Contents

I. AUTOMATA

1. Automata and Formal Languages

Automata and formal languages play an important role in projecting and realizing compilers. In the first section grammars and formal languages are defined. The different grammars and languages are discussed based on Chomsky hierarchy. In the second section we deal in detail with the finite automata and the languages accepted by them, while in the third section the pushdown automata and the corresponding accepted languages are discussed. Finally, references from a rich bibliography are given.

1.1. Languages and grammars

A finite and nonempty set of symbols is called an ***alphabet***. The elements of an alphabet are ***letters,*** but sometimes are named also ***symbols.***

With the letters of an alphabet words are composed. If $a_1, a_2, \ldots, a_n \in \Sigma, n \geq 0$, then $a_1 a_2 \ldots a_n$ a Σ is a ***word*** over the alphabet Σ (the letters a_i are not necessary distinct). The number of letters of a word, with their multiplicities, constitutes the ***length*** of the word. If $w = a_1 a_2 \ldots a_n$, then the length of w is $|w| = n$. If $n = 0$, then the word is an ***empty word,*** which will be denoted by ε (sometimes λ in other books). The set of words over the alphabet Σ will be denoted by Σ^*:

$$\Sigma^* = \left\{ a_1 a_2 \ldots a_n \mid a_1, a_2, \ldots, a_n \in \Sigma,\ n \geq 0 \right\} .$$

For the set of nonempty words over Σ the notation $\Sigma^+ = \Sigma^* \setminus \{\varepsilon\}$ will be used. The set of words of length n over Σ will be denoted by Σ^n, and $\Sigma^0 = \{\varepsilon\}$. Then

$$\Sigma^* = \Sigma^0 \cup \Sigma^1 \cup \cdots \cup \Sigma^n \cup \cdots \quad \text{and} \quad \Sigma^+ = \Sigma^1 \cup \Sigma^2 \cup \cdots \cup \Sigma^n \cup \cdots .$$

The words $u = a_1 a_2 \ldots a_m$ and $v = b_1 b_2 \ldots b_n$ are equal (i.e. $u = v$), if $m = n$ and $a_i = b_i,\ \ i = 1, 2, \ldots, n$.

We define in Σ^* the binary operation called ***concatenation.*** The concatenation (or product) of the words $u = a_1 a_2 \ldots a_m$ and $v = b_1 b_2 \ldots b_n$ is the word $uv = a_1 a_2 \ldots a_m b_1 b_2 \ldots b_n$. It is clear that $|uv| = |u| + |v|$. This operation is associative but not commutative. Its neutral element is ε, because $\varepsilon u = u\varepsilon = u$ for all $u \in \Sigma^*$. Σ^* with the concatenation is a monoid.

We introduce the power operation. If $u \in \Sigma^*$, then $u^0 = \varepsilon$, and $u^n = u^{n-1} u$

for $n \geq 1$. The ***reversal*** (or ***mirror image***) of the word $u = a_1 a_2 \dots a_n$ is $u^{-1} = a_n a_{n-1} \dots a_1$. The reversal of u sometimes is denoted by u^R or $\tilde{u}$. It is clear that $\left(u^{-1}\right)^{-1} = u$ and $(uv)^{-1} = v^{-1}u^{-1}$.

Word v is a ***prefix*** of the word u if there exists a word z such that $u = vz$. If $z \neq \varepsilon$ then v is a proper prefix of u. Similarly v is a ***suffix*** of u if there exists a word x such that $u = xv$. The proper suffix can also be defined. Word v is a ***subword*** of the word u if there are words p and q such that $u = pvq$. If $pq \neq \varepsilon$ then v is a ***proper subword.***

A subset L of Σ^* is called a ***language*** over the alphabet Σ. Sometimes this is called a ***formal language*** because the words are here considered without any meanings. Note that $\emptyset$ is the empty language while $\{\varepsilon\}$ is a language which contains the empty word.

1.1.1. Operations on languages

If L, L_1, L_2 are languages over Σ we define the following operations

- *union*
$$L_1 \cup L_2 = \{u \in \Sigma^* \mid u \in L_1 \text{ or } u \in L_2\} ,$$
- *intersection*
$$L_1 \cap L_2 = \{u \in \Sigma^* \mid u \in L_1 \text{ and } u \in L_2\} ,$$
- *difference*
$$L_1 \setminus L_2 = \{u \in \Sigma^* \mid u \in L_1 \text{ and } u \notin L_2\} ,$$
- *complement*
$$\overline{L} = \Sigma^* \setminus L ,$$
- *multiplication*
$$L_1 L_2 = \{uv \mid u \in L_1, v \in L_2\} ,$$
- *power*
$$L^0 = \{\varepsilon\}, \qquad L^n = L^{n-1}L, \text{ if } n \geq 1 ,$$
- *iteration* or *star operation*
$$L^* = \bigcup_{i=0}^{\infty} L^i = L^0 \cup L \cup L^2 \cup \dots \cup L^i \cup \dots ,$$
- *mirror*
$$L^{-1} = \{u^{-1} \mid u \in L\} .$$

We will use also the notation L^+
$$L^+ = \bigcup_{i=1}^{\infty} L^i = L \cup L^2 \cup \dots \cup L^i \cup \dots .$$
The union, product and iteration are called ***regular operations.***

1.1.2. Specifying languages

Languages can be specified in several ways. For example a language can be specified using

1) the enumeration of its words,

2) a property, such that all words of the language have this property but other word have not,

3) a grammar.

Specifying languages by listing their elements. For example the following are languages

$L_1 = \{\varepsilon, 0, 1\}$,

$L_2 = \{a, aa, aaa, ab, ba, aba\}$.

Even if we cannot enumerate the elements of an infinite set infinite languages can be specified by enumeration if after enumerating the first some elements we can continue the enumeration using a rule. The following is such a language

$L_3 = \{\varepsilon, ab, aabb, aaabbb, aaaabbbb, \ldots\}$.

Specifying languages by properties. The following sets are languages

$L_4 = \{a^n b^n \mid n = 0, 1, 2, \ldots\}$,

$L_5 = \{uu^{-1} \mid u \in \Sigma^*\}$,

$L_6 = \{u \in \{a, b\}^* \mid n_a(u) = n_b(u)\}$,

where $n_a(u)$ denotes the number of letters a in word u and $n_b(u)$ the number of letters b.

Specifying languages by grammars. Define the ***generative grammar*** or shortly the ***grammar***.

Definition 1.1 *A **grammar** is an ordered quadruple $G = (N, T, P, S)$, where*

- *N is the alphabet of **variables** (or **nonterminal symbols**),*
- *T is the alphabet of **terminal symbols,** where $N \cap T = \emptyset$,*
- *$P \subseteq (N \cup T)^* N (N \cup T)^* \times (N \cup T)^*$ is a finite set, that is P is the finite set of **productions** of the form (u, v) , where $u, v \in (N \cup T)^*$ and u contains at least a nonterminal symbol,*
- *$S \in N$ is the **start symbol.***

Remarks. Instead of the notation (u, v) sometimes $u \to v$ is used.

In the production $u \to v$ or (u, v) word u is called the left-hand side of the production while v the right-hand side. If for a grammar there are more than one production with the same left-hand side, then these production

$$u \to v_1,\ u \to v_2, \ldots, u \to v_r \quad \text{can be written as} \quad u \to v_1 \mid v_2 \mid \ldots \mid v_r\ .$$

We define on the set $(N \cup T)^*$ the relation called ***direct derivation***

$$u \Longrightarrow v, \qquad \text{if} \quad u = p_1 p p_2, \quad v = p_1 q p_2 \quad \text{and} \quad (p, q) \in P\ .$$

In fact we replace in u an appearance of the subword p by q and we get v. Another notations for the same relation can be $\vdash$ or $\models$.

If we want to emphasize the used grammar G, then the notation $\Longrightarrow$ can be replaced by $\underset{G}{\Longrightarrow}$. Relation $\overset{*}{\Longrightarrow}$ is the reflexive and transitive closure of $\Longrightarrow$, while $\overset{+}{\Longrightarrow}$ denotes its transitive closure. Relation $\overset{*}{\Longrightarrow}$ is called a ***derivation.***

From the definition of a reflexive and transitive relation we can deduce the following: $u \overset{*}{\Longrightarrow} v$, if there exist the words $w_0, w_1, \ldots, w_n \in (N \cup T)^*$, $n \geq 0$ and $u = w_0$, $w_0 \Longrightarrow w_1$, $w_1 \Longrightarrow w_2$, $\ldots, w_{n-1} \Longrightarrow w_n$, $w_n = v$. This can be written shortly $u = w_0 \Longrightarrow w_1 \Longrightarrow w_2 \Longrightarrow \ldots \Longrightarrow w_{n-1} \Longrightarrow w_n = v$. If $n = 0$ then $u = v$. The same way we can define the relation $u \overset{+}{\Longrightarrow} v$ except that $n \geq 1$ always, so at least one direct derivation will de used.

Definition 1.2 *The* ***language generated*** *by grammar $G = (N, T, P, S)$ is the set*

$$L(G) = \{u \in T^* \mid S \overset{*}{\Longrightarrow} u\} .$$

So $L(G)$ contains all words over the alphabet T which can be derived from the start symbol S using the productions from P.

Example 1.1 Let $G = (N, T, P, S)$ where
$N = \{S\}$,
$T = \{a, b\}$,
$P = \{S \to aSb,\ S \to ab\}$.
It is easy to see than $L(G) = \{a^n b^n \mid n \geq 1\}$ because

$$S \underset{G}{\Longrightarrow} aSb \underset{G}{\Longrightarrow} a^2Sb^2 \underset{G}{\Longrightarrow} \cdots \underset{G}{\Longrightarrow} a^{n-1}Sb^{n-1} \underset{G}{\Longrightarrow} a^n b^n ,$$

where up to the last but one replacement the first production ($S \to aSb$) was used, while at the last replacement the production $S \to ab$. This derivation can be written $S \overset{*}{\underset{G}{\Longrightarrow}} a^n b^n$. Therefore $a^n b^n$ can be derived from S for all n and no other words can be derived from S.

Definition 1.3 *Two grammars G_1 and G_2 are* ***equivalent,*** *and this is denoted by $G_1 \cong G_2$ if $L(G_1) = L(G_2)$.*

Example 1.2 The following two grammars are equivalent because both of them generate the language $\{a^n b^n c^n \mid n \geq 1\}$.
$G_1 = (N_1, T, P_1, S_1)$, where
$N_1 = \{S_1, X, Y\}$, $T = \{a, b, c\}$,
$P_1 = \{S_1 \to abc,\ S_1 \to aXbc,\ Xb \to bX,\ Xc \to Ybcc,\ bY \to Yb,\ aY \to aaX,\ aY \to aa\}$.
$G_2 = (N_2, T, P_2, S_2)$, where
$N_2 = \{S_2, A, B, C\}$,
$P_2 = \{S_2 \to aS_2BC,\ S_2 \to aBC,\ CB \to BC,\ aB \to ab,\ bB \to bb,\ bC \to bc,\ cC \to cc\}$.
First let us prove by mathematical induction that for $n \geq 2$ $S_1 \overset{*}{\underset{G_1}{\Longrightarrow}} a^{n-1}Yb^n c^n$. If $n = 2$ then

$$S_1 \underset{G_1}{\Longrightarrow} aXbc \underset{G_1}{\Longrightarrow} abXc \underset{G_1}{\Longrightarrow} abYbcc \underset{G_1}{\Longrightarrow} aYb^2c^2 .$$

The inductive hypothesis is $S_1 \overset{*}{\underset{G_1}{\Longrightarrow}} a^{n-2}Yb^{n-1}c^{n-1}$. We use production $aY \to aaX$, then $(n-1)$ times production $Xb \to bX$, and then production $Xc \to Ybcc$, afterwards again $(n-1)$ times production $bY \to Yb$. Therefore

$$S_1 \underset{G_1}{\Longrightarrow} a^{n-2}Yb^{n-1}c^{n-1} \underset{G_1}{\Longrightarrow} a^{n-1}Xb^{n-1}c^{n-1} \overset{*}{\underset{G_1}{\Longrightarrow}}$$

$$a^{n-1}b^{n-1}Xc^{n-1} \underset{G_1}{\Longrightarrow} a^{n-1}b^{n-1}Ybc^n \overset{*}{\underset{G_1}{\Longrightarrow}} a^{n-1}Yb^n c^n .$$

If now we use production $aY \to aa$ we get $S_1 \overset{*}{\underset{G_1}{\Longrightarrow}} a^n b^n c^n$ for $n \geq 2$, but $S_1 \underset{G_1}{\Longrightarrow} abc$ by the production $S_1 \to abc$, so $a^n b^n c^n \in L(G_1)$ for any $n \geq 1$. We have to prove in addition that using the productions of the grammar we cannot derive only words of the form $a^n b^n c^n$. It is easy to see that a successful derivation (which ends in a word containing only terminals) can be obtained only in the presented way.
Similarly for $n \geq 2$

$$S_2 \underset{G_2}{\Longrightarrow} aS_2BC \overset{*}{\underset{G_2}{\Longrightarrow}} a^{n-1}S_2(BC)^{n-1} \underset{G_2}{\Longrightarrow} a^n(BC)^n \overset{*}{\underset{G_2}{\Longrightarrow}} a^n B^n C^n$$

$$\underset{G_2}{\Longrightarrow} a^n bB^{n-1}C^n \overset{*}{\underset{G_2}{\Longrightarrow}} a^n b^n C^n \underset{G_2}{\Longrightarrow} a^n b^n cC^{n-1} \overset{*}{\underset{G_2}{\Longrightarrow}} a^n b^n c^n \ .$$

Here orderly were used the productions $S_2 \to aS_2BC$ ($n-1$ times), $S_2 \to aBC$, $CB \to BC$ ($n-1$ times), $aB \to ab$, $bB \to bb$ ($n-1$ times), $bC \to bc$, $cC \to cc$ ($n-1$ times). But $S_2 \underset{G_2}{\Longrightarrow} aBC \underset{G_2}{\Longrightarrow} abC \underset{G_2}{\Longrightarrow} abc$, So $S_2 \overset{*}{\underset{G_2}{\Longrightarrow}} a^n b^n c^n$, $n \geq 1$. It is also easy to see than other words cannot be derived using grammar G_2.

The grammars
$G_3 = (\{S\}, \{a, b\}, \{S \to aSb,\ S \to \varepsilon\}, S)$ and
$G_4 = (\{S\}, \{a, b\}, \{S \to aSb,\ S \to ab\}, S)$
are not equivalent because $L(G_3) \setminus \{\varepsilon\} = L(G_4)$.

Theorem 1.4 *Not all languages can be generated by grammars.*

Proof We encode grammars for the proof as words on the alphabet $\{0, 1\}$. For a given grammar $G = (N, T, P, S)$ let $N = \{S_1, S_2, \ldots, S_n\}$, $T = \{a_1, a_2, \ldots, a_m\}$ and $S = S_1$. The encoding is the following:

the code of S_i is $10\underbrace{11\ldots11}_{i \text{ times}}01$, the code of a_i is $100\underbrace{11\ldots11}_{i \text{ times}}001$.

In the code of the grammar the letters are separated by 000, the code of the arrow is 0000, and the productions are separated by 00000.

It is enough, of course, to encode the productions only. For example, consider the grammar

$G = (\{S\}, \{a, b\}, \{S \to aSb,\ S \to ab\}, S)$.

The code of S is 10101, the code of a is 1001001, the code of b is 10011001. The code of the grammar is

$$\underbrace{10101}\,0000\,\underbrace{1001001}\,000\,\underbrace{10101}\,000\,\underbrace{10011001}\,00000\,\underbrace{10101}\,0000\,\underbrace{1001001}\,000$$

$$\underbrace{10011001}\ .$$

From this encoding results that the grammars with terminal alphabet T can be enumerated [1] as $G_1, G_2, \ldots, G_k, \ldots$, and the set of these grammars is a denumerable

[1] Let us suppose that in the alphabet $\{0, 1\}$ there is a linear order $<$, let us say $0 < 1$. The words which are codes of grammars can be enumerated by ordering them first after their lengths, and inside the equal length words, alphabetically, using the order of their letters. But we can use equally the lexicographic order, which means that $u < v$ (u is before v) if u is a proper prefix of v or there exists the decompositions $u = xay$ and $v = xby'$, where x, y, y' are subwords, a and b letters with $a < b$.

infinite set.

Consider now the set of all languages over T denoted by $\mathcal{L}_T = \{L \mid L \subseteq T^*\}$, that is $\mathcal{L}_T = \mathcal{P}(T^*)$. The set T^* is denumerable because its words can be ordered. Let this order $s_0, s_1, s_2, \ldots$, where $s_0 = \varepsilon$. We associate to each language $L \in \mathcal{L}_T$ an infinite binary sequence $b_0, b_1, b_2, \ldots$ the following way:

$$b_i = \begin{cases} 1, & \text{if } s_i \in L \\ 0, & \text{if } s_i \notin L \end{cases} \qquad i = 0, 1, 2, \ldots \ .$$

It is easy to see that the set of all such binary sequences is not denumerable, because each sequence can be considered as a positive number less than 1 using its binary representation (The decimal point is considered to be before the first digit). Conversely, to each positive number less than 1 in binary representation a binary sequence can be associated. So, the cardinality of the set of infinite binary sequences is equal to cardinality of interval $[0, 1]$, which is of continuum power. Therefore the set $\mathcal{L}_T$ is of continuum cardinality. Now to each grammar with terminal alphabet T associate the corresponding generated language over T. Since the cardinality of the set of grammars is denumerable, there will exist a language from $\mathcal{L}_T$, without associated grammar, a language which cannot be generated by a grammar. ■

1.1.3. Chomsky hierarchy of grammars and languages

Putting some restrictions on the form of productions, four type of grammars can be distinguished.

Definition 1.5 *Define for a grammar $G = (N, T, P, S)$ the following four types.*

*A grammar G is of type 0 (**phrase-structure grammar**) if there are no restrictions on productions.*

*A grammar G is of type 1 **(context-sensitive grammar)** if all of its productions are of the form $\alpha A\gamma \to \alpha\beta\gamma$, where $A \in N$, $\alpha, \gamma \in (N \cup T)^*$, $\beta \in (N \cup T)^+$. A production of the form $S \to \varepsilon$ can also be accepted if the start symbol S does not occur in the right-hand side of any production.*

*A grammar G is of type 2 **(context-free grammar)** if all of its productions are of the form $A \to \beta$, where $A \in N$, $\beta \in (N \cup T)^+$. A production of the form $S \to \varepsilon$ can also be accepted if the start symbol S does not occur in the right-hand side of any production.*

*A grammar G is of type 3 **(regular grammar)** if its productions are of the form $A \to aB$ or $A \to a$, where $a \in T$ and $A, B \in N$. A production of the form $S \to \varepsilon$ can also be accepted if the start symbol S does not occur in the right-hand side of any production.*

If a grammar G is of type i then language $L(G)$ is also of type i.

This classification was introduced by Noam Chomsky.

A language L is of type i $(i = 0, 1, 2, 3)$ if there exists a grammar G of type i which generates the language L, so $L = L(G)$.

Denote by $\mathcal{L}_i$ $(i = 0, 1, 2, 3)$ the class of the languages of type i. Can be proved that

$$\mathcal{L}_0 \supset \mathcal{L}_1 \supset \mathcal{L}_2 \supset \mathcal{L}_3 \ .$$

By the definition of different type of languages, the inclusions $(\supseteq)$ are evident, but the strict inclusions $(\supset)$ must be proved.

Example 1.3 We give an example for each type of context-sensitive, context-free and regular grammars.
Context-sensitive grammar. $G_1 = (N_1, T_1, P_1, S_1)$, where $N_1 = \{S_1, A, B, C\}$, $T_1 = \{a, 0, 1\}$.
Elements of P_1 are:

$$\begin{array}{lcl} S_1 & \to & ACA, \\ AC & \to & AACA \mid ABa \mid AaB, \\ B & \to & AB \mid A, \\ A & \to & 0 \mid 1. \end{array}$$

Language $L(G_1)$ contains words of the form uav with $u, v \in \{0, 1\}^*$ and $|u| \neq |v|$.
Context-free grammar. $G_2 = (N_2, T_2, P_2, S)$, where $N_2 = \{S, A, B\}$, $T_2 = \{+, *, (,), a\}$.
Elements of P_2 are:

$$\begin{array}{lcl} S & \to & S + A \mid A, \\ A & \to & A * B \mid B, \\ B & \to & (S) \mid a. \end{array}$$

Language $L(G_2)$ contains algebraic expressions which can be correctly built using letter a, operators $+$ and $*$ and brackets.
Regular grammar. $G_3 = (N_3, T_3, P_3, S_3)$, where $N_3 = \{S_3, A, B\}$, $T_3 = \{a, b\}$.
Elements of P_3 are:

$$\begin{array}{lcl} S_3 & \to & aA \\ A & \to & aB \mid a \\ B & \to & aB \mid bB \mid a \mid b. \end{array}$$

Language $L(G_3)$ contains words over the alphabet $\{a, b\}$ with at least two letters a at the beginning.

It is easy to prove that any finite language is regular. The productions will be done to generate all words of the language. For example, if $u = a_1 a_2 \dots a_n$ is in the language, then we introduce the productions: $S \to a_1 A_1$, $A_1 \to a_2 A_2$, $\dots A_{n-2} \to a_{n-1} A_{n-1}$, $A_{n-1} \to a_n$, where S is the start symbol of the language and $A_1, \dots, A_{n-1}$ are distinct nonterminals. We define such productions for all words of the language using different nonterminals for different words, excepting the start symbol S. If the empty word is also an element of the language, then the production $S \to \varepsilon$ is also considered.

The empty set is also a regular language, because the regular grammar $G = (\{S\}, \{a\}, \{S \to aS\}, S)$ generates it.

Eliminating unit productions. A production of the form $A \to B$ is called a ***unit production,*** where $A, B \in N$. Unit productions can be eliminated from a grammar in such a way that the new grammar will be of the same type and equivalent to the first one.

Let $G = (N, T, P, S)$ be a grammar with unit productions. Define an equivalent grammar $G' = (N, T, P', S)$ without unit productions. The following algorithm will

construct the equivalent grammar.

ELIMINATE-UNIT-PRODUCTIONS(G)

1 if the unit productions $A \to B$ and $B \to C$ are in P put also the unit production $A \to C$ in P while P can be extended,
2 if the unit production $A \to B$ and the production $B \to \alpha$ ($\alpha \notin N$) are in P put also the production $A \to \alpha$ in P,
3 let P' be the set of productions of P except unit productions
4 **return** G'

Clearly, G and G' are equivalent. If G is of type $i \in \{0, 1, 2, 3\}$ then G' is also of type i.

Example 1.4 Use the above algorithm in the case of the grammar $G = \big(\{S, A, B, C\}, \{a, b\}, P, S\big)$, where P contains

$S \to A$, $A \to B$, $B \to C$, $C \to B$, $D \to C$,
$S \to B$, $A \to D$, $C \to Aa$,
$A \to aB$,
$A \to b$.

Using the first step of the algorithm, we get the following new unit productions:

$S \to D$ (because of $S \to A$ and $A \to D$),
$S \to C$ (because of $S \to B$ and $B \to C$),
$A \to C$ (because of $A \to B$ and $B \to C$),
$B \to B$ (because of $B \to C$ and $C \to B$),
$C \to C$ (because of $C \to B$ and $B \to C$),
$D \to B$ (because of $D \to C$ and $C \to B$).

In the second step of the algorithm will be considered only productions with A or C in the right-hand side, since productions $A \to aB$, $A \to b$ and $C \to Aa$ can be used (the other productions are all unit productions). We get the following new productions:

$S \to aB$ (because of $S \to A$ and $A \to aB$),
$S \to b$ (because of $S \to A$ and $A \to b$),
$S \to Aa$ (because of $S \to C$ and $C \to Aa$),
$A \to Aa$ (because of $A \to C$ and $C \to Aa$),
$B \to Aa$ (because of $B \to C$ and $C \to Aa$).

The new grammar $G' = \big(\{S, A, B, C\}, \{a, b\}, P', S\big)$ will have the productions:

$S \to b$, $A \to b$, $B \to Aa$, $C \to Aa$,
$S \to aB$, $A \to aB$,
$S \to Aa$ $A \to Aa$,

Grammars in normal forms. A grammar is to be said a ***grammar in normal form*** if its productions have no terminal symbols in the left-hand side.

We need the following notions. For alphabets Σ_1 and Σ_2 a *homomorphism* is a function $h : \Sigma_1^* \to \Sigma_2^*$ for which $h(u_1u_2) = h(u_1)h(u_2)$, $\forall u_1, u_2 \in \Sigma_1^*$. It is easy to see that for arbitrary $u = a_1a_2 \dots a_n \in \Sigma_1^*$ value $h(u)$ is uniquely determined by the restriction of h on Σ_1, because $h(u) = h(a_1)h(a_2) \dots h(a_n)$.

If a homomorphism h is a bijection then h is an *isomorphism*.

Theorem 1.6 *To any grammar an equivalent grammar in normal form can be associated.*

Proof Grammars of type 2 and 3 have in left-hand side of any productions only a nonterminal, so they are in normal form. The proof has to be done for grammars of type 0 and 1 only.

Let $G = (N, T, P, S)$ be the original grammar and we define the grammar in normal form as $G' = (N', T, P', S)$.

Let $a_1, a_2, \ldots, a_k$ be those terminal symbols which occur in the left-hand side of productions. We introduce the new nonterminals $A_1, A_2, \ldots, A_k$. The following notation will be used: $T_1 = \{a_1, a_2, \ldots, a_k\}$, $T_2 = T \setminus T_1$, $N_1 = \{A_1, A_2, \ldots, A_k\}$ and $N' = N \cup N_1$.

Define the isomorphism $h : N \cup T \longrightarrow N' \cup T_2$, where

$$\begin{array}{ll} h(a_i) = A_i, & \text{if} \quad a_i \in T_1, \\ h(X) = X, & \text{if} \quad X \in N \cup T_2. \end{array}$$

Define the set P' of production as

$$P' = \Big\{ h(\alpha) \to h(\beta) \;\Big|\; (\alpha \to \beta) \in P \Big\} \cup \Big\{ A_i \longrightarrow a_i \;\Big|\; i = 1, 2, \ldots, k \Big\} .$$

In this case $\alpha \overset{*}{\underset{G}{\Longrightarrow}} \beta$ if and only if $h(\alpha) \overset{*}{\underset{G'}{\Longrightarrow}} h(\beta)$. From this the theorem immediately results because $S \overset{*}{\underset{G}{\Longrightarrow}} u \;\Leftrightarrow\; S = h(S) \overset{*}{\underset{G'}{\Longrightarrow}} h(u) = u$. ∎

Example 1.5 Let $G = (\{S, D, E\}, \{a, b, c, d, e\}, P, S)$, where P contains

$$\begin{array}{lcl} S & \to & aebc \mid aDbc \\ Db & \to & bD \\ Dc & \to & Ebccd \\ bE & \to & Eb \\ aE & \to & aaD \mid aae. \end{array}$$

In the left-hand side of productions the terminals a, b, c occur, therefore consider the new nonterminals A, B, C, and include in P' also the new productions $A \to a$, $B \to b$ and $C \to c$.

Terminals a, b, c will be replaced by nonterminals A, B, C respectively, and we get the set P' as

$$\begin{array}{lcl} S & \to & AeBC \mid ADBC \\ DB & \to & BD \\ DC & \to & EBCCd \\ BE & \to & EB \\ AE & \to & AAD \mid AAe \\ A & \to & a \\ B & \to & b \\ C & \to & c. \end{array}$$

Let us see what words can be generated by this grammars. It is easy to see that $aebc \in L(G')$, because $S \Longrightarrow AeBC \overset{*}{\Longrightarrow} aebc$.

$S \Longrightarrow ADBC \Longrightarrow ABDC \Longrightarrow ABEBCCd \Longrightarrow AEBBCCd \Longrightarrow AAeBBCCd \overset{*}{\Longrightarrow} aaebbccd$, so $aaebbccd \in L(G')$.

We prove, using the mathematical induction, that $S \overset{*}{\Longrightarrow} A^{n-1}EB^nC(Cd)^{n-1}$ for $n \geq 2$. For $n = 2$ this is the case, as we have seen before. Continuing the derivation we get $S \overset{*}{\Longrightarrow} A^{n-1}EB^nC(Cd)^{n-1} \Longrightarrow A^{n-2}AADB^nC(Cd)^{n-1} \overset{*}{\Longrightarrow} A^nB^nDC(Cd)^{n-1} \Longrightarrow A^nB^nEBCCd(Cd)^{n-1} \overset{*}{\Longrightarrow} A^nEB^{n+1}CCd(Cd)^{n-1} = A^nEB^{n+1}C(Cd)^n$, and this is what we had to prove.

But $S \overset{*}{\Longrightarrow} A^{n-1}EB^nC(Cd)^{n-1} \Longrightarrow A^{n-2}AAeB^nC(Cd)^{n-1} \overset{*}{\Longrightarrow} a^neb^nc(cd)^{n-1}$. So $a^neb^nc(cd)^{n-1} \in L(G'),\ n \geq 1$. These words can be generated also in G.

1.1.4. Extended grammars

In this subsection extended grammars of type 1, 2 and 3 will be presented.

Extended grammar of type 1. All productions are of the form $\alpha \to \beta$, where $|\alpha| \leq |\beta|$, excepted possibly the production $S \to \varepsilon$.

Extended grammar of type 2. All productions are of the form $A \to \beta$, where $A \in N, \beta \in (N \cup T)^*$.

Extended grammar of type 3. All productions are of the form $A \to uB$ or $A \to u$, Where $A, B \in N, u \in T^*$.

Theorem 1.7 *To any extended grammar an equivalent grammar of the same type can be associated.*

Proof Denote by G_{ext} the extended grammar and by G the corresponding equivalent grammar of the same type.

Type 1. Define the productions of grammar G by rewriting the productions $\alpha \to \beta$, where $|\alpha| \leq |\beta|$ of the extended grammar G_{ext} in the form $\gamma_1\delta\gamma_2 \to \gamma_1\gamma\gamma_2$ allowed in the case of grammar G by the following way.

Let $X_1X_2 \ldots X_m \to Y_1Y_2 \ldots Y_n$ $(m \leq n)$ be a production of G_{ext}, which is not in the required form. Add to the set of productions of G the following productions, where $A_1,\ A_2,\ \ldots,\ A_m$ are new nonterminals:

$$
\begin{array}{lcl}
X_1X_2 \ldots X_m & \to & A_1X_2X_3 \ldots X_m \\
A_1X_2 \ldots X_m & \to & A_1A_2X_3 \ldots X_m \\
 & \ldots & \\
A_1A_2 \ldots A_{m-1}X_m & \to & A_1A_2 \ldots A_{m-1}A_m \\
A_1A_2 \ldots A_{m-1}A_m & \to & Y_1A_2 \ldots A_{m-1}A_m \\
Y_1A_2 \ldots A_{m-1}A_m & \to & Y_1Y_2 \ldots A_{m-1}A_m \\
 & \ldots & \\
Y_1Y_2 \ldots Y_{m-2}A_{m-1}A_m & \to & Y_1Y_2 \ldots Y_{m-2}Y_{m-1}A_m \\
Y_1Y_2 \ldots Y_{m-1}A_m & \to & Y_1Y_2 \ldots Y_{m-1}Y_mY_{m+1} \ldots Y_n.
\end{array}
$$

Furthermore, add to the set of productions of G without any modification the productions of G_{ext} which are of permitted form, i.e. $\gamma_1\delta\gamma_2 \to \gamma_1\gamma\gamma_2$.

Inclusion $L(G_{ext}) \subseteq L(G)$ can be proved because each used production of G_{ext} in a derivation can be simulated by productions G obtained from it. Furthermore, since the productions of G can be used only in the prescribed order, we could not obtain other words, so $L(G) \subseteq L(G_{ext})$ also is true.

Type 2. Let $G_{ext} = (N, T, P, S)$. Productions of form $A \to \varepsilon$ have to be eliminated, only $S \to \varepsilon$ can remain, if S doesn't occur in the right-hand side of produc-

tions. For this define the following sets:

$U_0 = \{A \in N \mid (A \to \varepsilon) \in P\}$

$U_i = U_{i-1} \cup \{A \in N \mid (A \to w) \in P,\ w \in U_{i-1}^+\}$.

Since for $i \geq 1$ we have $U_{i-1} \subseteq U_i$, $U_i \subseteq N$ and N is a finite set, there must exists such a k for which $U_{k-1} = U_k$. Let us denote this set as U. It is easy to see that a nonterminal A is in U if and only if $A \stackrel{*}{\Longrightarrow} \varepsilon$. (In addition $\varepsilon \in L(G_{ext})$ if and only if $S \in U$.)

We define the productions of G starting from the productions of G_{ext} in the following way. For each production $A \to \alpha$ with $\alpha \neq \varepsilon$ of G_{ext} add to the set of productions of G this one and all productions which can be obtained from it by eliminating from α one or more nonterminals which are in U, but only in the case when the right-hand side does not become ε.

It in not difficult to see that this grammar G generates the same language as G_{ext} does, except the empty word ε. So, if $\varepsilon \notin L(G_{ext})$ then the proof is finished. But if $\varepsilon \in L(G_{ext})$, then there are two cases. If the start symbol S does not occur in any right-hand side of productions, then by introducing the production $S \to \varepsilon$, grammar G will generate also the empty word. If S occurs in a production in the right-hand side, then we introduce a new start symbol S' and the new productions $S' \to S$ and $S' \to \varepsilon$. Now the empty word ε can also be generated by grammar G.

Type 3. First we use for G_{ext} the procedure defined for grammars of type 2 to eliminate productions of the form $A \to \varepsilon$. From the obtained grammar we eliminate the unit productions using the algorithm ELIMINATE-UNIT-PRODUCTIONS (see page 21).

In the obtained grammar for each production $A \to a_1 a_2 \dots a_n B$, where $B \in N \cup \{\varepsilon\}$, add to the productions of G also the followings

$$\begin{array}{lcl} A & \to & a_1 A_1, \\ A_1 & \to & a_2 A_2, \\ & \dots & \\ A_{n-1} & \to & a_n B, \end{array}$$

where $A_1, A_2, \dots, A_{n-1}$ are new nonterminals. It is easy to prove that grammar G built in this way is equivalent to G_{ext}. ■

Example 1.6 Let $G_{ext} = (N, T, P, S)$ be an extended grammar of type 1, where $N = \{S, B, C\}$, $T = \{a, b, c\}$ and P contains the following productions:

$$\begin{array}{lcl@{\qquad}lcl} S & \to & aSBC \mid aBC & CB & \to & BC \\ aB & \to & ab & bB & \to & bb \\ bC & \to & bc & cC & \to & cc\,. \end{array}$$

The only production which is not context-sensitive is $CB \to BC$. Using the method given in the proof, we introduce the productions:

$$\begin{array}{lcl} CB & \to & AB \\ AB & \to & AD \\ AD & \to & BD \\ BD & \to & BC \end{array}$$

Now the grammar $G = (\{S, A, B, C, D\}, \{a, b, c\}, P', S)$ is context-sensitive, where the elements of P' are

$S \to aSBC \mid aBC$
$CB \to AB$ $\qquad aB \to ab$
$AB \to AD$ $\qquad bB \to bb$
$AD \to BD$ $\qquad bC \to bc$
$BD \to BC$ $\qquad cC \to cc.$

It can be proved that $L(G_{ext}) = L(G) = \{a^n b^n c^n \mid n \geq 1\}$.

Example 1.7 Let $G_{ext} = (\{S, B, C\}, \{a, b, c\}, P, S)$ be an extended grammar of type 2, where P contains:

$S \to aSc \mid B$
$B \to bB \mid C$
$C \to Cc \mid \varepsilon.$

Then $U_0 = \{C\}$, $U_1 = \{B, C\}$, $U_3 = \{S, B, C\} = U$. The productions of the new grammar are:

$S \to aSc \mid ac \mid B$
$B \to bB \mid b \mid C$
$C \to Cc \mid c.$

The original grammar generates also the empty word and because S occurs in the right-hand side of a production, a new start symbol and two new productions will be defined: $S' \to S, S' \to \varepsilon$. The context-free grammar equivalent to the original grammar is $G = (\{S', S, B, C\}, \{a, b, c\}, P', S')$ with the productions:

$S' \to S \mid \varepsilon$
$S \to aSc \mid ac \mid B$
$B \to bB \mid b \mid C$
$C \to Cc \mid c.$

Both of these grammars generate language $\{a^m b^n c^p \mid p \geq m \geq 0, n \geq 0\}$.

Example 1.8 Let $G_{ext} = (\{S, A, B\}, \{a, b\}, P, S)$ be the extended grammar of type 3 under examination, where P:

$S \to abA$
$A \to bB$
$B \to S \mid \varepsilon.$

First, we eliminate production $B \to \varepsilon$. Since $U_0 = U = \{B\}$, the productions will be

$S \to abA$
$A \to bB \mid b$
$B \to S.$

The latter production (which a unit production) can also be eliminated, by replacing it with $B \to abA$. Productions $S \to abA$ and $B \to abA$ have to be transformed. Since, both productions have the same right-hand side, it is enough to introduce only one new nonterminal and to use the productions $S \to aC$ and $C \to bA$ instead of $S \to abA$. Production $B \to abA$ will be replaced by $B \to aC$. The new grammar is $G = (\{S, A, B, C\}, \{a, b\}, P', S)$, where P':

$S \to aC$
$A \to bB \mid b$
$B \to aC$
$C \to bA.$

Can be proved that $L(G_{ext}) = L(G) = \{(abb)^n \mid n \geq 1\}$.

1.1.5. Closure properties in the Chomsky-classes

We will prove the following theorem, by which the Chomsky-classes of languages are closed under the regular operations, that is, the union and product of two languages of type i is also of type i, the iteration of a language of type i is also of type i $(i = 0, 1, 2, 3)$.

Theorem 1.8 *The class $\mathcal{L}_i$ $(i = 0, 1, 2, 3)$ of languages is closed under the regular operations.*

Proof For the proof we will use extended grammars. Consider the extended grammars $G_1 = (N_1, T_1, P_1, S_1)$ and $G_2 = (N_2, T_2, P_2, S_2)$ of type i each. We can suppose that $N_1 \cap N_2 = \emptyset$.

Union. Let $G_\cup = (N_1 \cup N_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \to S_1, S \to S_2\}, S)$.

We will show that $L(G_\cup) = L(G_1) \cup L(G_2)$. If $i = 0, 2, 3$ then from the assumption that G_1 and G_2 are of type i follows by definition that $G_\cup$ also is of type i. If $i = 1$ and one of the grammars generates the empty word, then we eliminate from $G_\cup$ the corresponding production (possibly the both) $S_k \to \varepsilon$ $(k = 1, 2)$ and replace it by production $S \to \varepsilon$.

Product. Let $G_\times = (N_1 \cup N_2 \cup \{S\}, T_1 \cup T_2, P_1 \cup P_2 \cup \{S \to S_1 S_2\}, S)$.

We will show that $L(G_\times) = L(G_1)L(G_2)$. By definition, if $i = 0, 2$ then $G_\times$ will be of the same type. If $i = 1$ and there is production $S_1 \to \varepsilon$ in P_1 but there is no production $S_2 \to \varepsilon$ in P_2 then production $S_1 \to \varepsilon$ will be replaced by $S \to S_2$. We will proceed the same way in the symmetrical case. If there is in P_1 production $S_1 \to \varepsilon$ and in P_2 production $S_2 \to \varepsilon$ then they will be replaced by $S \to \varepsilon$.

In the case of regular grammars $(i = 3)$, because $S \to S_1 S_2$ is not a regular production, we need to use another grammar $G_\times = (N_1 \cup N_2, T_1 \cup T_2, P_1' \cup P_2, S_1)$, where the difference between P_1' and P_1 lies in that instead of productions in the form $A \to u, u \in T^*$ in P_1' will exist production of the form $A \to uS_2$.

Iteration. Let $G_* = (N_1 \cup \{S\}, T_1, P, S)$.

In the case of grammars of type 2 let $P = P_1 \cup \{S \to S_1 S,\ S \to \varepsilon\}$. Then G_* also is of type 2.

In the case of grammars of type 3, as in the case of product, we will change the productions, that is $P = P_1' \cup \{S \to S_1,\ S \to \varepsilon\}$, where the difference between P_1' and P_1 lies in that for each $A \to u$ $(u \in T^*)$ will be replaced by $A \to uS$, and the others will be not changed. Then G_* also will be of type 3.

The productions given in the case of type 2 are not valid for $i = 0, 1$, because when applying production $S \to S_1 S$ we can get the derivations of type $S \overset{*}{\Longrightarrow} S_1 S_1$, $S_1 \overset{*}{\Longrightarrow} \alpha_1 \beta_1$, $S_1 \overset{*}{\Longrightarrow} \alpha_2 \beta_2$, where $\beta_1 \alpha_2$ can be a left-hand side of a production. In this case, replacing $\beta_1 \alpha_2$ by its right-hand side in derivation $S \overset{*}{\Longrightarrow} \alpha_1 \beta_1 \alpha_2 \beta_2$, we can generate a word which is not in the iterated language. To avoid such situations, first let us assume that the language is in normal form, i.e. the left-hand side of productions does not contain terminals (*see* page 21), second we introduce a new nonterminal S', so the set of nonterminals now is $N_1 \cup \{S, S'\}$, and the productions are the following:

$$P = P_1 \cup \{S \to \varepsilon,\ S \to S_1 S'\} \cup \{aS' \to aS \mid a \in T_1\}\ .$$

Now we can avoid situations in which the left-hand side of a production can extend over the limits of words in a derivation because of the iteration. The above derivations can be used only by beginning with $S \Longrightarrow S_1 S'$ and getting derivation $S \overset{*}{\Longrightarrow} \alpha_1\beta_1 S'$. Here we can not replace S' unless the last symbol in β_1 is a terminal symbol, and only after using a production of the form $aS' \to aS$.

It is easy to show that $L(G_*) = L(G_1)^*$ for each type.type ■

Exercises

1.1-1 Give a grammar which generates language $L = \{uu^{-1} \mid u \in \{a,b\}^*\}$ and determine its type.

1.1-2 Let $G = (N, T, P, S)$ be an extended context-free grammar, where

$N = \{S, A, C, D\}$, $T = \{a, b, c, d, e\}$,

$P = \{S \to abCADe,\ C \to cC,\ C \to \varepsilon,\ D \to dD,\ D \to \varepsilon,\ A \to \varepsilon,\ A \to dDcCA\}$.

Give an equivalent context-free grammar.

1.1-3 Show that Σ^* and Σ^+ are regular languages over arbitrary alphabet Σ.

1.1-4 Give a grammar to generate language $L = \{u \in \{0,1\}^* \mid n_0(u) = n_1(u)\}$, where $n_0(u)$ represents the number of 0's in word u and $n_1(u)$ the number of 1's.

1.1-5 Give a grammar to generate all natural numbers.

1.1-6 Give a grammar to generate the following languages, respectively:

$L_1 = \{a^n b^m c^p \mid n \geq 1, m \geq 1, p \geq 1\}$,

$L_2 = \{a^{2n} \mid n \geq 1\}$,

$L_3 = \{a^n b^m \mid n \geq 0, m \geq 0\ \}$,

$L_4 = \{a^n b^m \mid n \geq m \geq 1\}$.

1.1-7 Let $G = (N, T, P, S)$ be an extended grammar, where $N = \{S, A, B, C\}$, $T = \{a\}$ and P contains the productions:

$$S \to BAB,\ BA \to BC,\ CA \to AAC,\ CB \to AAB,\ A \to a,\ B \to \varepsilon\ .$$

Determine the type of this grammar. Give an equivalent, not extended grammar with the same type. What language it generates?

1.2. Finite automata and regular languages

Finite automata are computing models with input tape and a finite set of states (Fig. 1.1). Among the states some are called initial and some final. At the beginning the automaton read the first letter of the input word written on the input tape. Beginning with an initial state, the automaton read the letters of the input word one after another while change its states, and when after reading the last input letter the current state is a final one, we say that the automaton accepts the given word. The set of words accepted by such an automaton is called the language accepted (recognized) by the automaton.

Definition 1.9 *A **nondeterministic finite automaton** (NFA) is a system* $\mathrm{A} = (Q, \Sigma, E, I, F)$, *where*

- Q *is a finite, nonempty set of **states**,*
- Σ *is the **input alphabet**,*
- E *is the set of **transitions** (or of edges), where* $E \subseteq Q \times \Sigma \times Q$,

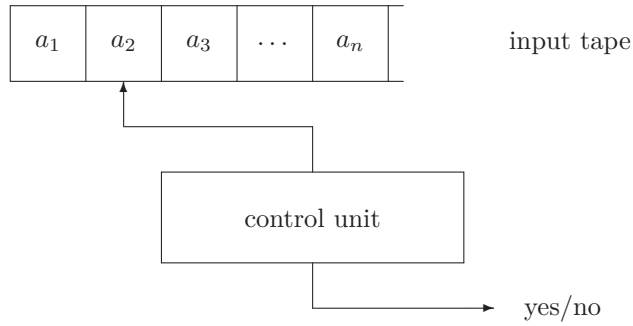


Figure 1.1 Finite automaton.

- $I \subseteq Q$ *is the set of* ***initial states****,*
- $F \subseteq Q$ *is the set of* ***final states.***

An NFA is in fact a directed, labelled graph, whose vertices are the states and there is a (directed) edge labelled with a from vertex p to vertex q if $(p, a, q) \in E$. Among vertices some are initial and some final states. Initial states are marked by a small arrow entering the corresponding vertex, while the final states are marked with double circles. If two vertices are joined by two edges with the same direction then these can be replaced by only one edge labelled with two letters. This graph can be called a transition graph.

Example 1.9 Let $\mathrm{A} = (Q, \Sigma, E, I, F)$, where $Q = \{q_0, q_1, q_2\}$, $\Sigma = \{0, 1, 2\}$, $E = \big\{(q_0, 0, q_0), (q_0, 1, q_1), (q_0, 2, q_2),$
$(q_1, 0, q_1), (q_1, 1, q_2), (q_1, 2, q_0),$
$(q_2, 0, q_2), (q_2, 1, q_0), (q_2, 2, q_1)\big\}$
$I = \{q_0\}$, $F = \{q_0\}$.
The automaton can be seen in Fig. 1.2.

In the case of an edge (p, a, q) vertex p is the start-vertex, q the end-vertex and a the label. Now define the notion of the ***walk*** as in the case of graphs. A sequence

$$(q_0, a_1, q_1), (q_1, a_2, q_2), \ldots, (q_{n-2}, a_{n-1}, q_{n-1}), (q_{n-1}, a_n, q_n)$$

of edges of a NFA is a walk with the label $a_1 a_2 \ldots a_n$. If $n = 0$ then $q_0 = q_n$ and $a_1 a_2 \ldots a_n = \varepsilon$. Such a walk is called an ***empty walk.*** For a walk the notation

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n$$

will be used, or if $w = a_1 a_2 \ldots a_n$ then we write shortly $q_0 \xrightarrow{w} q_n$. Here q_0 is the start-vertex and q_n the end-vertex of the walk. The states in a walk are not necessary distinct.

A walk is ***productive*** if its start-vertex is an initial state and its end-vertex is a final state. We say that an NFA ***accepts*** or ***recognizes*** a word if this word is the label of a productive walk. The empty word ε is accepted by an NFA if there is an

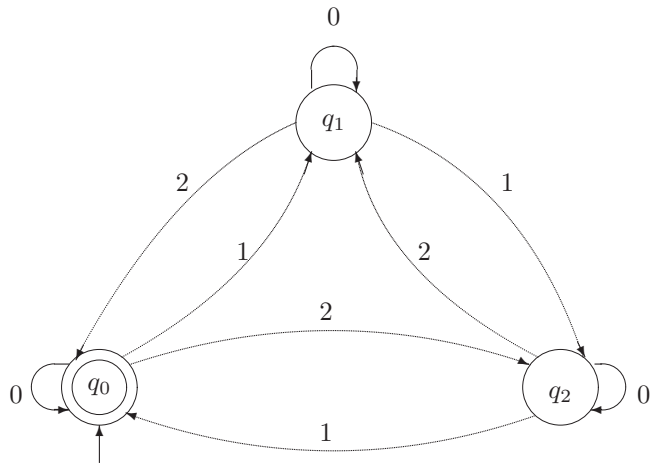


Figure 1.2 The finite automaton of Example 1.9.

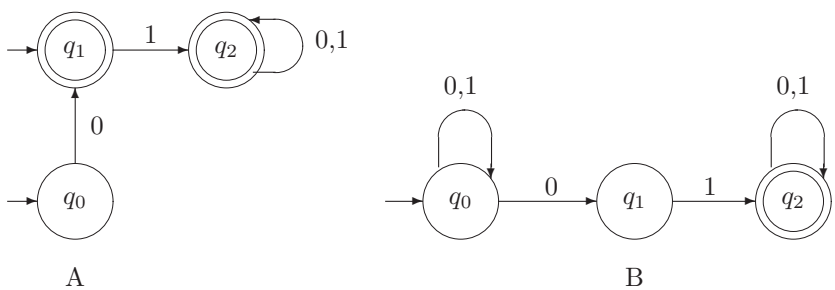


Figure 1.3 Nondeterministic finite automata.

empty productive walk, i.e. there is an initial state which is also a final state.

The set of words accepted by an NFA will be called the language accepted by this NFA. The ***language accepted*** or ***recognized*** by NFA A is

$$L(\mathrm{A}) = \left\{ w \in \Sigma^* \mid \exists p \in I,\ \exists q \in F,\ \exists p \xrightarrow{w} q \right\} .$$

The NFA A_1 and A_2 are ***equivalent*** if $L(\mathrm{A}_1) = L(\mathrm{A}_2)$.

Sometimes it is useful the following ***transition function:***

$$\delta : Q \times \Sigma \to P(Q), \quad \delta(p, a) = \{q \in Q \mid (p, a, q) \in E\} .$$

This function associate to a state p and input letter a the set of states in which the automaton can go if its current state is p and the head is on input letter a.

δ	0	1
q_0	$\{q_1\}$	$\emptyset$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$

A

δ	0	1
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\{q_2\}$

B

Figure 1.4 Transition tables of the NFA in Fig. 1.3.

Denote by $|H|$ the cardinal (the number of elements) of H.[2] An NFA is a ***deterministic finite automaton*** (DFA) if

$$|I| = 1 \text{ and } |\delta(q,a)| \leq 1,\ \forall q \in Q,\ \forall a \in \Sigma\ .$$

In Fig. 1.2 a DFA can be seen.

Condition $|\delta(q,a)| \leq 1$ can be replaced by

$$(p,a,q) \in E,\ (p,a,r) \in E \implies q = r\ , \forall p,q,r \in Q, \forall a \in \Sigma\ .$$

If for a DFA $|\delta(q,a)| = 1$ for each state $q \in Q$ and for each letter $a \in \Sigma$ then it is called a ***complete DFA.***

Every DFA can be transformed in a complete DFA by introducing a new state, which can be called a snare state. Let $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ be a DFA. An equivalent and complete DFA will be $\mathrm{A}' = (Q \cup \{s\}, \Sigma, E', \{q_0\}, F)$, where s is the new state and $E' = E \cup \big\{(p,a,s) \mid \delta(p,a) = \emptyset, p \in Q, a \in \Sigma\big\} \cup \big\{(s,a,s) \mid a \in \Sigma\big\}$. It is easy to see that $L(\mathrm{A}) = L(\mathrm{A}')$.

Using the transition function we can easily define the transition table. The rows of this table are indexed by the elements of Q, its columns by the elements of Σ. At the intersection of row $q \in Q$ and column $a \in \Sigma$ we put $\delta(q,a)$. In the case of Fig. 1.2, the transition table is:

δ	0	1	2
q_0	$\{q_0\}$	$\{q_1\}$	$\{q_2\}$
q_1	$\{q_1\}$	$\{q_2\}$	$\{q_0\}$
q_2	$\{q_2\}$	$\{q_0\}$	$\{q_1\}$

The NFA in Fig. 1.3 are not deterministic: the first (automaton A) has two initial states, the second (automaton B) has two transitions with 0 from state q_0 (to states q_0 and q_1). The transition table of these two automata are in Fig. 1.4. $L(\mathrm{A})$ is set of words over $\Sigma = \{0,1\}$ which do not begin with two zeroes (of course ε is in language), $L(\mathrm{B})$ is the set of words which contain 01 as a subword.

Eliminating inaccessible states. Let $\mathrm{A} = (Q, \Sigma, E, I, F)$ be a finite automaton. A state is *accessible* if it is on a walk which starts by an initial state. The following

[2] The same notation is used for the cardinal of a set and length of a word, but this is no matter of confusion because for word we use lowercase letters and for set capital letters. The only exception is $\delta(q,a)$, but this could not be confused with a word.

algorithm determines the inaccessible states building a sequence U_0, U_1, $U_2, \ldots$ of sets, where U_0 is the set of initial states, and for any $i \geq 1$ U_i is the set of accessible states, which are at distance at most i from an initial state.

INACCESSIBLE-STATES(A)

```
1  U_0 ← I
2  i ← 0
3  repeat
4     i ← i + 1
5     for all q ∈ U_{i−1}
6           do for all a ∈ Σ
7                 do U_i ← U_{i−1} ∪ δ(q, a)
8  until U_i = U_{i−1}
9  U ← Q \ U_i
10 return U
```

The inaccessible states of the automaton can be eliminated without changing the accepted language.

If $|Q| = n$ and $|\Sigma| = m$ then the running time of the algorithm (the number of steps) in the worst case is $O(n^2m)$, because the number of steps in the two embedded loops is at most nm and in the loop **repeat** at most n.

Set U has the property that $L(\mathrm{A}) \neq \emptyset$ if and only if $U \cap F \neq \emptyset$. The above algorithm can be extended by inserting the $U \cap F \neq \emptyset$ condition to decide if language $L(\mathrm{A})$ is or not empty.

Eliminating nonproductive states. Let $\mathrm{A} = (Q, \Sigma, E, I, F)$ be a finite automaton. A state is *productive* if it is on a walk which ends in a terminal state. For finding the productive states the following algorithm uses the function δ^{-1}:

$$\delta^{-1} : Q \times \Sigma \to \mathcal{P}(Q), \quad \delta^{-1}(p, a) = \{q \mid (q, a, p) \in E\} .$$

This function for a state p and a letter a gives the set of all states from which using this letter a the automaton can go into the state p.

NONPRODUCTIVE-STATES(A)

```
1  V_0 ← F
2  i ← 0
3  repeat
4     i ← i + 1
5     for all p ∈ V_{i−1}
6           do for all a ∈ Σ
7                 do V_i ← V_{i−1} ∪ δ^{−1}(p, a)
```

8 **until** $V_i = V_{i-1}$
9 $V \leftarrow Q \setminus V_i$
10 **return** V

The nonproductive states of the automaton can be eliminated without changing the accepted language.

If n is the number of states, m the number of letters in the alphabet, then the running time of the algorithm is also $O(n^2 m)$ as in the case of the algorithm INACCESSIBLE-STATES.

The set V given by the algorithm has the property that $L(\mathrm{A}) \neq \emptyset$ if and only if $V \cap I \neq \emptyset$. So, by a little modification it can be used to decide if language $L(\mathrm{A})$ is or not empty.

1.2.1. Transforming nondeterministic finite automata

As follows we will show that any NFA can be transformed in an equivalent DFA.

Theorem 1.10 *For any NFA one may construct an equivalent DFA.*

Proof Let $\mathrm{A} = (Q, \Sigma, E, I, F)$ be an NFA. Define a DFA $\overline{\mathrm{A}} = (\overline{Q}, \Sigma, \overline{E}, \overline{I}, \overline{F})$, where

- $\overline{Q} = \mathcal{P}(Q) \setminus \emptyset$,
- edges of $\overline{E}$ are those triplets (S, a, R) for which $R, S \in \overline{Q}$ are not empty, $a \in \Sigma$ and $R = \bigcup_{p \in S} \delta(p, a)$,
- $\overline{I} = \{I\}$,
- $\overline{F} = \{S \subseteq Q \mid S \cap F \neq \emptyset\}$.

We prove that $L(\mathrm{A}) = L(\overline{\mathrm{A}})$.

a) First prove that $L(\mathrm{A}) \subseteq L(\overline{\mathrm{A}})$. Let $w = a_1 a_2 \ldots a_k \in L(\mathrm{A})$. Then there exists a walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} q_{k-1} \xrightarrow{a_k} q_k, \quad q_0 \in I,\ q_k \in F\ .$$

Using the transition function $\overline{\delta}$ of NFA $\overline{\mathrm{A}}$ we construct the sets $S_0 = \{q_0\}$, $\overline{\delta}(S_0, a_1) = S_1$, $\ldots \overline{\delta}(S_{k-1}, a_k) = S_k$. Then $q_1 \in S_1, \ldots, q_k \in S_k$ and since $q_k \in F$ we get $S_k \cap F \neq \emptyset$, so $S_k \in \overline{F}$. Thus, there exists a walk

$$S_0 \xrightarrow{a_1} S_1 \xrightarrow{a_2} S_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} S_{k-1} \xrightarrow{a_k} S_k, \quad S_0 \subseteq I,\ S_k \in \overline{F}\ .$$

There are sets $S'_0, \ldots, S'_k$ for which $S'_0 = I$, and for $i = 0, 1, \ldots, k$ we have $S_i \subseteq S'_i$, and

$$S'_0 \xrightarrow{a_1} S'_1 \xrightarrow{a_2} S'_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} S'_{k-1} \xrightarrow{a_k} S'_k$$

is a productive walk. Therefore $w \in L(\overline{\mathrm{A}})$. That is $L(\mathrm{A}) \subseteq L(\overline{\mathrm{A}})$.

b) Now we show that $L(\overline{\mathrm{A}}) \subseteq L(\mathrm{A})$. Let $w = a_1 a_2 \ldots a_k \in L(\overline{\mathrm{A}})$. Then there is a walk

$$\overline{q_0} \xrightarrow{a_1} \overline{q_1} \xrightarrow{a_2} \overline{q_2} \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} \overline{q}_{k-1} \xrightarrow{a_k} \overline{q}_k, \quad \overline{q}_0 \in \overline{I},\ \overline{q}_k \in \overline{F}\ .$$

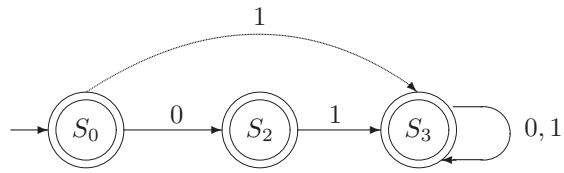


Figure 1.5 The equivalent DFA with NFA A in Fig. 1.3.

Using the definition of $\overline{F}$ we have $\overline{q}_k \cap F \neq \emptyset$, i.e. there exists $q_k \in \overline{q}_k \cap F$, that is by the definitions of $q_k \in F$ and $\overline{q}_k$ there is q_{k-1} such that $(q_{k-1}, a_k, q_k) \in E$. Similarly, there are the states $q_{k-2}, \ldots, q_1, q_0$ such that $(q_{k-2}, a_k, q_{k-1}) \in E, \ldots, (q_0, a_1, q_1) \in E$, where $q_0 \in \overline{q}_0 = I$, thus, there is a walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{k-1}} q_{k-1} \xrightarrow{a_k} q_k, \quad q_0 \in I,\ q_k \in F\ ,$$

so $L(\overline{\mathrm{A}}) \subseteq L(\mathrm{A})$. ∎

In constructing DFA we can use the corresponding transition function $\overline{\delta}$:

$$\overline{\delta}(\overline{q}, a) = \left\{ \bigcup_{q \in \overline{q}} \delta(q, a) \right\}, \ \forall \overline{q} \in \overline{Q}, \forall a \in \Sigma\ .$$

The empty set was excluded from the states, so we used here $\emptyset$ instead of $\{\emptyset\}$.

Example 1.10 Apply Theorem 1.10 to transform NFA A in Fig. 1.3. Introduce the following notation for the states of the DFA:

$$S_0 := \{q_0, q_1\}, \quad S_1 := \{q_0\}, \quad S_2 := \{q_1\}, \quad S_3 := \{q_2\},$$
$$S_4 := \{q_0, q_2\}, \quad S_5 := \{q_1, q_2\}, \quad S_6 := \{q_0, q_1, q_2\}\ ,$$

where S_0 is the initial state. Using the transition function we get the transition table:

$\overline{\delta}$	0	1
S_0	$\{S_2\}$	$\{S_3\}$
S_1	$\{S_2\}$	$\emptyset$
S_2	$\emptyset$	$\{S_3\}$
S_3	$\{S_3\}$	$\{S_3\}$
S_4	$\{S_5\}$	$\{S_3\}$
S_5	$\{S_3\}$	$\{S_3\}$
S_6	$\{S_5\}$	$\{S_3\}$

This automaton contains many inaccessible states. By algorithm Inaccessible-States we determine the accessible states of DFA:

$$U_0 = \{S_0\}, \quad U_1 = \{S_0, S_2, S_3\}, \quad U_2 = \{S_0, S_2, S_3\} = U_1 = U.$$

Initial state S_0 is also a final state. States S_2 and S_3 are final states. States S_1, S_4, S_5, S_6 are inaccessible and can be removed from the DFA. The transition table of the resulted DFA is

$\overline{\delta}$	0	1
S_0	$\{S_2\}$	$\{S_3\}$
S_2	$\emptyset$	$\{S_3\}$
S_3	$\{S_3\}$	$\{S_3\}$

The corresponding transition graph is in Fig. 1.5.

The algorithm given in Theorem 1.10 can be simplified. It is not necessary to consider all subset of the set of states of NFA. The states of DFA $\overline{\mathrm{A}}$ can be obtained successively. Begin with the state $\overline{q}_0 = I$ and determine the states $\overline{\delta}(\overline{q}_0, a)$ for all $a \in \Sigma$. For the newly obtained states we determine the states accessible from them. This can be continued until no new states arise.

In our previous example $\overline{q}_0 := \{q_0, q_1\}$ is the initial state. From this we get

$\overline{\delta}(\overline{q}_0, 0) = \{\overline{q_1}\}$, where $\overline{q}_1 := \{q_1\}$, $\quad \overline{\delta}(\overline{q}_0, 1) = \{\overline{q}_2\}$, where $\overline{q}_2 := \{q_2\}$,
$\overline{\delta}(\overline{q}_1, 0) = \emptyset$, $\quad \overline{\delta}(\overline{q}_1, 1) = \{\overline{q}_2\}$,
$\overline{\delta}(\overline{q}_2, 0) = \{\overline{q}_2\}$, $\quad \overline{\delta}(\overline{q}_2, 1) = \{\overline{q}_2\}$.

The transition table is

$\overline{\delta}$	0	1
$\overline{q}_0$	$\{\overline{q}_1\}$	$\{\overline{q}_2\}$
$\overline{q}_1$	$\emptyset$	$\{\overline{q}_2\}$
$\overline{q}_2$	$\{\overline{q}_2\}$	$\{\overline{q}_2\}$

which is the same (excepted the notation) as before.

The next algorithm will construct for an NFA $\mathrm{A} = (Q, \Sigma, E, I, F)$ the transition table M of the equivalent DFA $\overline{\mathrm{A}} = (\overline{Q}, \Sigma, \overline{E}, \overline{I}, \overline{F})$, but without to determine the final states (which can easily be included). Value of $\textsc{IsIn}(\overline{q}, \overline{Q})$ in the algorithm is true if state $\overline{q}$ is already in $\overline{Q}$ and is false otherwise. Let $a_1, a_2, \ldots, a_m$ be an ordered list of the letters of Σ.

NFA-DFA(A)

```
1  q̄_0 ← I
2  Q̄ ← {q̄_0}
3  i ← 0                          ▷ i counts the rows.
4  k ← 0                          ▷ k counts the states.
```

```
5  repeat
6        for j = 1, 2, ..., m                          ▷ j counts the columns.
7            do q̄ ← ⋃_{p∈q̄_i} δ(p, a_j)
8               if q̄ ≠ ∅
9                  then if IsIn(q̄, Q̄)
10                           then M[i, j] ← {q̄}
11                           else k ← k + 1
12                                q̄_k ← q̄
13                                M[i, j] ← {q̄_k}
14                                Q̄ ← Q̄ ∪ {q̄_k}
15                 else M[i, j] ← ∅
16       i ← i + 1
17 until i = k + 1
18 return transition table M of Ā
```

Since loop **repeat** is executed as many times as the number of states of new automaton, in worst case the running time can be exponential, because, if the number of states in NFA is n, then DFA can have even $2^n - 1$ states. (The number of subsets of a set of n elements is 2^n, including the empty set.)

Theorem 1.10 will have it that to any NFA one may construct an equivalent DFA. Conversely, any DFA is also an NFA by definition. So, the nondeterministic finite automata accepts the same class of languages as the deterministic finite automata.

1.2.2. Equivalence of deterministic finite automata

In this subsection we will use complete deterministic finite automata only. In this case $\delta(q, a)$ has a single element. In formulae, sometimes, instead of set $\delta(q, a)$ we will use its single element. We introduce for a set $A = \{a\}$ the function $elem(A)$ which give us the single element of set A, so $elem(A) = a$. Using walks which begin with the initial state and have the same label in two DFA's we can determine the equivalence of these DFA's. If only one of these walks ends in a final state, then they could not be equivalent.

Consider two DFA's over the same alphabet $A = (Q, \Sigma, E, \{q_0\}, F)$ and $A' = (Q', \Sigma, E', \{q_0'\}, F')$. We are interested to determine if they are or not equivalent. We construct a table with elements of form (q, q'), where $q \in Q$ and $q' \in Q'$. Beginning with the second column of the table, we associate a column to each letter of the alphabet Σ. If the first element of the ith row is (q, q') then at the cross of ith row and the column associated to letter a will be the pair $\Big(elem\big(\delta(q, a)\big), elem\big(\delta'(q', a)\big)\Big)$.

	$\dots \quad a \quad \dots$
$\dots$	$\dots$
(q, q')	$\Big(elem\big(\delta(q, a)\big), elem\big(\delta'(q', a)\big)\Big)$
$\dots$	$\dots$

In the first column of the first row we put (q_0, q_0') and complete the first row using the above method. If in the first row in any column there occur a pair of states from which one is a final state and the other not then the algorithm ends, the two automata are ***not equivalent.*** If there is no such a pair of states, every new pair is written in the first column. The algorithm continues with the next unfilled row. If no new pair of states occurs in the table and for each pair both of states are final or both are not, then the algorithm ends and the two DFA are ***equivalent.***

If $|Q| = n$, $|Q'| = n'$ and $|\Sigma| = m$ then taking into account that in worst case loop **repeat** is executed nn' times, loop **for** m times, the running time of the algorithm in worst case will be $O(nn'm)$, or if $n = n'$ then $O(n^2m)$.

Our algorithm was described to determine the equivalence of two complete DFA's. If we have to determine the equivalence of two NFA's, first we transform them into complete DFA's and after this we can apply the above algorithm.

DFA-EQUIVALENCE(A, A′)

write in the first column of the first row the pair (q_0, q_0')
$i \leftarrow 0$
repeat
    $i \leftarrow i + 1$
    let (q, q') be the pair in the first column of the ith row
    for all $a \in \Sigma$
        do write in the column associated to a in the ith row
            the pair $\Big(elem\big(\delta(q,a)\big), elem\big(\delta'(q',a)\big)\Big)$
            if one state in $\Big(elem\big(\delta(q,a)\big), elem\big(\delta'(q',a)\big)\Big)$ is final
                and the other not
                then return NO
                else write pair $\Big(elem\big(\delta(q,a)\big), elem\big(\delta'(q',a)\big)\Big)$
                    in the next empty row of the first column,
                    if not occurred already in the first column
until the first element of $(i+1)$th row becomes empty
return YES

Example 1.11 Determine if the two DFA's in Fig. 1.6 are equivalent or not. The algorithm gives the table

	a	b
(q_0, p_0)	(q_2, p_3)	(q_1, p_1)
(q_2, p_3)	(q_1, p_2)	(q_2, p_3)
(q_1, p_1)	(q_2, p_3)	(q_0, p_0)
(q_1, p_2)	(q_2, p_3)	(q_0, p_0)

The two DFA's are equivalent because all possible pairs of states are considered and in every pair both states are final or both are not final.

Example 1.12 The table of the two DFA's in Fig. 1.7 is:

	a	b
(q_0, p_0)	(q_1, p_3)	(q_2, p_1)
(q_1, p_3)	(q_2, p_2)	(q_0, p_3)
(q_2, p_1)		
(q_2, p_2)		

These two DFA's are not equivalent, because in the last column of the second row in the pair (q_0, p_3) the first state is final and the second not.

1.2.3. Equivalence of finite automata and regular languages.

We have seen that NFA's accept the same class of languages as DFA's. The following theorem states that this class is that of regular languages.

Theorem 1.11 *If L is a language accepted by a DFA, then one may construct a regular grammar which generates language L.*

Proof Let $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ be the DFA accepting language L, that is $L = L(\mathrm{A})$. Define the regular grammar $G = (Q, \Sigma, P, q_0)$ with the productions:

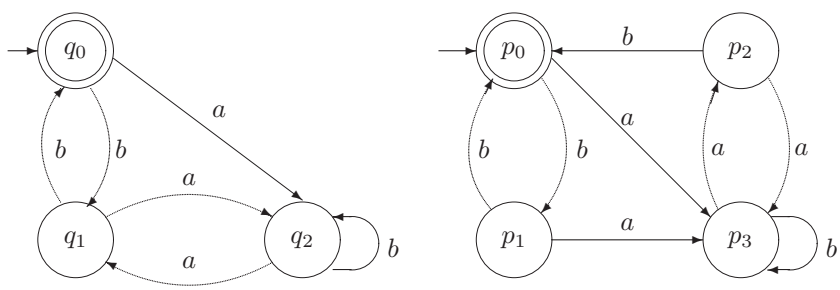


Figure 1.6 Equivalent DFA's (Example 1.11).

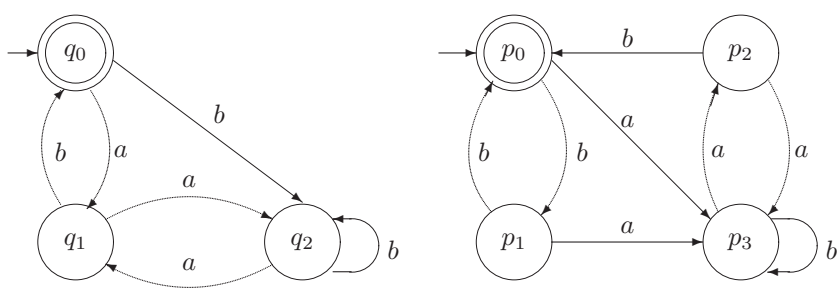


Figure 1.7 Non equivalent DFA's (Example 1.12).

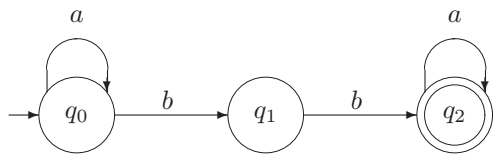


Figure 1.8 DFA of the Example 1.13.

• If $(p, a, q) \in E$ for $p, q \in Q$ and $a \in \Sigma$, then put production $p \to aq$ in P.
• If $(p, a, q) \in E$ and $q \in F$, then put also production $p \to a$ in P.

Prove that $L(G) = L(\mathrm{A}) \setminus \{\varepsilon\}$.

Let $u = a_1 a_2 \dots a_n \in L(\mathrm{A})$ and $u \neq \varepsilon$. Thus, since A accepts word u, there is a walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n, \quad q_n \in F \ .$$

Then there are in P the productions

$$q_0 \to a_1 q_1, \quad q_1 \to a_2 q_2, \quad \dots, \quad q_{n-2} \to a_{n-1} q_{n-1}, \quad q_{n-1} \to a_n$$

(in the right-hand side of the last production q_n does not occur, because $q_n \in F$), so there is the derivation

$$q_0 \Longrightarrow a_1 q_1 \Longrightarrow a_1 a_2 q_2 \Longrightarrow \dots \Longrightarrow a_1 a_2 \dots a_{n-1} q_{n-1} \Longrightarrow a_1 a_2 \dots a_n .$$

Therefore, $u \in L(G)$.

Conversely, let $u = a_1 a_2 \dots a_n \in L(G)$ and $u \neq \varepsilon$. Then there exists a derivation

$$q_0 \Longrightarrow a_1 q_1 \Longrightarrow a_1 a_2 q_2 \Longrightarrow \dots \Longrightarrow a_1 a_2 \dots a_{n-1} q_{n-1} \Longrightarrow a_1 a_2 \dots a_n \ ,$$

in which productions

$$q_0 \to a_1 q_1, \quad q_1 \to a_2 q_2, \quad \dots, \quad q_{n-2} \to a_{n-1} q_{n-1}, \quad q_{n-1} \to a_n$$

were used, which by definition means that in DFA A there is a walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} q_{n-1} \xrightarrow{a_n} q_n \ ,$$

and since q_n is a final state, $u \in L(\mathrm{A}) \setminus \{\varepsilon\}$.

If the DFA accepts also the empty word ε, then in the above grammar we introduce a new start symbol q_0' instead of q_0, consider the new production $q_0' \to \varepsilon$ and for each production $q_0 \to \alpha$ introduce also $q_0' \to \alpha$. ■

Example 1.13 Let $\mathrm{A} = (\{q_0, q_1, q_2\}, \{a, b\}, E, \{q_0\}, \{q_2\})$ be a DFA, where $E = \big\{(q_0, a, q_0), (q_0, b, q_1), (q_1, b, q_2), (q_2, a, q_2)\big\}$. The corresponding transition table is

δ	a	b
q_0	$\{q_0\}$	$\{q_1\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\{q_2\}$	$\emptyset$

The transition graph of A is in Fig. 1.8. By Theorem 1.11 we define regular grammar $G = (\{q_0, q_1, q_2\}, \{a, b\}, P, q_0)$ with the productions in P

$$q_0 \to aq_0 \mid bq_1, \qquad q_1 \to bq_2 \mid b, \qquad q_2 \to aq_2 \mid a \ .$$

One may prove that $L(\mathrm{A}) = \{a^m bba^n \mid m \geq 0, n \geq 0\}$.

The method described in the proof of Theorem 1.11 easily can be given as an algorithm. The productions of regular grammar $G = (Q, \Sigma, P, q_0)$ obtained from the DFA $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ can be determined by the following algorithm.

REGULAR-GRAMMAR-FROM-DFA(A)

```
1  P ← ∅
2  for all p ∈ Q
3       do for all a ∈ Σ
4            do for all q ∈ Q
5                 do if (p, a, q) ∈ E
6                      then P ← P ∪ {p → aq}
7                           if q ∈ F
8                              then P ← P ∪ {p → a}
9  if q_0 ∈ F
10    then P ← P ∪ {q_0 → ε}
11 return G
```

It is easy to see that the running time of the algorithm is $\Theta(n^2 m)$, if the number of states is n and the number of letter in alphabet is m. In lines 2–4 we can consider only one loop, if we use the elements of E. Then the worst case running time is $\Theta(p)$, where p is the number of transitions of DFA. This is also $O(n^2 m)$, since all transitions are possible. This algorithm is:

REGULAR-GRAMMAR-FROM-DFA'(A)

```
1  P ← ∅
2  for all (p, a, q) ∈ E
3       do P ← P ∪ {p → aq}
4          if q ∈ F
5             then P ← P ∪ {p → a}
6  if q_0 ∈ F
7       then P ← P ∪ {q_0 → ε}
8  return G
```

Theorem 1.12 *If $L = L(G)$ is a regular language, then one may construct an NFA that accepts language L.*

Proof Let $G = (N, T, P, S)$ be the grammar which generates language L. Define NFA $\mathrm{A} = (Q, T, E, \{S\}, F)$:

- $Q = N \cup \{Z\}$, where $Z \notin N \cup T$ (i.e. Z is a new symbol),
- For every production $A \to aB$, define transition (A, a, B) in E.
- For every production $A \to a$, define transition (A, a, Z) in E.

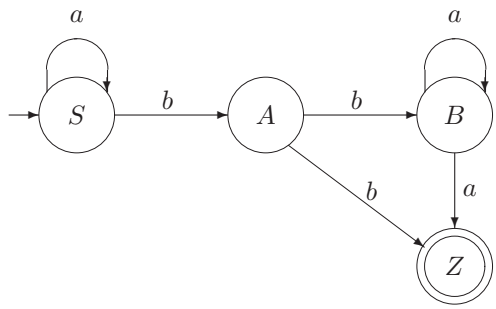


Figure 1.9 NFA associated to grammar in Example 1.14.

- $F = \begin{cases} \{Z\} & \text{if production } S \to \varepsilon \text{ does not occur in } G, \\ \{Z, S\} & \text{if production } S \to \varepsilon \text{ occurs in } G. \end{cases}$

Prove that $L(G) = L(A)$.
Let $u = a_1 a_2 \dots a_n \in L(G)$, $u \neq \varepsilon$. Then there is in G a derivation of word u:

$$S \Longrightarrow a_1 A_1 \Longrightarrow a_1 a_2 A_2 \Longrightarrow \dots \Longrightarrow a_1 a_2 \dots a_{n-1} A_{n-1} \Longrightarrow a_1 a_2 \dots a_n \ .$$

This derivation is based on productions

$$S \to a_1 A_1, \ \ A_1 \to a_2 A_2, \ \ \dots, \ \ A_{n-2} \to a_{n-1} A_{n-1}, \ \ A_{n-1} \to a_n \ .$$

Then, by the definition of the transitions of NFA A there exists a walk

$$S \xrightarrow{a_1} A_1 \xrightarrow{a_2} A_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} A_{n-1} \xrightarrow{a_n} Z, \ \ Z \in F \ .$$

Thus, $u \in L(\mathrm{A})$. If $\varepsilon \in L(G)$, there is production $S \to \varepsilon$, but in this case the initial state is also a final one, so $\varepsilon \in L(A)$. Therefore, $L(G) \subseteq L(A)$.

Let now $u = a_1 a_2 \dots a_n \in L(\mathrm{A})$. Then there exists a walk

$$S \xrightarrow{a_1} A_1 \xrightarrow{a_2} A_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{n-1}} A_{n-1} \xrightarrow{a_n} Z, \ \ Z \in F \ .$$

If u is the empty word, then instead of Z we have in the above formula S, which also is a final state. In other cases only Z can be as last symbol. Thus, in G there exist the productions

$$S \to a_1 A_1, \ \ A_1 \to a_2 A_2, \ \ \dots, \ \ A_{n-2} \to a_{n-1} A_{n-1}, \ \ A_{n-1} \to a_n \ ,$$

and there is the derivation

$$S \Longrightarrow a_1 A_1 \Longrightarrow a_1 a_2 A_2 \Longrightarrow \dots \Longrightarrow a_1 a_2 \dots a_{n-1} A_{n-1} \Longrightarrow a_1 a_2 \dots a_n \ ,$$

thus, $u \in L(G)$ and therefore $L(A) \subseteq L(G)$. ■

Example 1.14 Let $G = (\{S, A, B\}, \{a, b\}, \{S \to aS, S \to bA, A \to bB, A \to b, B \to aB, B \to a\}, S)$ be a regular grammar. The NFA associated is $A = (\{S, A, B, Z\}, \{a, b\}, E, \{S\}, \{Z\})$, where $E = \big\{(S, a, S), (S, b, A), (A, b, B), (A, b, Z), (B, a, B), (B, a, Z)\big\}$. The corresponding transition table is

δ	a	b
S	$\{S\}$	$\{A\}$
A	$\emptyset$	$\{B, Z\}$
B	$\{B, Z\}$	$\emptyset$
E	$\emptyset$	$\emptyset$

The transition graph is in Fig. 1.9. This NFA can be simplified, states B and Z can be contracted in one final state.

Using the above theorem we define an algorithm which associate an NFA A = $(Q, T, E, \{S\}, F)$ to a regular grammar $G = (N, T, P, S)$.

NFA-FROM-REGULAR-GRAMMAR(A)

```
1  E ← ∅
2  Q ← N ∪ {Z}
3  for all A ∈ N
4       do for all a ∈ T
5            do if (A → a) ∈ P
6                 then E ← E ∪ {(A, a, Z)}
7            for all B ∈ N
8                 do if (A → aB) ∈ P
9                      then E ← E ∪ {(A, a, B)}
10 if (S → ε) ∉ P
11    then F ← {Z}
12    else F ← {Z, S}
13 return A
```

As in the case of algorithm REGULAR-GRAMMAR-FROM-DFA, the running time is $\Theta(n^2 m)$, where n is number of nonterminals and m the number of terminals. Loops in lines 3, 4 and 7 can be replaced by only one, which uses productions. The running time in this case is better and is equal to $\Theta(p)$, if p is the number of productions. This algorithm is:

NFA-FROM-REGULAR-GRAMMAR'(A)

```
1  E ← ∅
2  Q ← N ∪ {Z}
3  for all (A → u) ∈ P
4       do if u = a
5            then E ← E ∪ {(A, a, Z)}
6          if u = aB
7            then E ← E ∪ {(A, a, B)}
8  if (S → ε) ∉ P
9     then F ← {Z}
10    else F ← {Z, S}
11 return A
```

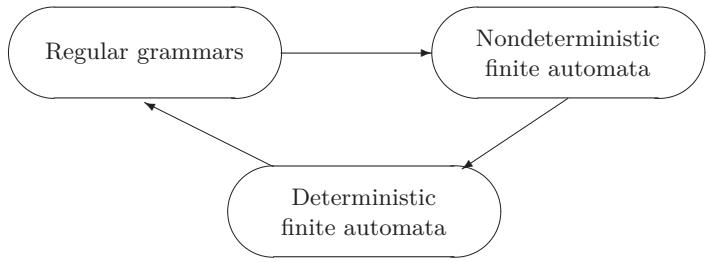


Figure 1.10 Relations between regular grammars and finite automata. To any regular grammar one may construct an NFA which accepts the language generated by that grammar. Any NFA can be transformed in an equivalent DFA. To any DFA one may construct a regular grammar which generates the language accepted by that DFA.

From Theorems 1.10, 1.11 and 1.12 results that the class of regular languages coincides with the class of languages accepted by NFA's and also with class of languages accepted by DFA's. The result of these three theorems is illustrated in Fig. 1.10 and can be summarised also in the following theorem.

Theorem 1.13 *The following three class of languages are the same:*
- *the class of regular languages,*
- *the class of languages accepted by DFA's,*
- *the class of languages accepted by NFA's.*

Operation on regular languages It is known (see Theorem 1.8) that the set $\mathcal{L}_3$ of regular languages is closed under the regular operations, that is if L_1, L_2 are regular languages, then languages $L_1 \cup L_2$, $L_1 L_2$ and L_1^* are also regular. For regular languages are true also the following statements.

The complement of a regular language is also regular. This is easy to prove using automata. Let L be a regular language and let $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ be a DFA which accepts language L. It is easy to see that the DFA $\overline{\mathrm{A}} = (Q, \Sigma, E, \{q_0\}, Q \setminus F)$ accepts language $\overline{L}$. So, $\overline{L}$ is also regular.

The intersection of two regular languages is also regular. Since $L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$, the intersection is also regular.

The difference of two regular languages is also regular. Since $L_1 \setminus L_2 = L_1 \cap \overline{L_2}$, the difference is also regular.

1.2.4. Finite automata with special moves

A finite automaton with ε-moves (FA with ε-moves) extends NFA in such way that it may have transitions on the empty input ε, i.e. it may change a state without reading any input symbol. In the case of a FA with ε-moves $\mathrm{A} = (Q, \Sigma, E, I, F)$ for the set of transitions it is true that $E \subseteq Q \times \big(\Sigma \cup \{\varepsilon\}\big) \times Q$.

The transition function of a FA with ε-moves is:

$$\delta : Q \times \big(\Sigma \cup \{\varepsilon\}\big) \to \mathcal{P}(Q), \quad \delta(p, a) = \{q \in Q \mid (p, a, q) \in E\} .$$

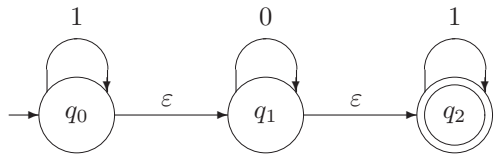


Figure 1.11 Finite automata ϵ-moves.

The FA with ε-moves in Fig. 1.11 accepts words of form uvw, where $u \in \{1\}^*$, $v \in \{0\}^*$ and $w \in \{1\}^*$.

Theorem 1.14 *To any FA with ε-moves one may construct an equivalent NFA (without ε-moves).*

Let $\mathrm{A} = (Q, \Sigma, E, I, F)$ be an FA with ε-moves and we construct an equivalent NFA $\overline{\mathrm{A}} = (Q, \Sigma, \overline{E}, I, \overline{F})$. The following algorithm determines sets $\overline{F}$ and $\overline{E}$.

For a state q denote by $\Lambda(q)$ the set of states (including even q) in which one may go from q using ε-moves only. This may be extended also to sets

$$\Lambda(S) = \bigcup_{q \in S} \Lambda(q), \quad \forall S \subseteq Q \ .$$

Clearly, for all $q \in Q$ and $S \subseteq Q$ both $\Lambda(q)$ and $\Lambda(S)$ may be computed. Suppose in the sequel that these are given.

The following algorithm determine the transitions using the transition function $\overline{\delta}$, which is defined in line 5.

If $|Q| = n$ and $|\Sigma| = m$,, then lines 2–6 show that the running time in worst case is $O(n^2 m)$.

ELIMINATE-EPSILON-MOVES(A)

1 $\overline{F} \leftarrow F \cup \{q \in I \mid \Lambda(q) \cap F \neq \emptyset\}$
2 **for** all $q \in Q$
3 **do for** all $a \in \Sigma$
4 **do** $\Delta \leftarrow \bigcup_{p \in \Lambda(q)} \delta(p, a)$
5 $\overline{\delta}(q, a) \leftarrow \Delta \cup \left(\bigcup_{p \in \Delta} \Lambda(p) \right)$
6 $\overline{E} \leftarrow \left\{ (p, a, q), \mid p, q \in Q,\ a \in \Sigma, q \in \overline{\delta}(p, a) \right\}$
7 **return** $\overline{\mathrm{A}}$

Example 1.15 Consider the FA with ε-moves in Fig. 1.11. The corresponding transition table is:

δ	0	1	ε
q_0	$\emptyset$	$\{q_0\}$	$\{q_1\}$
q_1	$\{q_1\}$	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\{q_2\}$	$\emptyset$

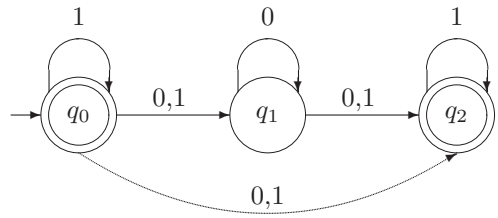


Figure 1.12 NFA equivalent to FA with ε-moves given in Fig. 1.11.

Apply algorithm ELIMINATE-EPSILON-MOVES.
$\Lambda(q_0) = \{q_0, q_1, q_2\}$, $\Lambda(q_1) = \{q_1, q_2\}$, $\Lambda(q_2) = \{q_2\}$
$\Lambda(I) = \Lambda(q_0)$, and its intersection with F is not empty, thus $\overline{F} = F \cup \{q_0\} = \{q_0, q_2\}$.
$(q_0, 0)$:
$\Delta = \delta(q_0, 0) \cup \delta(q_1, 0) \cup \delta(q_2, 0) = \{q_1\}$, $\{q_1\} \cup \Lambda(q_1) = \{q_1, q_2\}$
$\overline{\delta}(q_0, 0) = \{q_1, q_2\}$.
$(q_0, 1)$:
$\Delta = \delta(q_0, 1) \cup \delta(q_1, 1) \cup \delta(q_2, 1) = \{q_0, q_2\}$, $\{q_0, q_2\} \cup (\Lambda(q_0) \cup \Lambda(q_2)) = \{q_0, q_1, q_2\}$
$\overline{\delta}(q_0, 1) = \{q_0, q_1, q_2\}$
$(q_1, 0)$:
$\Delta = \delta(q_1, 0) \cup \delta(q_2, 0) = \{q_1\}$, $\{q_1\} \cup \Lambda(q_1) = \{q_1, q_2\}$
$\overline{\delta}(q_1, 0) = \{q_1, q_2\}$
$(q_1, 1)$:
$\Delta = \delta(q_1, 1) \cup \delta(q_2, 1) = \{q_2\}$, $\{q_2\} \cup \Lambda(q_2) = \{q_2\}$
$\overline{\delta}(q_1, 1) = \{q_2\}$
$(q_2, 0)$: $\Delta = \delta(q_2, 0) = \emptyset$
$\overline{\delta}(q_2, 0) = \emptyset$
$(q_2, 1)$:
$\Delta = \delta(q_2, 1) = \{q_2\}$, $\{q_2\} \cup \Lambda(q_2) = \{q_2\}$
$\overline{\delta}(q_2, 1) = \{q_2\}$.
The transition table of NFA $\overline{\mathrm{A}}$ is:

$\overline{\delta}$	0	1
q_0	$\{q_1, q_2\}$	$\{q_0, q_1, q_2\}$
q_1	$\{q_1, q_2\}$	$\{q_2\}$
q_2	$\emptyset$	$\{q_2\}$

and the transition graph is in Fig. 1.12.

Define regular operations on NFA: union, product and iteration. The result will be an FA with ε-moves.

Operation will be given also by diagrams. An NFA is given as in Fig. 1.13(a). Initial states are represented by a circle with an arrow, final states by a double circle.

Let $\mathrm{A}_1 = (Q_1, \Sigma_1, E_1, I_1, F_1)$ and $\mathrm{A}_2 = (Q_2, \Sigma_2, E_2, I_2, F_2)$ be NFA. The result of any operation is a FA with ε-moves $\mathrm{A} = (Q, \Sigma, E, I, F)$. Suppose that $Q_1 \cap Q_2 = \emptyset$ always. If not, we can rename the elements of any set of states.

Union. $\mathrm{A} = \mathrm{A}_1 \cup \mathrm{A}_2$, where

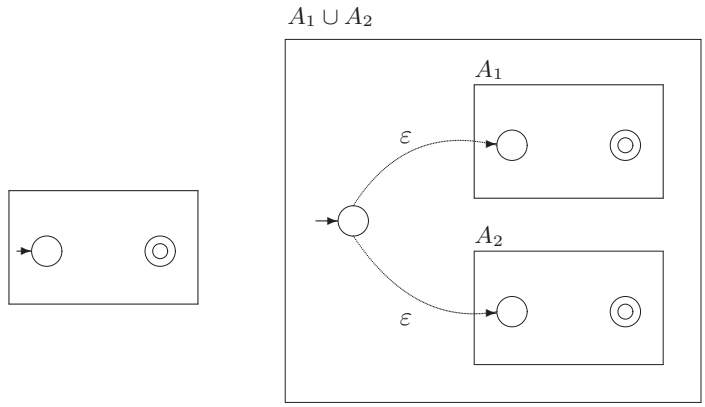


Figure 1.13 **(a)** Representation of an NFA. Initial states are represented by a circle with an arrow, final states by a double circle. **(b)** Union of two NFA's.

$$
\begin{aligned}
Q &= Q_1 \cup Q_2 \cup \{q_0\} \ , \\
\Sigma &= \Sigma_1 \cup \Sigma_2 \ , \\
I &= \{q_0\} \ , \\
F &= F_1 \cup F_2 \ , \\
E &= E_1 \cup E_2 \cup \bigcup_{q \in I_1 \cup I_2} \{(q_0, \varepsilon, q)\} \ .
\end{aligned}
$$

For the result of the union see Fig. 1.13(b). The result is the same if instead of a single initial state we choose as set of initial states the union $I_1 \cup I_2$. In this case the result automaton will be without ε-moves. By the definition it is easy to see that $L(A_1 \cup A_2) = L(A_1) \cup L(A_2)$.

Product. $\mathrm{A} = \mathrm{A}_1 \cdot \mathrm{A}_2$, where

$$
\begin{aligned}
Q &= Q_1 \cup Q_2, \\
\Sigma &= \Sigma_1 \cup \Sigma_2, \\
F &= F_2, \\
I &= I_1, \\
E &= E_1 \cup E_2 \cup \bigcup_{\substack{p \in F_1 \\ q \in I_2}} \{(p, \varepsilon, q)\} \ .
\end{aligned}
$$

For the result automaton see Fig. 1.14(a). Here also $L(A_1 \cdot A_2) = L(A_1)L(A_2)$.

Iteration. $\mathrm{A} = \mathrm{A}_1{}^*$, where

$$
\begin{aligned}
Q &= Q_1 \cup \{q_0\}, \\
\Sigma &= \Sigma_1, \\
F &= F_1 \cup \{q_0\}, \\
I &= \{q_0\}
\end{aligned}
$$

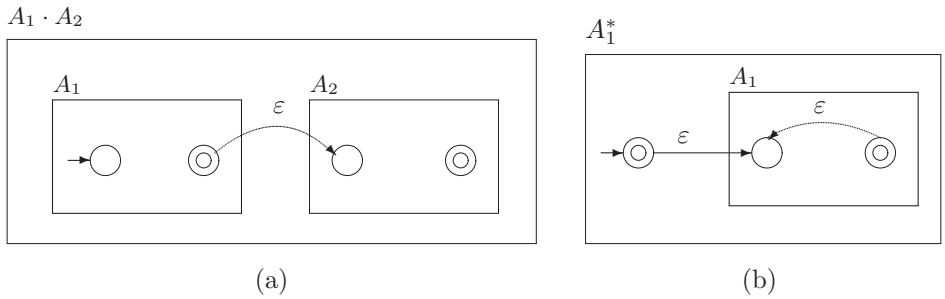


Figure 1.14 **(a)** Product of two FA. **(b)** Iteration of an FA.

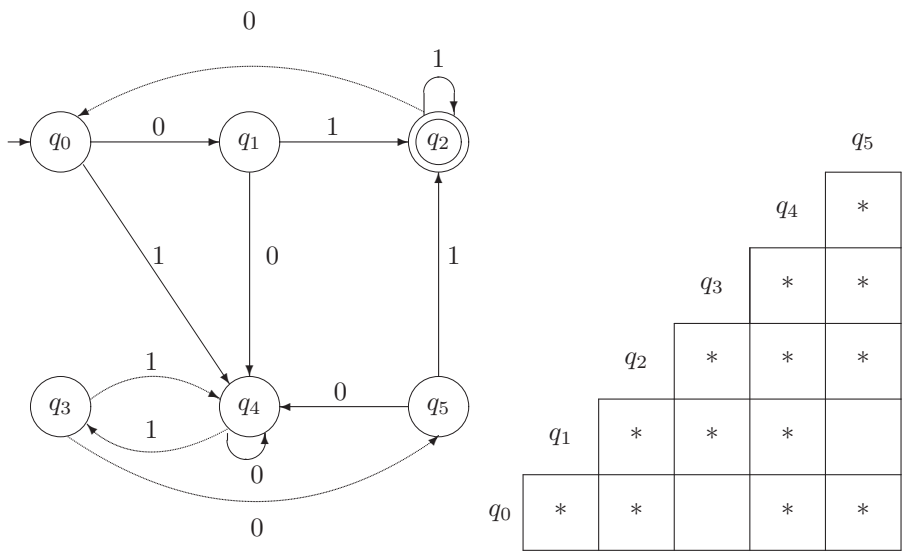


Figure 1.15 Minimization of DFA.

$$E = E_1 \cup \bigcup_{p \in I_1} \{(q_0, \varepsilon, p)\} \cup \bigcup_{\substack{q \in F_1 \\ p \in I_1}} \{(q, \varepsilon, p)\} .$$

The iteration of an FA can be seen in Fig. 1.14(b). For this operation it is also true that $L(A_1^*) = \big(L(A_1)\big)^*$.

The definition of these tree operations proves again that regular languages are closed under the regular operations.

1.2.5. Minimization of finite automata

A DFA $A = (Q, \Sigma, E, \{q_0\}, F)$ is called ***minimum state automaton*** if for any equivalent complete DFA $A' = (Q', \Sigma, E', \{q_0'\}, F')$ it is true that $|Q| \leq |Q'|$. We give an algorithm which builds for any complete DFA an equivalent minimum state automaton.

States p and q of an DFA $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ are ***equivalent*** if for arbitrary word u we reach from both either final or nonfinal states, that is

$$p \equiv q \text{ if for any word } u \in \Sigma^* \begin{cases} p \xrightarrow{u} r,\ r \in F \text{ and } q \xrightarrow{u} s,\ s \in F \text{ or} \\ p \xrightarrow{u} r,\ r \notin F \text{ and } q \xrightarrow{u} s,\ s \notin F\ . \end{cases}$$

If two states are not equivalent, then they are distinguishable. In the following algorithm the distinguishable states will be marked by a star, and equivalent states will be merged. The algorithm will associate list of pair of states with some pair of states expecting a later marking by a star, that is if we mark a pair of states by a star, then all pairs on the associated list will be also marked by a star. The algorithm is given for DFA without inaccessible states. The used DFA is complete, so $\delta(p, a)$ contains exact one element, function *elem* defined on page 35, which gives the unique element of the set, will be also used here.

AUTOMATON-MINIMIZATION(A)

1 mark with a star all pairs of states $\{p, q\}$ for which
  $p \in F$ and $q \notin F$ or $p \notin F$ and $q \in F$
2 associate an empty list with each unmarked pair $\{p, q\}$
3 **for** all unmarked pair of states $\{p, q\}$ and for all symbol $a \in \Sigma$
    examine pairs of states $\{elem(\delta(p, a)), elem(\delta(q, a))\}$
    if any of these pairs is marked,
      then mark also pair $\{p, q\}$ with all the elements on the list before
          associated with pair $\{p, q\}$
      else if all the above pairs are unmarked
          then put pair $\{p, q\}$ on each list associated with pairs
              $\{elem(\delta(p, a)), elem(\delta(q, a))\}$, unless $\delta(p, a) = \delta(q, a)$
4 merge all unmarked (equivalent) pairs

After finishing the algorithm, if a cell of the table does not contain a star, then the states corresponding to its row and column index, are equivalent and may be merged. Merging states is continued until it is possible. We can say that the equivalence relation decomposes the set of states in equivalence classes, and the states in such a class may be all merged.

Remark. The above algorithm can be used also in the case of an DFA which is not complete, that is there are states for which does not exist transition. Then a pair $\{\emptyset, \{q\}\}$ may occur, and if q is a final state, consider this pair marked.

Example 1.16 Let be the DFA in Fig. 1.15. We will use a table for marking pairs with a star. Marking pair $\{p, q\}$ means putting a star in the cell corresponding to row p and column q (or row q and column p).

First we mark pairs $\{q_2, q_0\}$, $\{q_2, q_1\}$, $\{q_2, q_3\}$, $\{q_2, q_4\}$ and $\{q_2, q_5\}$ (because q_2 is the single final state). Then consider all unmarked pairs and examine them as the algorithm requires. Let us begin with pair $\{q_0, q_1\}$. Associate with it pairs $\{elem(\delta(q_0, 0)), elem(\delta(q_1, 0))\}, \{elem(\delta(q_0, 1)), elem(\delta(q_1, 1))\}$, that is $\{q_1, q_4\}$, $\{q_4, q_2\}$. Because pair $\{q_4, q_2\}$ is already marked, mark also pair $\{q_0, q_1\}$.

In the case of pair $\{q_0, q_3\}$ the new pairs are $\{q_1, q_5\}$ and $\{q_4, q_4\}$. With pair $\{q_1, q_5\}$

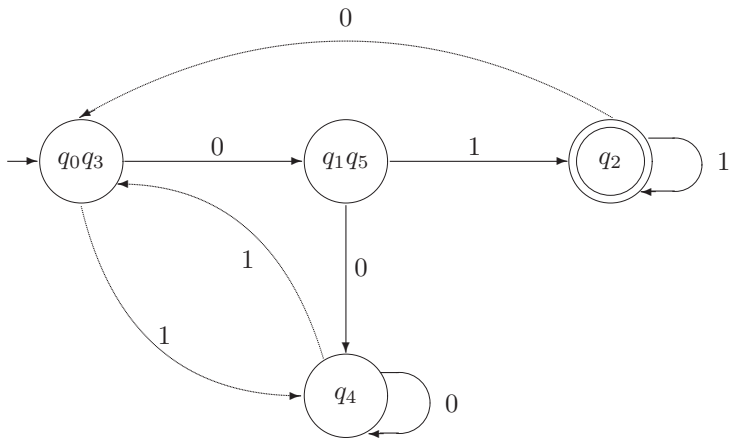


Figure 1.16 Minimum automaton equivalent with DFA in Fig. 1.15.

associate pair $\{q_0, q_3\}$ on a list, that is

$$\{q_1, q_5\} \longrightarrow \{q_0, q_3\} \ .$$

Now continuing with $\{q_1, q_5\}$ one obtain pairs $\{q_4, q_4\}$ and $\{q_2, q_2\}$, with which nothing are associated by algorithm.

Continue with pair $\{q_0, q_4\}$. The associated pairs are $\{q_1, q_4\}$ and $\{q_4, q_3\}$. None of them are marked, so associate with them on a list pair $\{q_0, q_4\}$, that is

$$\{q_1, q_4\} \longrightarrow \{q_0, q_4\}, \qquad \{q_4, q_3\} \longrightarrow \{q_0, q_4\} \ .$$

Now continuing with $\{q_1, q_4\}$ we get the pairs $\{q_4, q_4\}$ and $\{q_2, q_3\}$, and because this latter is marked we mark pair $\{q_1, q_4\}$ and also pair $\{q_0, q_4\}$ associated to it on a list. Continuing we will get the table in Fig. 1.15, that is we get that $q_0 \equiv q_3$ and $q_1 \equiv q_5$. After merging them we get an equivalent minimum state automaton (see Fig. 1.16).

1.2.6. Pumping lemma for regular languages

The following theorem, called *pumping lemma* for historical reasons, may be efficiently used to prove that a language is not regular. It is a sufficient condition for a regular language.

Theorem 1.15 (pumping lemma). *For any regular language L there exists a natural number $n \geq 1$ (depending only on L), such that any word u of L with length at least n may be written as $u = xyz$ such that*

(1) $|xy| \leq n$,
(2) $|y| \geq 1$,
(3) $xy^iz \in L$ *for all* $i = 0, 1, 2, \ldots$.

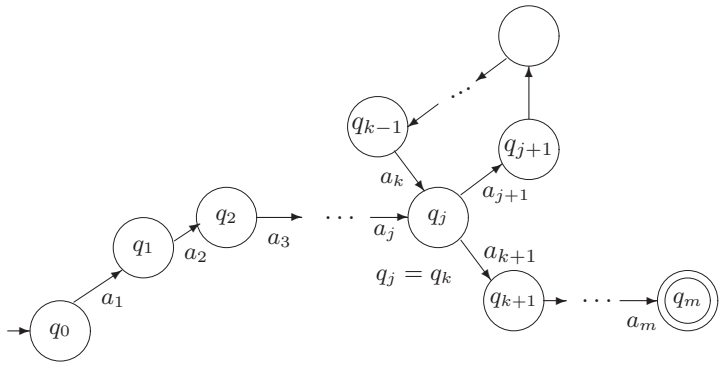


Figure 1.17 Sketch of DFA used in the proof of the pumping lemma.

Proof If L is a regular language, then there is such an DFA which accepts L (by Theorems 1.12 and 1.10). Let $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ be this DFA, so $L = L(\mathrm{A})$. Let n be the number of its states, that is $|Q| = n$. Let $u = a_1 a_2 \dots a_m \in L$ and $m \geq n$. Then, because the automaton accepts word u, there are states $q_0, q_1, \dots, q_m$ and walk

$$q_0 \xrightarrow{a_1} q_1 \xrightarrow{a_2} q_2 \xrightarrow{a_3} \cdots \xrightarrow{a_{m-1}} q_{m-1} \xrightarrow{a_m} q_m,\ q_m \in F\ .$$

Because the number of states is n and $m \geq n$, by the pigeonhole principle[3] states $q_0, q_1, \dots, q_m$ can not all be distinct (see Fig. 1.17), there are at least two of them which are equal. Let $q_j = q_k$, where $j < k$ and k is the least such index. Then $j < k \leq n$. Decompose word u as:

$x = a_1 a_2 \dots a_j$
$y = a_{j+1} a_{j+2} \dots a_k$
$z = a_{k+1} a_{k+2} \dots a_m$.

This decomposition immediately yields to $|xy| \leq n$ and $|y| \geq 1$. We will prove that $xy^i z \in L$ for any i.
Because $u = xyz \in L$, there exists an walk

$$q_0 \xrightarrow{x} q_j \xrightarrow{y} q_k \xrightarrow{z} q_m,\ q_m \in F,$$

and because of $q_j = q_k$, this may be written also as

$$q_0 \xrightarrow{x} q_j \xrightarrow{y} q_j \xrightarrow{z} q_m,\ q_m \in F\ .$$

From this walk $q_j \xrightarrow{y} q_j$ can be omitted or can be inserted many times. So, there are the following walks:

$$q_0 \xrightarrow{x} q_j \xrightarrow{z} q_m,\ q_m \in F\ ,$$

$$q_0 \xrightarrow{x} q_j \xrightarrow{y} q_j \xrightarrow{y} \dots \xrightarrow{y} q_j \xrightarrow{z} q_m,\ q_m \in F\ .$$

[3] *Pigeonhole principle:* If we have to put more than k objects into k boxes, then at least one box will contain at least two objects.

Therefore $xy^iz \in L$ for all i, and this proves the theorem. ■

Example 1.17 We use the pumping lemma to show that $L_1 = \{a^k b^k \mid k \geq 1\}$ is not regular. Assume that L_1 is regular, and let n be the corresponding natural number given by the pumping lemma. Because the length of the word $u = a^n b^n$ is $2n$, this word can be written as in the lemma. We prove that this leads to a contradiction. Let $u = xyz$ be the decomposition as in the lemma. Then $|xy| \leq n$, so x and y can contain no other letters than a, and because we must have $|y| \geq 1$, word y contains at least one a. Then xy^iz for $i \neq 1$ will contain a different number of a's and b's, therefore $xy^iz \notin L_1$ for any $i \neq 1$. This is a contradiction with the third assertion of the lemma, this is why that assumption that L_1 is regular, is false. Therefore $L_1 \notin \mathcal{L}_3$.

Because the context-free grammar $G_1 = (\{S\}, \{a, b\}, \{S \to ab, S \to aSb\}, S)$ generates language L_1, we have $L_1 \in \mathcal{L}_2$. From these two follow that $\mathcal{L}_3 \subset \mathcal{L}_2$.

Example 1.18 We show that $L_2 = \left\{u \in \{0,1\}^* \mid n_0(u) = n_1(u)\right\}$ is not regular. ($n_0(u)$ is the number of 0's in u, while $n_1(u)$ the number of 1's).

We proceed as in the previous example using here word $u = 0^n 1^n$, where n is the natural number associated by lemma to language L_2.

Example 1.19 We prove, using the pumping lemma, that $L_3 = \left\{uu \mid u \in \{a, b\}^*\right\}$ is not a regular language. Let $w = a^n b a^n b = xyz$ be, where n here is also the natural number associated to L_3 by the pumping lemma. From $|xy| \leq n$ we have that y contains no other letters than a, but it contains at least one. By lemma we have $xz \in L_3$, that is not possible. Therefore L_3 is not regular.

Pumping lemma has several interesting consequences.

Corollary 1.16 *Regular language L is not empty if and only if there exists a word $u \in L$, $|u| < n$, where n is the natural number associated to L by the pumping lemma.*

Proof The assertion in a direction is obvious: if there exists a word shorter than n in L, then $L \neq \emptyset$. Conversely, let $L \neq \emptyset$ and let u be the shortest word in L. We show that $|u| < n$. If $|u| \geq n$, then we apply the pumping lemma, and give the decomposition $u = xyz$, $|y| > 1$ and $xz \in L$. This is a contradiction, because $|xz| < |u|$ and u is the shortest word in L. Therefore $|u| < n$. ■

Corollary 1.17 *There exists an algorithm that can decide if a regular language is or not empty.*

Proof Assume that $L = L(\mathrm{A})$, where $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ is a DFA. By Consequence 1.16 and Theorem 1.15 language L is not empty if and only if it contains a word shorter than n, where n is the number of states of automaton A. By this it is enough to decide that there is a word shorter than n which is accepted by automaton A. Because the number of words shorter than n is finite, the problem can be decided. ■

When we had given an algorithm for inaccessible states of a DFA, we remarked that the procedure can be used also to decide if the language accepted by that automaton is or not empty. Because finite automata accept regular languages, we can consider to have already two procedures to decide if a regular languages is or not empty. Moreover, we have a third procedure, if we take into account that the algorithm for finding productive states also can be used to decide on a regular language when it is empty.

Corollary 1.18 *A regular language L is infinite if and only if there exists a word $u \in L$ such that $n \leq |u| < 2n$, where n is the natural number associated to language L, given by the pumping lemma.*

Proof If L is infinite, then it contains words longer than $2n$, and let u be the shortest word longer than $2n$ in L. Because L is regular we can use the pumping lemma, so $u = xyz$, where $|xy| \leq n$, thus $|y| \leq n$ is also true. By the lemma $u' = xz \in L$. But because $|u'| < |u|$ and the shortest word in L longer than $2n$ is u, we get $|u'| < 2n$. From $|y| \leq n$ we get also $|u'| \geq n$.

Conversely, if there exists a word $u \in L$ such that $n \leq |u| < 2n$, then using the pumping lemma, we obtain that $u = xyz$, $|y| \geq 1$ and $xy^iz \in L$ for any i, therefore L is infinite. ∎

Now, the question is: how can we apply the pumping lemma for a finite regular language, since by pumping words we get an infinite number of words? The number of states of a DFA accepting language L is greater than the length of the longest word in L. So, in L there is no word with length at least n, when n is the natural number associated to L by the pumping lemma. Therefore, no word in L can be decomposed in the form xyz, where $|xyz| \geq n$, $|xy| \leq n$, $|y| \geq 1$, and this is why we can not obtain an infinite number of words in L.

1.2.7. Regular expressions

In this subsection we introduce for any alphabet Σ the notion of regular expressions over Σ and the corresponding representing languages. A regular expression is a formula, and the corresponding language is a language over Σ. For example, if $\Sigma = \{a, b\}$, then a^*, b^*, $a^* + b^*$ are regular expressions over Σ which represent respectively languages $\{a\}^*$, $\{b\}^*$, $\{a\}^* \cup \{b\}^*$. The exact definition is the following.

Definition 1.19 *Define recursively a regular expression over Σ and the language it represent.*

- *$\emptyset$ is a regular expression representing the empty language.*
- *ε is a regular expression representing language $\{\varepsilon\}$.*
- *If $a \in \Sigma$, then a is a regular expression representing language $\{a\}$.*
- *If x, y are regular expressions representing languages X and Y respectively, then $(x + y)$, (xy), (x^*) are regular expressions representing languages $X \cup Y$, XY and X^* respectively.*

Regular expression over Σ can be obtained only by using the above rules a finite number of times.

$$
\begin{array}{rcl}
x+y & \equiv & y+x \\
(x+y)+z & \equiv & x+(y+z) \\
(xy)z & \equiv & x(yz) \\
(x+y)z & \equiv & xz+yz \\
x(y+z) & \equiv & xy+xz \\
(x+y)^* \equiv (x^*+y)^* & \equiv & (x+y^*)^* \equiv (x^*+y^*)^* \\
(x+y)^* & \equiv & (x^*y^*)^* \\
(x^*)^* & \equiv & x^* \\
x^*x & \equiv & xx^* \\
xx^*+\varepsilon & \equiv & x^*
\end{array}
$$

Figure 1.18 Properties of regular expressions.

Some brackets can be omitted in the regular expressions if taking into account the priority of operations (iteration, product, union) the corresponding languages are not affected. For example instead of $((x^*)(x+y))$ we can consider $x^*(x+y)$.

Two regular expressions are ***equivalent*** if they represent the same language, that is $x \equiv y$ if $X = Y$, where X and Y are the languages represented by regular expressions x and y respectively. Figure 1.18 shows some equivalent expressions.

We show that to any finite language L can be associated a regular expression x which represent language L. If $L = \emptyset$, then $x = \emptyset$. If $L = \{w_1, w_2, \ldots, w_n\}$, then $x = x_1 + x_2 + \ldots + x_n$, where for any $i = 1, 2, \ldots, n$ expression x_i is a regular expression representing language $\{w_i\}$. This latter can be done by the following rule. If $w_i = \varepsilon$, then $x_i = \varepsilon$, else if $w_i = a_1 a_2 \ldots a_m$, where $m \geq 1$ depends on i, then $x_i = a_1 a_2 \ldots a_m$, where the brackets are omitted.

We prove the theorem of Kleene which refers to the relationship between regular languages and regular expression.

Theorem 1.20 (Kleene's theorem). *Language $L \subseteq \Sigma^*$ is regular if and only if there exists a regular expression over Σ representing language L.*

Proof First we prove that if x is a regular expression, then language L which represents x is also regular. The proof will be done by induction on the construction of expression.

If $x = \emptyset$, $x = \varepsilon$, $x = a, \forall a \in \Sigma$, then $L = \emptyset$, $L = \{\varepsilon\}$, $L = \{a\}$ respectively. Since L is finite in all three cases, it is also regular.

If $x = (x_1 + x_2)$, then $L = L_1 \cup L_2$, where L_1 and L_2 are the languages which represent the regular expressions x_1 and x_2 respectively. By the induction hypothesis languages L_1 and L_2 are regular, so L is also regular because regular languages are closed on union. Cases $x = (x_1 x_2)$ and $x = (x_1^*)$ can be proved by similar way.

Conversely, we prove that if L is a regular language, then a regular expression x can be associated to it, which represent exactly the language L. If L is regular, then there exists a DFA $\mathrm{A} = (Q, \Sigma, E, \{q_0\}, F)$ for which $L = L(\mathrm{A})$. Let $q_0, q_1, \ldots, q_n$ the

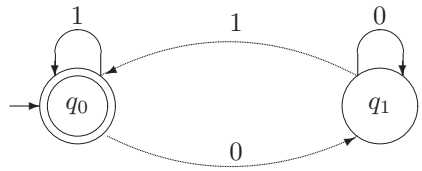


Figure 1.19 DFA from Example 1.20, to which regular expression is associated by Method 1.

states of the automaton A. Define languages R_{ij}^k for all $-1 \leq k \leq n$ and $0 \leq i, j \leq n$. R_{ij}^k is the set of words, for which automaton A goes from state q_i to state q_j without using any state with index greater than k. Using transition graph we can say: a word is in R_{ij}^k, if from state q_i we arrive to state q_j following the edges of the graph, and concatenating the corresponding labels on edges we get exactly that word, not using any state $q_{k+1}, \ldots q_n$. Sets R_{ij}^k can be done also formally:

$$R_{ij}^{-1} = \{a \in \Sigma \mid (q_i, a, q_j) \in E\}, \text{ if } i \neq j,$$
$$R_{ii}^{-1} = \{a \in \Sigma \mid (q_i, a, q_i) \in E\} \cup \{\varepsilon\},$$
$$R_{ij}^k = R_{ij}^{k-1} \cup R_{ik}^{k-1} \left(R_{kk}^{k-1}\right)^* R_{kj}^{k-1} \text{ for all } i, j, k \in \{0, 1, \ldots, n\}.$$

We can prove by induction that sets R_{ij}^k can be described by regular expressions. Indeed, if $k = -1$, then for all i and j languages R_{ij}^k are finite, so they can be expressed by regular expressions representing exactly these languages. Moreover, if for all i and j language R_{ij}^{k-1} can be expressed by regular expression, then language R_{ij}^k can be expressed also by regular expression, which can be corresponding constructed from regular expressions representing languages R_{ij}^{k-1}, R_{ik}^{k-1}, R_{kk}^{k-1} and R_{kj}^{k-1} respectively, using the above formula for R_{ij}^k.

Finally, if $F = \{q_{i_1}, q_{i_2}, \ldots, q_{i_p}\}$ is the set of final states of the DFA A, then $L = L(\mathrm{A}) = R_{0i_1}^n \cup R_{0i_2}^n \cup \ldots \cup R_{0i_p}^n$ can be expressed by a regular expression obtained from expressions representing languages $R_{0i_1}^n, R_{0i_2}^n, \ldots, R_{0i_p}^n$ using operation $+$. ■

Further on we give some procedures which associate DFA to regular expressions and conversely regular expression to DFA.

Associating regular expressions to finite automata. We present here three methods, each of which associate to a DFA the corresponding regular expression.

Method 1. Using the result of the theorem of Kleene, we will construct the sets R_{ij}^k, and write a regular expression which represents the language $L = R_{0i_1}^n \cup R_{0i_2}^n \cup \ldots \cup R_{0i_p}^n$, where $F = \{q_{i_1}, q_{i_2}, \ldots, q_{i_p}\}$ is the set of final states of the automaton.

Example 1.20 Consider the DFA in Fig. 1.19.

$L(\mathrm{A}) = R_{00}^1 = R_{00}^0 \cup R_{01}^0 \left(R_{11}^0\right)^* R_{10}^0$

R_{00}^0 : $1^* + \varepsilon \equiv 1^*$

R_{01}^0 : 1^*0

R_{11}^0 : $11^*0 + \varepsilon + 0 \equiv (11^* + \varepsilon)0 + \varepsilon \equiv 1^*0 + \varepsilon$

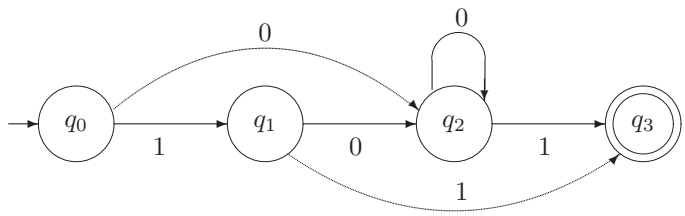


Figure 1.20 DFA in Example 1.21 to which a regular expression is associated by Method 1. The computation are in Figure 1.21.

R^0_{10} : 11^*

Then the regular expression corresponding to $L(\mathrm{A})$ is $1^* + 1^*0(1^*0 + \varepsilon)^*11^* \equiv 1^* + 1^*0(1^*0)^*11^*$.

Example 1.21 Find a regular expression associated to DFA in Fig. 1.20. The computations are in Figure 1.21. The regular expression corresponding to R^3_{03} is $11 + (0 + 10)0^*1$.

Method 2. Now we generalize the notion of finite automaton, considering words instead of letters as labels of edges. In such an automaton each walk determine a regular expression, which determine a regular language. The regular language accepted by a generalized finite automaton is the union of regular languages determined by the productive walks. It is easy to see that the generalized finite automata accept regular languages.

The advantage of generalized finite automata is that the number of its edges can be diminuted by equivalent transformations, which do not change the accepted language, and leads to a graph with only one edge which label is exactly the accepted language.

The possible equivalent transformations can be seen in Fig. 1.22. If some of the vertices 1, 2, 4, 5 on the figure coincide, in the result they are merged, and a loop will arrive.

First, the automaton is transformed by corresponding ε-moves to have only one initial and one final state. Then, applying the equivalent transformations until the graph will have only one edge, we will obtain as the label of this edge the regular expression associated to the automaton.

Example 1.22 In the case of Fig. 1.19 the result is obtained by steps illustrated in Fig. 1.23. This result is $(1+00^*1)^*$, which represents the same language as obtained by Method 1 (See example 1.20).

Example 1.23 In the case of Fig. 1.20 is not necessary to introduce new initial and final state. The steps of transformations can be seen in Fig. 1.24. The resulted regular expression can be written also as $(0 + 10)0^*1 + 11$, which is the same as obtained by the

	$k=-1$	$k=0$	$k=1$	$k=2$	$k=3$
R_{00}^k	ε	ε	ε	ε	
R_{01}^k	1	1	1	1	
R_{02}^k	0	0	$0+10$	$(0+10)0^*$	
R_{03}^k	$\emptyset$	$\emptyset$	11	$11+(0+10)0^*1$	$11+(0+10)0^*1$
R_{11}^k	ε	ε	ε	ε	
R_{12}^k	0	0	0	00^*	
R_{13}^k	1	1	1	$1+00^*1$	
R_{22}^k	$0+\varepsilon$	$0+\varepsilon$	$0+\varepsilon$	0^*	
R_{23}^k	1	1	1	0^*1	
R_{33}^k	ε	ε	ε	ε	

Figure 1.21 Determining a regular expression associated to DFA in Figure 1.20 using sets R_{ij}^k.

previous method.

Method 3. The third method for writing regular expressions associated to finite automata uses formal equations. A variable X is associated to each state of the automaton (to different states different variables). Associate to each state an equation which left side contains X, its right side contains sum of terms of form Ya or ε, where Y is a variable associated to a state, and a is its corresponding input symbol. If there is no incoming edge in the state corresponding to X then the right side of the equation with left side X contains ε, otherwise is the sum of all terms of the form Ya for which there is a transition labelled with letter a from state corresponding to Y to the state corresponding to X. If the state corresponding to X is also an initial

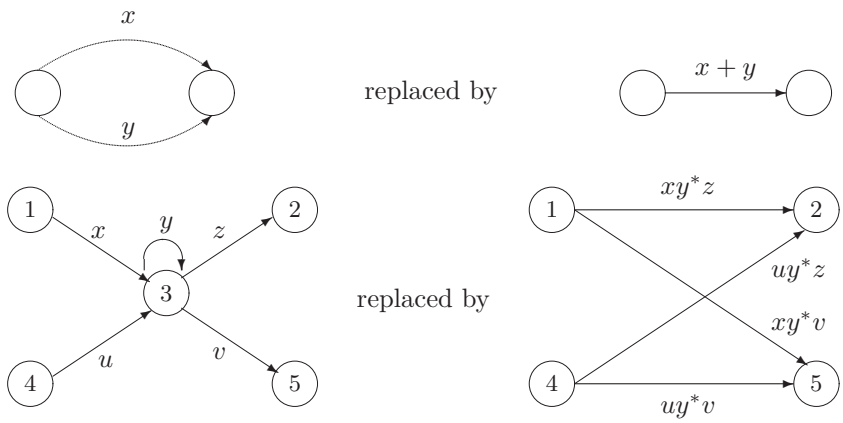


Figure 1.22 Possible equivalent transformations for finding regular expression associated to an automaton.

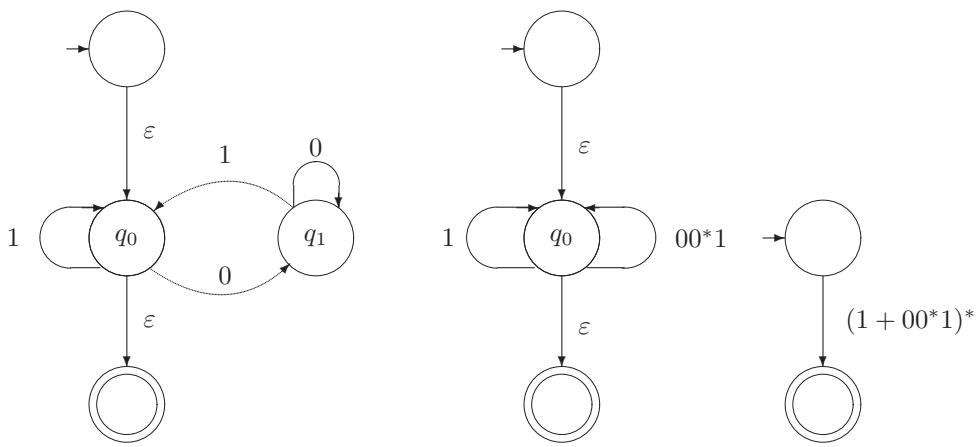


Figure 1.23 Transformation of the finite automaton in Fig. 1.19.

and a final state, then on right side of the equation with the left side X will be also a term equal to ε. For example in the case of Fig. 1.20 let these variable X, Y, Z, U corresponding to the states q_0, q_1, q_2, q_3. The corresponding equation are

$X = \varepsilon$
$Y = X1$
$Z = X0 + Y0 + Z0$
$U = Y1 + Z1.$

If an equation is of the form $X = X\alpha + \beta$, where α, β are arbitrary words not containing X, then it is easy to see by a simple substitution that $X = \beta\alpha^*$ is a solution of the equation.

Because these equations are linear, all of them can be written in the form $X = X\alpha + \beta$ or $X = X\alpha$, where α do not contain any variable. Substituting this in the

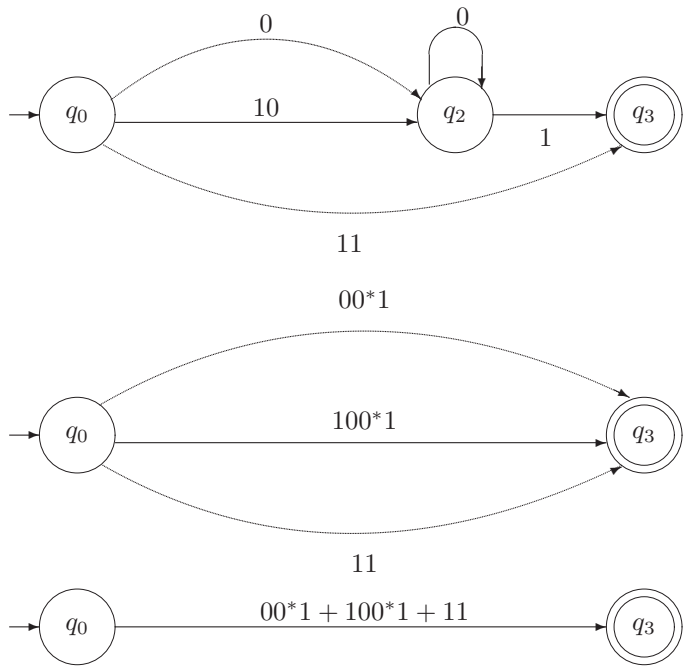


Figure 1.24 Steps of Example 1.23.

other equations the number of remaining equations will be diminuted by one. In such a way the system of equation can be solved for each variable.

The solution will be given by variables corresponding to final states summing the corresponding regular expressions.

In our example from the first equation we get $Y = 1$. From here $Z = 0 + 10 + Z0$, or $Z = Z0 + (0 + 10)$, and solving this we get $Z = (0 + 10)0^*$. Variable U can be obtained immediately and we obtain $U = 11 + (0 + 10)0^*1$.

Using this method in the case of Fig. 1.19, the following equations will be obtained

$X = \varepsilon + X1 + Y1$

$Y = X0 + Y0$

Therefore

$X = \varepsilon + (X + Y)1$

$Y = (X + Y)0$.

Adding the two equations we will obtain

$X + Y = \varepsilon + (X + Y)(0 + 1)$, from where (considering ε as β and $(0 + 1)$ as α) we get the result

$X + Y = (0 + 1)^*$.

From here the value of X after the substitution is

$X = \varepsilon + (0 + 1)^*1$,

which is equivalent to the expression obtained using the other methods.

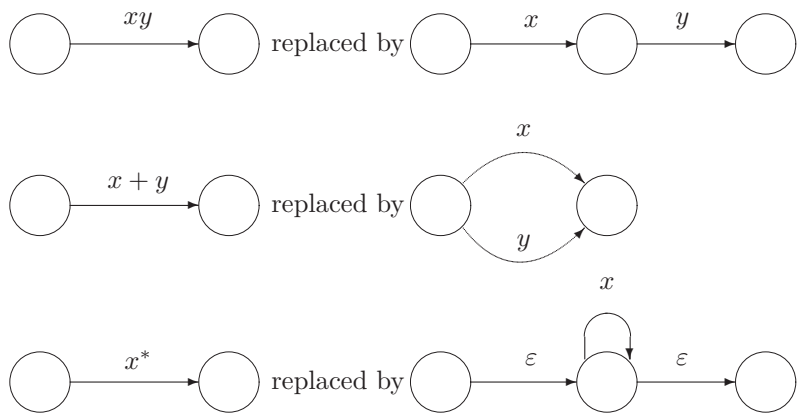
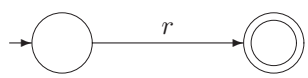


Figure 1.25 Possible transformations to obtain finite automaton associated to a regular expression.

Associating finite automata to regular expressions. Associate to the regular expression r a generalized finite automaton:



After this, use the transformations in Fig. 1.25 step by step, until an automaton with labels equal to letters from Σ or ε will be obtained.

Example 1.24 Get started from regular expression $\varepsilon + (0+1)^*1$. The steps of transformations are in Fig. 1.26(a)-(e). The last finite automaton (see Fig. 1.26(e)) can be done in a simpler form as can be seen in Fig. 1.26(f). After eliminating the ε-moves and transforming in a deterministic finite automaton the DFA in Fig. 1.27 will be obtained, which is equivalent to DFA in Fig. 1.19.

Exercises

1.2-1 Give a DFA which accepts natural numbers divisible by 9.

1.2-2 Give a DFA which accepts the language containing all words formed by

a. an even number of 0's and an even number of 1's,

b. an even number of 0's and an odd number of 1's,

c. an odd number of 0's and an even number of 1's,

d. an odd number of 0's and an odd number of 1's.

1.2-3 Give a DFA to accept respectively the following languages:

$L_1 = \{a^n b^m \mid n \geq 1, m \geq 0\}$, $L_2 = \{a^n b^m \mid n \geq 1, m \geq 1\}$,

$L_3 = \{a^n b^m \mid n \geq 0, m \geq 0\}$, $L_4 = \{a^n b^m \mid n \geq 0, m \geq 1\}$.

1.2-4 Give an NFA which accepts words containing at least two 0's and any number of 1's. Give an equivalent DFA.

1.2-5 Minimize the DFA's in Fig. 1.28.

1.2-6 Show that the DFA in 1.29.(a) is a minimum state automaton.

1.2-7 Transform NFA in Fig. 1.29.(b) in a DFA, and after this minimize it.

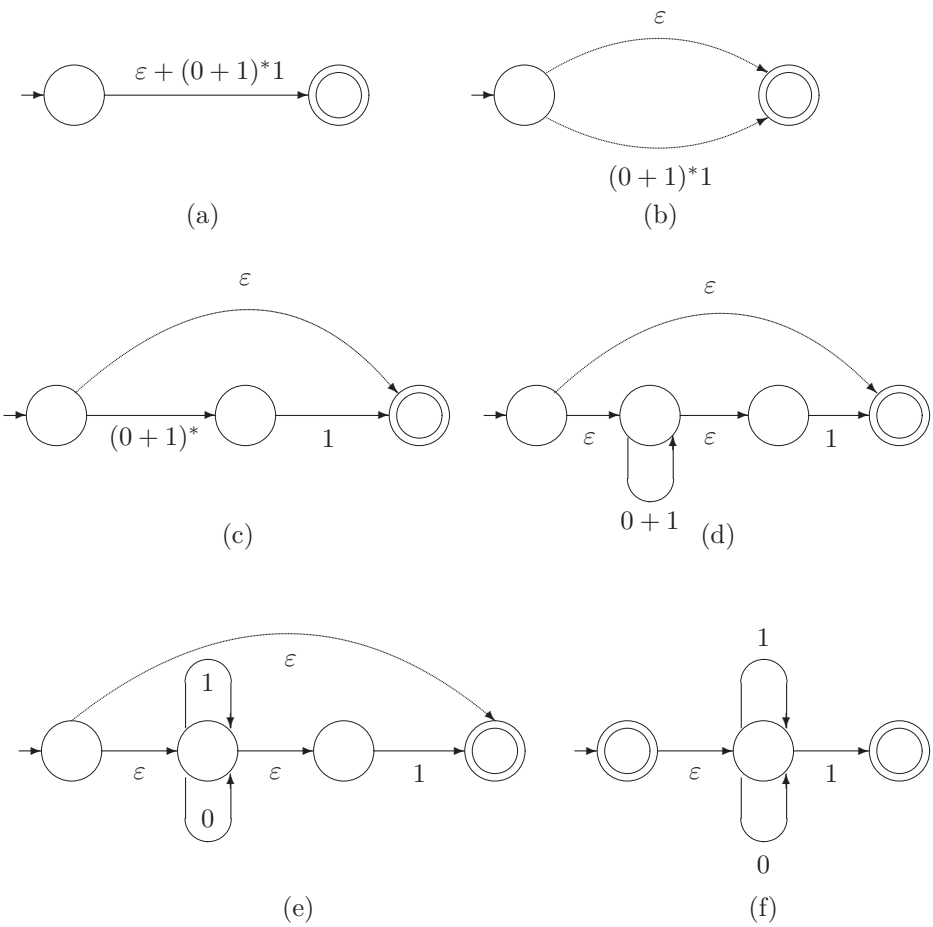


Figure 1.26 Associating finite automaton to regular expression $\varepsilon + (0+1)^*1$.

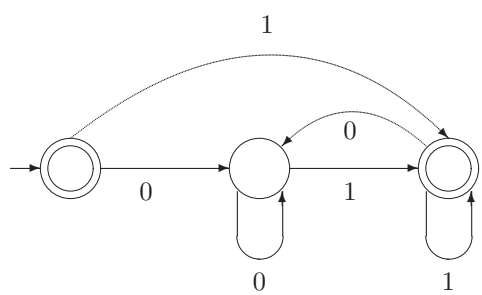


Figure 1.27 Finite automaton associated to regular expression $\varepsilon + (0+1)^*1$.

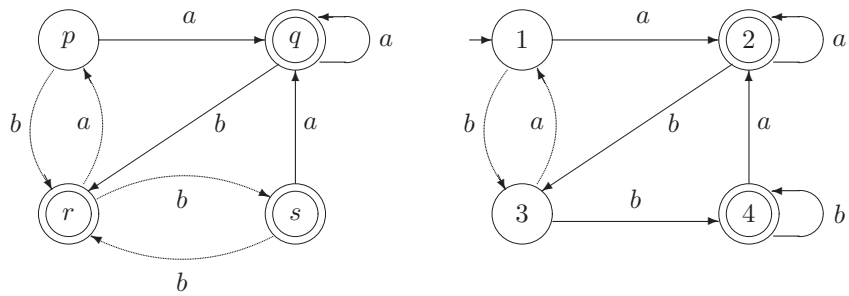


Figure 1.28 DFA's to minimize for Exercise 1.2-5

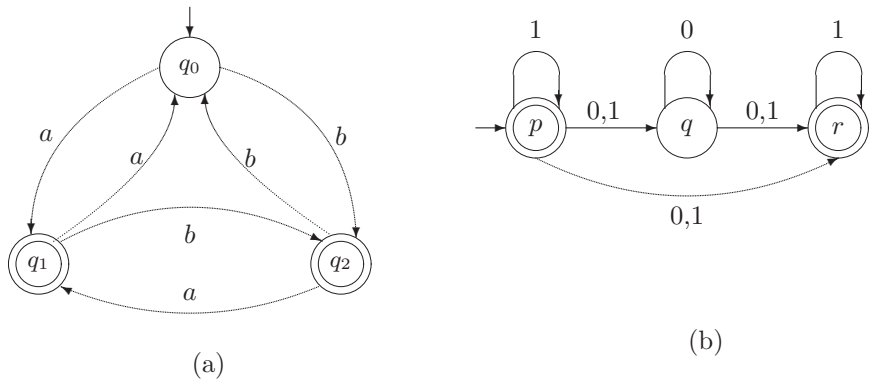


Figure 1.29 Finite automata for Exercises 1.2-6 and 1.2-7.

1.2-8 Define finite automaton A_1 which accepts all words of the form $0(10)^n$ $(n \geq 0)$, and finite automaton A_2 which accepts all words of the form $1(01)^n$ $(n \geq 0)$. Define the union automaton $A_1 \cup A_2$, and then eliminate the ε-moves.

1.2-9 Associate to DFA in Fig. 1.30 a regular expression.

1.2-10 Associate to regular expression $ab^*ba^* + b + ba^*a$ a DFA.

1.2-11 Prove, using the pumping lemma, that none of the following languages are regular:

$$L_1 = \{a^n c b^n \mid n \geq 0\}, \qquad L_2 = \{a^n b^n a^n \mid n \geq 0\}, \qquad L_3 = \{a^p \mid p \text{ prím}\}.$$

1.2-12 Prove that if L is a regular language, then $\{u^{-1} \mid u \in L\}$ is also regular.

1.2-13 Prove that if $L \subseteq \Sigma^*$ is a regular language, then the following languages are

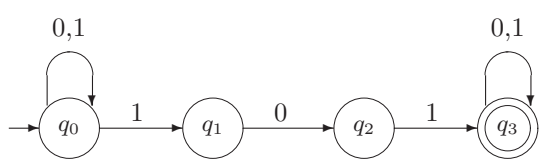


Figure 1.30 DFA for Exercise 1.2-9.

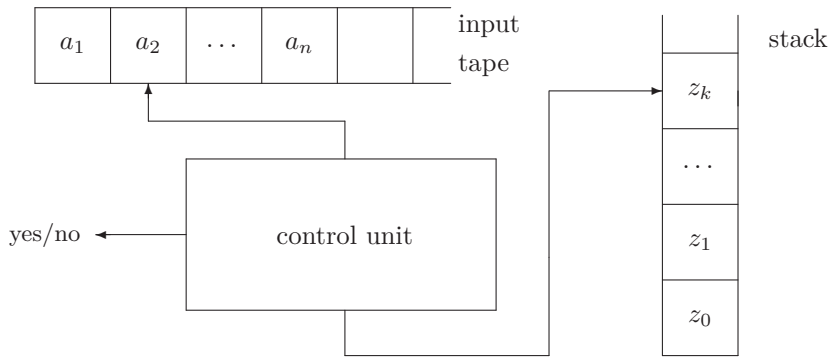


Figure 1.31 Pushdown automaton.

also regular.

$\text{pre}(L) = \{w \in \Sigma^* \mid \exists u \in \Sigma^*, wu \in L\}$, $\text{suf}(L) = \{w \in \Sigma^* \mid \exists u \in \Sigma^*, uw \in L\}$.

1.2-14 Show that the following languages are all regular.

$L_1 = \{ab^n cd^m \mid n > 0, m > 0\}$,
$L_2 = \{(ab)^n \mid n \geq 0\}$,
$L_3 = \{a^{kn} \mid n \geq 0, \ \ k \text{ constant}\}$.

1.3. Pushdown automata and context-free languages

In this section we deal with the pushdown automata and the class of languages — the context-free languages — accepted by them.

As we have been seen in Section 1.1, a context-free grammar $G = (N, T, P, S)$ is one with the productions of the form $A \to \beta$, $A \in N$, $\beta \in (N \cup T)^+$. The production $S \to \varepsilon$ is also permitted if S does not appear in right hand side of any productions. Language $L(G) = \{u \in T^* \mid S \overset{*}{\underset{G}{\Longrightarrow}} u\}$ is the context-free language generated by grammar G.

1.3.1. Pushdown automata

We have been seen that finite automata accept the class of regular languages. Now we get to know a new kind of automata, the so-called ***pushdown automata,*** which accept context-free languages. The pushdown automata differ from finite automata mainly in that to have the possibility to change states without reading any input symbol (i.e. to read the empty symbol) and possess a stack memory, which uses the so-called stack symbols (See Fig. 1.31).

The pushdown automaton get a word as input, start to function from an initial state having in the stack a special symbol, the initial stack symbol. While working, the pushdown automaton change its state based on current state, next input symbol

(or empty word) and stack top symbol and replace the top symbol in the stack with a (possibly empty) word.

There are two type of acceptances. The pushdown automaton accepts a word by final state when after reading it the automaton enter a final state. The pushdown automaton accepts a word by empty stack when after reading it the automaton empties its stack. We show that these two acceptances are equivalent.

Definition 1.21 *A **nondeterministic pushdown automaton** is a system*

$$\mathrm{V} = (Q, \Sigma, W, E, q_0, z_0, F) \ ,$$

where

- Q *is the finite, non-empty set of* ***states***
- Σ *is the* ***input alphabet,***
- W *is the* ***stack alphabet,***
- $E \subseteq Q \times \big(\Sigma \cup \{\varepsilon\}\big) \times W \times W^* \times Q$ *is the set of* ***transitions*** *or* ***edges****,*
- $q_0 \in Q$ *is the* ***initial state,***
- $z_0 \in W$ *is the* ***start symbol of stack,***
- $F \subseteq Q$ *is the set of* ***final states****.*

A transition (p, a, z, w, q) means that if pushdown automaton V is in state p, reads from the input tape letter a (instead of input letter we can also consider the empty word ε), and the top symbol in the stack is z, then the pushdown automaton enters state q and replaces in the stack z by word w. Writing word w in the stack is made by natural order (letters of word w will be put in the stack letter by letter from left to right). Instead of writing transition (p, a, z, w, q) we will use a more suggestive notation $\big(p, (a, z/w), q\big)$.

Here, as in the case of finite automata, we can define a transition function

$$\delta \ : \ Q \times (\Sigma \cup \{\varepsilon\}) \times W \to \mathcal{P}(W^* \times Q) \ ,$$

which associate to current state, input letter and top letter in stack pairs of the form (w, q), where $w \in W^*$ is the word written in stack and $q \in Q$ the new state.

Because the pushdown automaton is nondeterministic, we will have for the transition function

$\delta(q, a, z) = \{(w_1, p_1), \ldots, (w_k, p_k)\}$ (if the pushdown automaton reads an input letter and moves to right), or

$\delta(q, \varepsilon, z) = \{(w_1, p_1), \ldots, (w_k, p_k)\}$ (without move on the input tape).

A pushdown automaton is ***deterministic,*** if for any $q \in Q$ and $z \in W$ we have

- $|\delta(q, a, z)| \leq 1$, $\forall a \in \Sigma \cup \{\varepsilon\}$ and
- if $\delta(q, \varepsilon, z) \neq \emptyset$, then $\delta(q, a, z) = \emptyset$, $\forall a \in \Sigma$.

We can associate to any pushdown automaton a transition table, exactly as in the case of finite automata. The rows of this table are indexed by elements of Q, the columns by elements from $\Sigma \cup \{\varepsilon\}$ and W (to each $a \in \Sigma \cup \{\varepsilon\}$ and $z \in W$ will correspond a column). At intersection of row corresponding to state $q \in Q$ and column corresponding to $a \in \Sigma \cup \{\varepsilon\}$ and $z \in W$ we will have pairs $(w_1, p_1), \ldots, (w_k, p_k)$ if $\delta(q, a, z) = \{(w_1, p_1), \ldots, (w_k, p_k)\}$.

The transition graph, in which the label of edge (p, q) will be $(a, z/w)$ corresponding to transition $\big(p, (a, z/w), q\big)$, can be also defined.

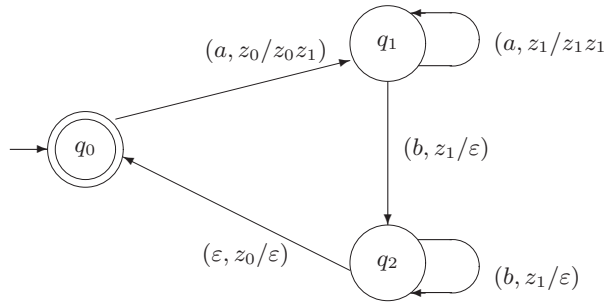


Figure 1.32 Example of pushdown automaton.

Example 1.25 $\mathrm{V}_1 = (\{q_0, q_1, q_2\}, \{a, b\}, \{z_0, z_1\}, E, q_0, z_0, \{q_0\})$. Elements of E are:

$$\begin{array}{ll} \big(q_0, (a, z_0/z_0z_1), q_1\big) & \\ \big(q_1, (a, z_1/z_1z_1), q_1\big) & \big(q_1, (b, z_1/\varepsilon), q_2\big) \\ \big(q_2, (b, z_1/\varepsilon), q_2\big) & \big(q_2, (\varepsilon, z_0/\varepsilon), q_0\big)\ . \end{array}$$

The transition function:

$$\begin{array}{ll} \delta(q_0, a, z_0) = \{(z_0z_1, q_1)\} & \\ \delta(q_1, a, z_1) = \{(z_1z_1, q_1)\} & \delta(q_1, b, z_1) = \{(\varepsilon, q_2)\} \\ \delta(q_2, b, z_1) = \{(\varepsilon, q_2)\} & \delta(q_2, \varepsilon, z_0) = \{(\varepsilon, q_0)\}\ . \end{array}$$

The transition table:

$\Sigma \cup \{\varepsilon\}$	a		b	ε
W	z_0	z_1	z_1	z_0
q_0	(z_0z_1, q_1)			
q_1		(z_1z_1, q_1)	(ε, q_2)	
q_2			(ε, q_2)	(ε, q_0)

Because for the transition function every set which is not empty contains only one element (e.g. $\delta(q_0, a, z_0) = \{(z_0z_1, q_1)\}$), in the above table each cell contains only one element, And the set notation is not used. Generally, if a set has more than one element, then its elements are written one under other. The transition graph of this pushdown automaton is in Fig. 1.32.

The current state, the unread part of the input word and the content of stack

constitutes a ***configuration*** of the pushdown automaton, i.e. for each $q \in Q$, $u \in \Sigma^*$ and $v \in W^*$ the triplet (q, u, v) can be a configuration.

If $u = a_1 a_2 \dots a_k$ and $v = x_1 x_2 \dots x_m$, then the pushdown automaton can change its configuration in two ways:

- $(q,\ a_1 a_2 \dots a_k,\ x_1 x_2 \dots x_{m-1} x_m) \Longrightarrow (p,\ a_2 a_3 \dots a_k,\ x_1, x_2 \dots x_{m-1} w)$, if $\big(q, (a_1, x_m/w), p\big) \in E$
- $(q,\ a_1 a_2 \dots a_k,\ x_1 x_2 \dots x_m) \Longrightarrow (p,\ a_1 a_2 \dots a_k,\ x_1, x_2 \dots x_{m-1} w)$, if $\big(q, (\varepsilon, x_m/w), p\big) \in E$.

The reflexive and transitive closure of the relation $\Longrightarrow$ will be denoted by $\overset{*}{\Longrightarrow}$. Instead of using $\Longrightarrow$, sometimes $\vdash$ is considered.

How does work such a pushdown automaton? Getting started with the initial configuration $(q_0, a_1 a_2 \dots a_n, z_0)$ we will consider all possible next configurations, and after this the next configurations to these next configurations, and so on, until it is possible.

Definition 1.22 *Pushdown automaton* V *accepts (recognizes) word* u ***by final state*** *if there exist a sequence of configurations of* V *for which the following are true:*

- *the first element of the sequence is* (q_0, u, z_0),
- *there is a going from each element of the sequence to the next element, excepting the case when the sequence has only one element,*
- *the last element of the sequence is* (p, ε, w), *where* $p \in F$ *and* $w \in W^*$.

Therefore pushdown automaton V accepts word u by final state, if and only if $(q_0, u, z_0) \overset{*}{\Longrightarrow} (p, \varepsilon, w)$ for some $w \in W^*$ and $p \in F$. The set of words accepted by final state by pushdown automaton V will be called the language accepted by V by final state and will be denoted by $L(\mathrm{V})$.

Definition 1.23 *Pushdown automaton* V *accepts (recognizes) word* u ***by empty stack*** *if there exist a sequence of configurations of* V *for which the following are true:*

- *the first element of the sequence is* (q_0, u, z_0),
- *there is a going from each element of the sequence to the next element,*
- *the last element of the sequence is* $(p, \varepsilon, \varepsilon)$ *and* p *is an arbitrary state.*

Therefore pushdown automaton V accepts a word u by empty stack if $(q_0, u, z_0) \overset{*}{\Longrightarrow} (p, \varepsilon, \varepsilon)$ for some $p \in Q$. The set of words accepted by empty stack by pushdown automaton V will be called the language accepted by empty stack by V and will be denoted by $L_\varepsilon(\mathrm{V})$.

Example 1.26 Pushdown automaton V_1 of Example 1.25 accepts the language $\{a^n b^n \mid n \geq 0\}$ by final state. Consider the derivation for words *aaabbb* and *abab*.

Word $a^3 b^3$ is accepted by the considered pushdown automaton because
$(q_0, aaabbb, z_0) \Longrightarrow (q_1, aabbb, z_0 z_1) \Longrightarrow (q_1, abbb, z_0 z_1 z_1) \Longrightarrow (q_1, bbb, z_0 z_1 z_1 z_1)$
$\Longrightarrow (q_2, bb, z_0 z_1 z_1) \Longrightarrow (q_2, b, z_0 z_1) \Longrightarrow (q_2, \varepsilon, z_0) \Longrightarrow (q_0, \varepsilon, \varepsilon)$ and because q_0 is a final state the pushdown automaton accepts this word. But the stack being empty, it accepts this word also by empty stack.

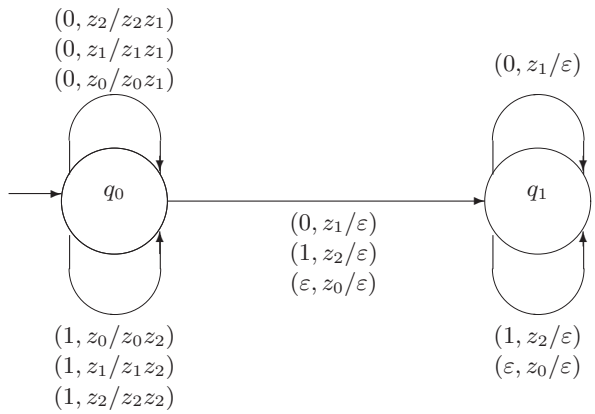


Figure 1.33 Transition graph of the Example 1.27

Because the initial state is also a final state, the empty word is accepted by final state, but not by empty stack.

To show that word $abab$ is not accepted, we need to study all possibilities. It is easy to see that in our case there is only a single possibility:

$(q_0, abab, z_0) \Longrightarrow (q_1, bab, z_0z_1) \Longrightarrow (q_2, ab, z_0) \Longrightarrow (q_0, ab, \varepsilon)$, but there is no further going, so word $abab$ is not accepted.

Example 1.27 The transition table of the pushdown automaton $\mathrm{V}_2 = (\{q_0, q_1\}, \{0, 1\}, \{z_0, z_1, z_2\}, E, q_0, z_0, \emptyset)$ is:

$\Sigma \cup \{\varepsilon\}$	0			1			ε
W	z_0	z_1	z_2	z_0	z_1	z_2	z_0
q_0	(z_0z_1, q_0)	(z_1z_1, q_0) (ε, q_1)	(z_2z_1, q_0)	(z_0z_2, q_0)	(z_1z_2, q_0)	(z_2z_2, q_0) (ε, q_1)	(ε, q_1)
q_1		(ε, q_1)				(ε, q_1)	(ε, q_1)

The corresponding transition graph can be seen in Fig. 1.33. Pushdown automaton V_2 accepts the language $\left\{uu^{-1} \mid u \in \{0,1\}^*\right\}$. Because V_2 is nemdeterministic, all the configurations obtained from the initial configuration (q_0, u, z_0) can be illustrated by a ***computation tree***. For example the computation tree associated to the initial configuration $(q_0, 1001, z_0)$ can be seen in Fig. 1.34. From this computation tree we can observe that, because $(q_1, \varepsilon, \varepsilon)$ is a leaf of the tree, pushdown automaton V_2 accepts word 1001 by empty stack. The computation tree in Fig. 1.35 shows that pushdown automaton V_2 does not accept word 101, because the configurations in leaves can not be continued and none of them has the form $(q, \varepsilon, \varepsilon)$.

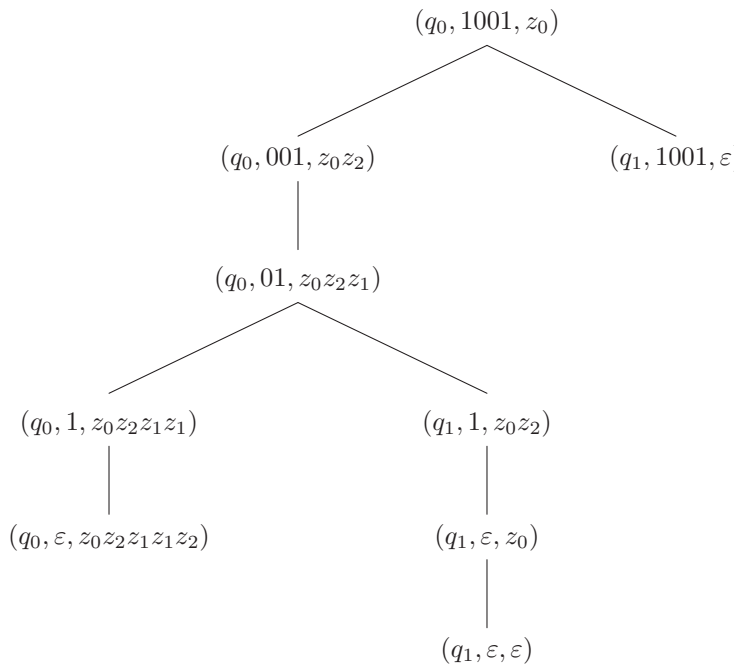


Figure 1.34 Computation tree to show acceptance of the word 1001 (see Example 1.27).

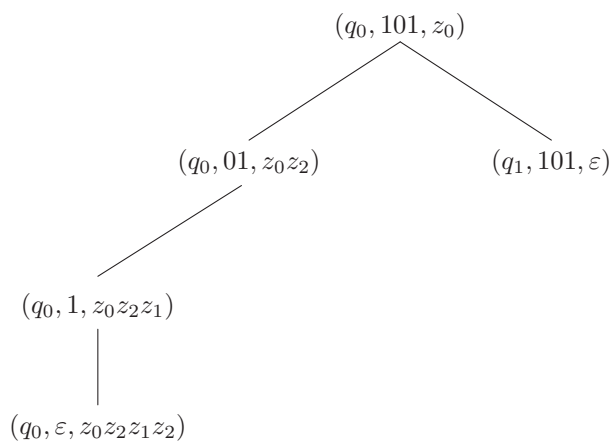


Figure 1.35 Computation tree to show that the pushdown automaton in Example 1.27 does not accept word 101.

Theorem 1.24 *A language L is accepted by a nondeterministic pushdown automaton V_1 by empty stack if and only if it can be accepted by a nondeterministic pushdown automaton V_2 by final state.*

Proof *a)* Let $\mathrm{V}_1 = (Q, \Sigma, W, E, q_0, z_0, \emptyset)$ be the pushdown automaton which accepts by empty stack language L. Define pushdown automaton $\mathrm{V}_2 = (Q \cup \{p_0, p\}, \Sigma, W \cup \{x\}, E', p_0, x, \{p\})$, where $p, p_0 \notin Q$, , $x \notin W$ and

$$E' = E \cup \Big\{\big(p_0, (\varepsilon, x/xz_0), q_0\big)\Big\} \cup \Big\{\big(q, (\varepsilon, x/\varepsilon), p\big) \mid q \in Q\Big\}.$$

Working of V_2: Pushdown automaton V_2 with an ε-move first goes in the initial state of V_1, writing z_0 (the initial stack symbol of V_1) in the stack (beside x). After this it is working as V_1. If V_1 for a given word empties its stack, then V_2 still has x in the stack, which can be deleted by V_2 using an ε-move, while a final state will be reached. V_2 can reach a final state only if V_1 has emptied the stack.

b) Let $\mathrm{V}_2 = (Q, \Sigma, W, E, q_0, z_0, F)$ be a pushdown automaton, which accepts language L by final state. Define pushdown automaton $\mathrm{V}_1 = (Q \cup \{p_0, p\}, \Sigma, W \cup \{x\}, E', p_0, x, \emptyset)$, where $p_0, p \notin Q$, $x \notin W$ and

$$E' = E \cup \Big\{\big(p_0, (\varepsilon, x/xz_0), q_0\big)\Big\} \cup \Big\{\big(q, (\varepsilon, z/\varepsilon), p\big) \mid q \in F, p \in Q, z \in W\Big\} \cup \Big\{\big(p, (\varepsilon, z/\varepsilon), p\big) \mid p \in Q, z \in W \cup \{x\}\Big\}$$

Working V_1: Pushdown automaton V_1 with an ε-move writes in the stack beside x the initial stack symbol z_0 of V_2, then works as V_2, i.e reaches a final state for each accepted word. After this V_1 empties the stack by an ε-move. V_1 can empty the stack only if V_2 goes in a final state. ∎

The next two theorems prove that the class of languages accepted by nondeterministic pushdown automata is just the set of context-free languages.

Theorem 1.25 *If G is a context-free grammar, then there exists such a nondeterministic pushdown automaton* V *which accepts $L(G)$ by empty stack, i.e. $L_\varepsilon(\mathrm{V}) = L(G)$.*

We outline the proof only. Let $G = (N, T, P, S)$ be a context-free grammar. Define pushdown automaton $\mathrm{V} = (\{q\}, T, N \cup T, E, q, S, \emptyset)$, where $q \notin N \cup T$, and the set E of transitions is:

• If there is in the set of productions of G a production of type $A \to \alpha$, then let put in E the transition $\big(q, (\varepsilon, A/\alpha^{-1}), q\big)$,

• For any letter $a \in T$ let put in E the transition $\big(q, (a, a/\varepsilon), q\big)$.

If there is a production $S \to \alpha$ in G, the pushdown automaton put in the stack the mirror of α with an ε-move. If the input letter coincides with that in the top of the stack, then the automaton deletes it from the stack. If in the top of the stack there is a nonterminal A, then the mirror of right-hand side of a production which has A in its left-hand side will be put in the stack. If after reading all letters of the input word, the stack will be empty, then the pushdown automaton recognized the input word.

The following algorithm builds for a context-free grammar $G = (N, T, P, S)$ the pushdown automaton $\mathrm{V} = (\{q\}, T, N \cup T, E, q, S, \emptyset)$, which accepts by empty stack the language generated by G.

FROM-CFG-TO-PUSHDOWN-AUTOMATON(G)

1 **for** all production $A \to \alpha$
  do put in E the transition $\big(q, (\varepsilon, A/\alpha^{-1}), q\big)$
2 **for** all terminal $a \in T$
  do put in E the transition $\big(q, (a, a/\varepsilon), q\big)$
3 **return** V

If G has n productions and m terminals, then the number of step of the algorithm is $\Theta(n+m)$.

Example 1.28 Let $G = (\{S, A\}, \{a, b\}, \{S \to \varepsilon,\ S \to ab,\ S \to aAb,\ A \to aAb,\ A \to ab\}, S)$. Then $\mathrm{V} = (\{q\}, \{a, b\}, \{a, b, A, S\}, E, q, S, \emptyset)$, with the following transition table.

$\Sigma \cup \{\varepsilon\}$	a	b	ε	
W	a	b	S	A
q	(ε, q)	(ε, q)	(ε, q) (ba, q) (bAa, q)	(bAa, q) (ba, q)

Let us see how pushdown automaton V accepts word $aabb$, which in grammar G can be derived in the following way:

$$S \Longrightarrow aAb \Longrightarrow aabb\ ,$$

where productions $S \to aAb$ and $A \to ab$ were used. Word is accepted by empty stack (see Fig. 1.36).

Theorem 1.26 *For a nondeterministic pushdown automaton* V *there exists always a context-free grammar* G *such that* V *accepts language* $L(G)$ *by empty stack, i.e.* $L_\varepsilon(\mathrm{V}) = L(G)$.

Instead of a proof we will give a method to obtain grammar G. Let $\mathrm{V} = (Q, \Sigma, W, E, q_0, z_0, \emptyset)$ be the nondeterministic pushdown automaton in question.

Then $G = (N, T, P, S)$, where
$N = \{S\} \cup \{S_{p,z,q} \mid p, q \in Q, z \in W\}$ and $T = \Sigma$.
Productions in P will be obtained as follows.

• For all state q put in P production $S \to S_{q_0,z_0,q}$.

• If $\big(q, (a, z/z_k \dots z_2 z_1), p\big) \in E$, where $q \in Q$, $z, z_1, z_2, \dots z_k \in W$ $(k \geq 1)$ and $a \in \Sigma \cup \{\varepsilon\}$, put in P for all possible states $p_1, p_2, \dots, p_k$ productions
$S_{q,z,p_k} \to a S_{p,z_1,p_1} S_{p_1,z_2,p_2} \dots S_{p_{k-1},z_k,p_k}$.

• If $\big(q, (a, z/\varepsilon), p\big) \in E$, where $p, q \in Q, z \in W$, and $a \in \Sigma \cup \{\varepsilon\}$, put in P production
$S_{q,z,p} \to a$.

The context-free grammar defined by this is an extended one, to which an

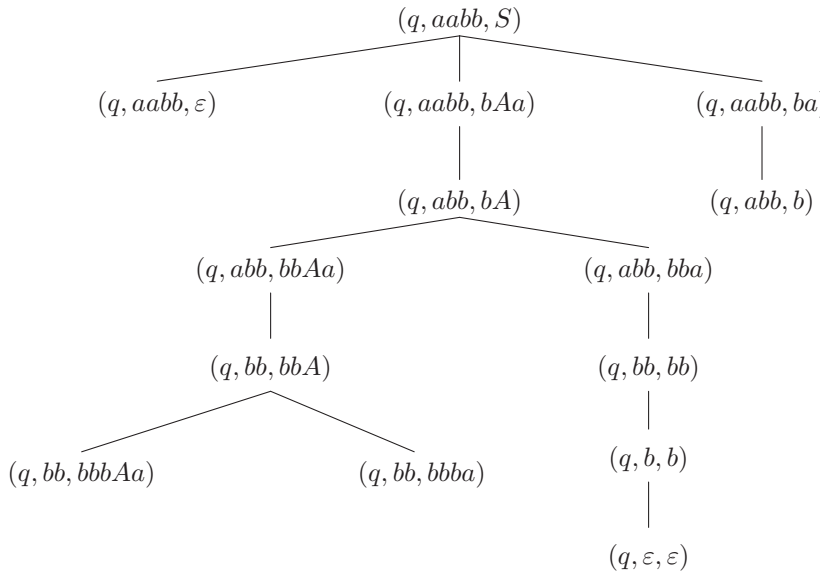


Figure 1.36 Recognising a word by empty stack (see Example 1.28).

equivalent context-free language can be associated. The proof of the theorem is based on the fact that to every sequence of configurations, by which the pushdown automaton V accepts a word, we can associate a derivation in grammar G. This derivation generates just the word in question, because of productions of the form $S_{q,z,p_k} \to aS_{p,z_1,p_1}S_{p_1,z_2,p_2}\dots S_{p_{k-1},z_k,p_k}$, which were defined for all possible states $p_1, p_2, \dots, p_k$. In Example 1.27 we show how can be associated a derivation to a sequence of configurations. The pushdown automaton defined in the example recognizes word 00 by the sequence of configurations

$$(q_0, 00, z_0) \Longrightarrow (q_0, 0, z_0z_1) \Longrightarrow (q_1, \varepsilon, z_0) \Longrightarrow (q_1, \varepsilon, \varepsilon),$$

which sequence is based on the transitions

$$\big(q_0, (0, z_0/z_0z_1), q_0\big),$$
$$\big(q_0, (0, z_1/\varepsilon), q_1\big),$$
$$\big(q_1, (\varepsilon, z_1/\varepsilon), q_1\big).$$

To these transitions, by the definition of grammar G, the following productions can be associated

(1) $S_{q_0,z_0,p_2} \longrightarrow 0S_{q_0,z_1,p_1}S_{p_1,z_0,p_2}$ for all states $p_1, p_2 \in Q$,
(2) $S_{q_0,z_1,q_1} \longrightarrow 0$,
(3) $S_{q_1,z_0,q_1} \longrightarrow \varepsilon$.

Furthermore, for each state q productions $S \longrightarrow S_{q_0,z_0,q}$ were defined.

By the existence of production $S \longrightarrow S_{q_0,z_0,q}$ there exists the derivation $S \Longrightarrow S_{q_0,z_0,q}$, where q can be chosen arbitrarily. Let choose in above production (1) state q to be equal to p_2. Then there exists also the derivation

$$S \Longrightarrow S_{q_0,z_0,q} \Longrightarrow 0S_{q_0,z_1,p_1}S_{p_1,z_0,q},$$

where $p_1 \in Q$ can be chosen arbitrarily. If $p_1 = q_1$, then the derivation

$$S \Longrightarrow S_{q_0,z_0,q} \Longrightarrow 0S_{q_0,z_1,q_1}S_{q_1,z_0,q} \Longrightarrow 00S_{q_1,z_0,q}$$

will result. Now let q equal to q_1, then

$S \Longrightarrow S_{q_0,z_0,q_1} \Longrightarrow 0S_{q_0,z_1,q_1}S_{q_1,z_0,q_1} \Longrightarrow 00S_{q_1,z_0,q_1} \Longrightarrow 00,$

which proves that word 00 can be derived used the above grammar.

The next algorithm builds for a pushdown automaton $\mathrm{V} = (Q, \Sigma, W, E, q_0, z_0, \emptyset)$ a context-free grammar $G = (N, T, P, S)$, which generates the language accepted by pushdown automaton V by empty stack.

FROM-PUSHDOWN-AUTOMATON-TO-CF-GRAMMAR(V, G)

1 **for** all $q \in Q$
2 **do** put in P production $S \to S_{q_0,z_0,q}$
3 **for** all $\big(q, (a, z/z_k \dots z_2 z_1), p\big) \in E$ $\quad \triangleright\ q \in Q,\ \ z, z_1, z_2, \dots z_k \in W$ $(k \geq 1),\ a \in \Sigma \cup \{\varepsilon\}$
4 **do for** all states $p_1, p_2, \dots, p_k$
5 **do** put in P productions $S_{q,z,p_k} \to aS_{p,z_1,p_1}S_{p_1,z_2,p_2} \dots S_{p_{k-1},z_k,p_k}$
6 **for** All $\big(q(a, z/\varepsilon), p\big) \in E$ $\quad \triangleright\ p, q \in Q, z \in W,\ a \in \Sigma \cup \{\varepsilon\}$
7 **do** put in P production $S_{q,z,p} \to a$

If the automaton has n states and m productions, then the above algorithm executes at most $n + mn + m$ steps, so in worst case the number of steps is $O(nm)$.

Finally, without proof, we mention that the class of languages accepted by deterministic pushdown automata is a proper subset of the class of languages accepted by nondeterministic pushdown automata. This points to the fact that pushdown automata behave differently as finite automata.

Example 1.29 As an example, consider pushdown automaton V from the Example 1.28: $\mathrm{V} = (\{q\}, \{a, b\}, \{a, b, A, S\}, E, q, S, \emptyset)$. Grammar G is:

$$G = (\{S, S_a, S_b, S_S, S_A, \}, \{a, b\}, P, S)\ ,$$

where for all $z \in \{a, b, S, A\}$ instead of $S_{q,z,q}$ we shortly used S_z. The transitions:

$$\begin{array}{lll} \big(q, (a, a/\varepsilon), q\big), & \big(q, (b, b/\varepsilon), q\big)\ , & \\ \big(q, (\varepsilon, S/\varepsilon), q\big), & \big(q, (\varepsilon, S/ba), q\big), & \big(q, (\varepsilon, S/bAa), q\big)\ , \\ \big(q, (\varepsilon, A/ba), q\big), & \big(q, (\varepsilon, A/bAa), q\big)\ . & \end{array}$$

Based on these, the following productions are defined:

$S \ \to S_S$
$S_a \ \to a$
$S_b \ \to b$
$S_S \to \varepsilon \mid S_aS_b \mid S_aS_AS_b$
$S_A \to S_aS_AS_b \mid S_aS_b.$

It is easy to see that S_S can be eliminated, and the productions will be:

$S \ \to \varepsilon \mid S_aS_b \mid S_aS_AS_b,$
$S_A \to S_aS_AS_b \mid S_aS_b,$
$S_a \ \to a,\ \ S_b \to b,$

and these productions can be replaced:

$S \to \varepsilon \mid ab \mid aAb,$
$A \to aAb \mid ab.$

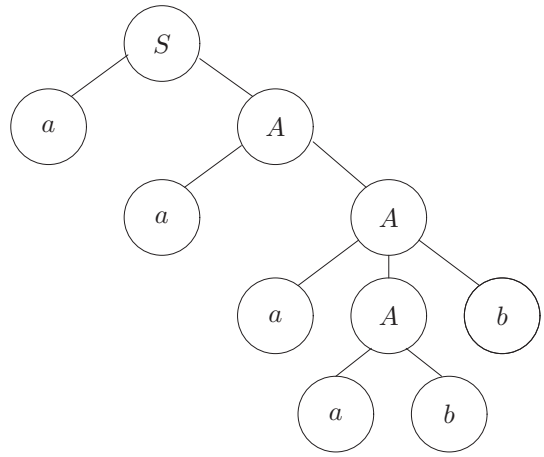


Figure 1.37 Derivation (or syntax) tree of word *aaaabb*.

1.3.2. Context-free languages

Consider context-free grammar $G = (N, T, P, S)$. A ***derivation tree*** of G is a finite, ordered, labelled tree, which root is labelled by the the start symbol S, every interior vertex is labelled by a nonterminal and every leaf by a terminal. If an interior vertex labelled by a nonterminal A has k descendents, then in P there exists a production $A \to a_1 a_2 \dots a_k$ such that the descendents are labelled by letters a_1, a_2, $\dots a_k$. The ***result*** of a derivation tree is a word over T, which can be obtained by reading the labels of the leaves from left to right. Derivation tree is also called ***syntax tree***.

Consider the context-free grammar $G = (\{S, A\}, \{a, b\}, \{S \to aA,\ S \to a,\ S \to \varepsilon,\ A \to aA,\ A \to aAb,\ A \to ab,\ A \to b\}, S)$. It generates language $L(G) = \{a^n b^m \mid n \geq m \geq 0\}$. Derivation of word $a^4 b^2 \in L(G)$ is:

$$S \Longrightarrow aA \Longrightarrow aaA \Longrightarrow aaaAb \Longrightarrow aaaabb.$$

In Fig. 1.37 this derivation can be seen, which result is *aaaabb*.

To every derivation we can associate a syntax tree. Conversely, to any syntax tree more than one derivation can be associated. For example to syntax tree in Fig. 1.37 the derivation

$$S \Longrightarrow aA \Longrightarrow aaAb \Longrightarrow aaaAb \Longrightarrow aaaabb$$

also can be associated.

Definition 1.27 *Derivation* $\alpha_0 \Longrightarrow \alpha_1 \Longrightarrow \dots \Longrightarrow \alpha_n$ *is a* ***leftmost derivation****, if for all* $i = 1, 2, \dots, n-1$ *there exist words* $u_i \in T^*$, $\beta_i \in (N \cup T)^*$ *and productions* $(A_i \to \gamma_i) \in P$, *for which we have*

$$\alpha_i = u_i A_i \beta_i \quad \text{and} \quad \alpha_{i+1} = u_i \gamma_i \beta_i \ .$$

Consider grammar:

$G = (\{S, A\}, \{a, b, c\}, \{S \to bA,\ S \to bAS,\ S \to a,\ A \to cS,\ A \to a\}, S)$.

In this grammar word *bcbaa* has two different leftmost derivations:

$S \Longrightarrow bA \Longrightarrow bcS \Longrightarrow bcbAS \Longrightarrow bcbaS \Longrightarrow bcbaa$,

$S \Longrightarrow bAS \Longrightarrow bcSS \Longrightarrow bcbAS \Longrightarrow bcbaS \Longrightarrow bcbaa.$

Definition 1.28 *A context-free grammar G is* ***ambiguous*** *if in $L(G)$ there exists a word with more than one leftmost derivation. Otherwise G is* ***unambiguous.***

The above grammar G is ambiguous, because word *bcbaa* has two different leftmost derivations. A language can be generated by more than one grammar, and between them can exist ambiguous and unambiguous too. A context-free language is ***inherently ambiguous,*** if there is no unambiguous grammar which generates it.

Example 1.30 Examine the following two grammars.
Grammar $G_1 = (\{S\}, \{a, +, *\}, \{S \to S + S,\ S \to S * S,\ S \to a\}, S)$ is ambiguous because
$S \Longrightarrow S + S \Longrightarrow a + S \Longrightarrow a + S * S \Longrightarrow a + a * S \Longrightarrow a + a * S + S \Longrightarrow a + a * a + S \Longrightarrow a + a * a + a$ and
$S \Longrightarrow S * S \Longrightarrow S + S * S \Longrightarrow a + S * S \Longrightarrow a + a * S \Longrightarrow a + a * S + S \Longrightarrow a + a * a + S \Longrightarrow a + a * a + a.$
Grammar $G_2 = (\{S, A\}, \{a, *, +\}, \{S \to A + S \mid A,\ A \to A * A \mid a\}, S)$ is unambiguous.
Can be proved that $L(G_1) = L(G_2)$.

1.3.3. Pumping lemma for context-free languages

Like for regular languages there exists a pumping lemma also for context-free languages.

Theorem 1.29 (pumping lemma). *For any context-free language L there exists a natural number n (which depends only on L), such that every word z of the language longer than n can be written in the form $uvwxy$ and the following are true:*
(1) $|w| \geq 1$,
(2) $|vx| \geq 1$,
(3) $|vwx| \leq n$,
(4) uv^iwx^iy *is also in L for all* $i \geq 0$.

Proof Let $G = (N, T, P, S)$ be a grammar without unit productions, which generates language L. Let $m = |N|$ be the number of nonterminals, and let ℓ be the maximum of lengths of right-hand sides of productions, i.e. $\ell = \max\{|\alpha| \mid \exists A \in N : (A \to \alpha) \in P\}$. Let $n = \ell^{m+1}$ and $z \in L(G)$, such that $|z| > n$. Then there exists a derivation tree T with the result z. Let h be the height of T (the maximum of path lengths from root to leaves). Because in T all interior vertices have at most ℓ descendents, T has at most ℓ^h leaves, i.e. $|z| \leq \ell^h$. On the other hand, because of $|z| > \ell^{m+1}$, we get that $h > m + 1$. From this follows that in derivation tree T there is a path from root to a leave in which there are more than $(m + 1)$ vertices. Consider such a path. Because in G the number of nonterminals is m and on this path vertices different from the leaf are labelled with nonterminals, by the pigeonhole principle, it must be a nonterminal on this path which occurs at least twice.

Let us denote by A the nonterminal being the first on this path from root to the leaf which firstly repeat. Denote by T' the subtree, which root is this occurrence

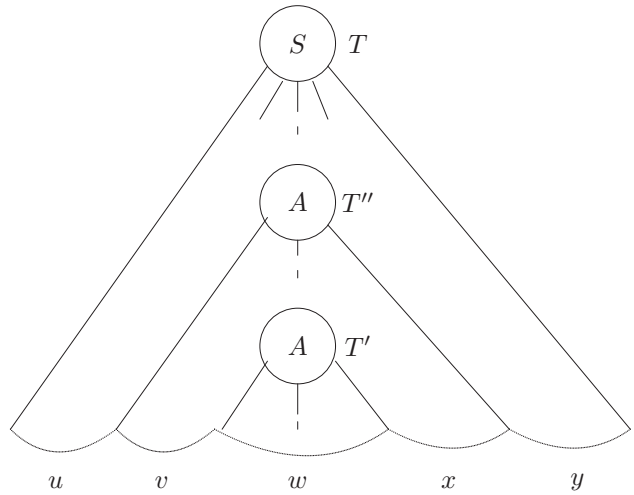


Figure 1.38 Decomposition of tree in the proof of pumping lemma.

of A. Similarly, denote by T'' the subtree, which root is the second occurrence of A on this path. Let w be the result of the tree T'. Then the result of T'' is in form vwx, while of T in $uvwxy$. Derivation tree T with this decomposition of z can be seen in Fig. 1.38. We show that this decomposition of z satisfies conditions (1)–(4) of lemma.

Because in P there are no ε-productions (except maybe the case $S \to \varepsilon$), we have $|w| \geq 1$. Furthermore, because each interior vertex of the derivation tree has at least two descendents (namely there are no unit productions), also the root of T'' has, hence $|vx| \geq 1$. Because A is the first repeated nonterminal on this path, the height of T'' is at most $m+1$, and from this $|vwx| \leq \ell^{m+1} = n$ results.

After eliminating from T all vertices of T'' excepting the root, the result of obtained tree is uAy, i.e. $S \overset{*}{\underset{G}{\Longrightarrow}} uAy$.

Similarly, after eliminating T' we get $A \overset{*}{\underset{G}{\Longrightarrow}} vAx$, and finally because of the definition of T' we get $A \overset{*}{\underset{G}{\Longrightarrow}} w$. Then $S \overset{*}{\underset{G}{\Longrightarrow}} uAy$, $A \overset{*}{\underset{G}{\Longrightarrow}} vAx$ and $A \overset{*}{\underset{G}{\Longrightarrow}} w$. Therefore $S \overset{*}{\underset{G}{\Longrightarrow}} uAy \overset{*}{\underset{G}{\Longrightarrow}} uwy$ and $S \overset{*}{\underset{G}{\Longrightarrow}} uAy \overset{*}{\underset{G}{\Longrightarrow}} uvAxy \overset{*}{\underset{G}{\Longrightarrow}} \ldots \overset{*}{\underset{G}{\Longrightarrow}} uv^iAx^iy \overset{*}{\underset{G}{\Longrightarrow}} uv^iwx^iy$ for all $i \geq 1$. Therefore, for all $i \geq 0$ we have $S \overset{*}{\Longrightarrow} uv^iwx^iy$, i.e. for all $i \geq 0$ $uv^iwx^iy \in L(G)$. ∎

Now we present two consequences of the lemma.

Corollary 1.30 $\mathcal{L}_2 \subset \mathcal{L}_1$.

Proof This consequence states that there exists a context-sensitive language which is not context-free. To prove this it is sufficient to find a context-sensitive language

for which the lemma is not true. Let this language be $L = \{a^m b^m c^m \mid m \geq 1\}$.

To show that this language is context-sensitive it is enough to give a convenient grammar. In Example 1.2 both grammars are extended context-sensitive, and we know that to each extended grammar of type i an equivalent grammar of the same type can be associated.

Let n be the natural number associated to L by lemma, and consider the word $z = a^n b^n c^n$. Because of $|z| = 3n > n$, if L is context-free z can be decomposed in $z = uvwxy$ such that conditions (1)–(4) are true. We show that this leads us to a contradiction.

Firstly, we will show that word v and x can contain only one type of letters. Indeed if either v or x contain more than one type of letters, then in word $uvvwxxy$ the order of the letters will be not the order a, b, c, so $uvvwxxy \notin L(G)$, which contradicts condition (4) of lemma.

If both v and x contain at most one type of letters, then in word uwy the number of different letters will be not the same, so $uwy \notin L(G)$. This also contradicts condition (4) in lemma. Therefore L is not context-free. ∎

Corollary 1.31 *The class of context-free languages is not closed under the intersection.*

Proof We give two context-free languages which intersection is not context-free. Let $N = \{S, A, B\}$, $T = \{a, b, c\}$ and

$G_1 = (N, T, P_1, S)$ where P_1 :
$S \to AB$,
$A \to aAb \mid ab$,
$B \to cB \mid c$,

and $G_2 = (N, T, P_2, S)$, where P_2 :
$S \to AB$,
$A \to Aa \mid a$,
$B \to bBc \mid bc$.

Languages $L(G_1) = \{a^n b^n c^m \mid n \geq 1, m \geq 1\}$ and $L(G_2) = \{a^n b^m c^m \mid n \geq 1, m \geq 1\}$ are context-free. But

$$L(G_1) \cap L(G_2) = \{a^n b^n c^n \mid n \geq 1\}$$

is not context-free (see the proof of the Consequence 1.30). ∎

1.3.4. Normal forms of the context-free languages

In the case of arbitrary grammars the normal form was defined (see page 21) as grammars with no terminals in the left-hand side of productions. The normal form in the case of the context-free languages will contains some restrictions on the right-hand sides of productions. Two normal forms (Chomsky and Greibach) will be discussed.

Chomsky normal form

Definition 1.32 *A context-free grammar $G = (N, T, P, S)$ is in Chomsky normal form, if all productions have form $A \to a$ or $A \to BC$, where $A, B, C \in N$, $a \in T$.*

Example 1.31 Grammar $G = (\{S, A, B, C\}, \{a, b\}, \{S \to AB,\ S \to CB,\ C \to AS,\ A \to a,\ B \to b\}, S)$ is in Chomsky normal form and $L(G) = \{a^n b^n \mid n \geq 1\}$.

To each ε-free context-free language can be associated an equivalent grammar is Chomsky normal form. The next algorithm transforms an ε-free context-free grammar $G = (N, T, P, S)$ in grammar $G' = (N', T, P', S)$ which is in Chomsky normal form.

CHOMSKY-NORMAL-FORM(G)

1 $N' \leftarrow N$
2 eliminate unit productions, and let P' the new set of productions
(see algorithm ELIMINATE-UNIT-PRODUCTIONS on page 21)
3 in P' replace in each production with at least two letters in right-hand side
all terminals a by a new nonterminal A, and add this nonterminal to N'
and add production $A \to a$ to P'
4 replace all productions $B \to A_1 A_2 \dots A_k$, where $k \geq 3$ and $A_1, A_2, \dots, A_k \in N$,
by the following:

$$\begin{aligned} B &\to A_1 C_1, \\ C_1 &\to A_2 C_2, \\ &\dots \\ C_{k-3} &\to A_{k-2} C_{k-2}, \\ C_{k-2} &\to A_{k-1} A_k, \end{aligned}$$

where $C_1, C_2, \dots, C_{k-2}$ are new nonterminals, and add them to N'
5 **return** G'

Example 1.32 Let $G = (\{S, D\}, \{a, b, c\}, \{S \to aSc,\ S \to D,\ D \to bD,\ D \to b\}, S)$. It is easy to see that $L(G) = \{a^n b^m c^n \mid n \geq 0, m \geq 1\}$. Steps of transformation to Chomsky normal form are the following:

Step 1: $N' = \{S, D\}$
Step 2: After eliminating the unit production $S \to D$ the productions are:
$S \to aSc \mid bD \mid b$,
$D \to bD \mid b$.
Step 3: We introduce three new nonterminals because of the three terminals in productions. Let these be A, B, C. Then the production are:
$S \to ASC \mid BD \mid b$,
$D \to BD \mid b$,
$A \to a$,
$B \to b$,
$C \to c$.
Step 4: Only one new nonterminal (let this E) must be introduced because of a single production with three letters in the right-hand side. Therefore $N' = \{S, A, B, C, D, E\}$,

and the productions in P' are:
$S \to AE \mid BD \mid b$,
$D \to BD \mid b$,
$A \to a$,
$B \to b$,
$C \to c$,
$E \to SC$.
All these productions are in required form.

Greibach normal form

Definition 1.33 *A context-free grammar $G = (N, T, P, S)$ is in **Greibach normal form** if all production are in the form $A \to aw$, where $A \in N$, $a \in T$, $w \in N^*$.*

Example 1.33 Grammar $G = (\{S, B\}, \{a, b\}, \{S \to aB,\ S \to aSB,\ B \to b\}, S)$ is in Greibach normal form and $L(G) = \{a^n b^n \mid n \geq 1\}$.

To each ε-free context-free grammar an equivalent grammar in Greibach normal form can be given. We give and algorithm which transforms a context-free grammar $G = (N, T, P, S)$ in Chomsky normal form in a grammar $G' = (N', T, P', S)$ in Greibach normal form.

First, we give an order of the nonterminals: $A_1, A_2, \ldots, A_n$, where A_1 is the start symbol. The algorithm will use the notations $x \in N'^+$, $\alpha \in TN'^* \cup N'^+$.

GREIBACH-NORMAL-FORM(G)

1 $N' \leftarrow N$
2 $P' \leftarrow P$
3 **for** $i \leftarrow 2$ **to** n ▷ Case $A_i \to A_j x$, $j < i$
4 **do for** $j \leftarrow 1$ **to** $i - 1$
5 **do** for all productions $A_i \to A_j x$ and $A_j \to \alpha$ (where α has no A_j as first letter) in P' productions $A_i \to \alpha x$,
delete from P' productions $A_i \to A_j x$
6 **if** there is a production $A_i \to A_i x$ ▷ Case $A_i \to A_i x$
7 **then** put in N' the new nonterminal B_i,
for all productions $A_i \to A_i x$ put in P' productions $B_i \to xB_i$ and $B_i \to x$,delete from P' production $A_i \to A_i x$,
for all production $A_i \to \alpha$ (where A_i is not the first letter of α) put in P' production $A_i \to \alpha B_i$
8 **for** $i \leftarrow n - 1$ **downto** 1 ▷ Case $A_i \to A_j x$, $j > i$
9 **do for** $j \leftarrow i + 1$ **to** n
10 **do** for all productions $A_i \to A_j x$ and $A_j \to \alpha$
put in P' production $A_i \to \alpha x$ and
delete from P' productions $A_i \to A_j x$,

11 **for** $i \leftarrow 1$ **to** n ▷ Case $B_i \to A_j x$
12 **do for** $j \leftarrow 1$ **to** n
13 **do** for all productions $B_i \to A_j x$ and $A_j \to \alpha$
put in P' production $B_i \to \alpha x$ and
delete from P' productions $B_i \to A_j x$
14 **return** G'

The algorithm first transform productions of the form $A_i \to A_j x,\ j < i$ such that $A_i \to A_j x,\ j \geq i$ or $A_i \to \alpha$, where this latter is in Greibach normal form. After this, introducing a new nonterminal, eliminate productions $A_i \to A_i x$, and using substitutions all production of the form $A_i \to A_j x,\ j > i$ and $B_i \to A_j x$ will be transformed in Greibach normal form.

Example 1.34 Transform productions in Chomsky normal form

$A_1 \to A_2A_3 \mid A_2A_4$
$A_2 \to A_2A_3 \mid a$
$A_3 \to A_2A_4 \mid b$
$A_4 \to c$

in Greibach normal form.

Steps of the algorithm:

3–5: Production $A_3 \to A_2A_4$ must be transformed. For this production $A_2 \to a$ is appropriate. Put $A_3 \to aA_4$ in the set of productions and eliminate $A_3 \to A_2A_4$.

The productions will be:

$A_1 \to A_2A_3 \mid A_2A_4$
$A_2 \to A_2A_3 \mid a$
$A_3 \to aA_4 \mid b$
$A_4 \to c$

6-7: Elimination of production $A_2 \to A_2A_3$ will be made using productions:

$B_2 \to A_3B_2$
$B_2 \to A_3$
$A_2 \to aB_2$

Then, after steps 6–7. the productions will be:

$A_1 \to A_2A_3 \mid A_2A_4$
$A_2 \to aB_2 \mid a$
$A_3 \to aA_4 \mid b$
$A_4 \to c$
$B_2 \to A_3B_2 \mid A_3$

8–10: We make substitutions in productions with A_1 in left-hand side. The results is:

$A_1 \to aA_3 \mid aB_2A_3 \mid aA_4 \mid aB_2A_4$

11–13: Similarly with productions with B_2 in left-hand side:

$B_2 \to aA_4B_2 \mid aA_3A_4B_2 \mid aA_4 \mid aA_3A_4$

After the elimination in steps 8–13 of productions in which substitutions were made, the following productions, which are now in Greibach normal form, result:

$A_1 \to aA_3 \mid aB_2A_3 \mid aA_4 \mid aB_2A_4$
$A_2 \to aB_2 \mid a$
$A_3 \to aA_4 \mid b$
$A_4 \to c$
$B_2 \to aA_4B_2 \mid aA_3A_4B_2 \mid aA_4 \mid aA_3A_4$

Example 1.35 Language

$$L = \left\{ a^n b^k c^{n+k} \mid n \geq 0, k \geq 0, n+k > 0 \right\}$$

can be generated by grammar

$$G = \left\{ \{S, R\}, \{a, b, c\}, \{S \to aSc,\ S \to ac,\ S \to R, R \to bRc,\ R \to bc\}, S \right\}$$

First, will eliminate the single unit production, and after this we will give an equivalent grammar in Chomsky normal form, which will be transformed in Greibach normal form.

Productions after the elimination of production $S \to R$:

$S \to aSc \mid ac \mid bRc \mid bc$

$R \to bRc \mid bc$.

We introduce productions $A \to a, B \to b, C \to c$, and replace terminals by the corresponding nonterminals:

$S \to ASC \mid AC \mid BRC \mid BC$,

$R \to BRC \mid BC$,

$A \to a,\ B \to b,\ C \to c$.

After introducing two new nonterminals (D, E):

$S \to AD \mid AC \mid BE \mid BC$,

$D \to SC$,

$E \to RC$,

$R \to BE \mid BC$,

$A \to a,\ B \to b,\ C \to c$.

This is now in Chomsky normal form. Replace the nonterminals to be letters A_i as in the algorithm. Then, after applying the replacements

S replaced by A_1, A replaced by A_2, B replaced by A_3, C replaced by A_4, D replaced by A_5,

E replaced by A_6, R replaced by A_7,

our grammar will have the productions:

$A_1 \to A_2A_5 \mid A_2A_4 \mid A_3A_6 \mid A_3A_4$,

$A_2 \to a,\ A_3 \to b,\ A_4 \to c$,

$A_5 \to A_1A_4$,

$A_6 \to A_7A_4$,

$A_7 \to A_3A_6 \mid A_3A_4$.

In steps 3–5 of the algorithm the new productions will occur:

$A_5 \to A_2A_5A_4 \mid A_2A_4A_4 \mid A_3A_6A_4 \mid A_3A_4A_4$ then

$A_5 \to aA_5A_4 \mid aA_4A_4 \mid bA_6A_4 \mid bA_4A_4$

$A_7 \to A_3A_6 \mid A_3A_4$, then

$A_7 \to bA_6 \mid bA_4$.

Therefore

$A_1 \to A_2A_5 \mid A_2A_4 \mid A_3A_6 \mid A_3A_4$,

$A_2 \to a,\ A_3 \to b,\ A_4 \to c$,

$A_5 \to aA_5A_4 \mid aA_4A_4 \mid bA_6A_4 \mid bA_4A_4$

$A_6 \to A_7A_4$,

$A_7 \to bA_6 \mid bA_4$.

Steps 6–7 will be skipped, because we have no left-recursive productions. In steps 8–10 after the appropriate substitutions we have:

$A_1 \to aA_5 \mid aA_4 \mid bA_6 \mid bA_4$,

$A_2 \to a$,

$A_3 \to b$,

$A_4 \to c$,

$A_5 \to aA_5A_4 \mid aA_4A_4 \mid bA_6A_4 \mid bA_4A_4$
$A_6 \to bA_6A_4 \mid bA_4A_4,$
$A_7 \to bA_6 \mid bA_4.$

Exercises

1.3-1 Give pushdown automata to accept the following languages:
$L_1 = \{a^n cb^n \mid n \geq 0\},$
$L_2 = \{a^n b^{2n} \mid n \geq 1\},$
$L_3 = \{a^{2n} b^n \mid n \geq 0\} \cup \{a^n b^{2n} \mid n \geq 0\},$

1.3-2 Give a context-free grammar to generate language $L = \{a^n b^n c^m \mid n \geq 1, m \geq 1\}$, and transform it in Chomsky and Greibach normal forms. Give a pushdown automaton which accepts L.

1.3-3 What languages are generated by the following context-free grammars?
$G_1 = (\{S\}, \{a, b\}, \{S \to SSa,\ S \to b\}, S)$, $G_2 = (\{S\}, \{a, b\}, \{S \to SaS,\ S \to b\}, S)$.

1.3-4 Give a context-free grammar to generate words with an equal number of letters a and b.

1.3-5 Prove, using the pumping lemma, that a language whose words contains an equal number of letters a, b and c can not be context-free.

1.3-6 Let the grammar $G = (V, T, P, S)$, where
$V = \{S\},$
$T = \{\textit{if, then, else, a, c}\},$
$P = \{S \to \textit{if a then } S,\quad S \to \textit{if a then } S \textit{ else } S,\quad S \to c\},$
Show that word *if a then if a then c else c* has two different leftmost derivations.

1.3-7 Prove that if L is context-free, then $L^{-1} = \{u^{-1} \mid u \in L\}$ is also context-free.

Problems

1-1 Linear grammars
A grammar $G = (N, T, P, S)$ which has productions only in the form $A \to u_1 B u_2$ or $A \to u$, where $A, B \in N,\ u, u_1, u_2 \in T^*$, is called a ***linear grammar.*** If in a linear grammar all production are of the form $A \to Bu$ or $A \to v$, then it is called a left-linear grammar. Prove that the language generated by a left-linear grammar is regular.

1-2 Operator grammars
An ε-free context-free grammar is called ***operator grammar*** if in the right-hand side of productions there are no two successive nonterminals. Show that, for all ε-free context-free grammar an equivalent operator grammar can be built.

1-3 Complement of context-free languages
Prove that the class of context-free languages is not closed on complement.

Chapter Notes

In the definition of finite automata instead of transition function we have used the transition graph, which in many cases help us to give simpler proofs.

There exist a lot of classical books on automata and formal languages. We mention from these the following: two books of Aho and Ullman [**?**, **?**] in 1972 and 1973, book of Gécseg and Peák [**?**] in 1972, two books of Salomaa [**?**, **?**] in 1969 and 1973, a book of Hopcroft and Ullman [**?**] in 1979, a book of Harrison [**?**] in 1978, a book of Manna [**?**], which in 1981 was published also in Hungarian. We notice also a book of Sipser [**?**] in 1997 and a monograph of Rozenberg and Salomaa [**?**]. In a book of Lothaire (common name of French authors) [**?**] on combinatorics of words we can read on other types of automata. Paper of Giammarresi and Montalbano [**?**] generalise the notion of finite automata. A new monograph is of Hopcroft, Motwani and Ullman [**?**]. In German we recommend the textbook of Asteroth and Baier [**?**]. The concise description of the transformation in Greibach normal form is based on this book.

A practical introduction to formal languages is written by Webber [**?**].

Other books in English: [**?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**, **?**].

At the end of the next chapter on compilers another books on the subject are mentioned.

Bibliography

This bibliography is made by HBibTEX. First key of the sorting is the name of the authors (first author, second author etc.), second key is the year of publication, third key is the title of the document.

Underlying shows that the electronic version of the bibliography on the homepage of the book contains a link to the corresponding address.

Index

This index uses the following conventions. Numbers are alphabetised as if spelled out; for example, “2-3-4-tree" is indexed as if were “two-three-four-tree". When an entry refers to a place other than the main text, the page number is followed by a tag: *ex* for exercise, *exa* for example, *fig* for figure, *pr* for problem and *fn* for footnote.

The numbers of pages containing a definition are printed in *italic* font, e.g.

time complexity, *583*.

Name Index

This index uses the following conventions. If we know the full name of a cited person, then we print it. If the cited person is not living, and we know the correct data, then we print also the year of her/his birth and death.