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,

$$(abaab)^R =$$
$$(saippuakauppias)^R =$$
$$\varepsilon^R =$$

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

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

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

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

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

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

$$\{\varepsilon\}$$

$$\emptyset$$

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.

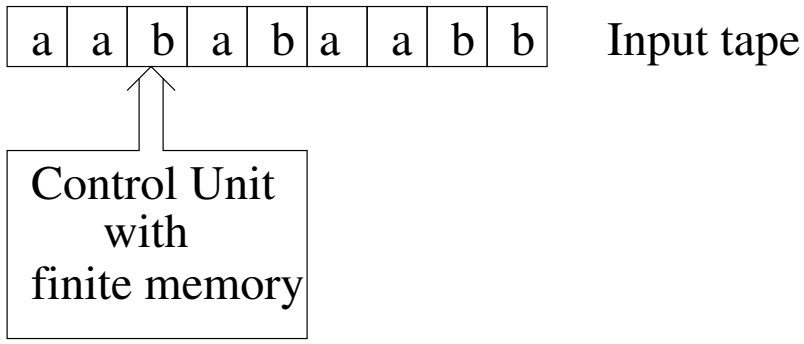
Deterministic Finite Automata (DFA)

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

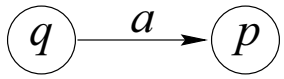
	a	b
p	q	p
q	r	p
r	r	r

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

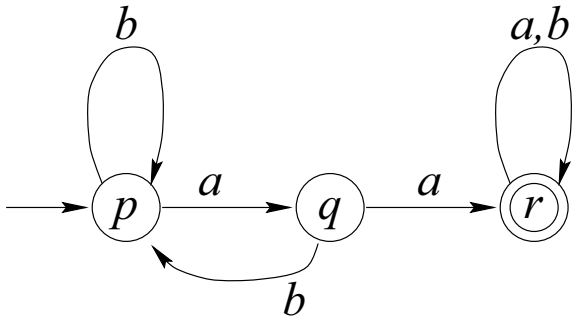
	a	b
p	q	p
q	r	p
r	r	r

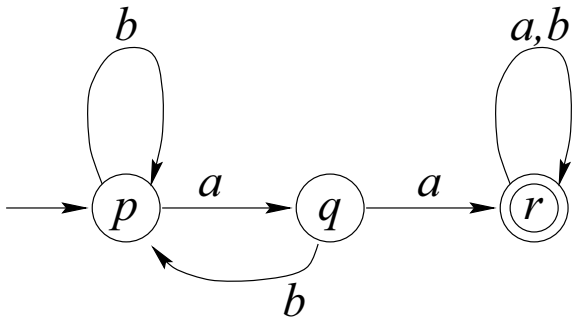
Example. The diagram representation of the DFA

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

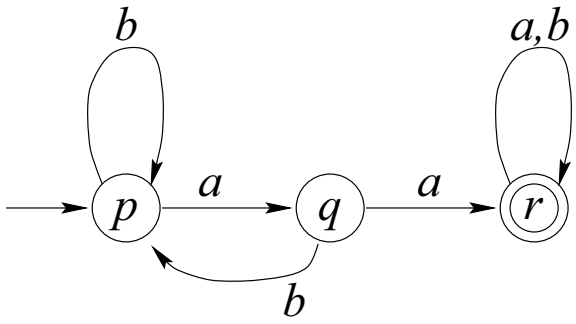
with the transition function

	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.

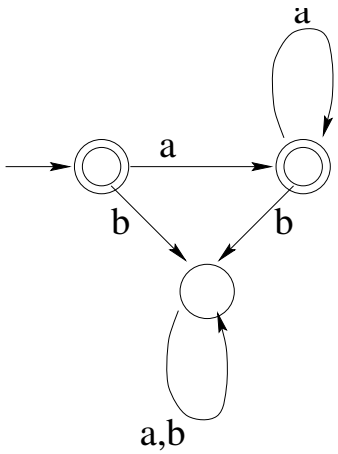
4. Words that start with a and end in a.

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

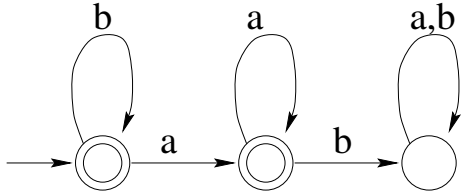
6. All words over $\{a, b\}$, i.e. $\{a, b\}^*$.

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

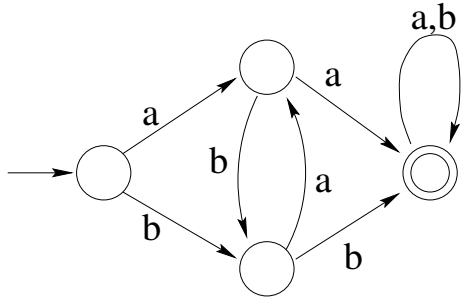
1)



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

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

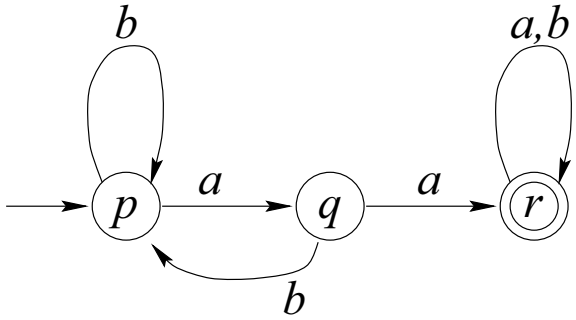
(If $p = \hat{\delta}(q, w)$ is the state after reading input w then $\delta(p, a)$ is the new state after reading input wa.)

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

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

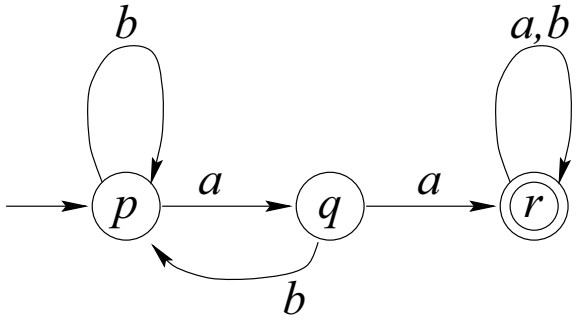
Therefore there is no danger of confusion if we simplify notations by removing the hat and indicate simply δ instead of $\hat{\delta}$.

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.)