Summary

Algorithm to convert an NFA to a RegEx:

1. Convert the NFA to a Generalised NFA (GNFA)
   - Only one start state, no incoming transitions
   - Only one accept state, no outgoing transitions
   - Transitions described by RegEx

Summary

Algorithm to convert an NFA to a RegEx:

1. Convert the NFA to a Generalised NFA (GNFA)
   - Only one start state, no incoming transitions
   - Only one accept state, no outgoing transitions
   - Transitions described by RegEx
2. Progressively *eliminate* all states between start and accept
   - For each pair (q_i, q_j) where $q_i, q_j \in Q \setminus \{q_{elim}\}$
   - Replace arrow $q_i \to q_j$ with $((R_1)(R_2)^\star(R_3) \mid (R_4))$, where
     - R_1 is the RegEx on transition $q_i \to q_{elim}$
     - R_2 is the RegEx on transition $q_{elim} \to q_{elim}$
     - R_3 is the RegEx on transition $q_{elim} \to q_j$
     - R_4 is the RegEx on transition $q_i \to q_j$

Summary

Algorithm to convert an NFA to a RegEx:

1. Convert the NFA to a Generalised NFA (GNFA)
   - Only one start state, no incoming transitions
   - Only one accept state, no outgoing transitions
   - Transitions described by RegEx
2. Progressively *eliminate* all states between start and accept
   - For each pair (q_i, q_j) where $q_i, q_j \in Q \setminus \{q_{elim}\}$
   - Replace arrow $q_i \to q_j$ with $((R_1)(R_2)^\star(R_3) \mid (R_4))$, where
     - R_1 is the RegEx on transition $q_i \to q_{elim}$
     - R_2 is the RegEx on transition $q_{elim} \to q_{elim}$
     - R_3 is the RegEx on transition $q_{elim} \to q_j$
     - R_4 is the RegEx on transition $q_i \to q_j$
3. The transition between the start and the accept state is now a regular expression describing L

Outline

- Regular Expressions

- Equivalence of FA and Regular Expressions

- **Proving if a language is, *or is not*, regular**

Proving if a language is regular *or not*

Recall the definition:

A language is *regular* if and only if there exists a finite automaton which recognises it.

Suppose a language is regular. How can we prove it?

Proving if a language is regular *or not*

Recall the definition:

A language is *regular* if and only if there exists a finite automaton which recognises it.

Suppose a language is regular. How can we prove it?

- Devise an NFA which recognises it, or
- Devise a DFA which recognises it, or
- Devise a RegEx which describes it.

Proving if a language is regular *or not*

Recall the definition:

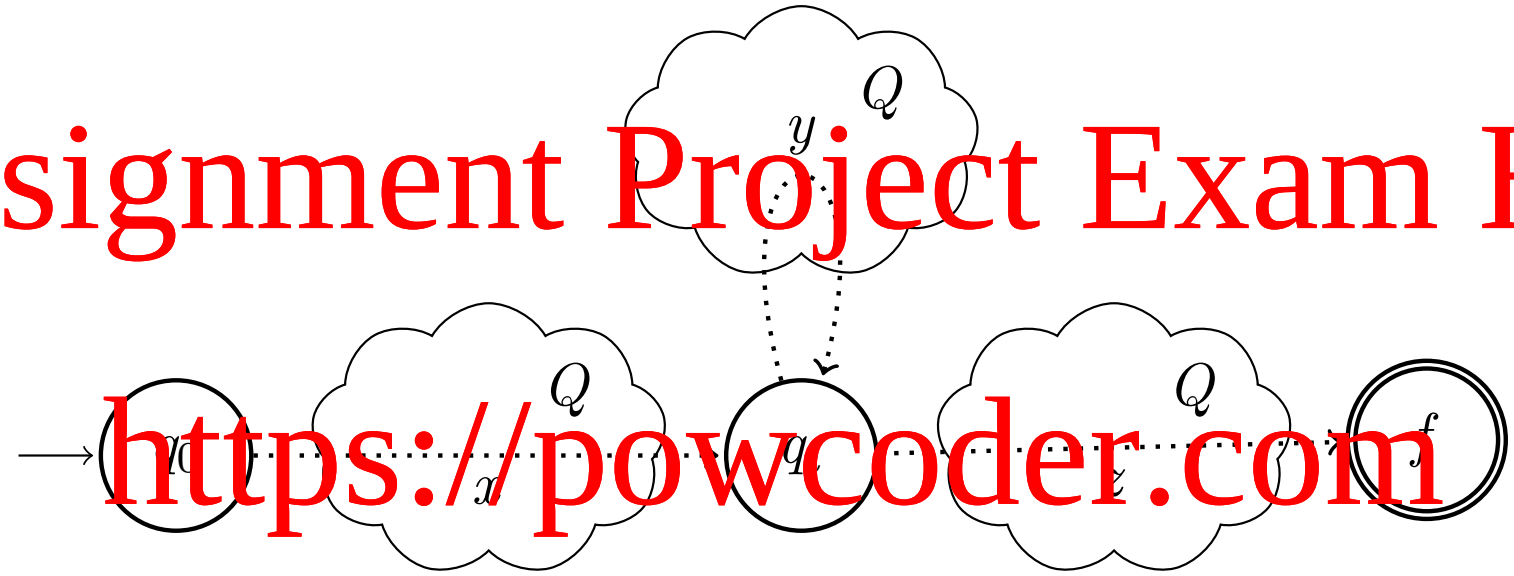
A language is *regular* if and only if there exists a finite automaton which recognises it.

Suppose a language is regular. How can we prove it?

- Devise an NFA which recognises it, or
- Devise a DFA which recognises it, or
- Devise a RegEx which describes it

What if the language was not regular? How can we *prove* that no suitable automata exists? We can use the *pumping lemma for regular langauges*

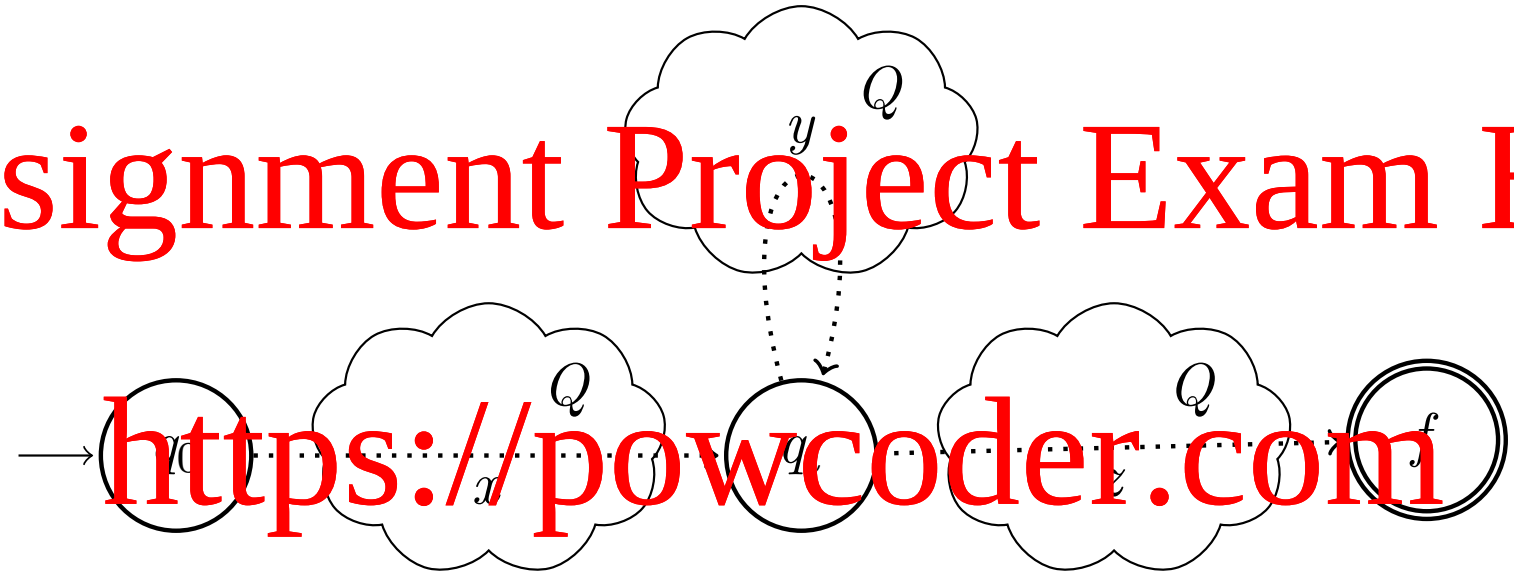
Pumping s using y



Suppose some string s exists, passing through M:

- From q_0 to q_i using a string x, then
- from q_i back to q_i, using a string $y \neq \varepsilon$, then
- from q_i to some accept state $f \in F$, using a string z

Pumping s using y

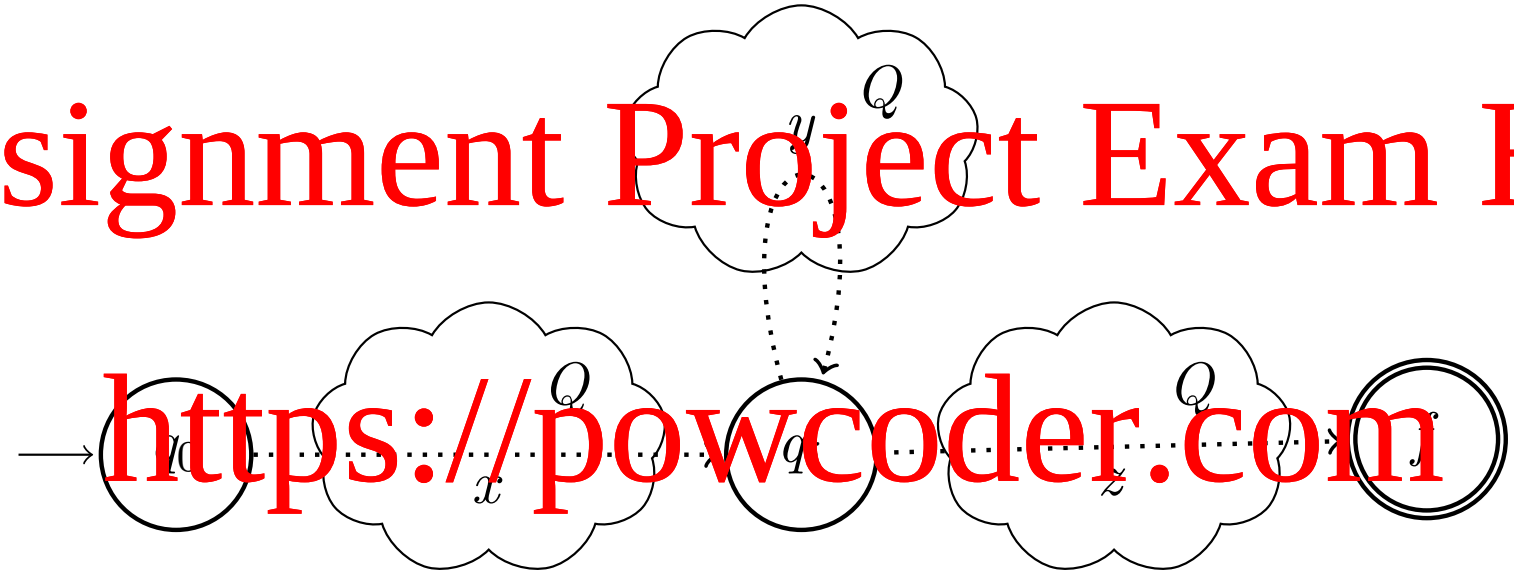


Suppose some string s exists, passing through M:

- From q_0 to q_i using a string x, then
- from q_i back to q_i, using a string $y \neq \varepsilon$, then
- from q_i to some accept state $f \in F$, using a string z

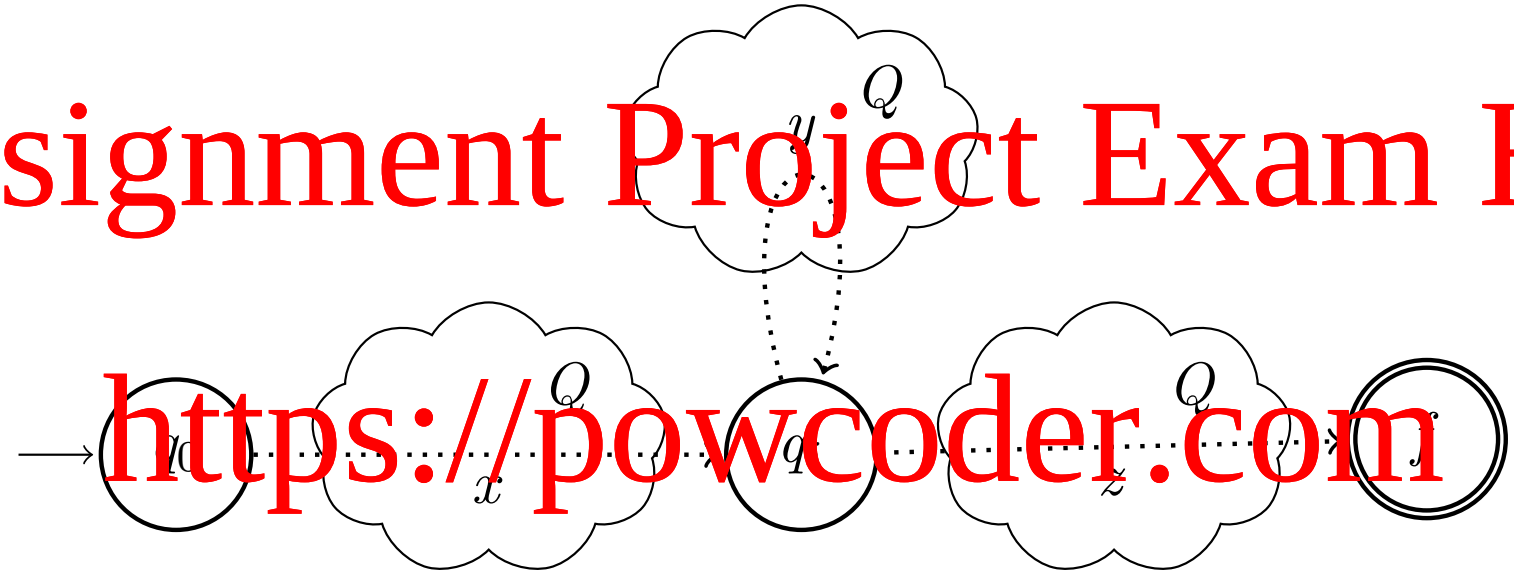
Then $s = xyz \in L$, and $xy^k z \in L$ for all $k \geq 0$
e.g. if $x = aa, y = b, z = c$, then $\{aac, aabc, aabbc, ...\} \subseteq L$

Long strings in finite automata



Suppose M is a DFA with n states, and $s \in L(M)$, but $|s| \geq n + 1$

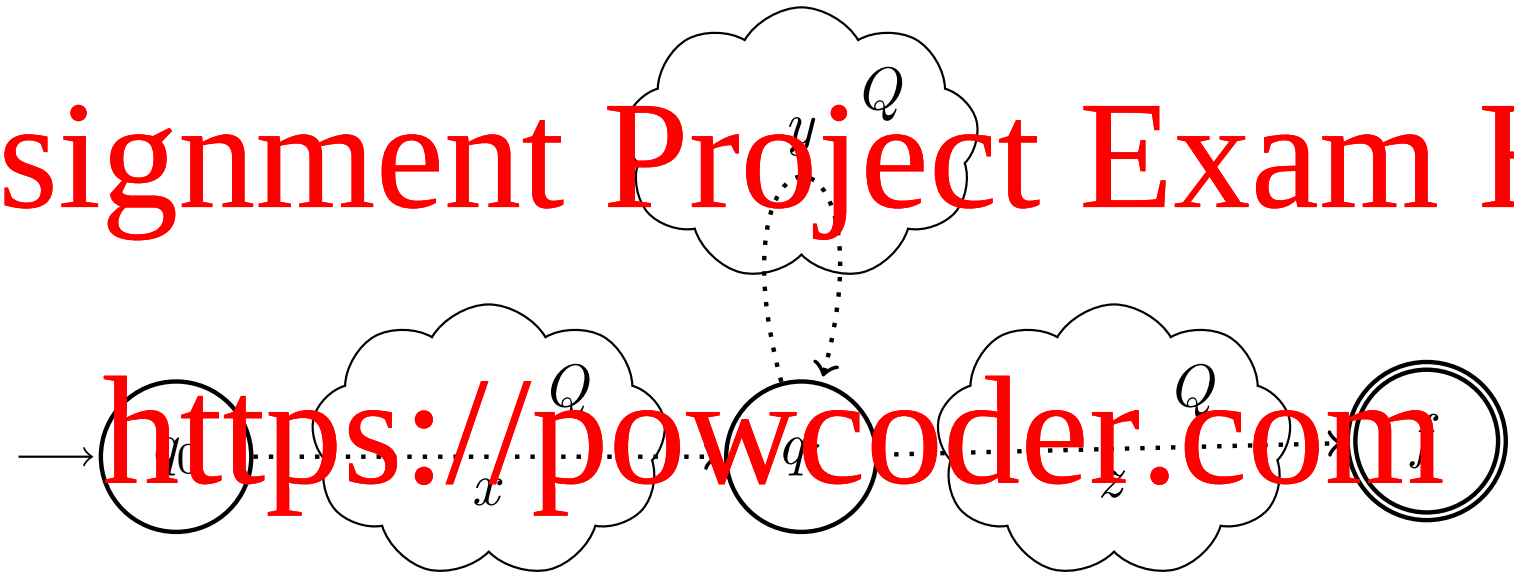
Long strings in finite automata



Suppose M is a DFA with n states, and $s \in L(M)$, but $|s| \geq n+1$

Then there exists at least one state which was visited more than once (q_i), and a substring $y \neq \varepsilon$ corresponding to the path followed between the two of those visits.

Long strings in finite automata



Suppose M is a DFA with n states, and $s \in L(M)$, but $|s| \geq n+1$

Then there exists at least one state which was visited more than once (q_i), and a substring $y \neq \varepsilon$ corresponding to the path followed between the two of those visits.

Hence we can *pump* s to find other strings in the language

Finite automata, infinite languages

All finite languages are regular. (Why?)

Finite automata, infinite languages

All finite languages are regular. (Why?)

Suppose L is an *infinite* regular language.

- L is regular, so a DFA M exists which recognises it.
- L is an infinite set over a finite alphabet, so it must include arbitrarily long words.
- Therefore words exist which can be *pumped*

Pumping lemma for regular languages

If L is a regular language, then there exists a number p (the *pumping length*) such that for any string $s \in L$ of length at least p, then s may be divided into three pieces, $s = xyz$, such that:

1. $xy^kz \in L$ for all $k \geq 0$ (i.e. we can use y to *pump* the string)
2. $|y| > 0$ (i.e. $y \neq \varepsilon$)
3. $|xy| \leq p$

If a language does not satisfy this lemma, then it cannot be regular

Using the pumping lemma

We can (only) use the pumping lemma to prove that a language is *not* regular

Proof by contradiction:

1. Assume that the langage is regular
2. Choose an appropriate string in the language
   - This is often the hardest part of the proof
   - We must find one which will allow us to:
3. Apply the lemma to find a contradiction
4. Thereby deduce that the assumption is false
   - i.e. The language cannot be regular

Example

Show that $L = \{a^n b^n | n \geq 0\}$ is not regular

Example

Show that $L = \{a^n b^n | n \geq 0\}$ is not regular

1. Assumption: L is regular.

Example

Show that $L = \{a^n b^n | n \geq 0\}$ is not regular

1. Assumption: L is regular.
2. Then there exists some $p \geq 0$ satisfying the pumping lemma.

Example

Show that $L = \{a^n b^n | n \geq 0\}$ is not regular

1. Assumption: L is regular.
2. Then there exists some $p \geq 0$ satisfying the pumping lemma.
3. Let $s = a^p b^p$ (note: the same p as above!)

Example

Show that $L = \{a^n b^n | n \geq 0\}$ is not regular

1. Assumption: L is regular.
2. Then there exists some $p \geq 0$ satisfying the pumping lemma.
3. Let $s = a^p b^p$ (note: the same p as above!)
4. $|s| \geq p$ therefore we can write $s = xyz$ such that
   - 4.1 $xy^k z \in L$ for all $k \geq 0$ (i.e. we can *pump* y)
   - 4.2 $|y| > 0$
   - 4.3 $|xy| \leq p$

Example

Show that $L = \{a^n b^n | n \geq 0\}$ is not regular

1. Assumption: L is regular.
2. Then there exists some $p \geq 0$ satisfying the pumping lemma.
3. Let $s = a^p b^p$ (note: the same p as above!)
4. $|s| \geq p$ therefore we can write $s = xyz$ such that
   - 4.1 $xy^k z \in L$ for all $k \geq 0$ (i.e. we can *pump* y)
   - 4.2 $|y| > 0$
   - 4.3 $|xy| \leq p$
5. Therefore $y = a^i$ for some $0 < i \leq p$ (at least one a, no b's)

Example

Show that $L = \{a^n b^n | n \geq 0\}$ is not regular

1. Assumption: L is regular.
2. Then there exists some $p \geq 0$ satisfying the pumping lemma.
3. Let $s = a^p b^p$ (note: the same p as above!)
4. $|s| \geq p$ therefore we can write $s = xyz$ such that
   - 4.1 $xy^k z \in L$ for all $k \geq 0$ (i.e. we can *pump* y)
   - 4.2 $|y| > 0$
   - 4.3 $|xy| \leq p$
5. Therefore $y = a^i$ for some $0 < i \leq p$ (at least one a, no b's)
6. Let $k = 2$. Then $xy^k z$ has more a's than b's, so $xy^k z \notin L$

Example

Show that $L = \{a^n b^n | n \geq 0\}$ is not regular

1. Assumption: L is regular.
2. Then there exists some $p \geq 0$ satisfying the pumping lemma.
3. Let $s = a^p b^p$ (note: the same p as above!)
4. $|s| \geq p$ therefore we can write $s = xyz$ such that
   - 4.1 $xy^k z \in L$ for all $k \geq 0$ (i.e. we can *pump* y)
   - 4.2 $|y| > 0$
   - 4.3 $|xy| \leq p$
5. Therefore $y = a^i$ for some $0 < i \leq p$ (at least one a, no b's)
6. Let $k = 2$. Then $xy^k z$ has more a's than b's, so $xy^k z \notin L$
7. Lines 6 and 4.1 form a contradiction, therefore the assumption is false (i.e. L is *not* regular.)

Introduction

So far we have seen two different, but equivalent, methods of describing languages: finite automata and regular expressions, which describe *regular languages*

We have already proven that some languages, such as $\{0^n 1^n \mid n \geq 0\}$, cannot be described using FA or RE.

Today we will introduce *context-free grammars (CFG)*, which describe the next category of languages, the *context-free languages*

Later, will see grammars called *regular grammars*, which describe exactly *regular languages*

Grammars

- Grammars are another way to describe a language

A *grammar* is a set of rules which can be used to *generate* a language

The language generated is the set of all strings which can be *derived* from the grammar

Introductory example (G_1)

$S \to 01$ Base case: $01 \in L$

$S \to 0S1$ Recursive case: if $S \in L$ then $0S1 \in L$

G_1 generates the language $L = \{0^n 1^n \mid n \geq 0\}$, which we already know is not regular

How does it derive 000111?

Introductory example (G_1)

$S \to 01$ Base case: $01 \in L$

$S \to 0S1$ Recursive case: if $S \in L$ then $0S1 \in L$

G_1 generates the language $L = \{0^n 1^n \mid n > 0\}$, which we already know is not regular

How does it derive 000111?

S

Introductory example (G_1)

$S \to 01$ Base case: $01 \in L$

$S \to 0S1$ Recursive case: if $S \in L$ then $0S1 \in L$

G_1 generates the language $L = \{0^n 1^n \mid n \geq 0\}$, which we already know is not regular

How does it derive 000111?

$S \Rightarrow 0S1$ using rule $S \to 0S1$

Introductory example (G_1)

$S \to 01$ Base case: $01 \in L$

$S \to 0S1$ Recursive case: if $S \in L$ then $0S1 \in L$

G_1 generates the language $L = \{0^n 1^n \mid n > 0\}$, which we already know is not regular

How does it derive 000111?

$S \Rightarrow 0S1$ using rule $S \to 0S1$

$\Rightarrow 00S11$ using rule $S \to 0S1$

Introductory example (G_1)

$S \to 01$ Base case: $01 \in L$

$S \to 0S1$ Recursive case: if $S \in L$ then $0S1 \in L$

G_1 generates the language $L = \{0^n 1^n \mid n > 0\}$, which we already know is not regular

How does it derive 000111?

$$
\begin{aligned}
S &\Rightarrow 0S1 && \text{using rule } S \to 0S1 \\
&\Rightarrow 00S11 && \text{using rule } S \to 0S1 \\
&\Rightarrow 000111 && \text{using rule } S \to 01
\end{aligned}
$$

Introductory example (G_2)

$$S \to NounPhrase\ VerbPhrase$$

$$NounPhrase \to \text{the}\ Noun$$

$$VerbPhrase \to Verb\ NounPhrase$$

$$Noun \to \text{girl} \mid \text{ball}$$

$$Verb \to \text{likes} \mid \text{sees}$$

What language does G_2 generate?

Introductory example (G_2)

$$S \to \mathit{NounPhrase}\ \mathit{VerbPhrase}$$
$$\mathit{NounPhrase} \to \text{the}\ \mathit{Noun}$$
$$\mathit{VerbPhrase} \to \mathit{Verb}\ \mathit{NounPhrase}$$
$$\mathit{Noun} \to \text{girl} \mid \text{ball}$$
$$\mathit{Verb} \to \text{likes} \mid \text{sees}$$

What language does G_2 generate?

{ the girl likes the girl, the girl likes the ball,
the girl sees the girl, the girl sees the ball,
the ball likes the girl, the ball likes the ball,
the ball sees the girl, the ball sees the ball }

Definitions

Terminals

- The finite set of symbols which make up strings of the language

Non-terminals / Variables

- A finite set of symbols used to generate the strings.
- They never appear in the language.

Start symbol

- The variable used to start every derivation

Definitions

Production rules

- Sometimes called substitution or derivation rules
- Define strings of *variables* and *terminals* which can be substituted for a *variable*:

$$Variable \rightarrow <\text{string of } Variables \text{ and } Terminals>$$

Definitions

Production rules

- Sometimes called substitution or derivation rules
- Define strings of *variables* and *terminals* which can be substituted for a *variable*:

$Variable \rightarrow <$string of $Variables$ and $Terminals>$

A variable can have many rules:

$Noun \rightarrow$ girl

$Noun \rightarrow$ ball

$Noun \rightarrow$ quokka

Definitions

Production rules

- Sometimes called substitution or derivation rules
- Define strings of *variables* and *terminals* which can be substituted for a *variable*:

$Variable \rightarrow <$string of $Variables$ and $Terminals>$

A variable can have many rules:

$Noun \rightarrow$ girl

$Noun \rightarrow$ ball

$Noun \rightarrow$ quokka

They can be written together:

$Noun \rightarrow$ girl | ball | quokka

Another example

$$S \to T \mid (S \cdot S) \mid (\lambda T.S)$$

$$T \to a \mid b \mid ...$$

This is a grammar for lambda calculus expressions

- The variables are S, T
- S is the start variable
- The terminals are $a, b, ..., (,), \lambda, ., \cdot$ (i.e. atoms and operators)

Some common notational conventions

If not stated otherwise:

- $A, B, C, \ldots$ and S are variables
- S is the start variable
- $a, b, c, \ldots$ are terminals
- $\ldots, X, Y, Z$ are either terminals or variables
- $\ldots w, x, y, z$ are strings of terminals *only*
- $\alpha, \beta, \gamma, \ldots$ are strings of terminals and/or variables

NIL

In your tree encoding, NIL should represent the empty tree (a tree with no nodes)

This is the same idea as NIL for a Church List

If you need to use both and they are different, then you could rename them. e.g.

- $NIL = PAIR\ TRUE\ TRUE$, for Church Lists
- $NILTREE$, for your tree encoding

Example answer

$(HEAD\ z)$ should be the head of list z

$HEAD = \lambda z.FIRST\ (SECOND\ z)$

The second pair in a Church list contains the data $[head, tail]$, so we just need to get the first expression from the second pair.

Example: head of $\{1, 2, 3\}$ should be 1

$HEAD\ \{1, 2, 3\}$

$= HEAD\ (CONS\ 1\ \{2, 3\})$ (list notation)

$= HEAD\ (PAIR\ TRUE\ (PAIR\ (1\ \{2, 3\})))$ (defn. of CONS)

$= (\lambda z.FIRST\ (SECOND\ z))\ (PAIR\ TRUE\ (PAIR\ 1\ \{2, 3\}))$ (defn. of HEAD)

$= FIRST\ (SECOND\ (PAIR\ TRUE\ (PAIR\ 1\ \{2, 3\})))$

$= FIRST\ (PAIR\ 1\ \{2, 3\})$ (reduced SECOND)

$= 1$ (reduced FIRST)

Immutable data structures

Data structures encoded in lambda calculus are immutable.

- We can't change them
- Instead, we return a new expression

Example: Delete the second element of a list

$$DELETE\_SECOND = \lambda z.CONS\ (HEAD\ z)\ (TAIL\ (TAIL\ z))$$

- CONS to make a new list using:
- the head of the old list
- the tail of the tail of the old list
  - i.e. we skipped the second element

Example: Insert e to the second position of a list

$$INSERT\_SECOND = \lambda ex.CONS\ (HEAD\ x)\ (CONS\ e\ (TAIL\ z))$$

- CONS a new list using:
- the head of the old list
  - e
  - the tail of the old list

Recursive data structures

Linked Lists

- Each position has a reference (link) to the next one

Church Lists are similar, except instead of linking to the next node, they link to the list which *starts with* the next node

- CONS head tail
  - head is the element stored at the start of the list
  - tail is the sublist containing all the remaining elements

Recursive data structures

You can implement a tree data structure in a very similar way.

Except instead of linking to a single tail, they have a left sub-tree and a right sub-tree.

In your assignment, *MAKETREE* needs to be an expression which will take a number to store, and two trees (which will be the left and right children).

Java example

```
public class Tree {
    public final int element;
    public final Tree left;
    public final Tree right;

    public Tree(int element, Tree left, Tree right) {
        this.element = element;
        this.left = left;
        this.right = right;
    }
}
```


MAKETREE in the assignment is like writing: `new Tree(e,a,b);`

Recursive functions

Simple recursion, condition decides if you're at the base case

$$H = \lambda f x y z.(\text{condition})(\text{base})(\text{recursive})$$

$$F = (Y\ H)$$

You can have more complicated logic if you want:

$$H = \lambda f x y.(<\ x\ 5)\ \Big((<\ y\ 5)\ y\ (f\ 1\ x)\Big)\ \Big((=\ x\ y)\ (f\ y\ x)\ 3\Big)$$

$$F = (Y\ H)$$

- If $x < 5$ and $y < 5$ then y (base case)
- If $x < 5$ and $y \geq 5$ then $f(1, x)$ (recursive case)
- If $x \geq 5$ and $x = y$ then $f(y, x)$ (recursive case)
- If $x \geq 5$ and $x \neq y$ then 3 (base case)

Helper functions

It's a good idea to define some extra expressions sometimes

- To make long/complex expressions easier to read
- To reduce repetition if you need the expression in several places

Example: a MAX function which returns the larger of two numbers would be helpful when calculating the height of a tree

LISP

I discourage you from trying to directly implement your lambda calculus expressions in LISP. It'll mostly work, but some things get messy (especially conditional logic) if you try to implement everything as `(lambda)` expressions.

I recommend using Lisp lists to store data, e.g.

- A Church pair can be represented by a Lisp list with only two elements
- A Church list can be represented by a Lisp list

Then you can just use standard Lisp list operations to manipulate your data, instead of needing to implement all your lambda calculus expressions directly.

LISP

Don't forget to indent your code to make it easier to see what is being applied to what. Most errors people have shown me were just scope errors.

```
;; example of doing an if statement
(defun foo (x y)
  (if
    (...) ;; some condition
    (...) ;; true case
    (...) ;; false case
  )
)
```

LISP

```
;; example of a nested if statement
(defun foo (x y)
  (if
    (...)    ;; condition 1
    (if      ;; true 1
      (...)  ;; condition 2
      (...)  ;; true 2
      (...)  ;; false 2
    )
    (...)    ;; false 1
  )
)
```

LISP

You don't need any notation we haven't provided in tutorials or lectures.

Look back at the earlier tutorials and solutions for some code examples.

Regular Expressions

- What they are and how to use them
- Regular operations

Equivalence of FA and Regular Expressions

- Convert an NFA to a RegEx
- Convert a RegEx to an NFA

Proving if a language is, *or is not*, regular

- Prove regularity by finding a DFA, NFA, or RegEx
- Prove non-regularity by finding a contradiction in the Pumping Lemma

Grammars (basic concepts)

Lambda calculus revision