Introduction

Theoretical computer science is the **mathematical foundation of computation**.

It investigates the **power and limitations** of computing devices.

In order to be able to use rigorous mathematical proofs, **abstract mathematical models** of computers are introduced. Models should be

- as simple as possible so that they can be easily analysed, but
- powerful enough to be able to perform any computation task.

Introduction

It turns out that that real computers can be modeled with very simple abstract machines. The models ignore the implementation details of individual computers and concentrate on the actual computation process.

In the last part of the course we investigate one such model — called **Turing machine** — and using it we are able prove that there are problems that no computer can solve.

Introduction

It turns out that that real computers can be modeled with very simple abstract machines. The models ignore the implementation details of individual computers and concentrate on the actual computation process.

In the last part of the course we investigate one such model — called **Turing machine** — and using it we are able prove that there are problems that no computer can solve.

Before Turing machines we investigate some weaker models of computation. We start with the most restricted models, called **finite automata**. Then we move up to intermediate level by introducing **pushdown automata**. We analyse the computation power of different models, and study their limitations.

All our abstract machines manipulate strings of symbols. **Formal languages** are basic mathematical objects used throughout this course. The power of any computation model will be determined by analysing how complex formal languages it can describe.

- Finite automata are able to define only very simple languages, called **regular languages**.
- Pushdown automata describe more complicated **context-free languages**.
- Full-powered computers like Turing machines can define any **recursively enumerable language**.

You will become very familiar with all these language types.

Let us start with basic notions:

An **alphabet** is a finite non-empty set of symbols. Often upper case greek letters are used to denote alphabets: Σ, Δ, Γ,...

The symbols that are used depend on the application in mind. In our examples we often use letters of the English alphabet or digits or other characters found on computer keyboards. But any other symbols could be used as well.

Here are some examples of alphabets:

$$\begin{aligned} \Sigma_1 &= \{a, b\}, \\ \Sigma_2 &= \{0, 1, 2\}, \\ \Sigma_3 &= \{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\} \end{aligned}$$

A **word** is a finite sequence of symbols. Examples of words over the alphabet $\Sigma = \{a, b\}$ are *abaab*, *aaa* and *b*. Lower case variable names $u, v, w, x, y, \ldots$ are used to represent words.

If w is a word then $|w|$ denotes its **length**, *i.e.*, the number of symbols in it. Note that the length of a word can be 0. Such word is called the **empty word**, and denoted by ε. (We cannot just write nothing: no one would know that the empty word is there!)

A **word** is a finite sequence of symbols. Examples of words over the alphabet $\Sigma = \{a, b\}$ are $abaab$, aaa and b. Lower case variable names $u, v, w, x, y, \ldots$ are used to represent words.

If w is a word then $|w|$ denotes its **length**, *i.e.*, the number of symbols in it. Note that the length of a word can be 0. Such word is called the **empty word**, and denoted by ε. (We cannot just write nothing: no one would know that the empty word is there!)

For example,

$$\begin{aligned} |abaab| &= \\ |\clubsuit\heartsuit\heartsuit| &= \\ |\varepsilon| &= \end{aligned}$$

A **word** is a finite sequence of symbols. Examples of words over the alphabet $\Sigma = \{a, b\}$ are $abaab$, aaa and b. Lower case variable names $u, v, w, x, y, \ldots$ are used to represent words.

If w is a word then $|w|$ denotes its **length**, *i.e.*, the number of symbols in it. Note that the length of a word can be 0. Such word is called the **empty word**, and denoted by ε. (We cannot just write nothing: no one would know that the empty word is there!)

For example,

$$\begin{aligned} |abaab| &= \mathbf{5} \\ |\clubsuit\heartsuit\heartsuit| &= \mathbf{3} \\ |\varepsilon| &= \mathbf{0} \end{aligned}$$

The **concatenation** of two words is the word obtained by writing the first word followed by the second one as a single word.

For example, the concatenation of **data** and **base** is the word **database**. (Finnish language is very good in forming concatenated compound words!)

The **concatenation** of two words is the word obtained by writing the first word followed by the second one as a single word.

For example, the concatenation of **data** and **base** is the word **database**. (Finnish language is very good in forming concatenated compound words!)

The notation for concatenation is similar to normal multiplication. The multiplication sign does not need to be written if the meaning is clear, i.e. uv is the concatenation of words u and v. For example,

$ab \cdot aab =$

The **concatenation** of two words is the word obtained by writing the first word followed by the second one as a single word.

For example, the concatenation of **data** and **base** is the word **database**. (Finnish language is very good in forming concatenated compound words!)

The notation for concatenation is similar to normal multiplication. The multiplication sign does not need to be written if the meaning is clear, i.e. uv is the concatenation of words u and v. For example,

$ab \cdot aab = abaab$

If $v = a$ and $w = ab$, then $vw =$

The **concatenation** of two words is the word obtained by writing the first word followed by the second one as a single word.

For example, the concatenation of **data** and **base** is the word **database**. (Finnish language is very good in forming concatenated compound words!)

The notation for concatenation is similar to normal multiplication. The multiplication sign does not need to be written if the meaning is clear, i.e. uv is the concatenation of words u and v. For example,

$ab \cdot aab = abaab$

If $v = a$ and $w = ab$, then $vw = aab$

The empty word is the identity element of concatenation, much the same way as number 1 is the identity element of multiplication: For any word w,

$$w\varepsilon =$$
$$\varepsilon w =$$

The empty word is the identity element of concatenation, much the same way as number 1 is the identity element of multiplication: For any word w,

$$w\varepsilon = w$$
$$\varepsilon w = w$$

The set of all words over the alphabet Σ with the concatenation operation is a **monoid**. This just means that

- concatenation is associative: $(uv)w = u(vw)$,
- the empty word ε is the identity: $w\varepsilon = \varepsilon w = w$.

It is the **free monoid** generated by Σ.

A concatenation of a word with itself is denoted the same way as the multiplication of a number with itself: For any integer n and word w the word w^n is the concatenation of n copies of w. For example, if $w = abba$ then

$$\begin{aligned} w^2 &= \\ a^5 &= \\ \varepsilon^3 &= \\ ab^2a^3b &= \\ w^0 &= \\ \varepsilon^0 &= \end{aligned}$$

A concatenation of a word with itself is denoted the same way as the multiplication of a number with itself: For any integer n and word w the word w^n is the concatenation of n copies of w. For example, if $w = abba$ then

$$\begin{aligned} w^2 &= abbaabba \\ a^5 &= aaaaa \\ \varepsilon^3 &= \varepsilon \\ ab^2a^3b &= abbaaab \\ w^0 &= \varepsilon \\ \varepsilon^0 &= \varepsilon \end{aligned}$$

An important difference between concatenation and multiplication is that concatenation is not commutative. There are words v and w for which

$$vw \neq wv.$$

(Find some examples!)

A **prefix** of a word is any sequence of leading symbols of the word. For example, word $abaab$ has 6 prefixes:

A **prefix** of a word is any sequence of leading symbols of the word. For example, word $abaab$ has 6 prefixes: $\varepsilon, a, ab, aba, abaa, abaab$

A **prefix** of a word is any sequence of leading symbols of the word. For example, word $abaab$ has 6 prefixes: $\varepsilon, a, ab, aba, abaa, abaab$

A **suffix** of a word is any sequence of trailing symbols of the word. The suffixes of word $abaab$ are:

A **prefix** of a word is any sequence of leading symbols of the word. For example, word $abaab$ has 6 prefixes: $\varepsilon, a, ab, aba, abaa, abaab$

A **suffix** of a word is any sequence of trailing symbols of the word. The suffixes of word $abaab$ are: $\varepsilon, b, ab, aab, baab, abaab$

A **prefix** of a word is any sequence of leading symbols of the word. For example, word $abaab$ has 6 prefixes: $\varepsilon, a, ab, aba, abaa, abaab$

A **suffix** of a word is any sequence of trailing symbols of the word. The suffixes of word $abaab$ are: $\varepsilon, b, ab, aab, baab, abaab$

A **subword** of a word is any sequence of consecutive symbols that appears in the word. The subwords of $abaab$ are:

A **prefix** of a word is any sequence of leading symbols of the word. For example, word $abaab$ has 6 prefixes: $\varepsilon, a, ab, aba, abaa, abaab$

A **suffix** of a word is any sequence of trailing symbols of the word. The suffixes of word $abaab$ are: $\varepsilon, b, ab, aab, baab, abaab$

A **subword** of a word is any sequence of consecutive symbols that appears in the word. The subwords of $abaab$ are:

$$\varepsilon, a, b, ab, aa, ba, aba, baa, aab, abaa, baab, abaab$$

A **prefix** of a word is any sequence of leading symbols of the word. For example, word $abaab$ has 6 prefixes: $\varepsilon, a, ab, aba, abaa, abaab$

A **suffix** of a word is any sequence of trailing symbols of the word. The suffixes of word $abaab$ are: $\varepsilon, b, ab, aab, baab, abaab$

A **subword** of a word is any sequence of consecutive symbols that appears in the word. The subwords of $abaab$ are:

$$\varepsilon, a, b, ab, aa, ba, aba, baa, aab, abaa, baab, abaab$$

A prefix, suffix or subword of a word is called **proper** if it is not the word itself. Each word w has $|w|$ different proper prefixes and suffixes.

The **mirror image** of a word is the word obtained by reversing the order of its letters. The mirror image of word w is denoted by w^R. For example,

$$\begin{aligned} (abaab)^R &= \\ (saippuakauppias)^R &= \\ \varepsilon^R &= \end{aligned}$$

The **mirror image** of a word is the word obtained by reversing the order of its letters. The mirror image of word w is denoted by w^R. For example,

$$\begin{aligned} (abaab)^R &= baaba \\ (saippuakauppias)^R &= saippuakauppias \\ \varepsilon^R &= \varepsilon \end{aligned}$$

A word w whose mirror image is the word itself is called a **palindrome**. In other words, word w is a palindrome iff $w = w^R$. For example

$$saippuakauppias$$

is a palindrome.

A formal **language** is a set of words from a fixed alphabet. The language is **finite** if it contains only a finite number of words, otherwise it is **infinite**.

A formal **language** is a set of words from a fixed alphabet. The language is **finite** if it contains only a finite number of words, otherwise it is **infinite**.

We are mainly interested in infinite languages. Let our alphabet be $\Sigma = \{a, b\}$. Examples of languages over Σ include

$$\{a, ab, abb\}$$

$$\{a, aa, aaa, aaaa, \dots\} = \{a^n \mid n \geq 1\}$$

$$\{a^n b^n \mid n \geq 0\}$$

$$\{w \mid w = w^R\} = \{w \mid w \textit{ is a palindrome }\}$$

$$\{a^p \mid p \textit{ is a prime number }\}$$

$$\{\varepsilon\}$$

$$\emptyset$$

Note that

- ε is a word,
- $\{\varepsilon\}$ is a language containing one element (the empty word), and
- $\emptyset$ is a language containing no words.

They are all different!

Note that

- ε is a word,
- $\{\varepsilon\}$ is a language containing one element (the empty word), and
- $\emptyset$ is a language containing no words.

They are all different!

The language of all words over alphabet Σ is denoted by Σ^*.
The language of all **non-empty** words over Σ is denoted by Σ^+:

$$\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}.$$

For example,

$$\begin{aligned} \{a, b\}^* &= \{\varepsilon, a, b, aa, ab, ba, bb, aaa, \dots\} \\ \{\heartsuit\}^* &= \{\varepsilon, \heartsuit, \heartsuit^2, \heartsuit^3, \heartsuit^4, \dots\} \\ \{\heartsuit\}^+ &= \{\heartsuit, \heartsuit^2, \heartsuit^3, \heartsuit^4, \dots\} \end{aligned}$$

A problem with infinite languages is how to describe them. We cannot just list the words as as we do in the case of finite languages. Formal language theory presents techniques for specifying infinite languages.

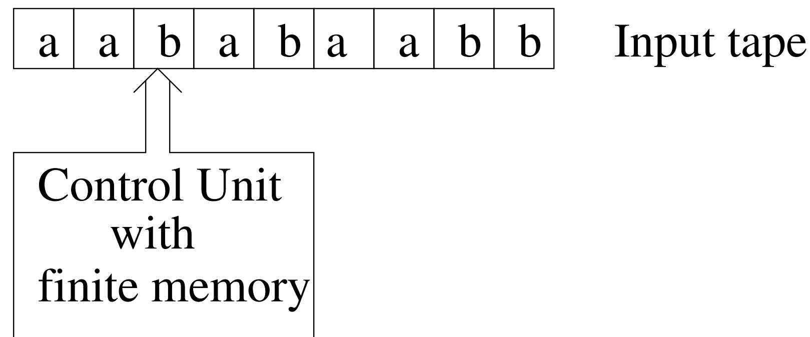
Deterministic Finite Automata (DFA)

DFA provide a simple way of describing languages. DFA are accepting devices: you give a word as input and after a while the DFA tells whether the input word is in the language (DFA **accepts** the word) or whether it is not in the language (DFA **rejects** it).

To decide whether to accept or reject the input word the DFA scans the letters of the word from left to right. The DFA has a finite internal memory available. At each input letter the state of the internal memory is changed depending on the letter scanned.

The **previous memory state** and the **next input letter** together determine what the next state of the memory is.

The word is accepted if the internal memory is in an **accepting state** after scanning the entire word.



Let us be precise: A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is specified by 5 items:

- Finite **state set** Q. At all times the internal memory is in some state $q \in Q$.

Let us be precise: A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is specified by 5 items:

- Finite **state set** Q. At all times the internal memory is in some state $q \in Q$.
- **Input alphabet** Σ. The machine only operates on words over alphabet Σ.

Let us be precise: A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is specified by 5 items:

- Finite **state set** Q. At all times the internal memory is in some state $q \in Q$.
- **Input alphabet** Σ. The machine only operates on words over alphabet Σ.
- **Transition function** δ. The transition function describes how the machine changes its internal state. It is a function

$$\delta : Q \times \Sigma \longrightarrow Q$$

from (state, input letter) -pairs to states. If the machine is in state q and next input letter is a then the machine changes its internal state to $\delta(q, a)$ and moves to the next input letter.

Let us be precise: A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is specified by 5 items:

- Finite **state set** Q. At all times the internal memory is in some state $q \in Q$.
- **Input alphabet** Σ. The machine only operates on words over alphabet Σ.
- **Transition function** δ. The transition function describes how the machine changes its internal state. It is a function

$$\delta : Q \times \Sigma \longrightarrow Q$$

  from (state, input letter) -pairs to states. If the machine is in state q and next input letter is a then the machine changes its internal state to $\delta(q, a)$ and moves to the next input letter.
- **Initial state** $q_0 \in Q$ is the internal state of the machine before any letters have been read.

Let us be precise: A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is specified by 5 items:

- Finite **state set** Q. At all times the internal memory is in some state $q \in Q$.
- **Input alphabet** Σ. The machine only operates on words over alphabet Σ.
- **Transition function** δ. The transition function describes how the machine changes its internal state. It is a function

$$\delta : Q \times \Sigma \longrightarrow Q$$

  from (state, input letter) -pairs to states. If the machine is in state q and next input letter is a then the machine changes its internal state to $\delta(q, a)$ and moves to the next input letter.
- **Initial state** $q_0 \in Q$ is the internal state of the machine before any letters have been read.
- Set $F \subseteq Q$ of **final states** specifies which states are accepting and which are rejecting. If the internal state of the machine, after reading the whole input, is some state of F then the word is accepted, otherwise rejected.

Let us be precise: A DFA $A = (Q, \Sigma, \delta, q_0, F)$ is specified by 5 items:

- Finite **state set** Q. At all times the internal memory is in some state $q \in Q$.
- **Input alphabet** Σ. The machine only operates on words over alphabet Σ.
- **Transition function** δ. The transition function describes how the machine changes its internal state. It is a function

  $$\delta : Q \times \Sigma \longrightarrow Q$$

  from (state, input letter) -pairs to states. If the machine is in state q and next input letter is a then the machine changes its internal state to $\delta(q, a)$ and moves to the next input letter.
- **Initial state** $q_0 \in Q$ is the internal state of the machine before any letters have been read.
- Set $F \subseteq Q$ of **final states** specifies which states are accepting and which are rejecting. If the internal state of the machine, after reading the whole input, is some state of F then the word is accepted, otherwise rejected.

The **language recognized by DFA** A consists of all words that A accepts. The language is denoted by $L(A)$.

Example. Consider the DFA

$$A = (\{p, q, r\}, \{a, b\}, \delta, p, \{r\})$$

where the transition function is given by the table

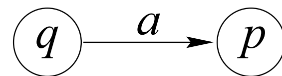
	a	b
p	q	p
q	r	p
r	r	r

Let us see the operation of the machine on input word $w = abaab$ on the blackboard.

A convenient way of displaying DFA is to use a **transition diagram**. It is a labeled directed graph whose nodes represent different states of Q, and whose edges indicate the transitions with different input symbols.

The edges are labeled with the input letters and nodes are labeled with states.

Transition $\delta(q, a) = p$ is represented by an arc labeled a going from node q into node p:



Final states are indicated as double circles, initial state is indicated by a short incoming arrow.

Example. The diagram representation of the DFA

$$A = (\{p, q, r\}, \{a, b\}, \delta, p, \{r\})$$

with the transition function

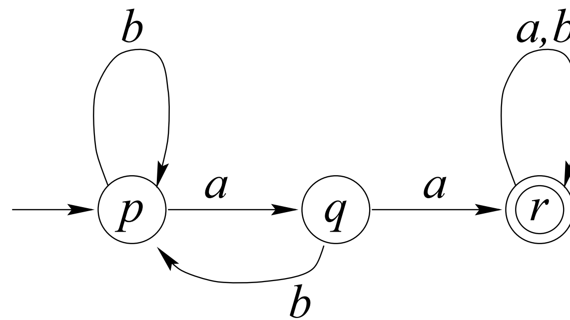
	a	b
p	q	p
q	r	p
r	r	r

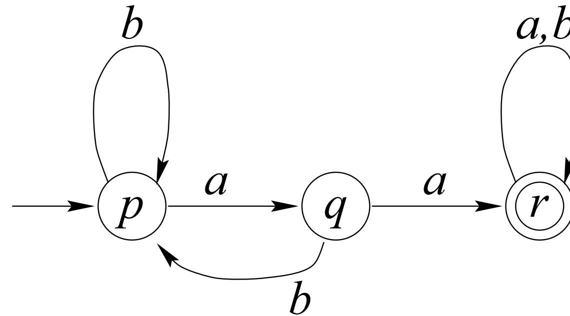
Example. The diagram representation of the DFA

$$A = (\{p, q, r\}, \{a, b\}, \delta, p, \{r\})$$

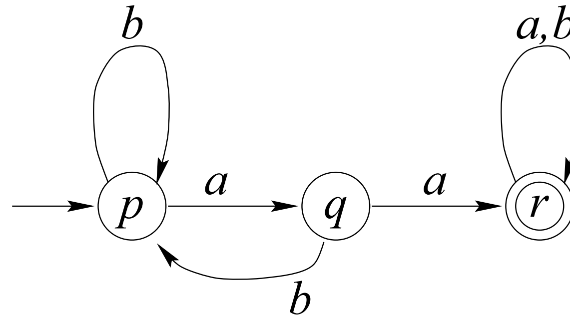
with the transition function

	a	b
p	q	p
q	r	p
r	r	r





To determine whether a given word is accepted by A one follows the **path labeled with the input letters**, starting from the initial state. If the state where the path ends is a final state, the word is accepted. Otherwise it is rejected.



To determine whether a given word is accepted by A one follows the **path labeled with the input letters**, starting from the initial state. If the state where the path ends is a final state, the word is accepted. Otherwise it is rejected.

In our example, path labeled with input word $w = abaab$ leads to state r so the word is accepted. Input word $abba$ is rejected since it leads to q which is not a final state.

A simple characterization in English of the words that are accepted by the DFA A above ?

Let us design transition diagrams for DFA that accept following languages over alphabet $\{a, b\}$. (The DFA has to accept the language **exactly**: all words of the language have to be accepted; all words not in the language have to be rejected.)

1. Words that end in ab.

2. Words with odd number of a's.

3. Words that contain aba as a subword.

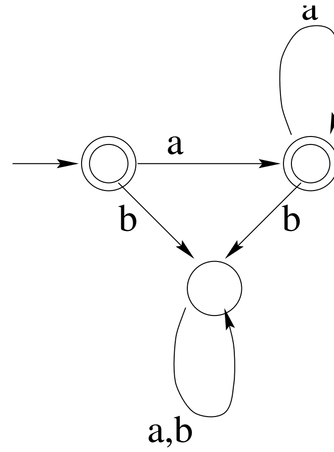
4. Words that start with a and end in a.

5. Finite language $\{\varepsilon, a, b\}$.

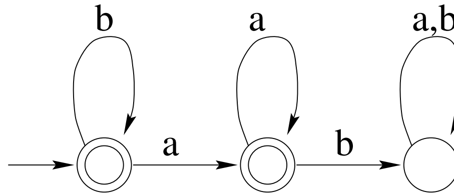
6. All words over $\{a, b\}$, i.e. $\{a, b\}^*$.

Then the inverse problem: describe in English the language accepted by the following DFA:

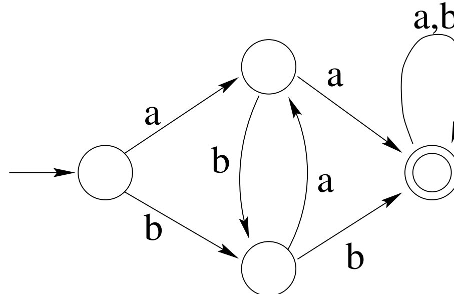
1)



2)



3)



Not all languages can be defined by a DFA. For example, it is impossible to build a DFA that would recognize the language

$$\{a^p \mid \ p \text{ is a prime number } \}$$

or the language

$$\{a^n b^n \mid n \geq 0\}.$$

Languages that can be recognized by DFA are called **regular**. By now, we know many regular languages.

To enable exact mathematical notations and proofs by mathematical induction, we **extend** the meaning of the transition function δ from single letters to words.

- The basic transition function δ gives the new state of the machine after a single letter is read.
- The extended function (that we will denote by $\hat{\delta}$) gives the new state after an arbitrary string of letters is read.

In other words, $\hat{\delta}$ is a function

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow Q,$$

and for every state q and word w the value of $\hat{\delta}(q, w)$ is the state that the DFA reaches if in state q it reads input word w.

Formally the extended function is defined recursively as follows:

1. $\hat{\delta}(q, \varepsilon) = q$ for every state q.

(The machine does not change its state if no input letters are consumed.)

2. For all words w and letters a

$$\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a).$$

(If $p = \hat{\delta}(q, w)$ is the state after reading input w then $\delta(p, a)$ is the new state after reading input wa.)

Formally the extended function is defined recursively as follows:

1. $\hat{\delta}(q, \varepsilon) = q$ for every state q.

(The machine does not change its state if no input letters are consumed.)

2. For all words w and letters a

$$\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a).$$

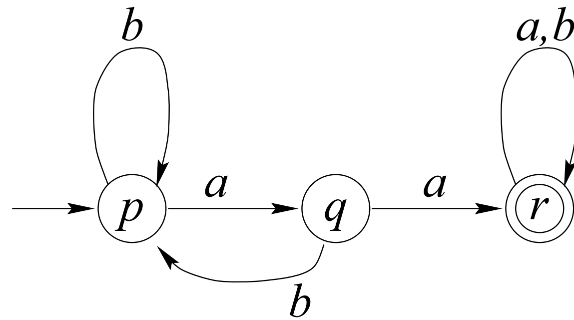
(If $p = \hat{\delta}(q, w)$ is the state after reading input w then $\delta(p, a)$ is the new state after reading input wa.)

Note that $\hat{\delta}$ is an extension of δ. They give same value for input words $w = a$ that contain only one letter:

$$\hat{\delta}(q, a) = \delta(q, a).$$

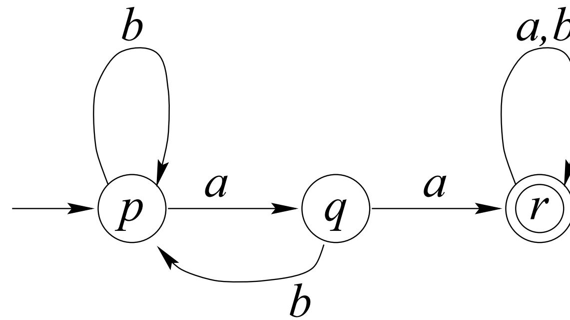
Therefore there is no danger of confusion if we simplify notations by removing the hat and indicate simply δ instead of $\hat{\delta}$.

Example.



$$
\begin{aligned}
\delta(r, ab) &= \\
\delta(q, bb) &= \\
\delta(p, abaab) &=
\end{aligned}
$$

Example.



$$
\begin{aligned}
\delta(r, ab) &= r \\
\delta(q, bb) &= p \\
\delta(p, abaab) &= r
\end{aligned}
$$

The **language recognized by DFA** $A = (Q, \Sigma, \delta, q_0, F)$ can now be formulated as follows:

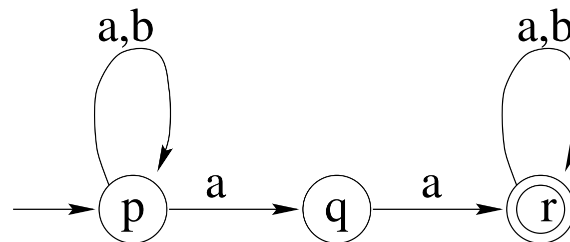
$$L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \in F\}$$

(=The set of words w over alphabet Σ such that, if the machine reads input w in the initial state q_0, then the state it reaches is a final state.)

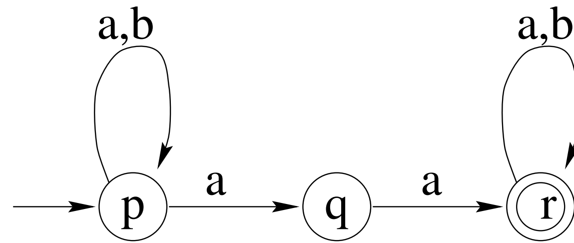
Nondeterministic finite automata (NFA)

Nondeterministic finite automata are generalizations of DFA. Instead of exactly one outgoing transition from each state by every input letter, NFA allow **several outgoing transitions** at the same time. A word is accepted by the NFA if **some** choice of transitions takes the machine to a final state. Some other choices may lead to a non-final state, but the word is accepted as long as there exists at least one accepting computation path in the automaton.

An example of a transition diagram of an NFA:

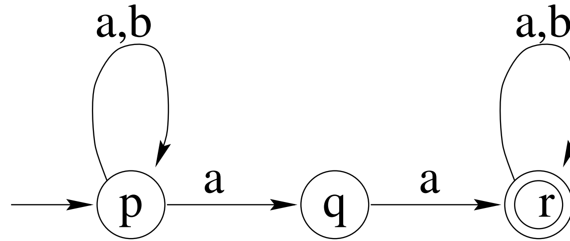


There are two transitions from state p with input letter a, into states p and q. Note also that there are no transitions from state q with input letter b. The number of transitions may be zero, one or more.



NFA may have several different computations for the same input word. Let us take for example word

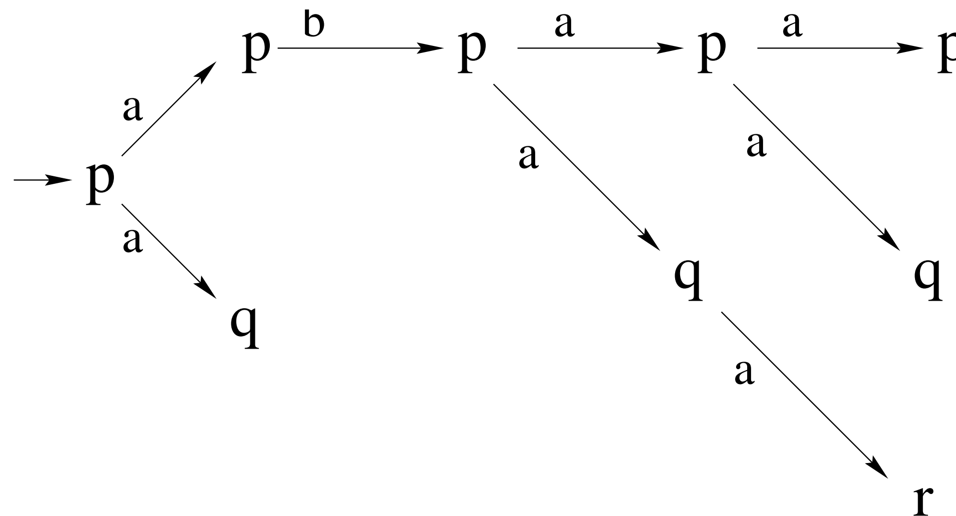
$$w = abaa.$$



NFA may have several different computations for the same input word. Let us take for example word

$$w = abaa.$$

The following computation tree summarizes all possible computations with input $abaa$:



Precise definition: An NFA $A = (Q, \Sigma, \delta, q_0, F)$ is specified by 5 items:

- State set Q,
- input alphabet Σ,
- initial state q_0, and
- final state set F

all have the same meaning as for a DFA.

- The transition function δ is defined differently. It gives for each state q and input letter a a **set** $\delta(q, a)$ of possible next states.

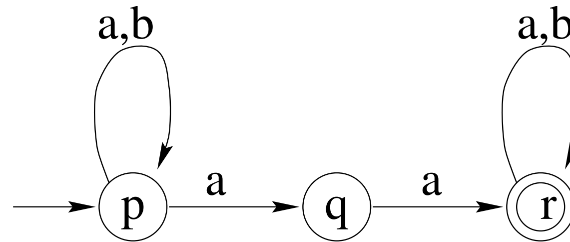
Using the power set notation

$$2^Q = \{S \mid S \subseteq Q\}$$

we can write

$$\delta : Q \times \Sigma \longrightarrow 2^Q.$$

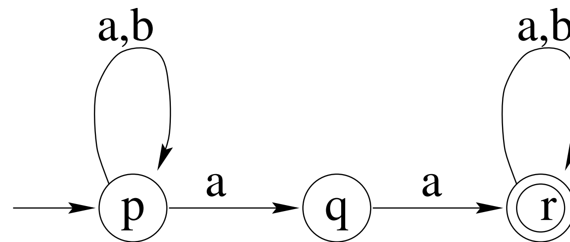
Example. The transition function δ of



is given by the table

	a	b
p	$\{p, q\}$	$\{p\}$
q	$\{r\}$	$\emptyset$
r	$\{r\}$	$\{r\}$

Example. The transition function δ of



is given by the table

	a	b
p	$\{p,q\}$	$\{p\}$
q	$\{r\}$	$\emptyset$
r	$\{r\}$	$\{r\}$

What is the language recognized by the sample NFA ? The language consists of all words for which there exists at least one accepting computation path.

Let us **extend** the meaning of the transition function δ the same way we did for DFA. We define

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow 2^Q$$

such that $\hat{\delta}(q, w)$ is the set of all states the machine can reach from state q reading input word w.

Let us **extend** the meaning of the transition function δ the same way we did for DFA. We define

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow 2^Q$$

such that $\hat{\delta}(q, w)$ is the set of all states the machine can reach from state q reading input word w.

The exact recursive definition goes like this:

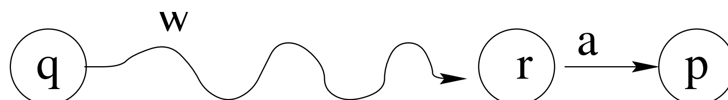
1. For every state q

$$\hat{\delta}(q, \varepsilon) = \{q\}.$$

(No state is changed if no input is read.)

2. For every state q, word w and letter a

$$\hat{\delta}(q, wa) = \bigcup_{r \in \hat{\delta}(q,w)} \delta(r, a) = \{p \mid \text{there exists state } r \text{ such that } r \in \hat{\delta}(q, w) \text{ and } p \in \delta(r, a)\ \}.$$

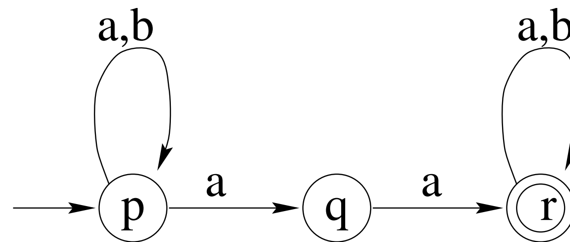


On single symbols δ and $\hat{\delta}$ have identical values:

$$\delta(q, a) = \hat{\delta}(q, a).$$

Therefore there is no risk of confusion if we drop the hat and write simply δ instead of $\hat{\delta}$.

Example. In our sample NFA



$$\begin{aligned}
\delta(p, a) &= \{p, q\}, \\
\delta(p, ab) &= \{p\}, \\
\delta(p, aba) &= \{p, q\}, \\
\delta(p, abaa) &= \{p, q, r\}.
\end{aligned}$$

The language recognized by NFA $A = (Q, \Sigma, \delta, q_0, F)$ is

$$L(A) = \{w \in \Sigma^* \mid \delta(q_0, w) \cap F \neq \emptyset\}.$$

(Words w such that there is a final state among the states $\delta(q_0, w)$ reachable from the initial state q_0 on input w.)

Let us design NFA over alphabet $\Sigma = \{a, b\}$ that recognize the following languages:

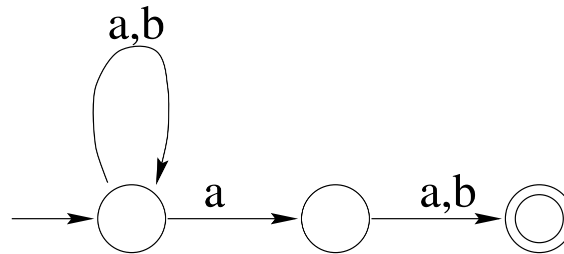
1. Words that end in ab.

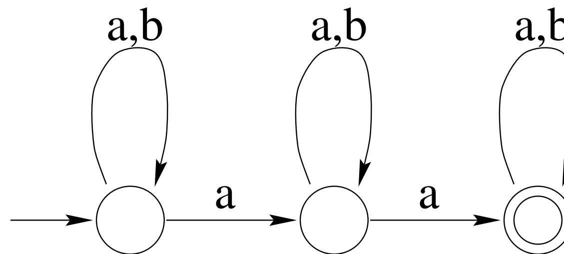
2. Words that contain aba as a subword.

3. Words that start with ab and end in ba

4. Words that contain two b's separated by an even number of a's.

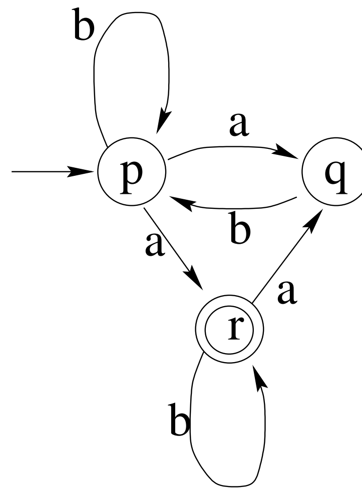
And the inverse problem: Describe in simple English sentence the words accepted by the following NFA:





Question: How does one go about checking if a given NFA accepts a given input word w ?

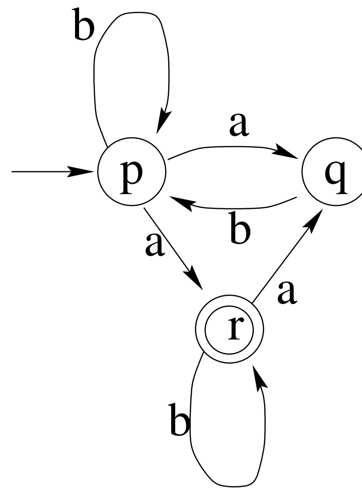
For example, how would one check if the NFA



accepts input $w = abbaabb$?

Question: How does one go about checking if a given NFA accepts a given input word w ?

For example, how would one check if the NFA

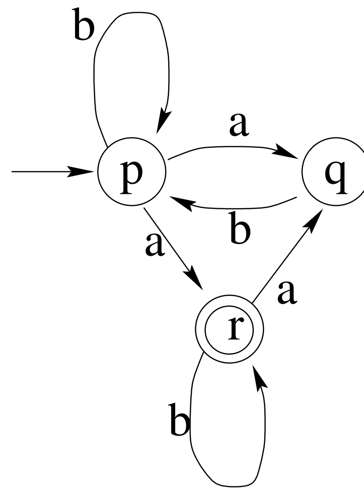


accepts input $w = abbaabb$?

One alternative is to try all possible computation paths for w and see if any of them ends in an accepting state. But the number of paths may be very large, and grow exponentially with the length of the input!

Question: How does one go about checking if a given NFA accepts a given input word w ?

For example, how would one check if the NFA

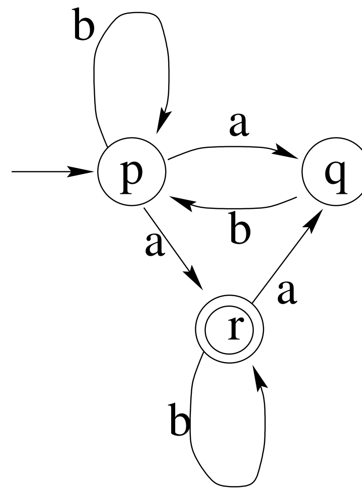


accepts input $w = abbaabb$?

A **better way** is to scan the input only once, and keep track of the **set** of possible states. For example, with our NFA and input $w = abbaabb$ we have:

Question: How does one go about checking if a given NFA accepts a given input word w ?

For example, how would one check if the NFA



accepts input $w = abbaabb$?

A **better way** is to scan the input only once, and keep track of the **set** of possible states. For example, with our NFA and input $w = abbaabb$ we have:

$$\{p\} \xrightarrow{a} \{q, r\} \xrightarrow{b} \{p, r\} \xrightarrow{b} \{p, r\} \xrightarrow{a} \{q, r\} \xrightarrow{a} \{q\} \xrightarrow{b} \{p\} \xrightarrow{b} \{p\}$$

The word is not accepted because one can only reach p which is not a final state.

Intuitively it may seem that NFA can recognize some languages that no DFA can recognize. But this is not the case: NFA and DFA recognize exactly the **same family of languages** (the regular languages).

Intuitively it may seem that NFA can recognize some languages that no DFA can recognize. But this is not the case: NFA and DFA recognize exactly the **same family of languages** (the regular languages).

1) First, any language recognized by some DFA is also recognized by some NFA; namely the same automaton. (Every DFA is also an NFA.)

Intuitively it may seem that NFA can recognize some languages that no DFA can recognize. But this is not the case: NFA and DFA recognize exactly the **same family of languages** (the regular languages).

1) First, any language recognized by some DFA is also recognized by some NFA; namely the same automaton. (Every DFA is also an NFA.)

2) Conversely, every language that is recognized by an NFA is also recognized by some DFA. To prove this we show how to construct a DFA that is equivalent to a given NFA, *i.e.*, it recognizes the same language.

Intuitively it may seem that NFA can recognize some languages that no DFA can recognize. But this is not the case: NFA and DFA recognize exactly the **same family of languages** (the regular languages).

1) First, any language recognized by some DFA is also recognized by some NFA; namely the same automaton. (Every DFA is also an NFA.)

2) Conversely, every language that is recognized by an NFA is also recognized by some DFA. To prove this we show how to construct a DFA that is equivalent to a given NFA, *i.e.*, it recognizes the same language.

The idea of the proof is to keep track of all possible states that the NFA can be in after reading the input. So we construct a DFA whose states are **sets of states of the original NFA**.

The states of the new DFA will be sets

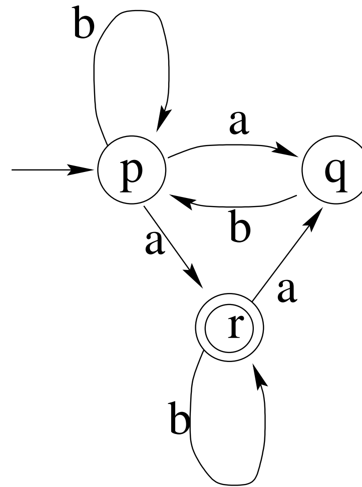
$$\{q_1, q_2, \ldots, q_k\}$$

where $q_1, q_2, \ldots, q_k$ are states of the NFA.

The interpretation of the state $\{q_1, q_2, \ldots, q_k\}$: Input w takes the DFA into state $\{q_1, q_2, \ldots, q_k\}$ if and only if $q_1, q_2, \ldots, q_k$ are precisely the states one can reach with input w in the NFA.

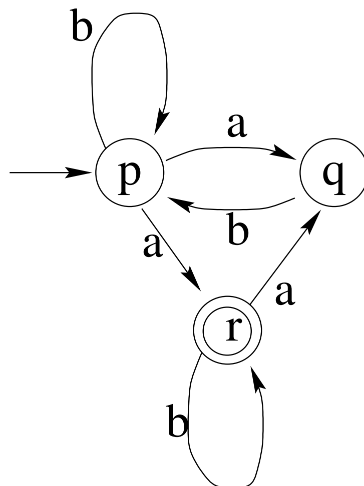
Before the general case, let us look at one example

Example. Let us construct a DFA that recognizes the same language as our sample NFA

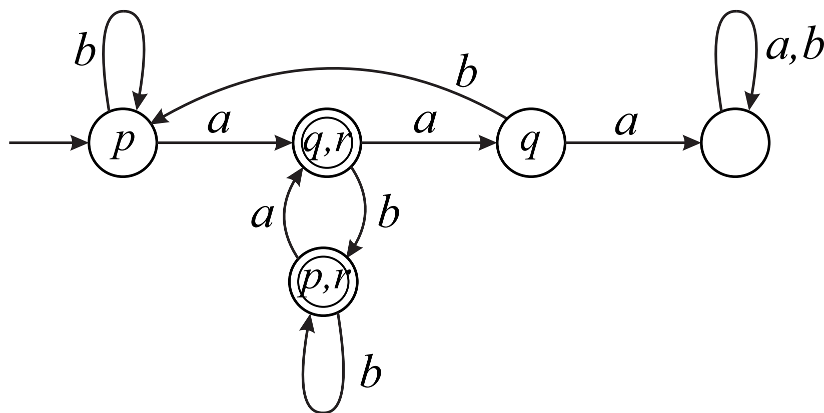


Before the general case, let us look at one example

Example. Let us construct a DFA that recognizes the same language as our sample NFA



The construction is called the **powerset** construction.



Remark. This is just one DFA that recognizes the same language; it is not necessarily the smallest one.

Let us proof the general case:

Theorem (The powerset construction). Given an NFA A one can effectively construct a DFA A' such that $L(A) = L(A')$.

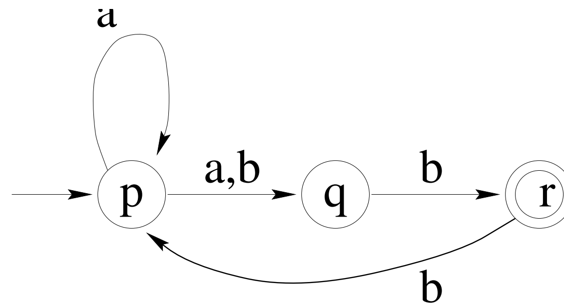
Proof.

Let us proof the general case:

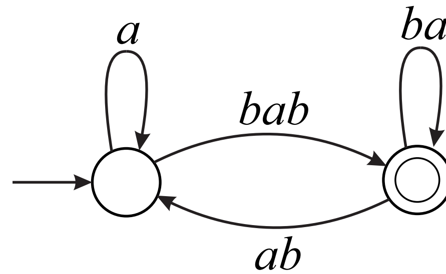
Theorem (The powerset construction). Given an NFA A one can **effectively** construct a DFA A' such that $L(A) = L(A')$.

Remark. "Effective construction" means that there is an algorithm (=mechanical procedure) to construct the DFA A' when the NFA A is given as input. The construction mechanically processes the NFA A, without any need to understand what A does. This is a stronger statement than simply stating that "there exists" a DFA that is equivalent to A.

Let us practice the powerset construction with the following NFA:



A simple extension of NFA: Allow transitions with words instead of just letters.

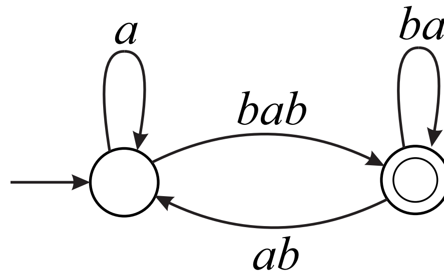


For example

$$babbaabbab = bab \cdot ba \cdot ab \cdot bab$$

is accepted.

A simple extension of NFA: Allow transitions with words instead of just letters.

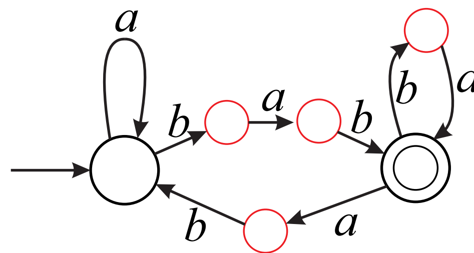


For example

$$babbaabbab = bab \cdot ba \cdot ab \cdot bab$$

is accepted.

Transitions with longer words can cut into single-letter transitions by adding new states:

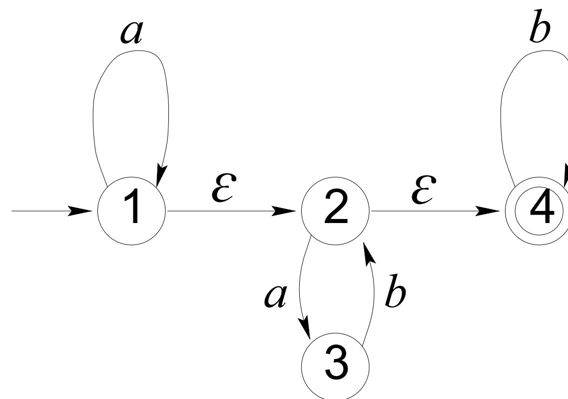


The construction cannot be used for transitions with the empty word ε.

NFA with ε-moves

In the next section, it turns out to be useful to allow NFA with **spontaneous transitions**. When an NFA executes a spontaneous transition (called an **ε-move**) it changes its state without reading any input letter. Any number of consecutive ε-moves are allowed.

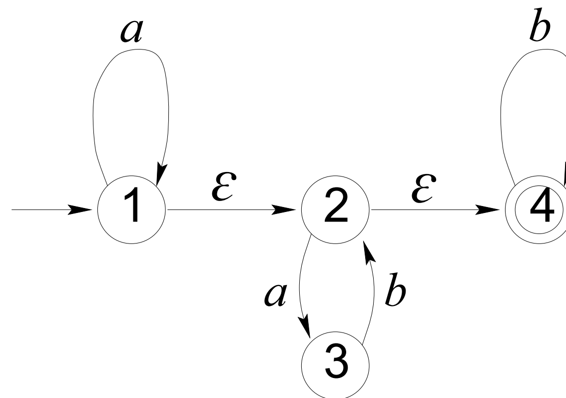
Here's an example of an NFA with ε-moves:



NFA with ε-moves

In the next section, it turns out to be useful to allow NFA with **spontaneous transitions**. When an NFA executes a spontaneous transition (called an **ε-move**) it changes its state without reading any input letter. Any number of consecutive ε-moves are allowed.

Here's an example of an NFA with ε-moves:



The automaton accepts any sequence of a's followed by any repetition of ab's followed by any number of b's:

$$L(A) = \{a^i(ab)^j b^k \mid i, j, k \geq 0\}.$$

Formally, an NFA with ε-moves (or **ε-NFA** for short) is $A = (Q, \Sigma, \delta, q_0, F)$ where Q, Σ, q_0 and F are as before, and δ is a function

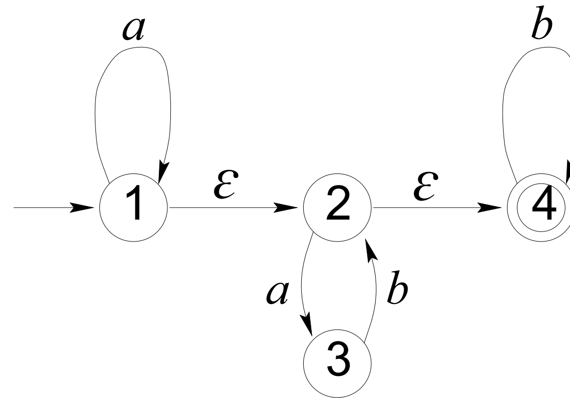
$$\mathbf{Q} \times (\mathbf{\Sigma} \cup \{\varepsilon\}) \longrightarrow \mathbf{2^Q}$$

that specifies for each state q transitions with all input letters a, and the empty word ε.

- $\delta(q, a)$ is the set of all states p such that there is a transition from q to p with input a, and

- $\delta(q, \varepsilon)$ is the set of states p such that there is a spontaneous transition from q to p.

Example.

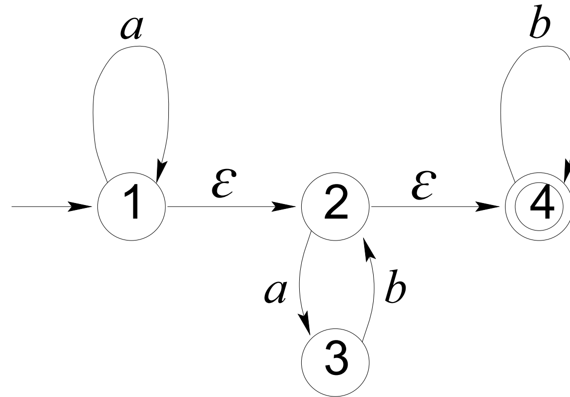


The transition function

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow 2^Q$$

is

Example.



The transition function

$$\delta : Q \times (\Sigma \cup \{\varepsilon\}) \longrightarrow 2^Q$$

is

	a	b	ε
1	$\{1\}$	$\emptyset$	$\{2\}$
2	$\{3\}$	$\emptyset$	$\{4\}$
3	$\emptyset$	$\{2\}$	$\emptyset$
4	$\emptyset$	$\{4\}$	$\emptyset$

We **extend** δ to function $\hat{\delta}$ that gives reached states for all input words:

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow 2^Q$$

Set $\hat{\delta}(q, w)$ contains all possible states after the automaton reads input word w, starting at state q. When processing w any number of ε-moves may be used.

We **extend** δ to function $\hat{\delta}$ that gives reached states for all input words:

$$\hat{\delta} : Q \times \Sigma^* \longrightarrow 2^Q$$

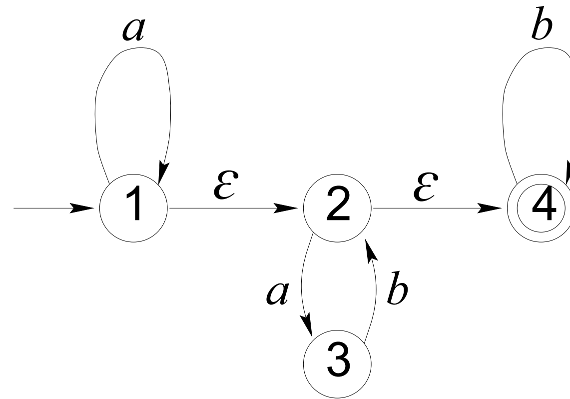
Set $\hat{\delta}(q, w)$ contains all possible states after the automaton reads input word w, starting at state q. When processing w any number of ε-moves may be used.

In the recursive definition of $\hat{\delta}$ we use the concept of an **ε-closure**. For any state q we let

$$\varepsilon\text{-CLOSURE }(q)$$

be the set of states reached from q using only ε-moves.

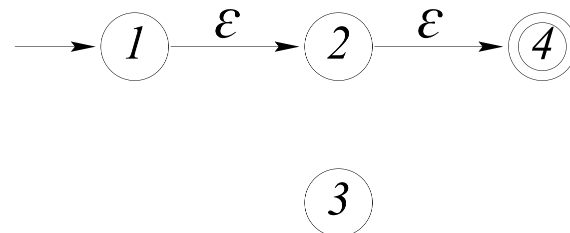
Example.



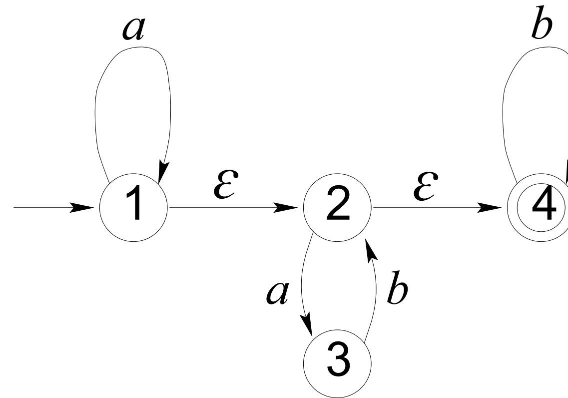
The ε-closures of the states are

$$\begin{aligned}
\varepsilon\text{-CLOSURE }(1) &= \\
\varepsilon\text{-CLOSURE }(2) &= \\
\varepsilon\text{-CLOSURE }(3) &= \\
\varepsilon\text{-CLOSURE }(4) &=
\end{aligned}$$

The set ε-CLOSURE(q) contains exactly the reachable states from q in the graph



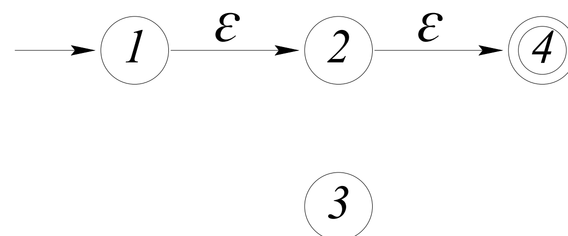
Example.



The ε-closures of the states are

$$\begin{aligned}
\varepsilon\text{-CLOSURE }(1) &= \{1,2,4\} \\
\varepsilon\text{-CLOSURE }(2) &= \{2,4\} \\
\varepsilon\text{-CLOSURE }(3) &= \{3\} \\
\varepsilon\text{-CLOSURE }(4) &= \{4\}
\end{aligned}$$

The set ε-CLOSURE(q) contains exactly the reachable states from q in the graph



For any **subset of states** $S \subseteq Q$ we denote ε-CLOSURE (S) for the set of states reachable from any element of S using only ε-moves:

$$\varepsilon\text{-CLOSURE } (S) = \bigcup_{q \in S} \varepsilon\text{-CLOSURE } (q).$$

Here is a recursive definition of $\hat{\delta}(q, w)$:

1. For every state q

$$\hat{\delta}(q, \varepsilon) = \varepsilon\text{-CLOSURE } (q).$$

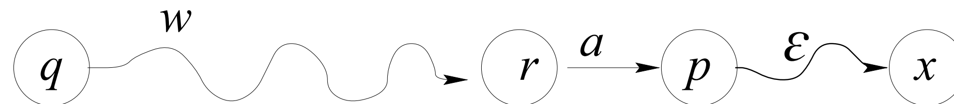
(By definition, the ε-CLOSURE consists of all states reachable with ε-moves.)

2. For every state q, word w and letter a

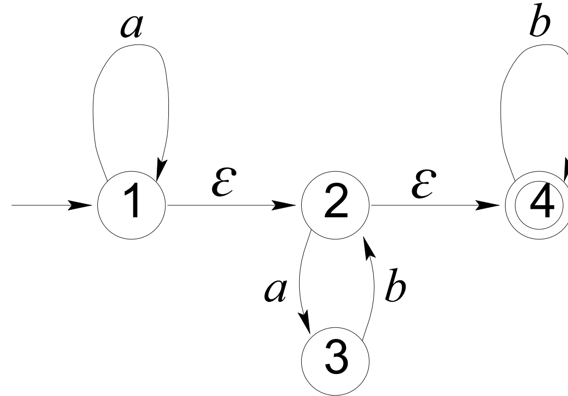
$$\hat{\delta}(q, wa) = \varepsilon\text{-CLOSURE } \left(\bigcup_{r \in \hat{\delta}(q,w)} \delta(r, a) \right) = \varepsilon\text{-CLOSURE } (S)$$

where

$$S = \{p \mid \exists\, r \in \hat{\delta}(q, w) \ : \ p \in \delta(r, a)\}.$$

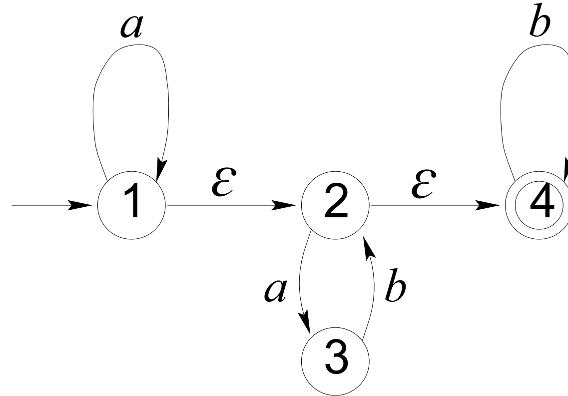


Example.



$$\begin{aligned}
\hat{\delta}(1,\varepsilon) &= \{1,2,4\} \\
\hat{\delta}(2,\varepsilon) &= \{2,4\} \\
\hat{\delta}(3,\varepsilon) &= \{3\} \\
\hat{\delta}(4,\varepsilon) &= \{4\} \\
\hat{\delta}(1,a) &= \\
\hat{\delta}(1,ab) &= \\
\hat{\delta}(1,aba) &= \\
&\cdots
\end{aligned}$$

Example.



$$
\begin{aligned}
\hat{\delta}(1, \varepsilon) &= \{1, 2, 4\} \\
\hat{\delta}(2, \varepsilon) &= \{2, 4\} \\
\hat{\delta}(3, \varepsilon) &= \{3\} \\
\hat{\delta}(4, \varepsilon) &= \{4\} \\
\hat{\delta}(1, a) &= \{1, 2, 3, 4\} \\
\hat{\delta}(1, ab) &= \{2, 4\} \\
\hat{\delta}(1, aba) &= \{3\} \\
&\cdots
\end{aligned}
$$

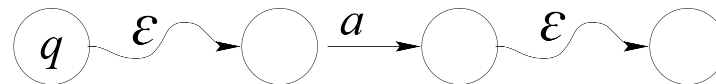
The language recognized by ε-NFA A is

$$L(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \cap F \neq \emptyset\}$$

The values of $\hat{\delta}(q, a)$ for single letters a are of special interest to us. First of all, they are **not** necessarily identical to $\delta(q, a)$, so we cannot remove the hat as we did with DFA and NFA:

- $\delta(q, a)$ contains only states you can reach with one transition labeled by a. It does not allow using ε-moves.

- $\hat{\delta}(q, a)$ contains all states you can reach from q by doing any number of ε-moves and one a-transition, in any order:

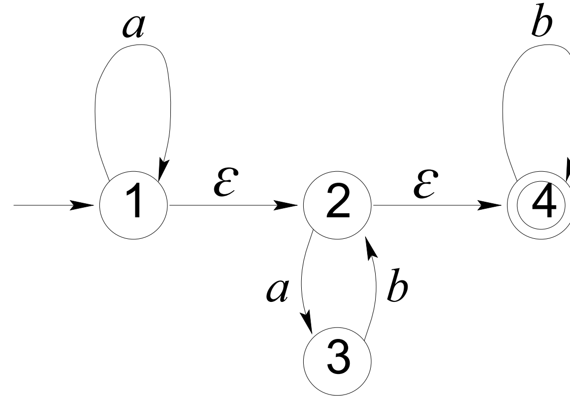


Introducing spontaneous transitions does not increase the power of NFA. Next we show that ε-NFA accept exactly the same regular languages as DFA and NFA:

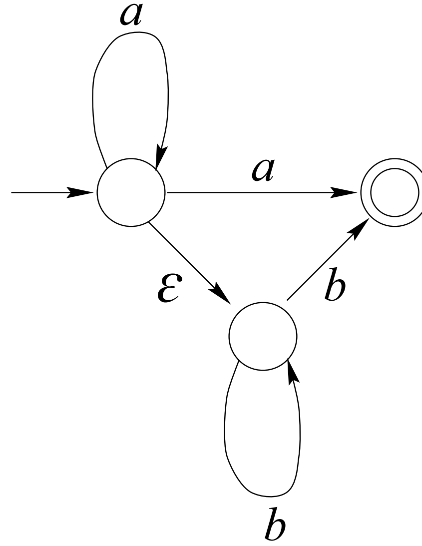
Theorem. For any given ε-NFA A one can effectively construct an NFA A' such that $L(A') = L(A)$.

Proof.

Example. Let us construct an NFA equivalent to



Example. Let us construct an NFA equivalent to



Now we know how to convert any ε-NFA into an equivalent NFA, and how to convert any NFA into an equivalent DFA, so we can convert any ε-NFA into a DFA. All three automata models recognize the same languages.

Regular expressions

Regular expressions provide a different technique to define languages. Instead of being accepting devices such as finite automata they are **generating** devices.

By **Kleene's theorem** (proved later) regular expressions define exactly regular languages (=the same languages recognized by finite automata).

We need some **operations** on languages.

- The **concatenation** L_1L_2 of languages L_1 and L_2 is the language containing all words obtained by concatenating a word from L_1 and a word from L_2:

$$L_1L_2 = \{uv \mid u \in L_1 \text{ and } v \in L_2\}$$

For example

$\{ab, b\}\{aa, ba\} =$

We need some **operations** on languages.

• The **concatenation** $L_1 L_2$ of languages L_1 and L_2 is the language containing all words obtained by concatenating a word from L_1 and a word from L_2:

$$L_1 L_2 = \{uv \mid u \in L_1 \text{ and } v \in L_2\}$$

For example

$\{ab, b\}\{aa, ba\} = \{abaa, abba, baa, bba\}$

• For every $n \geq 0$ we define L^n to be the set of words obtained by concatenating n words from language L. A precise recursive definition:

$$
\begin{aligned}
L^0 &= \{\varepsilon\} \\
L^n &= L^{n-1}L \text{ for every } n \geq 1.
\end{aligned}
$$

For example,

$\{ab, b\}^3 =$

• For every $n \geq 0$ we define L^n to be the set of words obtained by concatenating n words from language L. A precise recursive definition:

$$\begin{aligned} L^0 &= \{\varepsilon\} \\ L^n &= L^{n-1}L \text{ for every } n \geq 1. \end{aligned}$$

For example,

$\{ab, b\}^3 = \{ababab, ababb, abbab, abbb, babab, babb, bbab, bbb\}$

• The **Kleene closure** L^* of language L is the set containing words obtained by concatenating any number of words from L together:

$$L^* = \bigcup_{i=0}^{\infty} L^i.$$

For example,

$$\{ab, b\}^* = \{\varepsilon, ab, b, abab, abb, bab, bb, ababab, ababb, \dots\}$$

Note that if $L = \Sigma$ then $L^* = \Sigma^*$ is the set of all words over alphabet Σ. We have already used this notation earlier!

- The **positive closure** L^+ of L is

$$L^+ = \bigcup_{i=1}^{\infty} L^i,$$

i.e., concatenations of one or more words from L. For example,

$$\{ab, b\}^+ = \{ab, b, abab, abb, bab, bb, ababab, ababb, \dots\}.$$

We have always

$$L^* = L^+ \cup \{\varepsilon\}.$$

Also, we have always

$$L^+ = LL^*.$$

- When are languages L^* and L^+ identical ?
- When are languages L^* and L^+ finite ?
- What language is $\emptyset^*$? What about $\emptyset^+$?

- The **union** $L_1 \cup L_2$ of languages L_1 and L_2 is just the usual union as sets.

Next we define **regular expressions** over alphabet Σ. They are syntactic expressions that represent certain languages. If r is a regular expression then we denote the language it represent as $L(r)$.

Next we define **regular expressions** over alphabet Σ. They are syntactic expressions that represent certain languages. If r is a regular expression then we denote the language it represent as $L(r)$.

$\emptyset$ is a regular expression representing the empty language.

Next we define **regular expressions** over alphabet Σ. They are syntactic expressions that represent certain languages. If r is a regular expression then we denote the language it represent as $L(r)$.

$\emptyset$ is a regular expression representing the empty language.

ε is a regular expression and it represents the singleton language $\{\varepsilon\}$.

Next we define **regular expressions** over alphabet Σ. They are syntactic expressions that represent certain languages. If r is a regular expression then we denote the language it represent as $L(r)$.

$\emptyset$ is a regular expression representing the empty language.

ε is a regular expression and it represents the singleton language $\{\varepsilon\}$.

Every letter a of Σ is a regular expression representing the singleton language $\{a\}$.

Next we define **regular expressions** over alphabet Σ. They are syntactic expressions that represent certain languages. If r is a regular expression then we denote the language it represent as $L(r)$.

$\emptyset$ is a regular expression representing the empty language.

ε is a regular expression and it represents the singleton language $\{\varepsilon\}$.

Every letter a of Σ is a regular expression representing the singleton language $\{a\}$.

If r and s are arbitrary regular expressions then $(r+s)$, (rs) and (r^*) are regular expressions. If $L(r) = R$ and $L(s) = S$ then

$$\begin{aligned} L(r+s) &= R \cup S \\ L(rs) &= RS \\ L(r^*) &= R^* \end{aligned}$$

We may remove parentheses from regular expressions using the following precedence rules:

- the Kleene star * has highest precedence,
- the concatenation has second highest precedence,
- the union $+$ has lowest precedence.

We may remove parentheses from regular expressions using the following precedence rules:

- the Kleene star * has highest precedence,
- the concatenation has second highest precedence,
- the union $+$ has lowest precedence.

Because concatenation and union are associative, we may

- simplify $r(st)$ and $(rs)t$ into rst, and
- simplify $r + (s + t)$ and $(r + s) + t$ into $r + s + t$.

Example.

$$(((ab)^*) + (a(ba)))((a^*)b) = ((ab)^* + aba)a^*b.$$

Often we do not distinguish between a regular expression r and its language $L(r)$. We simplify notations by talking about language r.

Other shorthand notations:

- Expression rr^* may be denoted as r^+.
- Expression $\overbrace{rr\ldots r}^{n}$ may be abbreviated as r^n.

Example. Construct a regular expression for the following languages over the alphabet $\Sigma = \{a, b\}$:

1. $L = \{ab, ba\}$.
2. All words of Σ.
3. All words that start with a and end in b.
4. All words that contain aba as a subword.
5. Words that start with ab and end in ba.
6. Words that contain two b's separated by an even number of a's.

Example. Conversely, describe in English the following languages:

1. $a^*(ab)^*b^*$

2. $(ab + b)^*$

3. $(\varepsilon + b)(ab)^*(\varepsilon + a)$

Regular expressions can define exactly the same languages as DFA, NFA and ε-NFA, *i.e.*, the regular languages.

Theorem (Kleene 1956). Language L is recognized by some DFA (NFA, ε-NFA) if and only if there exists a regular expression for it.

Proof.

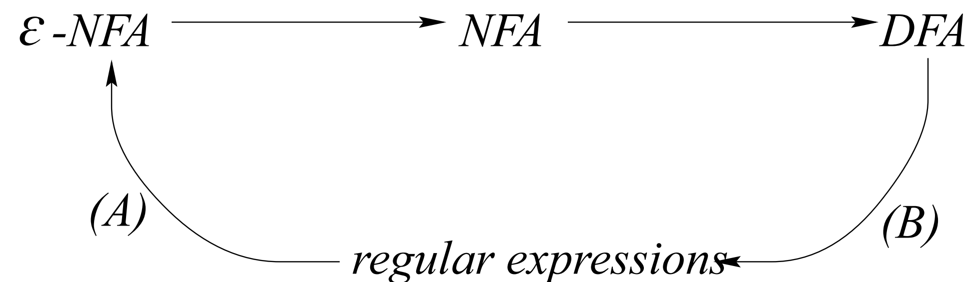
Theorem (Kleene 1956). Language L is recognized by some DFA (NFA, ε-NFA) if and only if there exists a regular expression for it.

Proof. To prove the theorem we show two directions:

(A) We show how to effectively construct for any given regular expression an equivalent ε-NFA.

(B) We show how to effectively construct for any given DFA an equivalent regular expression.

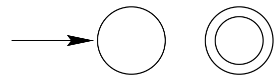
Together with our earlier proofs ε-NFA $\longrightarrow$ NFA and NFA $\longrightarrow$ DFA, constructions (A) and (B) allow us to transform any of the 4 devices into each other:

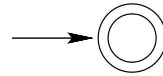


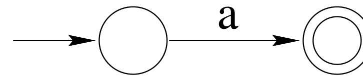
(A) Regular expression $\longrightarrow$ ε-NFA.

By induction on the size of the regular expression r we construct an ε-NFA A that has a **single final state** such that $L(A) = L(r)$.

1. Base case. Below are diagrams for automata that recognize $\emptyset$, $\{\varepsilon\}$ and $\{a\}$. Every machine has exactly one final state:

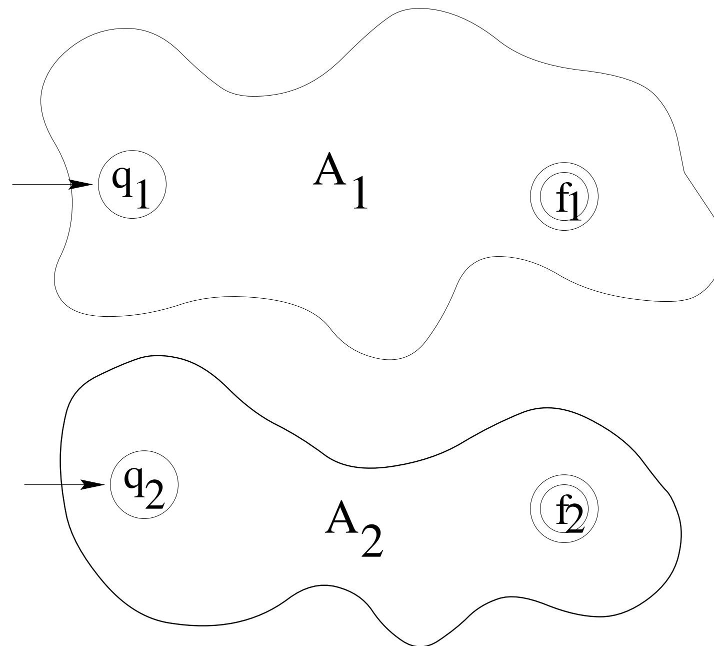




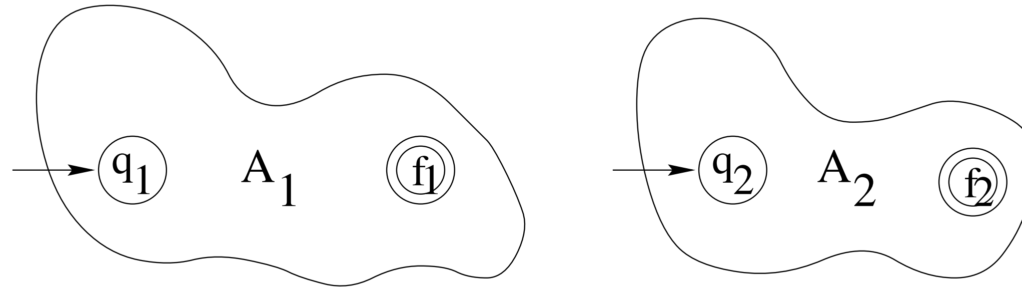


2. Inductive step. Assume we have ε-NFA A_1 and A_2 for regular expressions s and t, both with a single final state.

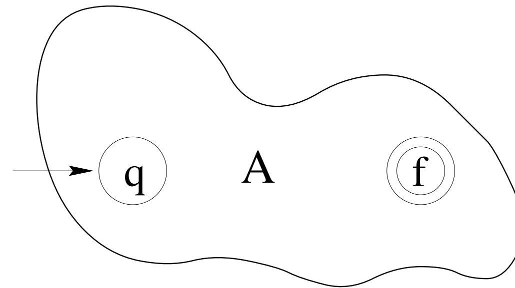
This is how we construct an ε-NFA for the union $s + t$:



This is an ε-NFA for the concatenation st:



Finally, let A accept s. This is an ε-NFA for the Kleene star s^*:



All constructions above work, and the resulting ε-NFA have exactly one final state. Using these constructions we can build an ε-NFA for any given regular expression. This completes the proof of (A).

Example. Let us construct an ε-NFA for regular expression

$$(abb^* + a)^*.$$

(B) DFA $\longrightarrow$ Regular expression

Let

$$A = (Q, \Sigma, \delta, q_0, F)$$

be the given DFA. States can be renamed as integers from 1 to n:

$$Q = \{1, 2, \ldots, n\}.$$

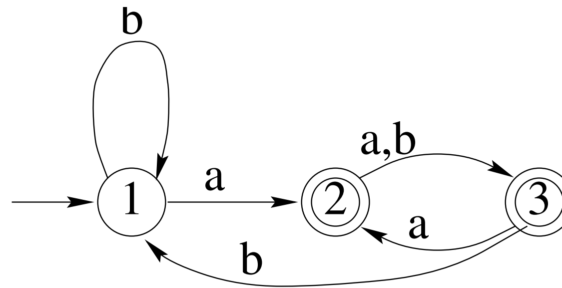
In this construction we build regular expressions for the following languages R_{ij}^k:

$$R_{ij}^k = \{w \mid \ w \text{ takes the automaton from state } i \text{ to state } j \text{ without passing through any states greater than } k.\}.$$

"Passing through" a state means that the node is along the computation path, excluding the starting and ending points.

$R_{ij}^k = \{w \mid$ w takes the automaton from state i to state j without passing through any states greater than $k.\}$.

Example. In



the word $bbab$ belongs to R_{33}^2 but the word word $baaab$ does not.

Also the word $bbaba$ is not in R_{33}^2 since its computation path does not end in state 3.

We construct regular expressions r_{ij}^k for the languages R_{ij}^k. We start with $k = 0$ and then move up to larger values of k.

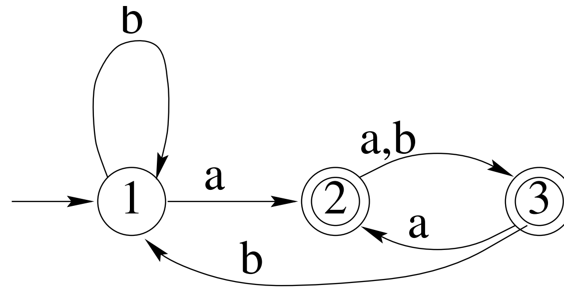
We construct regular expressions r_{ij}^k for the languages R_{ij}^k. We start with $k = 0$ and then move up to larger values of k.

1) $k = 0$**.** Language R_{ij}^0 contains all words that take the automaton from state i into state j without going through **any** states at all. Only possible input words are ε (if $i = j$) and single letters a (if $\delta(i, a) = j$).

If $a_1, a_2, \ldots, a_p$ are the input letters with transitions from state i to j then R_{ij}^0 has regular expression

- $a_1 + a_2 + \cdots + a_p$ if $i \neq j$ and $p \geq 1$,
- $\emptyset$ if $i \neq j$ and $p = 0$,
- $\varepsilon + a_1 + a_2 + \cdots + a_p$ if $i = j$ and $p \geq 1$,
- $\varepsilon+$ if $i = j$ and $p = 1$.

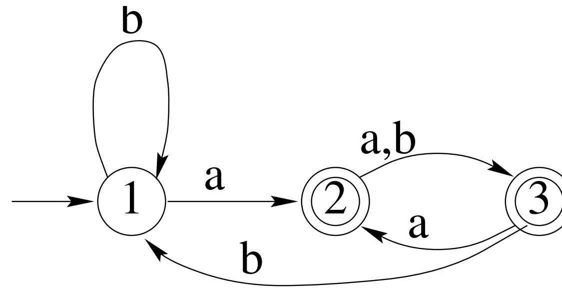
Example. In



we have

$$r_{11}^{0} =$$
$$r_{12}^{0} =$$
$$r_{13}^{0} =$$
$$r_{21}^{0} =$$
$$r_{22}^{0} =$$
$$r_{23}^{0} =$$
$$r_{31}^{0} =$$
$$r_{32}^{0} =$$
$$r_{33}^{0} =$$

Example. In

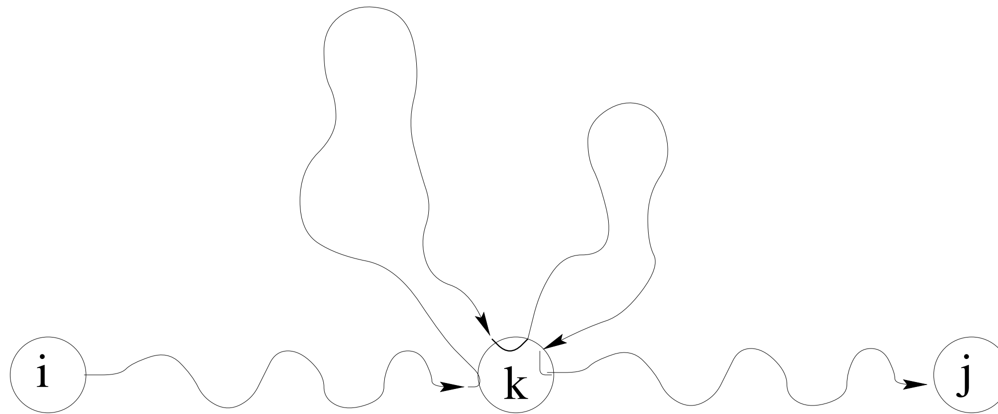


we have

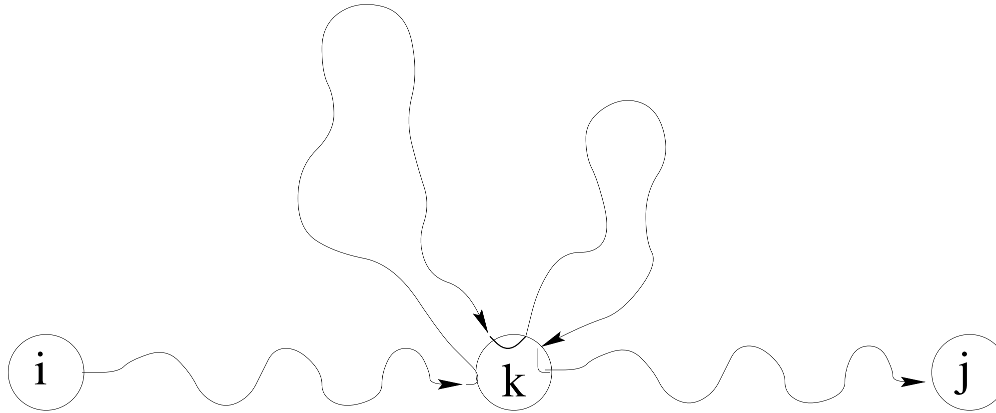
$$
\begin{aligned}
r_{11}^0 &= \varepsilon + b \\
r_{12}^0 &= a \\
r_{13}^0 &= \emptyset \\
r_{21}^0 &= \emptyset \\
r_{22}^0 &= \varepsilon \\
r_{23}^0 &= a + b \\
r_{31}^0 &= b \\
r_{32}^0 &= a \\
r_{33}^0 &= \varepsilon
\end{aligned}
$$

2) $k > 0$, and assume we have constructed regular expressions for all R_{ij}^{k-1}.

Consider an arbitrary computation path from state i into state j that only goes through states $\{1, 2, \ldots, k\}$. Let us cut the path into segments at points where it goes through state k:

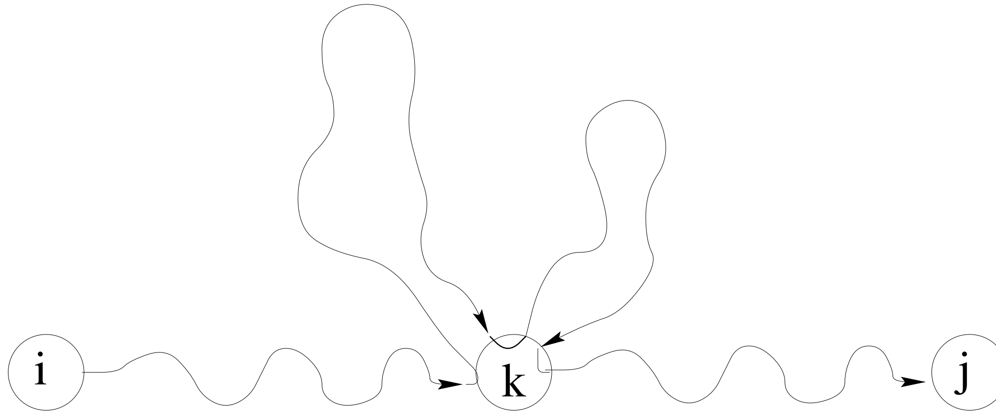


All segments only go through states $\{1, 2, \ldots, k-1\}$. The first segment is in the set R_{ik}^{k-1}, and the last segment belongs to R_{kj}^{k-1}. All middle segments start and end in state k, so they belong to R_{kk}^{k-1}. Number of middle segments can be arbitrary.



So words of R_{ij}^{k} that visit k at least once form the language $R_{ik}^{k-1}\left(R_{kk}^{k-1}\right)^{*}R_{kj}^{k-1}$.

We also must include words that do not visit state k even once, i.e., R_{ij}^{k-1}.



So words of R_{ij}^{k} that visit k at least once form the language $R_{ik}^{k-1}\left(R_{kk}^{k-1}\right)^{*}R_{kj}^{k-1}$.

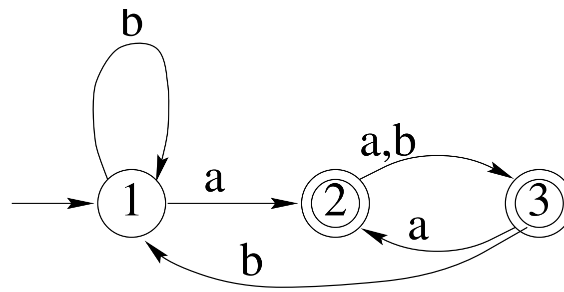
We also must include words that do not visit state k even once, i.e., R_{ij}^{k-1}.

In total: $R_{ij}^{k} = R_{ik}^{k-1}\left(R_{kk}^{k-1}\right)^{*}R_{kj}^{k-1} \ \cup \ R_{ij}^{k-1}$

So the language R_{ij}^{k} is defined by the regular expression

$$r_{ik}^{k} = r_{ik}^{k-1}\,(r_{kk}^{k-1})^{*}\,r_{kj}^{k-1} \ + \ r_{ij}^{k-1}$$

Example.



$r^0_{11} = \varepsilon + b$	$r^1_{11} =$	$r^2_{11} =$	$r^3_{11} =$
$r^0_{12} = a$	$r^1_{12} =$	$r^2_{12} =$	$r^3_{12} =$
$r^0_{13} = \emptyset$	$r^1_{13} =$	$r^2_{13} =$	$r^3_{13} =$
$r^0_{21} = \emptyset$	$r^1_{21} =$	$r^2_{21} =$	$r^3_{21} =$
$r^0_{22} = \varepsilon$	$r^1_{22} =$	$r^2_{22} =$	$r^3_{22} =$
$r^0_{23} = a + b$	$r^1_{23} =$	$r^2_{23} =$	$r^3_{23} =$
$r^0_{31} = b$	$r^1_{31} =$	$r^2_{31} =$	$r^3_{31} =$
$r^0_{32} = a$	$r^1_{32} =$	$r^2_{32} =$	$r^3_{32} =$
$r^0_{33} = \varepsilon$	$r^1_{33} =$	$r^2_{33} =$	$r^3_{33} =$

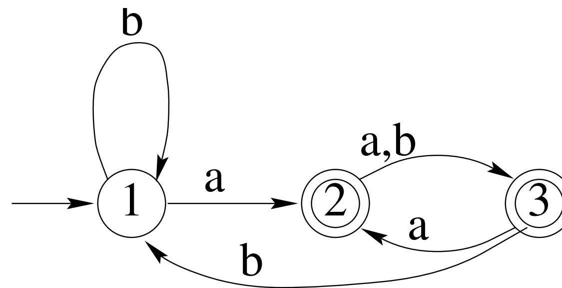
The regular expression r_{ij}^n represents all strings that take the automaton from state i to state j through any states. If $i = 1$ is the initial state then r_{1j}^n represents strings whose computation paths finish in state j.

If states $j_1, j_2, \ldots, j_f$ are the final states of the DFA then the language recognized by the automaton is represented by the expression

$$r_{1j_1}^n + r_{1j_2}^n + \cdots + r_{1j_f}^n.$$

These are all words that take the machine from the initial state to some final state, using any states whatsoever on the way.

Example.



The language recognized by this DFA is given by regular expression

$r^3_{12} + r^3_{13} =$

The construction provides huge regular expressions even for small automata. It is a good idea to simplify the expressions r_{ij}^k as you go along. Following simplifications are especially useful.

For any regular expressions r, s and t, both sides of following simplification rules describe the same language:

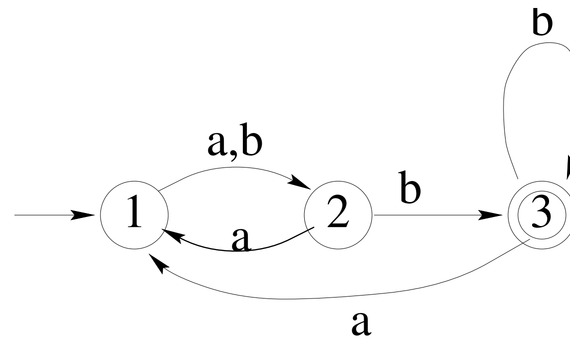
$$
\begin{array}{lcl}
\varepsilon^* & \longrightarrow & \varepsilon \\
\varepsilon r, r\varepsilon & \longrightarrow & r \\
(r+\varepsilon)^* & \longrightarrow & r^* \\
\emptyset r, r\emptyset & \longrightarrow & \emptyset \\
\emptyset + r, r + \emptyset & \longrightarrow & r \\
r(s+t) & \longrightarrow & rs + rt \\
(s+t)r & \longrightarrow & sr + tr \\
(r+\varepsilon)r^*, r^*(r+\varepsilon) & \longrightarrow & r^*
\end{array}
$$

Also, one may only construct only those expressions r_{ij}^k that are actually needed.

For example, in our sample DFA we only need two regular expressions r_{12}^3 and r_{13}^3 at level $k = 3$.

At level $k = 2$ we only need to construct expressions that are used at level 3: In our case r_{12}^2, r_{13}^2, r_{33}^2 and r_{32}^2. And so on.

Example. Let us find a regular expression equivalent to the following DFA:



Another **language equation** based method to construct a regular expression from a DFA. Again,we number the states from 1 to n:

$$Q = \{1, 2, \ldots, n\},$$

and assume that 1 is the initial state.

For every $i, j \in Q$, denote

$$L_i = \{w \in \Sigma^* \mid \delta(1, w) = i\}$$

and

$$K_{ij} = \{a \in \Sigma \mid \delta(i, a) = j\}.$$

Then the following equalities of languages hold:

$$\begin{aligned}
L_1 &= L_1K_{11} \cup L_2K_{21} \cup \ldots \cup L_nK_{n1} \cup \{\varepsilon\}, \\
L_2 &= L_1K_{12} \cup L_2K_{22} \cup \ldots \cup L_nK_{n2}, \\
&\vdots \\
L_n &= L_1K_{1n} \cup L_2K_{2n} \cup \ldots \cup L_nK_{nn}.
\end{aligned}$$

In each equation the language on the left and on the right are identical. The languages K_{ij} can be read directly from the given DFA – they are a representation of the transitions. The languages L_i are unknowns that we want to solve from the system of equations.

$$
\begin{aligned}
L_1 &= L_1K_{11} \cup L_2K_{21} \cup \ldots \cup L_nK_{n1} \cup \{\varepsilon\}, \\
L_2 &= L_1K_{12} \cup L_2K_{22} \cup \ldots \cup L_nK_{n2}, \\
&\vdots \\
L_n &= L_1K_{1n} \cup L_2K_{2n} \cup \ldots \cup L_nK_{nn}.
\end{aligned}
$$

Note that the languages K_{ij} do not contain the empty word ε. It turns out for such K_{ij} the languages L_i are uniquely determined from the system of equations.

The following lemma states this for the case $n = 1$:

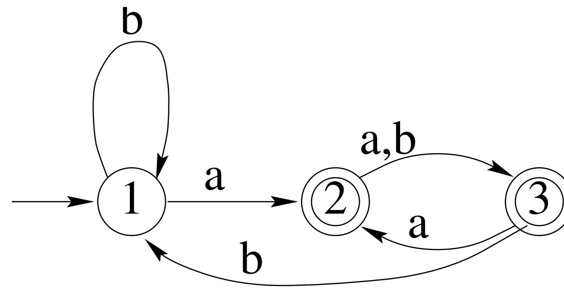
Lemma. Let $K, L \subseteq \Sigma^*$ be languages, and assume that $\varepsilon \notin K$. Then $X = LK^*$ is the unique language for which the equality $X = XK \cup L$ holds.

Proof.

$$
\begin{aligned}
L_1 &= L_1K_{11} \cup L_2K_{21} \cup \ldots \cup L_nK_{n1} \cup \{\varepsilon\}, \\
L_2 &= L_1K_{12} \cup L_2K_{22} \cup \ldots \cup L_nK_{n2}, \\
&\vdots \\
L_n &= L_1K_{1n} \cup L_2K_{2n} \cup \ldots \cup L_nK_{nn}.
\end{aligned}
$$

Using the Lemma on the first equation one may solve L_1, substitute the solution to the other equations, solve L_2 from the second equation, substitute it, and so on.

Example. Let us use the method of language equations to find a regular expression for the language recognized by



The pumping lemma

We have learned four types of devices for defining formal languages, and we have proved that all of them are able to describe exactly same languages, called **regular languages**.

We have procedures for converting any device into an equivalent device of any other type. The conversion procedures are mechanical algorithms in the sense that one can write a computer program that performs any conversion.

We say the devices are **effectively** equivalent.

It is usually fairly straightforward to prove that some language L is regular: one only needs to design a finite automaton (DFA, NFA or ε-NFA) or a regular expression, and to show that it defines language L.

But what about the **opposite problem**: How does one show that some language L is not regular? One needs to show that there does not exist any device that defines L, but one cannot try all DFA one-by-one because there are infinitely many of them.

(In general, proving that something cannot be done is harder than proving that it can be done, and that makes the negative question more interesting.)

Example. The language

$$L = \{a^n b^n \mid n \geq 0\}$$

contains all words that begin with any number of a's, followed by equally many b's. Let us show that there does not exist any DFA that recognizes L.

Example. The language

$$L = \{a^n b^n \mid n \geq 0\}$$

contains all words that begin with any number of a's, followed by equally many b's. Let us show that there does not exist any DFA that recognizes L.

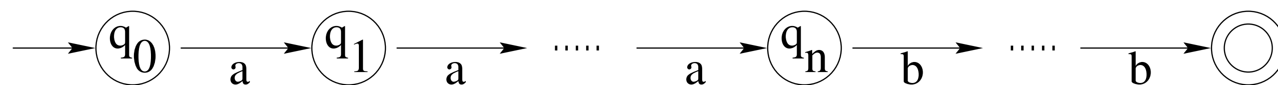
Informally: A DFA that accepts L should first count the number of a's in the beginning of the word. But a DFA has only finitely many states, so it will get confused: Some a^i and a^j take the DFA to the same state, and then it can no longer remember whether it saw i or j letters a. Since the DFA accepts input word $a^i b^i$, it also accepts input word $a^j b^i$, which is not in the language. So the DFA works incorrectly.

Example. $L = \{a^n b^n \mid n \geq 0\}$

More precisely: Assume that there exists a DFA

$$A = (Q, \Sigma, \delta, q_0, F)$$

such that $L = L(A)$. Let n be the number of states in Q. Consider the accepting path for the input word $a^n b^n$.

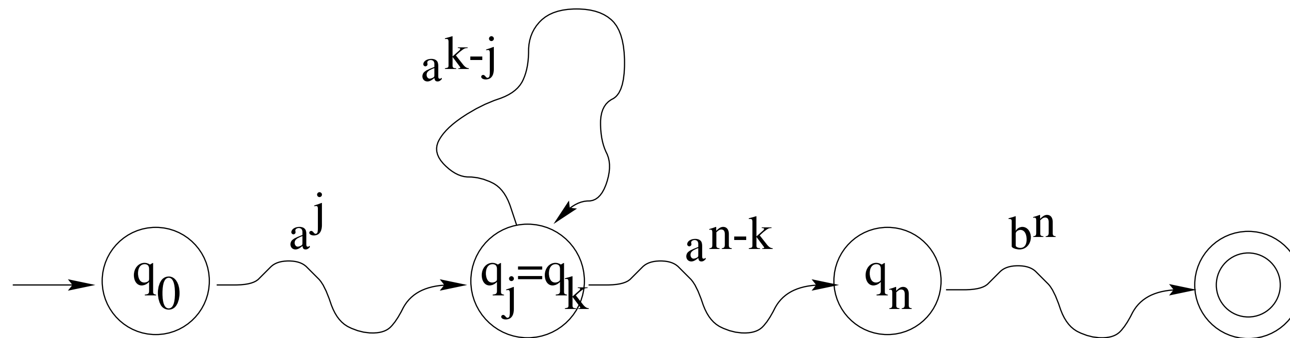


Let q_j be the state of A after reading the first j input letters a.

Example. $L = \{a^n b^n \mid n \geq 0\}$

There are $n+1$ states q_0, q_1, $\ldots$, q_n but the DFA has only n different states, so two of the states must be identical. (This is the **pigeon hole principle**: if you have more pigeons than holes then two or more pigeons have to share a hole.)

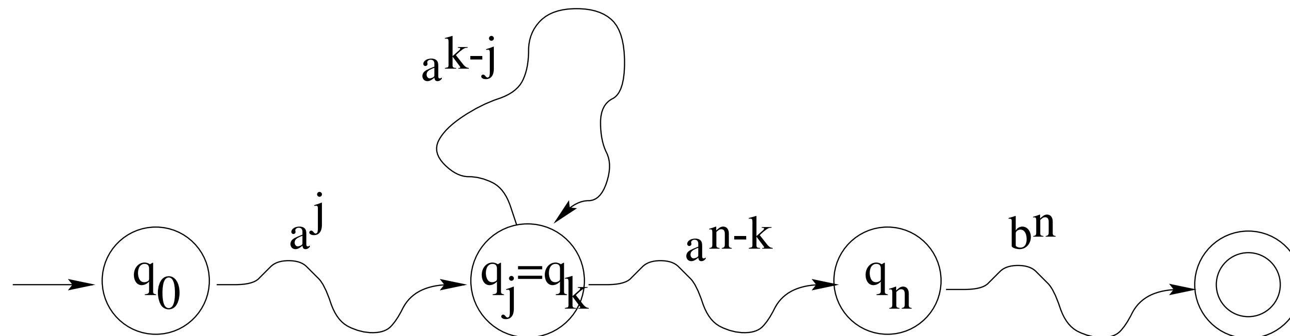
Let q_j and q_k be two identical states, where $j < k$:



Example. $L = \{a^n b^n \mid n \geq 0\}$

There are $n+1$ states q_0, q_1, $\dots$, q_n but the DFA has only n different states, so two of the states must be identical. (This is the **pigeon hole principle**: if you have more pigeons than holes then two or more pigeons have to share a hole.)

Let q_j and q_k be two identical states, where $j < k$:



The word a^{k-j} **loops** the DFA at state $q_j = q_k$. The loop can be repeated arbitrarily many times, always getting an accepting computation. For example, if we repeat the loop 0 times we have accepting computation for the word

$$a^j a^{n-k} b^n = a^{n-(k-j)} b^n.$$

This word is not in language L because $k - j \neq 0$. Therefore the DFA A is **not correct**: the language it recognizes is not L.

Similar **”confused in counting”** argument works with many other languages as well. Instead of always repeating the argument like above, we formulate the argumentation as a theorem known as the **pumping lemma**. (Word ”pumping” refers to the fact that the loop can be repeated, or ”pumped”, arbitrarily many times.)

Pumping lemma states a property that every regular language has. If a language does not have that property then the language is not regular.

Pumping lemma. Let L be a regular language. Then there exists some positive constant n such that every word z of length n or greater that belongs to language L can be divided into three segments

$$z = uvw$$

in such a way that

$$\begin{cases} |uv| \leq n, \text{ and} \\ v \neq \varepsilon, \end{cases} \qquad (*)$$

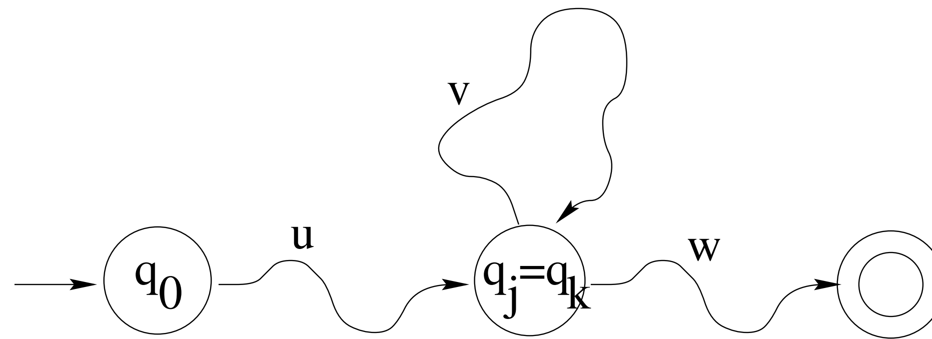
and for all $i \geq 0$ the word uv^iw is in language L.

(So every long enough word in L contains a non-empty subword that can be pumped arbitrarily many times, and the result always belongs to L.)

Proof of the pumping lemma. Let L be a regular language. Then it is accepted by some DFA A. Let n be the number of states in A.

Consider an arbitrary word $z \in L$ such that $|z| \geq n$. We have to show how to divide z into three segments in such a way that $(*)$ is satisfied.

We argue as follows: Let q_j be the state of the machine after the first j input letters of z have been read. The machine has n states so the pigeon hole principal tells that there must exist two identical states among $q_0, q_1, \ldots, q_n$, say $q_j = q_k$ for some $j < k \leq n$:



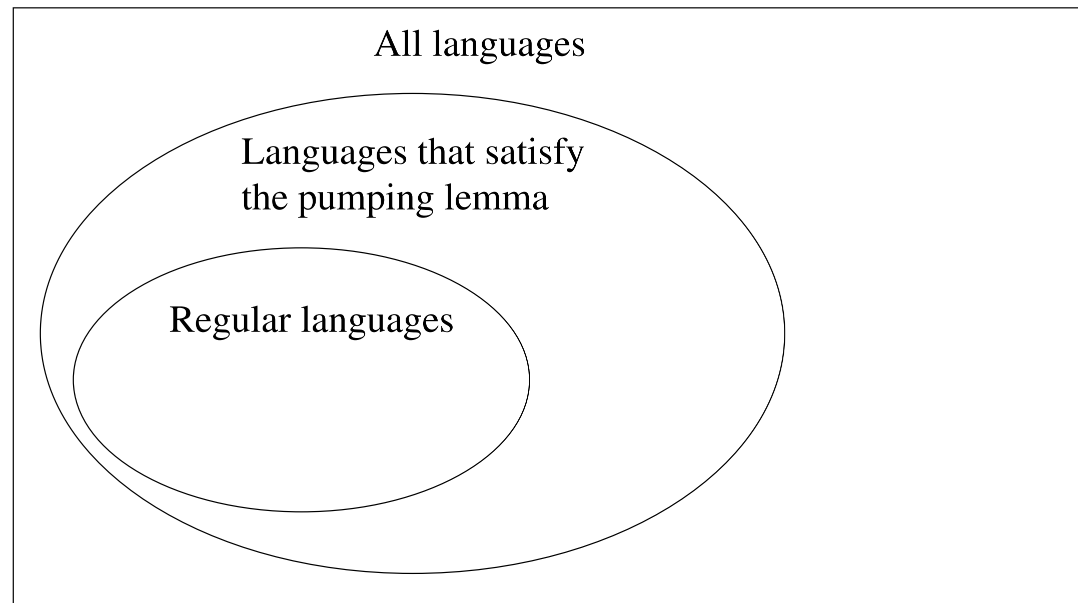
We divide the input word z into three segments in a natural way: u consists of the first j letters, v of the following $k - j$ letters, and w is the remaining tail. This division satisfies $(*)$:

$$|uv| = k \leq n, \text{ and}$$
$$v \neq \varepsilon \text{ because } |v| = k - j > 0.$$

The loop can be repeated i times for any i, so $uv^iw \in L$ for all $i \geq 0$.

The pumping lemma gives a property satisfied by every regular language. Therefore, if language L does not satisfy the pumping lemma then L is not regular. We can use the pumping lemma to show that certain languages are not regular.

We can **not** use the pumping lemma to prove that a language L is regular. The property is not ”if and only if”: There are some non-regular languages that satisfy the pumping lemma.



If you show that L satisfies the pumping lemma then language L can still be non-regular. But if you show that L does not satisfy the pumping lemma then L is guaranteed not to be regular.

So we are more interested in the negation of the pumping lemma. The pumping lemma says that if L is regular then

$$(\exists n)(\forall z \ldots)(\exists u, v, w \ldots)(\forall i) uv^i w \in L.$$

Therefore if L satisfies the opposite statement

$$(\forall n)(\exists z \ldots)(\forall u, v, w \ldots)(\exists i) uv^i w \notin L$$

then L is not regular.

So we are more interested in the negation of the pumping lemma. The pumping lemma says that if L is regular then

$$(\exists n)(\forall z \ldots)(\exists u, v, w \ldots)(\forall i) uv^i w \in L.$$

Therefore if L satisfies the opposite statement

$$(\forall n)(\exists z \ldots)(\forall u, v, w \ldots)(\exists i) uv^i w \notin L$$

then L is not regular.

This is what you do to show that L is not regular:

(1) For **every** n, select a word $z \in L$, $|z| \geq n$.

(2) Show that for **every** division $z = uvw$ where $|uv| \leq n$ and $v \neq \varepsilon$ **there exists** a number i such that

$$uv^i w$$

is **not** in language L.

If you can do (1) and (2) then L does not satisfy the pumping lemma, and L is not regular.

Example. Consider again $L = \{a^n b^n \mid n \geq 0\}$.

(1) For any given n select $z = a^n b^n$. The choice is good since $z \in L$ and $|z| \geq n$.

(2) Consider an arbitrary division of z into three part $z = uvw$ where

$$|uv| \leq n \text{ and } v \neq \varepsilon.$$

Then necessarily $u = a^j$ and $uv = a^k$ for some j and k, and $j < k$.

Choosing $i = 0$ gives

$$uv^i w = uw = a^{n-(k-j)} b^n$$

which does not belong to L.

Example 2. Let us prove that

$$L = \{a^{m^2} \mid m \geq 1\} = \{a, a^4, a^9, a^{16}, \dots\}$$

is not regular.

Example 2. Let us prove that

$$L = \{a^{m^2} \mid m \geq 1\} = \{a, a^4, a^9, a^{16}, \dots\}$$

is not regular.

(1) For given n let us choose

$$z = a^{n^2}.$$

Example 2. Let us prove that

$$L = \{a^{m^2} \mid m \geq 1\} = \{a, a^4, a^9, a^{16}, \dots\}$$

is not regular.

(1) For given n let us choose

$$z = a^{n^2}.$$

(2) Consider an arbitrary division $z = uvw$ that satisfies

$$|uv| \leq n \text{ and } v \neq \varepsilon.$$

Let k denote the length of v. It follows from that $0 < k \leq n$.

Pump v with $i =$

Example 2. Let us prove that

$$L = \{a^{m^2} \mid m \geq 1\} = \{a, a^4, a^9, a^{16}, \dots\}$$

is not regular.

(1) For given n let us choose

$$z = a^{n^2}.$$

(2) Consider an arbitrary division $z = uvw$ that satisfies

$$|uv| \leq n \text{ and } v \neq \varepsilon.$$

Let k denote the length of v. It follows from that $0 < k \leq n$.

Pump v with $i = 2$. Then

$$uv^iw = uvvw = a^{n^2+k}$$

Because

$$n^2 = n^2 + 0 < n^2 + k \leq n^2 + n < (n+1)^2$$

number $n^2 + k$ is not a square of any integer. (It falls between two consecutive squares.) Therefore uv^2w does not belong to L.

Some more examples:

1. Is the language $L = \{ww \mid w \in \{a, b\}^*\}$ regular ?

2. Is the language $L = \{ww \mid w \in \{a\}^*\}$ regular ?

3. Is the language $L = \{a^p \mid p \text{ is a prime number}\}$ regular ?

Closure properties

It follows from the definition of regular expressions that the **union** $L_1 \cup L_2$ of any two regular languages L_1 and L_2 is regular. Namely, if r_1 and r_2 are regular expressions for L_1 and L_2 then $r_1 + r_2$ is a regular expression for the union $L_1 \cup L_2$.

In the same way, the **concatenation** of two regular languages is always regular: $r_1 r_2$ is a regular expression for the concatenation.

We say that the family of regular languages is **closed** under union and concatenation.

In general, let Op denote some **language operation**, that is, an operation whose operands and result are all formal languages.

We say that a family $\mathcal{F}$ of languages is **closed under the operation** Op if

$$L_1, L_2, \dots \in \mathcal{F} \Longrightarrow Op(L_1, L_2, \dots) \in \mathcal{F}.$$

(*E.g.*, the family of regular languages is closed under operation Op if the result after applying Op on regular languages is always a regular language.)

In general, let Op denote some **language operation**, that is, an operation whose operands and result are all formal languages.

We say that a family $\mathcal{F}$ of languages is **closed under the operation** Op if

$$L_1, L_2, \dots \in \mathcal{F} \Longrightarrow Op(L_1, L_2, \dots) \in \mathcal{F}.$$

(*E.g.*, the family of regular languages is closed under operation Op if the result after applying Op on regular languages is always a regular language.)

Next we investigate the closure properties of regular languages under some common operations. We start the familiar boolean operations: **union**, **intersection** and **complement**. We already know the closure under union.

Consider next the **complement**. Let $L \subseteq \Sigma^*$ be a regular language over alphabet Σ. Is its complement

$$\bar{L} = \Sigma^* \setminus L = \{w \in \Sigma^* \mid w \notin L\}$$

necessarily regular?

Consider next the **complement**. Let $L \subseteq \Sigma^*$ be a regular language over alphabet Σ. Is its complement

$$\bar{L} = \Sigma^* \setminus L = \{w \in \Sigma^* \mid w \notin L\}$$

necessarily regular?

The answer is **yes**, the family of regular languages is closed under complement.

Consider a **DFA** A **that recognizes** L. Without loss of generality we may assume that A uses alphabet Σ, the same alphabet relative to which the complement is defined.

Every input word w takes A to some state q. The word w is accepted if and only if state q is among the final states.

Let us change A by making every final state a non-final state, and vice versa:

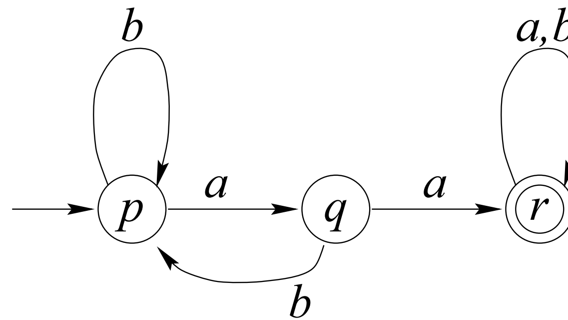
final $\longleftrightarrow$ non-final

The new DFA accepts word w if and only if the original DFA did not accept w. So the new DFA recognizes the complement of L.

Example. Let us find a DFA for the complement

$$\{a, b\}^* \setminus L(A)$$

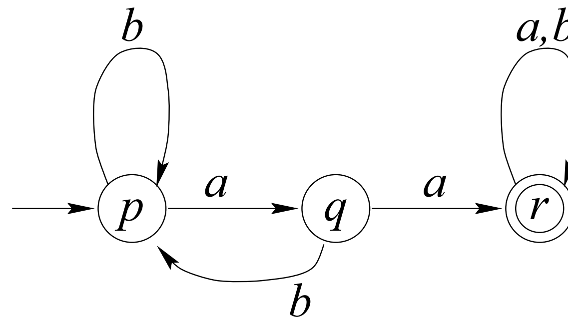
of the language $L(A)$ recognized by DFA



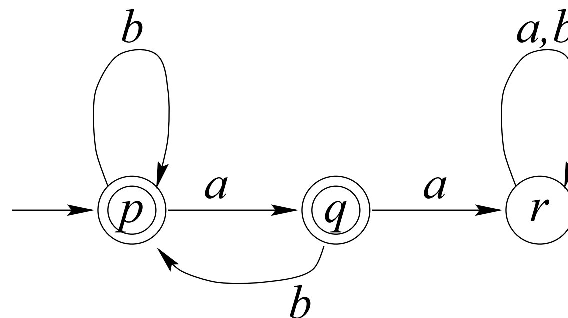
Example. Let us find a DFA for the complement

$$\{a, b\}^* \setminus L(A)$$

of the language $L(A)$ recognized by DFA



This DFA recognizes the complement of $L(A)$:



What about **intersection** ? Elementary set theory (**de Morgan's law**) tells us how to use union and complement to do intersection. For any languages L_1 and L_2

$$L_1 \cap L_2 = \overline{\bar{L}_1 \cup \bar{L}_2}$$

So

L_1 and L_2 regular $\Longrightarrow$ $\bar{L}_1$ and $\bar{L}_2$ regular $\Longrightarrow$ $\bar{L}_1 \cup \bar{L}_2$ regular $\Longrightarrow$ $\overline{\bar{L}_1 \cup \bar{L}_2}$ regular

Remark. All closures so far are **effective** in the sense that we can algorithmically construct the result of the operation on any given regular languages.

• The union can be constructed by inserting a plus sign between the regular expressions of the input languages,

• the complement can be formed by swapping the final states of a DFA,

• the intersection can be done by performing a sequence of effective operations for union and complementation.

It is irrelevant which description we use for the operands: they may be given as a DFA, NFA, ε-NFA or regular expression. All formats are effectively equivalent since we can mechanically convert any device into an equivalent device of any other type.

We say that the family of regular languages is **effectively closed** under operation *Op* if there exists an algorithm (=mechanical procedure) that produces the result of the operation for any given regular input languages. The format of the inputs and outputs can be arbitrary since we have algorithms for converting from one format to any other.

We have proved the following theorem:

Theorem. The family of regular languages is effectively closed under union, concatenation, Kleene star, complementation and intersection.

The closure properties can be used in proving languages regular. It is enough to show how to build the language from known regular languages using above operations.

Example. The language L containing all words over the English alphabet

$$\Sigma = \{a, b, c, \ldots, z\}$$

that do not contain the word utu as a subword is regular. Namely, it is the complement of the language

$$\Sigma^* utu \Sigma^*$$

of all words that do contain utu as a subword.

We can also use closure properties to show that some languages are **not** regular.

Example. Let us show that

$$L = \{a^n c^m b^n \mid n, m \geq 0\}$$

is not regular.

Assume the contrary: assume that L is regular. Then also the language

$$L \cap a^* b^*$$

is regular because the family of regular languages is closed under intersection and $a^* b^*$ is regular. But we already know that

$L \cap a^* b^* = \{a^n b^n \mid n \geq 0\}$

is **not** regular. Therefore our assumption that L is regular has to be false.

Another example. The language

$$L = \{a^{p-1} \mid p \text{ is a prime number } \}$$

is not regular.

+ If you start with known regular languages, apply operations (under which reg. languages are closed) and end up with language L, then L is a regular language.

+ If you start with L, apply operations to L and known regular languages, and end up with a language that is known to be non-regular, then L is non-regular.

+ If you start with known regular languages, apply operations (under which reg. languages are closed) and end up with language L, then L is a regular language.

+ If you start with L, apply operations to L and known regular languages, and end up with a language that is known to be non-regular, then L is non-regular.

BUT DO NOT USE THE WRONG DIRECTION:

– Starting with L and applying operations on L and regular languages you end up with a known regular language. This does not prove a thing about L: it can be either regular or non-regular.

– Starting with known non-regular languages you apply operations and end up with L. Also this does not prove anything about L.

Next we introduce some new language operations.

Let Σ and Δ be two alphabets. A **homomorphism** is a function

$$h : \Sigma \longrightarrow \Delta^*,$$

that assigns to each letter $a \in \Sigma$ a word $h(a) \in \Delta^*$.

Homomorphisms are **applied to words** by coding each letter separately and concatenating the results together:

$$h(a_1 a_2 \dots a_n) = h(a_1) h(a_2) \dots h(a_n)$$

Example. Homomorphism

$$h(a) = 0,$$
$$h(b) = 01$$

is from $\Sigma = \{a, b\}$ to $\Delta = \{0, 1\}$. We have

$$\begin{aligned} h(ba) &= \\ h(babba) &= \\ h(\varepsilon) &= \end{aligned}$$

Example. Homomorphism

$h(a) = 0,$
$h(b) = 01$

is from $\Sigma = \{a, b\}$ to $\Delta = \{0, 1\}$. We have

$$
\begin{aligned}
h(ba) &= 010 \\
h(babba) &= 01001010 \\
h(\varepsilon) &= \varepsilon
\end{aligned}
$$

Homomorphisms are **applied to languages** as well: The homomorphic image $h(L)$ of a language L consists of all homomorphic images of all L's words:

$$h(L) = \{h(w) \mid w \in L\} = \bigcup_{w \in L} \{h(w)\}$$

Homomorphisms are **applied to languages** as well: The homomorphic image $h(L)$ of a language L consists of all homomorphic images of all L's words:

$$h(L) = \{h(w) \mid w \in L\} = \bigcup_{w \in L} \{h(w)\}$$

Example. Our sample homomorphism

$$h(a) = 0,$$
$$h(b) = 01$$

gives

$$\begin{aligned} h(b + ab + bbb) &= h(\{b, ab, bbb\}) && = \\ h(b^*) &= h(\{\varepsilon, b, bb, bbb, \ldots\}) && = \\ h(a^*b + bb) &= \end{aligned}$$

Homomorphisms are **applied to languages** as well: The homomorphic image $h(L)$ of a language L consists of all homomorphic images of all L's words:

$$h(L) = \{h(w) \mid w \in L\} = \bigcup_{w \in L} \{h(w)\}$$

Example. Our sample homomorphism

$h(a) = 0,$
$h(b) = 01$

gives

$$\begin{aligned}
h(b + ab + bbb) &= h(\{b, ab, bbb\}) &&= \{01, 001, 010101\} = 01 + 001 + 010101 \\
h(b^*) &= h(\{\varepsilon, b, bb, bbb, \dots\}) &&= \{\varepsilon, 01, 0101, 010101, \dots\} = (01)^* \\
h(a^*b + bb) &= 0^*01 + 0101
\end{aligned}$$

A **substitution** is a generalization of the notion of homomorphism. A substitution assigns a **language to each letter**. For example

$$s(a) = 0 + 11$$
$$s(b) = 0^*10^*$$

A **substitution** is a generalization of the notion of homomorphism. A substitution assigns a **language to each letter**. For example

$$s(a) = 0 + 11$$
$$s(b) = 0^*10^*$$

A substitution s is **regular** if $s(a)$ is a regular language for each $a \in \Sigma$. It is a **finite** substitution if $s(a)$ is a finite language for each $a \in \Sigma$.

A **substitution** is a generalization of the notion of homomorphism. A substitution assigns a **language to each letter**. For example

$$s(a) = 0 + 11$$
$$s(b) = 0^*10^*$$

A substitution s is **regular** if $s(a)$ is a regular language for each $a \in \Sigma$. It is a **finite** substitution if $s(a)$ is a finite language for each $a \in \Sigma$.

Substitution s is **applied to words** by applying it to each letter separately and concatenating the results (which are languages!) together:

$$s(a_1 a_2 \dots a_n) = s(a_1)s(a_2)\dots s(a_n).$$

A **substitution** is a generalization of the notion of homomorphism. A substitution assigns a **language to each letter**. For example

$$s(a) = 0 + 11$$
$$s(b) = 0^*10^*$$

A substitution s is **regular** if $s(a)$ is a regular language for each $a \in \Sigma$. It is a **finite** substitution if $s(a)$ is a finite language for each $a \in \Sigma$.

Substitution s is **applied to words** by applying it to each letter separately and concatenating the results (which are languages!) together:

$$s(a_1 a_2 \ldots a_n) = s(a_1)s(a_2) \ldots s(a_n).$$

Finally, substitution s **applied to a language** L is the language consisting of all words that can be obtained by applying the substitution on L's words:

$$s(L) = \{u \mid u \in s(w) \text{ for some } w \in L\} = \bigcup_{w \in L} s(w).$$

A **substitution** is a generalization of the notion of homomorphism. A substitution assigns a **language to each letter**. For example

$$s(a) = 0 + 11$$
$$s(b) = 0^*10^*$$

A substitution s is **regular** if $s(a)$ is a regular language for each $a \in \Sigma$. It is a **finite** substitution if $s(a)$ is a finite language for each $a \in \Sigma$.

Substitution s is **applied to words** by applying it to each letter separately and concatenating the results (which are languages!) together:

$$s(a_1 a_2 \ldots a_n) = s(a_1)s(a_2)\ldots s(a_n).$$

Finally, substitution s **applied to a language** L is the language consisting of all words that can be obtained by applying the substitution on L's words:

$$s(L) = \{u \mid u \in s(w) \text{ for some } w \in L\} = \bigcup_{w \in L} s(w).$$

Remark: Every homomorphism is a special type of substitution where each $s(a)$ consists of a single word.

Example.

$$s(a) = 0 + 11 \qquad f(a) = 000 + 11$$
$$s(b) = 0^*10^* \qquad f(b) = 01 + \varepsilon$$

are substitutions from $\Sigma = \{a, b\}$ to $\Delta = \{0, 1\}$.

Here, s is a regular substitution and f is a finite substitution. (Of course, every finite substitution is also regular.)

Example.

$$s(a) = 0 + 11 \qquad f(a) = 000 + 11$$
$$s(b) = 0^*10^* \qquad f(b) = 01 + \varepsilon$$

are substitutions from $\Sigma = \{a, b\}$ to $\Delta = \{0, 1\}$.

Here, s is a regular substitution and f is a finite substitution. (Of course, every finite substitution is also regular.)

$$\begin{aligned} f(ba) &= \\ s(aba) &= \\ s(\varepsilon) &= \end{aligned}$$

Example.

$$s(a) = 0 + 11 \qquad f(a) = 000 + 11$$
$$s(b) = 0^*10^* \qquad f(b) = 01 + \varepsilon$$

are substitutions from $\Sigma = \{a, b\}$ to $\Delta = \{0, 1\}$.

Here, s is a regular substitution and f is a finite substitution. (Of course, every finite substitution is also regular.)

$$\begin{aligned}
f(ba) &= (01 + \varepsilon)(000 + 11) = \{01000, 0111, 000, 11\} \\
s(aba) &= (0 + 11)0^*10^*(0 + 11) \\
s(\varepsilon) &= \{\varepsilon\}
\end{aligned}$$

$$\begin{aligned}
s(ba + aba) &= s(ba) \cup s(aba) = \\
f(ba^*) &= \\
s(ab + b^*) &= \\
s(\varepsilon) &= \\
s(\emptyset) &=
\end{aligned}$$

Example.

$$s(a) = 0 + 11 \qquad f(a) = 000 + 11$$
$$s(b) = 0^*10^* \qquad f(b) = 01 + \varepsilon$$

are substitutions from $\Sigma = \{a, b\}$ to $\Delta = \{0, 1\}$.

Here, s is a regular substitution and f is a finite substitution. (Of course, every finite substitution is also regular.)

$$\begin{aligned}
f(ba) &= (01 + \varepsilon)(000 + 11) = \{01000, 0111, 000, 11\} \\
s(aba) &= (0 + 11)0^*10^*(0 + 11) \\
s(\varepsilon) &= \{\varepsilon\}
\end{aligned}$$

$$\begin{aligned}
s(ba + aba) &= s(ba) \cup s(aba) = 0^*10^*(0 + 11) + (0 + 11)0^*10^*(0 + 11) \\
f(ba^*) &= (01 + \varepsilon)(000 + 11)^* \\
s(ab + b^*) &= (0 + 11)0^*10^* + (0^*10^*)^* \\
s(\varepsilon) &= \{\varepsilon\} \\
s(\emptyset) &= \emptyset
\end{aligned}$$

Lemma. For any substitution s and languages L and K

$$\begin{aligned} s(L \cup K) &= s(L) \cup s(K), \\ s(LK) &= s(L)s(K), \\ s(L^*) &= s(L)^*. \end{aligned}$$

Lemma. For any substitution s and languages L and K

$$
\begin{aligned}
s(L \cup K) &= s(L) \cup s(K), \\
s(LK) &= s(L)s(K), \\
s(L^*) &= s(L)^*.
\end{aligned}
$$

Corollary. If L is a regular language and s a regular substitution then $s(L)$ is (effectively) a regular language. In other words, the family of regular languages is effectively closed under regular substitutions.

Proof. Induction on the size of the regular expression r for the language L.

Lemma. For any substitution s and languages L and K

$$\begin{aligned} s(L \cup K) &= s(L) \cup s(K), \\ s(LK) &= s(L)s(K), \\ s(L^*) &= s(L)^*. \end{aligned}$$

Corollary. If L is a regular language and s a regular substitution then $s(L)$ is (effectively) a regular language. In other words, the family of regular languages is effectively closed under regular substitutions.

Proof. Induction on the size of the regular expression r for the language L.

A regular expression for $s(L)$ is obtained by replacing every occurrence of every letter a in the regular expression r for L by the regular expression r_a for the language $s(a)$. For example,

$$\begin{aligned} s(a^*b + bb) &= s(a^*b) + s(bb) \\ &= s(a^*)s(b) + s(b)s(b) \\ &= s(a)^*s(b) + s(b)s(b) \\ &= r_a^* r_b + r_b r_b. \end{aligned}$$

Corollary. The family of regular languages is closed under homomorphisms.

Proof. Every homomorphism is a regular substitution.

Corollary. The family of regular languages is closed under homomorphisms.

Proof. Every homomorphism is a regular substitution.

Example. Using homomorphisms and substitutions

$h(a) = 0$	$s(a) = 0 + 11$	$f(a) = 000 + 11$
$h(b) = 01$	$s(b) = 0^*10^*$	$f(b) = 01 + \varepsilon$

we have

$$
\begin{aligned}
h((ab^*aa + b)^* + aba) &= \\
s(ab^* + bba) &= \\
f(\emptyset + \varepsilon + a) &=
\end{aligned}
$$

Corollary. The family of regular languages is closed under homomorphisms.

Proof. Every homomorphism is a regular substitution.

Example. Using homomorphisms and substitutions

$$h(a) = 0 \qquad s(a) = 0 + 11 \qquad f(a) = 000 + 11$$
$$h(b) = 01 \qquad s(b) = 0^*10^* \qquad f(b) = 01 + \varepsilon$$

we have

$$\begin{aligned}
h((ab^*aa + b)^* + aba) &= (0(01)^*00 + 01)^* + 0010 \\
s(ab^* + bba) &= (0 + 11)(0^*10^*)^* + 0^*10^*0^*10^*(0 + 11) \\
f(\emptyset + \varepsilon + a) &= \emptyset + \varepsilon + 000 + 11
\end{aligned}$$

Closure under homomorphisms and regular substitutions can be used in proving non-regularity of languages.

Example. Let us prove that

$$L = \{(ab)^p \mid p \text{ is a prime number } \}$$

is not regular. We already know that

$$L_p = \{a^p \mid p \text{ is a prime number } \}$$

is not regular, so let us **reduce** L into L_p.

Closure under homomorphisms and regular substitutions can be used in proving non-regularity of languages.

Example. Let us prove that

$$L = \{(ab)^p \mid p \text{ is a prime number }\}$$

is not regular. We already know that

$$L_p = \{a^p \mid p \text{ is a prime number }\}$$

is not regular, so let us **reduce** L into L_p.

Assume that L is regular. Then also $h(L)$ is regular where h is the homomorphism

$$h(a) = a,$$
$$h(b) = \varepsilon.$$

But $h(L) = L_p$, which is not regular. Therefore L is not regular either.

Our next language operation: the **inverse homomorphism**.

Let h be a homomorphism from Σ to Δ. The inverse homomorphism h^{-1} is defined on languages of alphabet Δ.

For any $L \subseteq \Delta^*$,

$$h^{-1}(L) = \{w \in \Sigma^* \mid h(w) \in L\}$$

(The language $h^{-1}(L)$ consists of all words over Σ that are mapped by the homomorphism h into L.)

If L contains only one word we may simply write $h^{-1}(w)$ instead of $h^{-1}(\{w\})$.

Example. Using the homomorphism

$$g(a) = 01,$$
$$g(b) = 011,$$
$$g(c) = 101$$

we have

$$\begin{aligned} g^{-1}(011101) &= \\ g^{-1}(01101) &= \\ g^{-1}(010) &= \\ g^{-1}((10)^*1) &= \end{aligned}$$

Example. Using the homomorphism

$$\begin{aligned} g(a) &= 01, \\ g(b) &= 011, \\ g(c) &= 101 \end{aligned}$$

we have

$$\begin{aligned} g^{-1}(011101) &= \{bc\} \\ g^{-1}(01101) &= \{ac, ba\} \\ g^{-1}(010) &= \emptyset \\ g^{-1}((10)^*1) &= ca^* \end{aligned}$$

Theorem. The family of regular languages is (effectively) closed under inverse homomorphism.

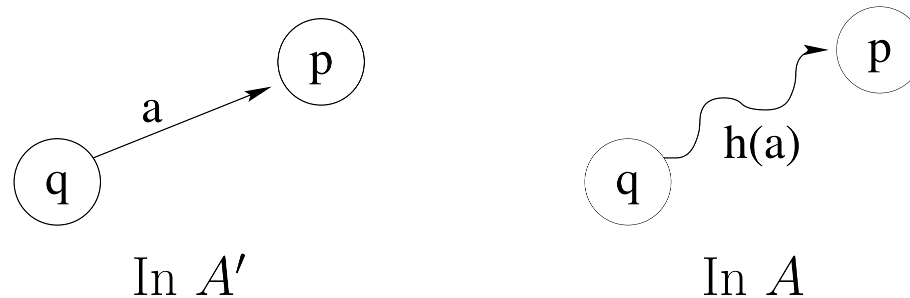
Proof. Let L be a regular language, and h a homomorphism. Let A be a DFA recognizing L. We construct a DFA A' that recognizes $h^{-1}(L)$.

Theorem. The family of regular languages is (effectively) closed under inverse homomorphism.

Proof. Let L be a regular language, and h a homomorphism. Let A be a DFA recognizing L. We construct a DFA A' that recognizes $h^{-1}(L)$.

A' has the same states as A. Also the initial state and final states are identical. Only the transitions are different: an input symbol a from state q takes A' into the same state that the input word $h(a)$ takes A from the same state q:

$$\delta'(q, a) = \delta(q, h(a)).$$



Computations by A' simulate computations by A: After reading input

$$w = a_1 a_2 \ldots a_n$$

A' is in the same state as A is after reading the input

$$h(w) = h(a_1)h(a_2)\ldots h(a_n).$$

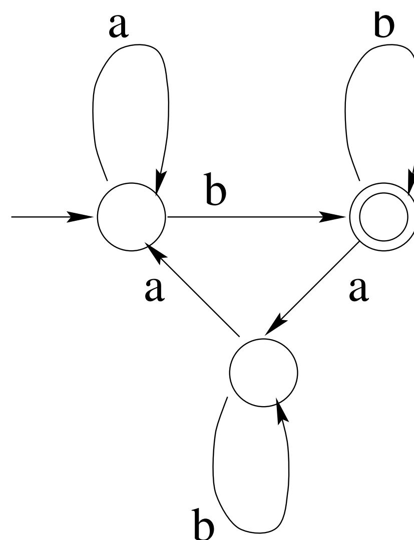
Since the same states are final states in both machines, DFA A' accepts w if and only if DFA A accepts $h(w)$.

In other words, the language recognized by A' is $h^{-1}(L)$.

Example. Let

$$\begin{aligned} h(0) &= ab, \\ h(1) &= abb, \\ h(2) &= bab, \end{aligned}$$

and let language L be defined by the DFA A



Let us construct a DFA A' that recognizes $h^{-1}(L)$.

Example. Let us prove that the language

$$L = \{0^n 1 0^{2n} \mid n \geq 0\}$$

is not regular. Define the homomorphism h:

$$\begin{aligned} h(a) &= 0, \\ h(b) &= 1, \\ h(c) &= 00. \end{aligned}$$

Then

$$h^{-1}(L) \cap a^* b c^* =$$

Example. Let us prove that the language

$$L = \{0^n 1 0^{2n} \mid n \geq 0\}$$

is not regular. Define the homomorphism h:

$$\begin{aligned} h(a) &= 0, \\ h(b) &= 1, \\ h(c) &= 00. \end{aligned}$$

Then

$$h^{-1}(L) \cap a^* b c^* = \{a^n b c^n \mid n \geq 0\}$$

Let us denote this language by L_1. If L is regular then also L_1 is regular. Define another homomorhism g:

$$\begin{aligned} g(a) &= a, \\ g(b) &= \varepsilon, \\ g(c) &= b. \end{aligned}$$

Then

$$g(L_1) =$$

Example. Let us prove that the language

$$L = \{0^n 1 0^{2n} \mid n \geq 0\}$$

is not regular. Define the homomorphism h:

$$\begin{aligned} h(a) &= 0, \\ h(b) &= 1, \\ h(c) &= 00. \end{aligned}$$

Then

$$h^{-1}(L) \cap a^*bc^* = \{a^n bc^n \mid n \geq 0\}$$

Let us denote this language by L_1. If L is regular then also L_1 is regular. Define another homomorhism g:

$$\begin{aligned} g(a) &= a, \\ g(b) &= \varepsilon, \\ g(c) &= b. \end{aligned}$$

Then

$$g(L_1) = \{a^n b^n \mid n \geq 0\}$$

But this language is not regular, so L_1 cannot be regular, which means that L cannot be regular either. We only used operations that preserve regularity (inverse homomorphism, intersection with a regular language, homomorphism).

Constructs of type

$$h^{-1}(L) \cap R$$

are very useful. They contain all words of R that are mapped to L by homomorphism h:

$$h^{-1}(L) \cap R = \{w \in R \mid h(w) \in L\}$$

Our next language operation is called **quotient**.

Let L_1 and L_2 be two languages. Their quotient L_1/L_2 is the language containing all words obtained by removing from the end of L_1's words suffix that belongs to L_2.

$$L_1/L_2 = \{w \mid wu \in L_1 \text{ for some } u \in L_2\}$$

Our next language operation is called **quotient**.

Let L_1 and L_2 be two languages. Their quotient L_1/L_2 is the language containing all words obtained by removing from the end of L_1's words suffix that belongs to L_2.

$$L_1/L_2 = \{w \mid wu \in L_1 \text{ for some } u \in L_2\}$$

Example. Let

$$\begin{aligned} L_1 &= abaa + aaa, \\ L_2 &= a + baa, \\ L_3 &= a^*ba^*. \end{aligned}$$

Then

$$\begin{aligned} L_1/L_2 &= \\ L_2/L_2 &= \\ L_3/L_1 &= \\ L_3/L_3 &= \\ L_2/L_1 &= \end{aligned}$$

Our next language operation is called **quotient**.

Let L_1 and L_2 be two languages. Their quotient L_1/L_2 is the language containing all words obtained by removing from the end of L_1's words suffix that belongs to L_2.

$$L_1/L_2 = \{w \mid wu \in L_1 \text{ for some } u \in L_2\}$$

Example. Let

$$\begin{aligned} L_1 &= abaa + aaa, \\ L_2 &= a + baa, \\ L_3 &= a^*ba^*. \end{aligned}$$

Then

$$\begin{aligned} L_1/L_2 &= aba + aa + a \\ L_2/L_2 &= \varepsilon + ba \\ L_3/L_1 &= a^* + a^*ba^* \\ L_3/L_3 &= a^* \\ L_2/L_1 &= \emptyset \end{aligned}$$

Theorem. The family of regular languages is closed under quotient with **arbitrary** languages.

In other words, if L_1 is a regular language, and L_2 any language (not necessarily even regular) then L_1/L_2 is regular.

Proof.

Theorem. The family of regular languages is closed under quotient with **arbitrary** languages.

In other words, if L_1 is a regular language, and L_2 any language (not necessarily even regular) then L_1/L_2 is regular.

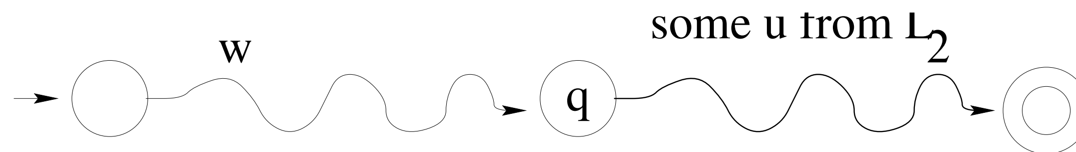
Proof. Let L_1 be a regular language, and A a DFA recognizing L_1. Let L_2 be an arbitrary language. We show that there is a DFA A' that recognizes the quotient $L' = L_1/L_2$.

Theorem. The family of regular languages is closed under quotient with **arbitrary** languages.

In other words, if L_1 is a regular language, and L_2 any language (not necessarily even regular) then L_1/L_2 is regular.

Proof. Let L_1 be a regular language, and A a DFA recognizing L_1. Let L_2 be an arbitrary language. We show that there is a DFA A' that recognizes the quotient $L' = L_1/L_2$.

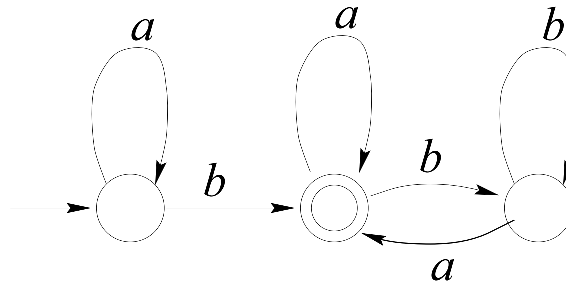
DFA A' has the same states and transitions as A. Also the initial state is the same. Only the final states are different: A state q is made final in A' if and only if some word $u \in L_2$ takes A from state q to some original final state.



Clearly word w is accepted by A' if and only if some wu is accepted by A for some $u \in L_2$. But this is equivalent to saying that w belongs to L_1/L_2: it is obtained from word $wu \in L_1$ by deleting word $u \in L_2$ from the end. So

$$L(A') = L_1/L_2.$$

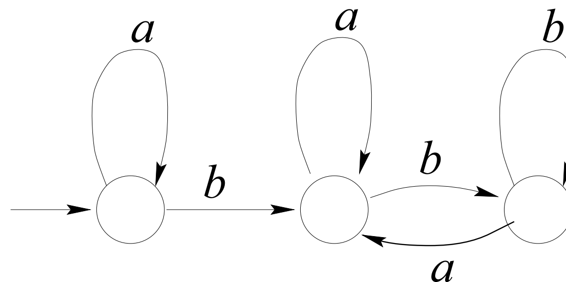
Example. Let L_1 be the regular language recognized by DFA



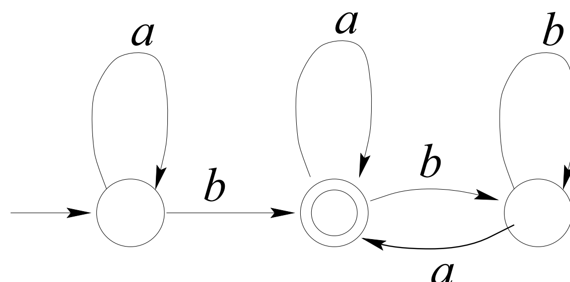
and let

$$L_2 = \{a^n b^n \mid n \geq 0\}.$$

Language L_1/L_2 is recognized by a DFA that has same transitions as A above. We only have to figure out which states are final:



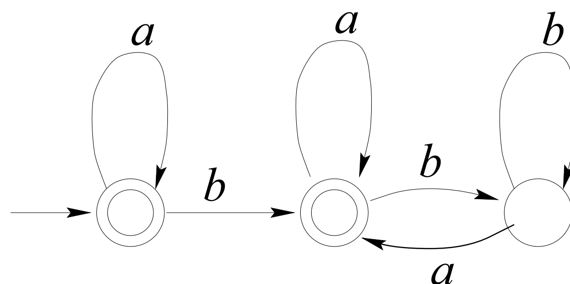
Example. Let L_1 be the regular language recognized by DFA



and let

$$L_2 = \{a^n b^n \mid n \geq 0\}.$$

Language L_1/L_2 is recognized by a DFA that has same transitions as A above. We only have to figure out which states are final:



Remark. Closure under quotient is not effective. If L_2 is some complicated language we may not have any way of determining which states to make final.

However, if both L_1 and L_2 are regular languages then L_1/L_2 can be constructed effectively.

Example. The quotient L/R can also be expressed using homomorphisms, finite substitutions, intersections and concatenations applied to L and R.

Let $L, R \subseteq \Sigma^*$. Let

$$\Sigma' = \{a' \mid a \in \Sigma\}$$

be a disjoint copy of Σ. (There is a' in place of a for each $a \in \Sigma$). For any word $u = a_1 \ldots a_n$ let us denote $u' = a'_1 \ldots a'_n$.

Define homomorphism h from Σ to Σ' by $h(a) = a'$ for all $a \in \Sigma$. Then for any word $u \in \Sigma^*$ we have $h(u) = u'$. (So h marks all letters of a word.)

Example. The quotient L/R can also be expressed using homomorphisms, finite substitutions, intersections and concatenations applied to L and R.

Let $L, R \subseteq \Sigma^*$. Let

$$\Sigma' = \{a' \mid a \in \Sigma\}$$

be a disjoint copy of Σ. (There is a' in place of a for each $a \in \Sigma$). For any word $u = a_1 \dots a_n$ let us denote $u' = a_1' \dots a_n'$.

Define homomorphism h from Σ to Σ' by $h(a) = a'$ for all $a \in \Sigma$. Then for any word $u \in \Sigma^*$ we have $h(u) = u'$. (So h marks all letters of a word.)

Define substitution s from Σ to $\Sigma \cup \Sigma'$ as follows: $s(a) = \{a, a'\}$ for all $a \in \Sigma$. (The substitution s may mark some letters of a word.)

$$s(L) \cap \Sigma^* h(R) = \{uv' \mid uv \in L \text{ and } v \in R\}$$

Example. The quotient L/R can also be expressed using homomorphisms, finite substitutions, intersections and concatenations applied to L and R.

Let $L, R \subseteq \Sigma^*$. Let

$$\Sigma' = \{a' \mid a \in \Sigma\}$$

be a disjoint copy of Σ. (There is a' in place of a for each $a \in \Sigma$). For any word $u = a_1 \ldots a_n$ let us denote $u' = a'_1 \ldots a'_n$.

Define homomorphism h from Σ to Σ' by $h(a) = a'$ for all $a \in \Sigma$. Then for any word $u \in \Sigma^*$ we have $h(u) = u'$. (So h marks all letters of a word.)

Define substitution s from Σ to $\Sigma \cup \Sigma'$ as follows: $s(a) = \{a, a'\}$ for all $a \in \Sigma$. (The substitution s may mark some letters of a word.)

$$s(L) \cap \Sigma^* h(R) = \{uv' \mid uv \in L \text{ and } v \in R\}$$

Define homomorphism g from $\Sigma \cup \Sigma'$ to Σ by $g(a) = a$ and $g(a') = \varepsilon$ for all $a \in \Sigma$. (Now g erases all marked letters.)

$$g(s(L) \cap \Sigma^* h(R)) = L/R.$$

Decision algorithms

In this section we present algorithms for determining whether a given regular language is empty, finite or infinite. We also present an algorithm for determining whether two given regular languages are identical, *i.e.*, whether they contain exactly same words.

First we make the same observation as before: it does not matter in which form the input language is represented. As a regular expression, DFA, NFA or ε-NFA. All representations are effectively equivalent.

Theorem. There is an algorithm to determine if a given regular language is empty.

Theorem. There is an algorithm to determine if a given regular language is empty.

Proof 1. Given a DFA A that recognizes L, check if there exists a path from the initial state to a final state. Such a path exists if and only if $L \neq \emptyset$.

Theorem. There is an algorithm to determine if a given regular language is empty.

Proof 2. Given a DFA A with n states that recognizes L, try all input words $w \in \Sigma^*$ of length $|w| < n$ and check if any of them is accepted by A. By a homework problem, $L \neq \emptyset$ if and only if some w is accepted by A.

(This is a very, very slow algorithm in practice. I gave it here only to show how the homework problem can be used here.)

Theorem. There is an algorithm to determine if a given regular language is empty.

Proof 3. We can also check emptyness directly from a regular expression r, without converting it to a DFA first. Observe that

- If $r = \varepsilon$ or $r = a \in \Sigma$ then $L(r)$ is **not** empty.
- If $r = \emptyset$ then $L(r)$ is empty.
- If $r = r_1 + r_2$ then $L(r)$ is empty if and only if both $L(r_1)$ **and** $L(r_2)$ are empty.
- If $r = r_1 r_2$ then $L(r)$ is empty if and only if $L(r_1)$ **or** $L(r_2)$ is empty.
- If $r = (r_1)^*$ then $L(r)$ is **not** empty. (Even $\emptyset^*$ is non-empty.)

A recursive algorithm based on these facts for the emptyness of $L(r)$:

```
Empty(r)
Begin
    if r = ε or r = a for some letter a then return( False )
    if r = ∅ then return( True )
    if r = r1 + r2 then return( Empty(r1) and Empty(r2) )
    if r = r1r2 then return( Empty(r1) or Empty(r2) )
    if r = r1* then return( False )
End
```

Example. Is the language represented by

$$(a^*\emptyset b + a)(\emptyset b)^*$$

empty ?

Consider then the equivalence problem. Some regular expressions stand for the same language. For example, $a(a + ba)^*$ and $(ab + a)^*a$ both define the same language. How can we determine for any given two regular expressions correctly if their languages are identical ?

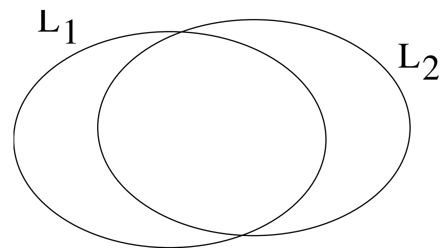
Theorem. There is an algorithm to determine if two reqular languages are the same.

Consider then the equivalence problem. Some regular expressions stand for the same language. For example, $a(a+ba)^*$ and $(ab+a)^*a$ both define the same language. How can we determine for any given two regular expressions correctly if their languages are identical ?

Theorem. There is an algorithm to determine if two reqular languages are the same.

Proof. Let L_1 and L_2 be two regular languages (represented by finite automata or regular expressions.) Using the effective closure properties proved earlier we can construct a regular expression for their symmetric difference

$$L = (L_1 \setminus L_2) \cup (L_2 \setminus L_1) = (L_1 \cap \bar{L}_2) \cup (L_2 \cap \bar{L}_1).$$



Since $L_1 = L_2$ if and only if L is empty, we can use the algorithm for emptyness to find out whether L_1 and L_2 are the same language.

Our last decision algorithm determines whether given regular language contains finitely or infinitely many words. We know that every finite language is regular:

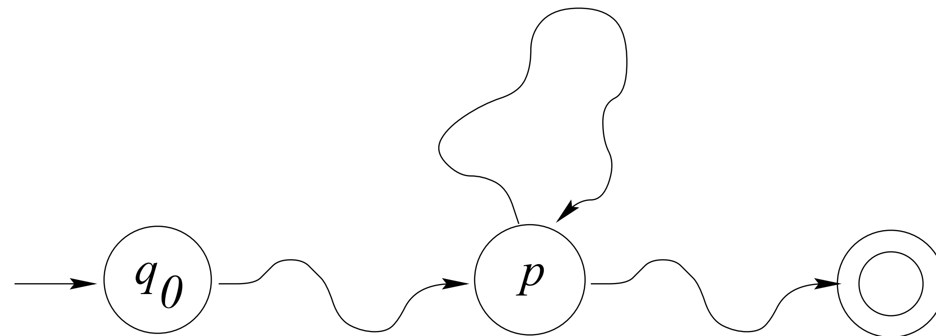
$$\{w_1, w_2, \ldots, w_n\} = L(w_1 + w_2 + \cdots + w_n).$$

Some regular languages are infinite, for example a^*. How do we determine whether given regular language has infinitely many words ?

Theorem. There is an algorithm to determine if a given regular language is infinite.

Theorem. There is an algorithm to determine if a given regular language is infinite.

Proof 1. Given a DFA A that recognizes L, check if for some state p there exists a path from the initial state to p, a cyacle containing p and a path from p to a final state. Such an accepting path containing a cycle exists if and only if L is infinite.



Theorem. There is an algorithm to determine if a given regular language is infinite.

Proof 2. Given a DFA A with n states that recognizes L, try all input words $w \in \Sigma^*$ of length $n \leq |w| < 2n$ and check if any of them is accepted by A. By a homework problem, L is infinite if and only if some w is accepted by A.

(Again, just to show how the homework problem can be used here. Do not try this in practice.)

Theorem. There is an algorithm to determine if a given regular language is infinite.

Proof 3. Testing if a given regular expression r represents an infinite language:

- If $r = \varepsilon$, $r = a \in \Sigma$ or $r = \emptyset$ then $L(r)$ is not infinite.
- If $r = r_1 + r_2$ then $L(r)$ is infinite if and only if $L(r_1)$ or $L(r_2)$ is infinite.
- If $r = r_1 r_2$ then $L(r)$ is infinite if and only if $L(r_1)$ and $L(r_2)$ are non-empty, and at least one of them is infinite.
- If $r = (r_1)^*$ then $L(r)$ is infinite unless $L(r_1)$ is $\emptyset$ or $\{\varepsilon\}$.

A recursive algorithm based on these facts uses also as a subroutine our earlier algorithms for emptyness and equivalence.

```
Infinite(r)
Begin
    if r = ∅ or r = ε or r = a for some letter a then
       return( False )
    if r = r1 + r2 then
       return( Infinite(r1) or Infinite(r2) )
    if r = r1r2 then
       return( (Infinite(r1) and not Empty(r2))
             or (Infinite(r2) and not Empty(r1)) )
    if r = r1* then
       return( not Empty(r1) and not Equal(r1,ε) )
End
```

Example. An algorithm to test if every word accepted a given DFA A is also accepted by another given DFA B.

Example. An algorithm to test if every word accepted a given DFA A is also accepted by another given DFA B.

Example. An algorithm to test, for a given regular expression r, whether every word is a concatenation of words that match r.

Myhill-Nerode theorem

A relation R on a set S is an **equivalence** relation if it is reflexive, symmetric and transitive, i.e. if

- aRa for all $a \in S$,
- aRb implies bRa, and
- aRb and bRc imply aRc.

Myhill-Nerode theorem

A relation R on a set S is an **equivalence** relation if it is reflexive, symmetric and transitive, i.e. if

- aRa for all $a \in S$,
- aRb implies bRa, and
- aRb and bRc imply aRc.

Example. With $S = \mathbb{Z}$, the set of integers, and n is any fixed positive integer, the congruence

$$a \equiv b \pmod{n}$$

is an equivalence relation. Namely,

- $a \equiv a \pmod{n}$ for all $a \in \mathbb{Z}$,
- if $a \equiv b \pmod{n}$ then $b \equiv a \pmod{n}$, and
- if $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$ then $a \equiv c \pmod{n}$.

An equivalence relation R partitions S into disjoint **equivalence classes** such that aRb if and only if a and b are in the same equivalence class. The equivalence class containing a is denoted by $[a]$:

$$[a] = \{b \in S \mid aRb\}.$$

An equivalence relation R partitions S into disjoint **equivalence classes** such that aRb if and only if a and b are in the same equivalence class. The equivalence class containing a is denoted by $[a]$:

$$[a] = \{b \in S \mid aRb\}.$$

Example. The equivalence class containing integer i in the congruence $\pmod n$ relation is

$$[i] = \{i + kn \mid k \in \mathbb{Z}\}.$$

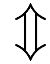
There are n different equivalence classes of the integer congruence $\pmod{}$:

$$\begin{array}{lcl} [0] & = & \{\dots, -2n, -n, 0, n, 2n, \dots\} \\ [1] & = & \{\dots, -2n+1, -n+1, 1, n+1, 2n+1, \dots\} \\ \dots & & \dots \\ [n-1] & = & \{\dots, -n-1, -1, n-1, 2n-1, 3n-1, \dots\} \end{array}$$

The **index** of an equivalence relation is the number of equivalence classes. The index can also be infinite.

Every language $L \subseteq \Sigma^*$ defines an equivalence relation R_L on words of Σ^*:

$$w \; R_L \; u$$



$$(\forall x \in \Sigma^*) \; wx \in L \text{ if and only if } ux \in L$$

Words w and u are in the relation R_L iff exactly the same extensions x take them to L.

Whether words w and u are in the relation R_L for a given language L depends on for which extensions x words wx and ux belong to L. Let us define the language of **good extensions** x as

$$\mathrm{Ext}(w, L) = \{x \mid wx \in L\}$$

Then $w\ R_L\ u$ if and only if

$$\mathrm{Ext}(w, L) = \mathrm{Ext}(u, L).$$

Example. Let $L = aa + baa$. Then

$$\begin{aligned}
&\mathrm{Ext}(a, L) = \\
&\mathrm{Ext}(b, L) = \\
&\mathrm{Ext}(ba, L) = \\
&\mathrm{Ext}(\varepsilon, L) = \\
&\mathrm{Ext}(aa, L) = \\
&\mathrm{Ext}(aba, L) =
\end{aligned}$$

Example. Let $L = aa + baa$. Then

$$\begin{aligned}
\text{Ext}(a, L) &= \{a\} \\
\text{Ext}(b, L) &= \{aa\} \\
\text{Ext}(ba, L) &= \{a\} \\
\text{Ext}(\varepsilon, L) &= \{aa, baa\} \\
\text{Ext}(aa, L) &= \{\varepsilon\} \\
\text{Ext}(aba, L) &= \emptyset
\end{aligned}$$

Are the following words in the relation R_L ?

a and b ?
a and ba ?
a and ε ?

Example. Let $L = aa + baa$. Then

$\mathrm{Ext}(a, L) = \{a\}$
$\mathrm{Ext}(b, L) = \{aa\}$
$\mathrm{Ext}(ba, L) = \{a\}$
$\mathrm{Ext}(\varepsilon, L) = \{aa, baa\}$
$\mathrm{Ext}(aa, L) = \{\varepsilon\}$
$\mathrm{Ext}(aba, L) = \emptyset$

Are the following words in the relation R_L ?

a and b ? No
a and ba ? Yes
a and ε ? No

What is the set of all words that are in relation R_L with word a: $[a] =?$

What are the equivalence classes of relation R_L ?

Example. Let $L = aa + baa$. Then

$\text{Ext}(a, L) = \{a\}$
$\text{Ext}(b, L) = \{aa\}$
$\text{Ext}(ba, L) = \{a\}$
$\text{Ext}(\varepsilon, L) = \{aa, baa\}$
$\text{Ext}(aa, L) = \{\varepsilon\}$
$\text{Ext}(aba, L) = \emptyset$

Are the following words in the relation R_L ?

a and b ? No
a and ba ? Yes
a and ε ? No

What is the set of all words that are in relation R_L with word a: $[a] = \{a, ba\}$

What are the equivalence classes of relation R_L ?

$\{\varepsilon\}, \{a, ba\}, \{aa, baa\}, \{b\}, \{u \mid u \text{ is not a prefix of any word in } L\}$

Another example. Let us find equivalence classes of R_L for $L = a^*b^*$.

First, what are

$\text{Ext}(a, L) =$
$\text{Ext}(a^{50}, L) =$
$\text{Ext}(a^{50}b, L) =$
$\text{Ext}(a^{50}b^{20}, L) =$
$\text{Ext}(b, L) =$
$\text{Ext}(ba, L) =$

Another example. Let us find equivalence classes of R_L for $L = a^*b^*$.

First, what are

$$\begin{aligned}
&\mathrm{Ext}(a, L) = a^*b^* \\
&\mathrm{Ext}(a^{50}, L) = a^*b^* \\
&\mathrm{Ext}(a^{50}b, L) = b^* \\
&\mathrm{Ext}(a^{50}b^{20}, L) = b^* \\
&\mathrm{Ext}(b, L) = b^* \\
&\mathrm{Ext}(ba, L) = \emptyset
\end{aligned}$$

Now, what is $\mathrm{Ext}(w, L)$ if

$$\begin{aligned}
&w \in a^* \quad ? \\
&w \in a^*b^+ \quad ? \\
&w \notin a^*b^* \quad ?
\end{aligned}$$

Another example. Let us find equivalence classes of R_L for $L = a^*b^*$.

First, what are

$\mathrm{Ext}(a, L) = a^*b^*$
$\mathrm{Ext}(a^{50}, L) = a^*b^*$
$\mathrm{Ext}(a^{50}b, L) = b^*$
$\mathrm{Ext}(a^{50}b^{20}, L) = b^*$
$\mathrm{Ext}(b, L) = b^*$
$\mathrm{Ext}(ba, L) = \emptyset$

Now, what is $\mathrm{Ext}(w, L)$ if

$w \in a^*$? a^*b^*
$w \in a^*b^+$? b^*
$w \notin a^*b^*$? $\emptyset$

So what are the equivalence classes of R_L ?

Another example. Let us find equivalence classes of R_L for $L = a^*b^*$.

First, what are

$\text{Ext}(a, L) = a^*b^*$
$\text{Ext}(a^{50}, L) = a^*b^*$
$\text{Ext}(a^{50}b, L) = b^*$
$\text{Ext}(a^{50}b^{20}, L) = b^*$
$\text{Ext}(b, L) = b^*$
$\text{Ext}(ba, L) = \emptyset$

Now, what is $\text{Ext}(w, L)$ if

$w \in a^*$? a^*b^*
$w \in a^*b^+$? b^*
$w \notin a^*b^*$? $\emptyset$

So what are the equivalence classes of R_L ?

$a^*, a^*b^+, \{u \mid u \text{ is not a prefix of any word in } L\}$

One more example. Let us find the equivalence classes of R_L for the language

$$L = \{w \in \{a, b\}^* \mid w \text{ contains equally many } a\text{'s and } b\text{'s } \}.$$

One more example. Let us find the equivalence classes of R_L for the language

$$L = \{w \in \{a,b\}^* \mid w \text{ contains equally many } a\text{'s and } b\text{'s } \}.$$

Let us denote

$$|w|_a = \text{ number of letters } a \text{ in word } w.$$

Consider languages

$$L_n = \{w \in \{a,b\}^* \mid |w|_a - |w|_b = n\}$$

for different integers n. Clearly $L = L_0$. Every word belongs to exactly one L_n. Observe also that

$$L_n L_m \subseteq L_{n+m}.$$

It follows from the considerations above that if $w \in L_n$ then

$$\text{Ext}(w, L) =$$

One more example. Let us find the equivalence classes of R_L for the language

$$L = \{w \in \{a,b\}^* \mid w \text{ contains equally many } a\text{'s and } b\text{'s }\}.$$

Let us denote

$$|w|_a = \text{ number of letters } a \text{ in word } w.$$

Consider languages

$$L_n = \{w \in \{a,b\}^* \mid |w|_a - |w|_b = n\}$$

for different integers n. Clearly $L = L_0$. Every word belongs to exactly one L_n. Observe also that

$$L_n L_m \subseteq L_{n+m}.$$

It follows from the considerations above that if $w \in L_n$ then

$$\mathrm{Ext}(w, L) = L_{-n}$$

Therefore $\mathrm{Ext}(w, L)$ and $\mathrm{Ext}(u, L)$ are identical if and only if w and u belong to the same set L_n. In other words, the equivalence classes of R_L are languages L_n for all $n \in \mathbb{Z}$.

Note that in this case there are **infinitely many equivalence classes**.

Theorem. Language L is regular if and only if the index of R_L is finite. The index is the smallest possible number of states of a DFA recognizing L.

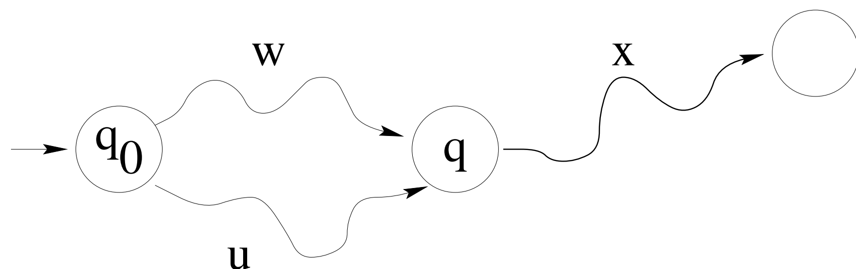
Proof.

Theorem. Language L is regular if and only if the index of R_L is finite. The index is the smallest possible number of states of a DFA recognizing L.

Proof. ($\Longrightarrow$) Let L be a regular language accepted by DFA A. If w and u are two words that take the automaton to the same state q, i.e.

$$\delta(q_0, w) = \delta(q_0, u),$$

then for every extension x, words wx and ux lead to same state.



This means that wR_Lu.

Since all words leading to same state are in relation R_L, the number of equivalence classes is at most the number of states, and therefore finite.

We have seen that the number of states in a DFA accepting L is at least the index of R_L.

Remark. Assign also to every DFA A its own equivalence relation R_A by

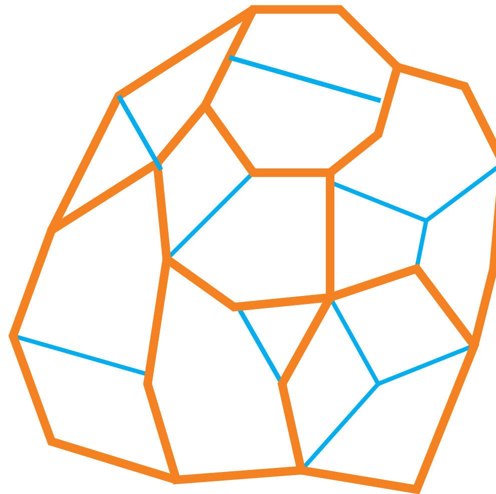
$$w\ R_A\ u \Longleftrightarrow \delta(q_0, w) = \delta(q_0, u).$$

That is, words w and u are in relation R_A if and only if they take the DFA A to the same state.

We proved that if A recognizes L then the relation R_A is a **refinement** of relation R_L. In other words,

$$w\ R_A\ u \Longrightarrow w\ R_L\ u.$$

The equivalence classes of a refinement define a finer partition of the set:



Theorem. Language L is regular if and only if the index of R_L is finite. The index is the smallest possible number of states of a DFA recognizing L.

Proof. ($\Longleftarrow$) Assume that R_L has finite index. Let $L_1, L_2, \ldots, L_n$ be its equivalence classes. We construct a DFA A with n states that recognizes L. The state set of A is labeled with the equivalence classes $[w]$:

$$Q = \{L_1, L_2, \ldots, L_n\}.$$

The idea of the construction is that every input word w will lead to state $[w]$, the equivalence class containing w.

Theorem. Language L is regular if and only if the index of R_L is finite. The index is the smallest possible number of states of a DFA recognizing L.

Proof. ($\Longleftarrow$) Assume that R_L has finite index. Let L_1, L_2, ..., L_n be its equivalence classes. We construct a DFA A with n states that recognizes L. The state set of A is labeled with the equivalence classes $[w]$:

$$Q = \{L_1, L_2, \ldots, L_n\}.$$

The idea of the construction is that every input word w will lead to state $[w]$, the equivalence class containing w.

The transitions are defined as follows:

$$\delta([w], a) = [wa].$$

The transitions are well defined, that is, the definition does not depend on which word w we selected from the equivalence class: If $[w] = [u]$ then $[wa] = [ua]$.

(If we would have $[wa] \neq [ua]$ then for some x, $wax \in L$ and $uax \notin L$ or vice versa. But then also $[w] \neq [u]$ since extension ax separates w and u.)

Theorem. Language L is regular if and only if the index of R_L is finite. The index is the smallest possible number of states of a DFA recognizing L.

Proof. ($\Longleftarrow$) Assume that R_L has finite index. Let L_1, L_2, $\ldots$, L_n be its equivalence classes. We construct a DFA A with n states that recognizes L. The state set of A is labeled with the equivalence classes $[w]$:

$$Q = \{L_1, L_2, \ldots, L_n\}.$$

The idea of the construction is that every input word w will lead to state $[w]$, the equivalence class containing w.

The transitions are defined as follows:

$$\delta([w], a) = [wa].$$

The transitions are well defined, that is, the definition does not depend on which word w we selected from the equivalence class: If $[w] = [u]$ then $[wa] = [ua]$.

Let the initial state q_0 be $[\varepsilon]$. Then

$$\delta(q_0, w) = [w]$$

for every w. The transitions were designed that way:

$$\begin{aligned}\delta([\varepsilon], a_1a_2\ldots a_n) &= \delta([a_1], a_2a_3\ldots a_n) = \delta([a_1a_2], a_3\ldots a_n)\\ &= \cdots = [a_1a_2\ldots a_n].\end{aligned}$$

Finally, the final states of our automaton are

$$\{[w] \mid w \in L\}.$$

Again, the choice of w does not matter since if $[w] = [u]$ and $w \in L$ then also $u \in L$.

(Otherwise the extension ε would separate them.)

Finally, the final states of our automaton are

$$\{[w] \mid w \in L\}.$$

Again, the choice of w does not matter since if $[w] = [u]$ and $w \in L$ then also $u \in L$.

The automaton we constructed recognizes the language L because

$$\delta(q_0, w) = [w]$$

and $[w]$ is a final state if and only if $w \in L$.

This completes the proof.

Example. Let us construct a DFA for $L = a^*b^*$ based on our knowledge about the equivalence classes of R_L:

$a^*, a^*b^+, \{u \mid\ u \text{ is not a prefix of any word in } L\}$

Example. Let us construct a DFA for $L = aa + baa$ based on our knowledge about the equivalence classes of R_L:

$\{\varepsilon\}, \{a, ba\}, \{aa, baa\}, \{b\}, \{u \mid \ u$ is not a prefix of any word in $L\}$

Example. Doing the construction on the non-regular language

$$L = \{w \in \{a,b\}^* \mid w \text{ contains equally many } a\text{'s and } b\text{'s }\}$$

leads to infinitely many states, corresponding to the equivalence classes

$$L_n = \{w \in \{a,b\}^* \mid |w|_a - |w|_b = n\}$$

We proved that the smallest possible number of states in a DFA recognizing L is the same as the index of the equivalence relation R_L.

A DFA with this smallest possible number of states is the **minimum-state DFA** of L. This DFA is unique:

Theorem. The minimum-state DFA of a language L is unique (up to renaming states).

Proof.

We proved that the smallest possible number of states in a DFA recognizing L is the same as the index of the equivalence relation R_L.

A DFA with this smallest possible number of states is the **minimum-state DFA** of L. This DFA is unique:

Theorem. The minimum-state DFA of a language L is unique (up to renaming states).

Proof. Let $A = (Q, \Sigma, \delta, q_0, F)$ be a DFA with n states for the language L, where n is the index of R_L. Let us see how to rename the states so that A becomes identical with the DFA A_0 we constructed in the previous proof.

Recall: the states of A_0 are the equivalence classes $[w]$ of relation R_L and transitions are defined by

$$\delta([w], a) = [wa].$$

As the automaton relation R_A is a refinement of R_L and has no more than n classes, we have that $R_A = R_L = R_{A_0}$. Thus for any words w and u we have

$$[w] = [u] \iff \delta(q_0, w) = \delta(q_0, u)$$

For every $w \in \Sigma^*$ we **rename** the state $\delta(q_0, w)$ as $[w]$.

As the automaton relation R_A is a refinement of R_L and has no more than n classes, we have that $R_A = R_L = R_{A_0}$. Thus for any words w and u we have

$$[w] = [u] \iff \delta(q_0, w) = \delta(q_0, u)$$

For every $w \in \Sigma^*$ we **rename** the state $\delta(q_0, w)$ as $[w]$.

- This renaming is well defined: if $\delta(q_0, w) = \delta(q_0, u)$ then $[w] = [u]$.

(So the name is independent of the choice of the the word w leading to the state.)

As the automaton relation R_A is a refinement of R_L and has no more than n classes, we have that $R_A = R_L = R_{A_0}$. Thus for any words w and u we have

$$[w] = [u] \iff \delta(q_0, w) = \delta(q_0, u)$$

For every $w \in \Sigma^*$ we **rename** the state $\delta(q_0, w)$ as $[w]$.

- This renaming is well defined: if $\delta(q_0, w) = \delta(q_0, u)$ then $[w] = [u]$.
- Every state gets a name, and every equivalence class is a name of a state.

(If a state gets no name then there is no word leading to that state. The state can be removed, yielding a smaller equivalent DFA. This is not possible since n is the smallest possible number of states.

Every equivalence class $[w]$ is trivially the name of the state $\delta(q_0, w)$.)

As the automaton relation R_A is a refinement of R_L and has no more than n classes, we have that $R_A = R_L = R_{A_0}$. Thus for any words w and u we have

$$[w] = [u] \iff \delta(q_0, w) = \delta(q_0, u)$$

For every $w \in \Sigma^*$ we **rename** the state $\delta(q_0, w)$ as $[w]$.

- This renaming is well defined: if $\delta(q_0, w) = \delta(q_0, u)$ then $[w] = [u]$.
- Every state gets a name, and every equivalence class is a name of a state.
- Different states get different names: if $[w] = [u]$ then $\delta(q_0, w) = \delta(q_0, u)$.

(If p and q get the same name $[w]$ then there is

- a word u such that $\delta(q_0, u) = p$ and $[u] = [w]$, and
- a word v such that $\delta(q_0, v) = q$ and $[v] = [w]$.

But since $[u] = [w] = [v]$ we have that $p = \delta(q_0, u) = \delta(q_0, v) = q$.)

As the automaton relation R_A is a refinement of R_L and has no more than n classes, we have that $R_A = R_L = R_{A_0}$. Thus for any words w and u we have

$$[w] = [u] \iff \delta(q_0, w) = \delta(q_0, u)$$

For every $w \in \Sigma^*$ we **rename** the state $\delta(q_0, w)$ as $[w]$.

- This renaming is well defined: if $\delta(q_0, w) = \delta(q_0, u)$ then $[w] = [u]$.
- Every state gets a name, and every equivalence class is a name of a state.
- Different states get different names: if $[w] = [u]$ then $\delta(q_0, w) = \delta(q_0, u)$.

With this renaming the state set of A becomes the same as the state set of A_0.

- The **initial state** $q_0 = \delta(q_0, \varepsilon)$ is renamed as $[\varepsilon]$, the initial state of A_0.
- A state renamed as $[w]$ is a **final state** of A if and only if $w \in L$, which is equivalent to $[w]$ being a final state in A_0.

As the automaton relation R_A is a refinement of R_L and has no more than n classes, we have that $R_A = R_L = R_{A_0}$. Thus for any words w and u we have

$$[w] = [u] \Longleftrightarrow \delta(q_0, w) = \delta(q_0, u)$$

For every $w \in \Sigma^*$ we **rename** the state $\delta(q_0, w)$ as $[w]$.

- This renaming is well defined: if $\delta(q_0, w) = \delta(q_0, u)$ then $[w] = [u]$.
- Every state gets a name, and every equivalence class is a name of a state.
- Different states get different names: if $[w] = [u]$ then $\delta(q_0, w) = \delta(q_0, u)$.

With this renaming the state set of A becomes the same as the state set of A_0.

- The **initial state** $q_0 = \delta(q_0, \varepsilon)$ is renamed as $[\varepsilon]$, the initial state of A_0.
- A state renamed as $[w]$ is a **final state** of A if and only if $w \in L$, which is equivalent to $[w]$ being a final state in A_0.
- With this renaming, **transitions** are exactly as in A_0: for all $w \in \Sigma^*$, $a \in \Sigma$

$$\delta([w], a) = [wa]$$

(Indeed: this simply states that $\delta(\delta(q_0, w), a) = \delta(q_0, wa)$.)

As the automaton relation R_A is a refinement of R_L and has no more than n classes, we have that $R_A = R_L = R_{A_0}$. Thus for any words w and u we have

$$[w] = [u] \iff \delta(q_0, w) = \delta(q_0, u)$$

For every $w \in \Sigma^*$ we **rename** the state $\delta(q_0, w)$ as $[w]$.

- This renaming is well defined: if $\delta(q_0, w) = \delta(q_0, u)$ then $[w] = [u]$.
- Every state gets a name, and every equivalence class is a name of a state.
- Different states get different names: if $[w] = [u]$ then $\delta(q_0, w) = \delta(q_0, u)$.

With this renaming the state set of A becomes the same as the state set of A_0.

- The **initial state** $q_0 = \delta(q_0, \varepsilon)$ is renamed as $[\varepsilon]$, the initial state of A_0.
- A state renamed as $[w]$ is a **final state** of A if and only if $w \in L$, which is equivalent to $[w]$ being a final state in A_0.
- With this renaming, **transitions** are exactly as in A_0: for all $w \in \Sigma^*$, $a \in \Sigma$

$$\delta([w], a) = [wa]$$

Conclusion: After the renaming, A and A_0 have the same states, initial and final states and identical transitions. Thus $A = A_0$.

Theorem. There is an algorithm to construct the minimum-state DFA for given regular language L.

Proof 1. A stupid but simple algorithm: enumerate all DFAs in the increasing order of number of states. (First 1-state DFA, then 2-state DFA, etc.) For each DFA A check whether $L(A) = L$. (Equivalence is decidable.) Stop when the first equivalent DFA is found.

Theorem. There is an algorithm to construct the minimum-state DFA for given regular language L.

Proof 2. There is also a practical state minimization algorithm that constructs the minimum-state DFA equivalent to a given DFA A.

1. The first step is to **remove all non-reachable states** from A. State q is non-reachable if there does not exist any input word that takes automaton A to state q.

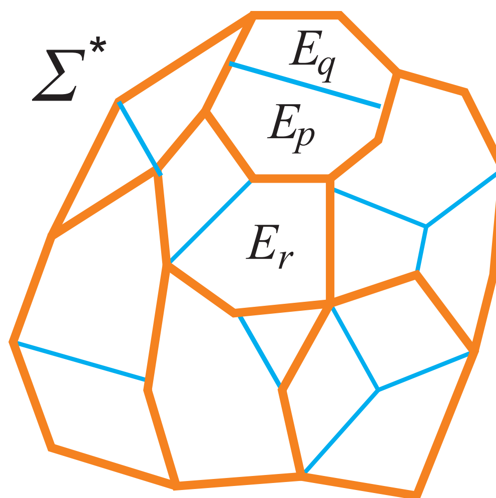
Reachable states can be found by a simple marking procedure: First mark the initial state only. Then mark all states that there is a transition into from some marked state. Continue this until no new states can be marked.

All non-marked states are non-reachable and can be deleted.

2. Now assume the DFA A only contains reachable states. Every state q defines an equivalence class E_q of the automaton relation R_A, consisting of all the words that take A from q_0 to the state q:

$$E_q = \{w \mid \delta(q_0, w) = q\}.$$

We know that R_A is a refinement of the relation R_L:



In the minimum-state DFA the equivalence classes of the automaton are the same as the equivalence classes of R_L. In A we can combine states q and p if their classes E_q and E_p belong to the same equivalence class of R_L. We call such states indistinguishable.

Precisely, p and q are **indistinguishable** if and only if

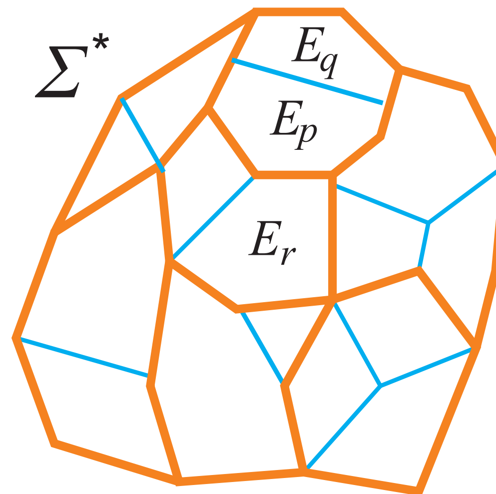
$$(\forall w \in \Sigma^*)\ \delta(p, w) \in F \iff \delta(q, w) \in F.$$

(It does not make any difference whether the DFA is in state p or q: Exactly the same words lead to a final state. Indistinguishable states can be merged.)

Complementarily, p and q are **distinguishable** if and only if

$$(\exists w \in \Sigma^*)\ \delta(p, w) \in F \text{ and } \delta(q, w) \notin F \text{ or vice versa.}$$

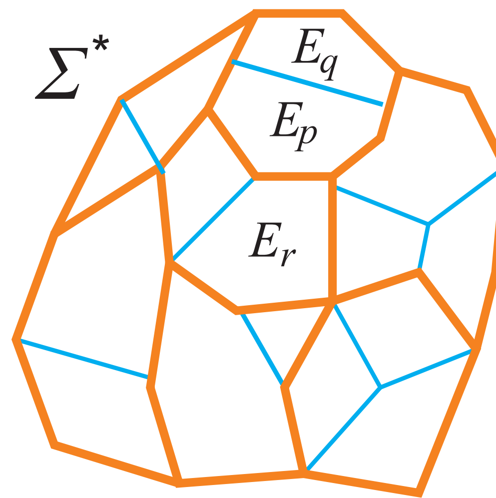
(There is a word w that distinguishes p and q. Distinguishable states cannot be merged.)



Here p and q are indistinguishable, while p and r are distinguishable.

If we find all pairs of indistinguishable states and combine them we end up with a DFA A' whose equivalence relation $R_{A'}$ is identical to R_L. This A' is the minimum-state DFA for language L.

The problem is to find out if two states are indistinguishable. It is easier to find states that are distinguishable. So rather than figuring out which states can be merged we search for states that cannot be merged!



We use the following property:

$$aw \text{ distinguishes } p \text{ and } q \iff w \text{ distinguishes } \delta(p,a) \text{ and } \delta(q,a)$$

The algorithm maintains a partition of the state set into classes $C_1, \ldots C_k$ such that for all $i \neq j$ the states of C_i are distinguishable from the states of C_j (even by the same word).

We use the following property:

$$aw \text{ distinguishes } p \text{ and } q \iff w \text{ distinguishes } \delta(p,a) \text{ and } \delta(q,a)$$

The algorithm maintains a partition of the state set into classes $C_1, \ldots C_k$ such that for all $i \neq j$ the states of C_i are distinguishable from the states of C_j (even by the same word).

- In the beginning we start with two classes: F (the set of final states) and $Q \setminus F$ (the set of non-final states). The empty word ε distinguishes these classes.

We use the following property:

$$aw \text{ distinguishes } p \text{ and } q \iff w \text{ distinguishes } \delta(p,a) \text{ and } \delta(q,a)$$

The algorithm maintains a partition of the state set into classes $C_1, \ldots C_k$ such that for all $i \neq j$ the states of C_i are distinguishable from the states of C_j (even by the same word).

• In the beginning we start with two classes: F (the set of final states) and $Q \setminus F$ (the set of non-final states). The empty word ε distinguishes these classes.

• At each round we look for a class C and a letter a such that for some $p, q \in C$ the states $\delta(q,a)$ and $\delta(p,a)$ belong to different classes. In this case, because $\delta(q,a)$ and $\delta(p,a)$ are distinguished by some word w, we can distinguish q and p by word aw. Thus we split C in such a way that any $p, q \in C$ go in the same part if and only if $\delta(q,a)$ and $\delta(p,a)$ are in the same class.

The algorithm is repeated until no class can be split further.

Claim: At the end all states that are in the same class are indistinguishable.

Proof. Assume the contrary: let w be the shortest word that distinguishes some states (say p and q) that belong to the same class (say C). Clearly $w \neq \varepsilon$ because otherwise p would be final and q non-final, or vice versa, but such pairs were put in different classes in the beginning.

So $w = au$ for some letter a.

But then $\delta(q, a)$ and $\delta(p, a)$ are distinguished by u that is shorter than w, and therefore $\delta(q, a)$ and $\delta(p, a)$ are in different classes. But this means that the class C can be split, contradicting the assumption that no classes can be split further.