Chapter 2

Formal Languages and Rewriting Systems

The purpose of this chapter is threefold. First, Section 2.1 introduces the terminology concerning formal languages. Second, Section 2.2 introduces rewriting systems. Then, based upon these systems, in an intuitive and preliminary way, it outlines major topics discussed later in this book in a more rigorous and thorough way. Third, Section 2.3 gives a synopsis of this book.

2.1 Formal Languages

An *alphabet* Σ is a finite nonempty set, whose members are called *symbols*. Any nonempty subset of Σ is a *subalphabet* of Σ. A finite sequence of symbols from Σ is a *string* over Σ; specifically, ε is referred to as the *empty string*—that is, the string consisting of zero symbols. By Σ^*, we denote the set of all strings over Σ; $\Sigma^+ = \Sigma^* - \{\varepsilon\}$. Let $x \in \Sigma^*$. Like for any sequence, $|x|$ denotes the *length of* x—that is, the number of symbols in x. For any $a \in \Sigma$, $occur(x, a)$ denotes the number of occurrences of as in x, so $occur(x, a)$ always satisfies $0 \leq occur(x, a) \leq |x|$. Furthermore, if $x \neq \varepsilon$, $symbol(x, i)$ denotes the ith symbol in x, where $i = 1, \ldots, |x|$. Any subset $L \subseteq \Sigma^*$ is a *formal language* or, briefly, a *language* over Σ. Set $symbol(L, i) = \{a \mid a = symbol(x, i), x \in L - \{\varepsilon\}, 1 \leq i \leq |x|\}$. Any subset of L is a *sublanguage* of L. If L represents a finite set of strings, L is a *finite language*; otherwise, L is an *infinite language*. For instance, Σ^*, which is called the *universal language* over Σ, is an infinite language while $\emptyset$ and $\{\varepsilon\}$ are finite; noteworthy, $\emptyset \neq \{\varepsilon\}$ because $card(\emptyset) = 0 \neq card(\{\varepsilon\}) = 1$. Sets whose members are languages are called *families of languages*.

Example 2.1 The English alphabet, consisting of 26 letters, illustrates the definition of an alphabet as stated earlier, except that we refer to its members as symbols in this book. Our definition of a language includes all common artificial and natural languages. For instance, programming languages represent formal languages in terms of this definition, and so do English, Navaho, and Japanese. Any family of natural languages, including Indo-European, Sino-Tibetan, Niger-Congo, Afro-Asiatic, Altaic, and Japonic families of languages, is a language family according to the definition.

Convention 2.1 In strings, for brevity, we simply juxtapose the symbols and omit the parentheses and all separating commas. That is, we write $a_1a_2\ldots a_n$ instead of $(a_1, a_2, \ldots, a_n)$.

Let ${}_{fin}\Phi$ and ${}_{infin}\Phi$ denote the families of finite and infinite languages, respectively. Let ${}_{all}\Phi$ denote the family of all languages; in other words, ${}_{all}\Phi = {}_{fin}\Phi \cup {}_{infin}\Phi$. ■

Operations. Let $x, y \in \Sigma^*$ be two strings over an alphabet Σ, and let $L, K \subseteq \Sigma^*$ be two languages over Σ. As languages are defined as sets, all set operations apply to them. Specifically, $L \cup K$, $L \cap K$, and $L - K$ denote the union, intersection, and difference of languages L and K, respectively. Perhaps most importantly, the *concatenation of x with y*, denoted by xy, is the string obtained by appending y to x. Notice that for every $w \in \Sigma^*$, $w\varepsilon = \varepsilon w = w$. The concatenation of L and K, denoted by LK, is defined as $LK = \{xy | \ x \in L, y \in K\}$.

Apart from binary operations, we also make some unary operations with strings and languages. Let $x \in \Sigma^*$ and $L \subseteq \Sigma^*$. The *complement* of L is denoted by $\sim L$ and defined as $\sim L = \Sigma^* - L$. The *reversal of x*, denoted by *reversal*(x), is x written in the reverse order, and the *reversal* of L, *reversal*(L), is defined as *reversal*(L) = $\{reversal(x) | \ x \in L\}$. For all $i \geq 0$, the ith *power of x*, denoted by x^i, is recursively defined as (1) $x^0 = \varepsilon$, and (2) $x^i = xx^{i-1}$, for $i \geq 1$. Observe that this definition is based on the *recursive definitional method*. To demonstrate the recursive aspect, consider, for instance, the ith power of x^i with $i = 3$. By the second part of the definition, $x^3 = xx^2$. By applying the second part to x^2 again, $x^2 = xx^1$. By another application of this part to x^1, $x^1 = xx^0$. By the first part of this definition, $x^0 = \varepsilon$. Thus, $x^1 = xx^0 = x\varepsilon = x$. Hence, $x^2 = xx^1 = xx$. Finally, $x^3 = xx^2 = xxx$. By using this recursive method, we frequently introduce new notions, including the *ith power of L*, L^i, which is defined as (1) $L^0 = \{\varepsilon\}$ and (2) $L^i = LL^{i-1}$, for $i \geq 1$. The *closure of L*, L^*, is defined as $L^* = L^0 \cup L^1 \cup L^2 \cup \ldots$, and the *positive closure of L*, L^+, is defined as $L^+ = L^1 \cup L^2 \cup \ldots$. Notice that $L^+ = LL^* = L^*L$, and $L^* = L^+ \cup \{\varepsilon\}$. Let $w, x, y, z \in \Sigma^*$. If $xz = y$, then x is a *prefix* of y; if in addition, $x \notin \{\varepsilon, y\}$, x is a *proper prefix* of y. By *prefixes*(y), we denote the set of all prefixes of y. Set *prefixes*(L) = $\{x |$ $x \in prefixes(y)$ for some $y \in L\}$. For $i = 0, \ldots, |y|$, *prefix*(y, i) denotes y's prefix of length i; notice that *prefix*(y, 0) = ε and *prefix*(y, $|y|$) = y. If $zx = y$, x is a *suffix* of y; if in addition, $x \notin \{\varepsilon, y\}$, x is a *proper suffix* of y. By *suffixes*(y), we denote the set of all suffixes of y. Set *suffixes*(L) = $\{x |$ $x \in suffixes(y)$ for some $y \in L\}$. For $i = 0, \ldots, |y|$, *suffix*(y, i) denotes y's suffix of length i. If $wxz = y$, x is a *substring* of y; if in addition, $x \notin \{\varepsilon, y\}$, x is a *proper substring* of y. By *substrings*(y), we denote the set of all substrings of y. Observe that for all $v \in \Sigma^*$, *prefixes*(v) $\subseteq$ *substrings*(v), *suffixes*(v) $\subseteq$ *substrings*(v), and $\{\varepsilon, v\} \in prefixes(v) \cap suffixes(v) \cap substrings(v)$. Set *symbols*($y$) = $\{a |$ $a \in substrings(y), |a| = 1\}$. Furthermore, set *substrings*(L) = $\{x |$ $x \in substrings(y)$ for some $y \in L\}$ and *symbols*(L) = $\{a |$ $a \in symbols(y)$ for some $y \in L\}$.

Example 2.2 Consider the alphabet {0, 1}. For instance, ε, 1, and 010 are strings over {0, 1}. Notice that $|\varepsilon| = 0$, $|1| = 1$, and $|010| = 3$. The concatenation of 1 and 010 is 1010. The third power of 1010 equals 101010101010. Observe that *reversal*(1010) = 0101. We have *prefixes*(1010) = {ε, 1, 10, 101, 1010}, where 1, 10, and 101 are proper prefixes of 1010 while ε and 1010 are not. We have *suffixes*(1010) = {ε, 0, 10, 010, 1010}, *substrings*(1010) = {ε, 0, 1, 01, 10, 010, 101, 1010}, and *symbols*(1010) = {0, 1}.

Set $K = \{0, 01\}$ and $L = \{1, 01\}$. Observe that $L \cup K$, $L \cap K$, and $L - K$ are equal to {0, 1, 01}, {01}, and {1}, respectively. The concatenation of K and L is $KL = \{01, 001, 011, 0101\}$. For L, $\sim L = \Sigma^* - L$, so $\sim L$ contains all strings in $\{0, 1\}^*$ but 1 and 01. Furthermore, *reversal*(L) = {1, 10} and L^2 = {11, 101, 011, 0101}. The strings in L^* that consists of four or fewer symbols are ε, 1, 01, 11, 011, 101, 111, 0101, 0111, 1011, 1101, and 1111. $L^+ = L^* - \{\varepsilon\}$. Notice that *prefixes*(L) = {ε, 1, 0, 01}, *suffixes*(L) = {ε, 1, 01}, *substrings*(L) = {ε, 0, 1, 01}, and *symbols*(L) = {0, 1}.

Letter	μ	Letter	μ	Letter	μ	Letter	μ
A	·–	*H*	····	*O*	–––	*V*	···–
B	–···	*I*	··	*P*	·––·	*W*	·––
C	–·–·	*J*	·–––	*Q*	––·–	*X*	–··–
D	–··	*K*	–·–	*R*	·–·	*Y*	–·––
E	·	*L*	·–··	*S*	···	*Z*	––··
F	··–·	*M*	––	*T*	–		
G	––·	*N*	–·	*U*	··–		

Figure 2.1 Morse code.

Let T and U be two alphabets. A total function τ from T^* to *power*(U^*) such that $\tau(uv) = \tau(u)\tau(v)$ for every $u, v \in T^*$ is a *substitution* from T^* to U^*. By this definition, $\tau(\varepsilon) = \{\varepsilon\}$ and $\tau(a_1a_2\ldots a_n) = \tau(a_1)\tau(a_2)\ldots\tau(a_n)$, where $a_i \in T$, $1 \leq i \leq n$, for some $n \geq 1$, so τ is completely specified by defining $\tau(a)$ for every $a \in T$. A total function υ from T^* to U^* such that $\upsilon(uv) = \upsilon(u)\upsilon(v)$ for every $u, v \in T^*$ is a *homomorphism* from T^* to U^*. As any homomorphism is obviously a special case of a substitution, we simply specify υ by defining $\upsilon(a)$ for every $a \in T$; if $\upsilon(a) \neq \varepsilon$ for all $a \in T$, υ is said to be an ε-*free homomorphism*. It is worth noting that a homomorphism from T^* to U^* may not represent an injection from T^* to U^* as illustrated in Example 2.3.

Example 2.3 Let ${}_{English}\Delta$ denote the English alphabet. The *Morse code*, denoted by μ, can be seen as a homomorphism from ${}_{English}\Delta^*$ to $\{\cdot, -\}^*$ (see Figure 2.1). For instance,

$$\mu(SOS) = \cdot\cdot\cdot - - - \cdot\cdot\cdot$$

Notice that μ is no injection from ${}_{English}\Delta^*$ to $\{\cdot, -\}^*$; for instance, μ(SOS) = μ(*IJS*).

We conclude this section by Example 2.4, which demonstrates how to represent nonnegative integers by strings in a very simple way. More specifically, it introduces function *unary*, which represents all nonnegative integers by strings consisting of *a*s. Later in this book, especially in Section IV, we frequently make use of *unary*.

Example 2.4 Let a be a symbol. To represent nonnegative integers by strings over $\{a\}$, define the total function *unary* from ${}_0\mathbb{N}$ to $\{a\}^*$ as $unary(i) = a^i$, for all $i \geq 0$**.** For instance, $unary(0) = \varepsilon$, $unary(2) = aa$, and $unary(1000000) = a^{1000000}$.

2.2 Rewriting Systems

Rewriting systems, defined and discussed in this section, are used in many mathematical and computer science areas, ranging from purely theoretically oriented areas, such as the investigation of computational principals in terms of logic, up to quite pragmatically oriented areas, such as the construction of translators. Considering the subject of this book, it comes as no surprise that we primarily make use of rewriting systems as language-defining models and computational models.

This section is divided into three subsections. Section 2.2.1 introduces rewriting systems in general. Section 2.2.2 treats them as language-defining models. Finally, Section 2.2.3 formalizes the intuitive notion of a procedure by them.

2.2.1 Rewriting Systems in General

Definition 2.2 A *rewriting system* is a pair, $M = (\Sigma, R)$, where Σ is an alphabet, and R is a finite relation on Σ^*. Σ is called the *total alphabet of M* or, simply, the *alphabet of M*. A member of R is called a *rule of M*, and accordingly, R is referred to as the *set of rules* in M.

The *rewriting relation* over Σ^* is denoted by $\Rightarrow$ and defined so that for every $u, v \in \Sigma^*$, $u \Rightarrow v$ in M iff there exist $(x, y) \in R$ and $w, z \in \Sigma^*$ such that $u = wxz$ and $v = wyz$. As usual, $\Rightarrow^*$ denotes the transitive and reflexive closure of $\Rightarrow$. ■

Convention 2.3 Let $M = (\Sigma, R)$ be a rewriting system. Each rule $(x, y) \in R$ is written as $x \rightarrow y$ throughout this book. We often denote $x \rightarrow y$ with a label r as $r: x \rightarrow y$, and instead of $r: x \rightarrow y \in R$, we just write $r \in R$. For $r: x \rightarrow y \in R$, x and y represent the *left-hand side of r*, denoted by ***lhs***(r), and the *right-hand side of r*, denoted by ***rhs***(r), respectively.

R^* denotes the set of all *sequences of rules* from R; accordingly, by $\rho \in R^*$, we briefly express that ρ is a sequence of rules from R. By analogy with strings (see Convention 2.1), in sequences of rules, we simply juxtapose the rules and omit the parentheses as well as all separating commas in them. That is, if $\rho = (r_1, r_2, \ldots, r_n)$, we simply write ρ as $r_1 r_2 \ldots r_n$. To explicitly express that Σ and R represent the components of M, we write ${}_M\Sigma$ and ${}_M R$ instead of Σ and R, respectively. To explicitly express that $\Rightarrow$ and $\Rightarrow^*$ concern M, we write ${}_M\Rightarrow$ and ${}_M\Rightarrow^*$ instead of $\Rightarrow$ and $\Rightarrow^*$, respectively. Furthermore, by $u \,{}_M\Rightarrow\, v\ [r]$, where $u, v \in \Sigma^*$ and $r \in R$, we express that M directly rewrites u as v according to r. To express that M makes $u \,{}_M\Rightarrow^*\, w$ according to a sequence of rules, $r_1 r_2 \ldots r_n$, we write $u \,{}_M\Rightarrow^*\, v\ [r_1 r_2 \ldots r_n]$. Of course, whenever the information regarding the applied rules is immaterial, we omit these rules; in other words, we simplify $u \,{}_M\Rightarrow\, v\ [r]$ and $u \,{}_M\Rightarrow^*\, v\ [r_1 r_2 \ldots r_n]$ to $u \,{}_M\Rightarrow\, v$ and $u \,{}_M\Rightarrow^*\, v$, respectively. Most often, however, M is understood, so we just write $\Rightarrow$ and $\Rightarrow^*$ instead of ${}_M\Rightarrow$ and ${}_M\Rightarrow^*$, respectively.

By underlining, we specify the substring rewritten during a rewriting step, if necessary. More formally, if $u = wxz$, $v = wyz$, $r: x \rightarrow y \in R$, where $u, v, x, y \in \Sigma^*$, then $w\underline{x}z \Rightarrow wyz\ [r]$ means that the x occurring behind w is rewritten during this step by using r (we usually specify the rewritten occurrence of x in this way when other occurrences of x appear in w and z). ■

Example 2.5 Let us introduce a rewriting system M that translates all strings $x \in {}_{English}\Delta^*$ to the corresponding Morse code $\mu(x)$, where ${}_{English}\Delta$ and μ have the same meaning as in Example 2.3. That is, we define $M = (\Sigma, R)$ with $\Sigma = {}_{English}\Delta \cup \{\cdot, -\}$ and $R = \{a \rightarrow \mu(a) \mid a \in {}_{English}\Delta\}$. Labeling the rules by l_1 through l_{26}, we list them in Figure 2.2.

Define the function $T(M)$ from ${}_{English}\Delta^*$ to $\{\cdot, -\}^*$ as

$$T(M) = \{(s, t) \mid s \Rightarrow^* t, s \in {}_{English}\Delta^*, t \in \{\cdot, -\}^*\}$$

Observe $T(M) = \mu$, so M actually translates strings from ${}_{English}\Delta^*$ to the corresponding Morse codes. For instance, $T(M)$ contains $(SOS, \ldots --- \ldots)$. Indeed, making use of Convention 2.3, we have

$$\begin{array}{lll} SO\underline{S} & \Rightarrow \underline{S}O \ldots & [l_{19}] \\ & \Rightarrow \ldots \underline{O} \ldots & [l_{15}] \\ & \Rightarrow \ldots --- \ldots & [l_{19}] \end{array}$$

Thus, $SOS \Rightarrow^* \cdots\!---\!\cdots [l_{19} l_{15} l_{19}]$. Therefore, $(SOS, \cdots\!---\!\cdots) \in T(M)$. To rephrase this less mathematically, M translates SOS to its Morse code $\cdots\!---\!\cdots$ as desired.

Rules in R			
l_1: $A \rightarrow \cdot -$	l_8: $H \rightarrow \cdot\cdot\cdot\cdot$	l_{15}: $O \rightarrow - - -$	l_{22}: $V \rightarrow \cdot\cdot\cdot -$
l_2: $B \rightarrow - \cdot\cdot\cdot$	l_9: $I \rightarrow \cdot\cdot$	l_{16}: $P \rightarrow \cdot - - \cdot$	l_{23}: $W \rightarrow \cdot - -$
l_3: $C \rightarrow - \cdot - \cdot$	l_{10}: $J \rightarrow \cdot - - -$	l_{17}: $Q \rightarrow - - \cdot -$	l_{24}: $X \rightarrow - \cdot\cdot -$
l_4: $D \rightarrow - \cdot\cdot$	l_{11}: $K \rightarrow - \cdot -$	l_{18}: $R \rightarrow \cdot - \cdot$	l_{25}: $Y \rightarrow - \cdot - -$
l_5: $E \rightarrow \cdot$	l_{12}: $L \rightarrow \cdot - \cdot\cdot$	l_{19}: $S \rightarrow \cdot\cdot\cdot$	l_{26}: $Z \rightarrow - - \cdot\cdot$
l_6: $F \rightarrow \cdot\cdot - \cdot$	l_{13}: $M \rightarrow - -$	l_{20}: $T \rightarrow -$	
l_7: $G \rightarrow - - \cdot$	l_{14}: $N \rightarrow - \cdot$	l_{21}: U $\rightarrow \cdot\cdot -$	

Figure 2.2 Rules of *M*.

2.2.2 Rewriting Systems as Language Models

In this book, we frequently discuss languages that are infinite, so we cannot specify them by an exhaustive enumeration of all their elements. Instead, we define them by formal models of finite size, and we base these models upon rewriting systems (see Definition 2.2).

Whenever we use a rewriting system, $M = (\Sigma, R)$, as a language-defining model, then for brevity, we denote the language that M defines by $L(M)$. In principal, M defines $L(M)$, so it either *generates* $L(M)$ or *accepts* $L(M)$. Next, we explain these two fundamental language-defining methods in a greater detail. Let $S \subseteq \Sigma^*$ and $F \subseteq \Sigma^*$ be a *start language* and a *final language*, respectively.

I. The *language generated by M* is defined as the set of all strings $y \in F$ such that $x \Rightarrow^* y$ in M for some $x \in S$. M used in this way is generally referred to as a *language-generating model*.
II. The *language accepted by M* is the set of all strings $x \in S$ such that $x \Rightarrow^* y$ in M for some $y \in F$. M used in this way is referred to as a *language-accepting model*.

Convention 2.4 Let D be the notion of a language model defined as a rewriting system, $M = ({}_M\Sigma, {}_MR)$, whose components ${}_M\Sigma$ and ${}_MR$ satisfy some prescribed properties, referred to as D-properties. By ${}_D\Psi$, we denote the entire set of all possible rewriting systems whose components satisfy D-properties; mathematically,

$${}_D\Psi = \{M \mid M = ({}_M\Sigma, {}_MR) \text{ is a rewriting system with } {}_M\Sigma \text{ and } {}_MR \text{ satisfying } D\text{-properties}\}.$$

By ${}_D\Phi$, we denote the family of languages defined by all the rewriting systems contained in ${}_D\Psi$; mathematically:

$${}_D\Phi = \{L(M) \mid M \in {}_D\Psi\}$$ ■

Example 2.6 To illustrate method I, we define the notion of a *parenthesis-generating language model* (*PGLM*) as a rewriting system, $M = ({}_M\Sigma, {}_MR)$, where ${}_M\Sigma$ consists of (and), and ${}_MR$ is a finite set of rules of the form $\varepsilon \rightarrow x$ with $x \in {}_M\Sigma^*$. Let $S = \{\varepsilon\}$ and $F = {}_M\Sigma^*$. We define the language generated by M as

$$L(M) = \{t \mid s \Rightarrow^* t, s \in S, t \in F\}$$

In other words,

$$L(M) = \{t | \ \varepsilon \Rightarrow^* t, t \in \Sigma^*\}$$

Following Convention 2.5, by ${}_{PGLM}\mathbf{\Psi}$, we denote the set of all rewriting systems that represent a *PGLM*—that is, their alphabets consist of (and), and each of their rules has ε as its left-hand side. Furthermore, we set ${}_{PGLM}\mathbf{\Phi} = \{L(Z) | \ Z \in {}_{PGLM}\mathbf{\Psi}\}$.

Next, we consider $X \in {}_{PGLM}\mathbf{\Psi}$ defined as $X = ({}_X\Sigma, {}_XR)$, ${}_X\Sigma$ consists of (and), and

$${}_XR = \{\varepsilon \rightarrow ()\}$$

For example,

$$\varepsilon \Rightarrow () \Rightarrow ()() \Rightarrow (())() \Rightarrow (()())()$$

in X, so $(()())() \in L(X)$. Observe that $L(X)$ consists of all properly nested parentheses; for instance, $() \in L$, but $((\notin L$.

To illustrate method II, we define the notion of a *parenthesis-accepting language model* (*PALM*) as a rewriting system $M = ({}_M\Sigma, {}_MR)$, where ${}_M\Sigma$ is an alphabet consisting of (and), and ${}_MR$ is a finite set of rules of the form $x \rightarrow \varepsilon$ with $x \in \Sigma^*$. Let $S = \Sigma^*$ and $F = \{\varepsilon\}$. We define the language accepted by M as

$$L(M) = \{s | \ s \Rightarrow^* t, s \in S, t \in F\}$$

In other words,

$$L(M) = \{s | \ s \Rightarrow^* \varepsilon, s \in \Sigma^*\}$$

Let ${}_{PALM}\mathbf{\Psi}$ denote the set of all rewriting systems that represent a *PALM*, and let ${}_{PALM}\mathbf{\Phi} = \{L(W) | \ W \in {}_{PALM}\mathbf{\Psi}\}$.

Next, we consider $Y \in {}_{PALM}\mathbf{\Psi}$ defined as $Y = ({}_Y\Sigma, {}_YR)$, where ${}_Y\Sigma$ consists of (and), and

$${}_YR = \{() \rightarrow \varepsilon\}$$

For instance, Y accepts $(())()$

$$(())\underline{()} \Rightarrow (\underline{()}) \Rightarrow \underline{()} \Rightarrow \varepsilon$$

where the underlined substrings denote the substrings that are rewritten (see Convention 2.3). On the other hand, observe that $(((\underline{()} \Rightarrow ((($, and $(($ cannot be rewritten by Y, so $((()$ is not accepted by Y. Observe that $L(X) = L(Y)$. More generally, as an exercise, prove that

$${}_{PGLM}\mathbf{\Phi} = {}_{PALM}\mathbf{\Phi}$$

If some language models define the same language, they are said to be *equivalent*. For instance, in Example 2.6, X and Y are equivalent. More generally, suppose we have defined two language-defining models, referred to as C and D. Let ${}_C\mathbf{\Phi} = {}_D\mathbf{\Phi}$ (see Convention 2.4); then, Cs and Ds are *equivalent*, or synonymously, Cs and Ds are *equally powerful*. If Cs and Ds are equivalent, we also say that Cs *characterize* ${}_D\mathbf{\Phi}$. For instance, in Example 2.6, *PGLM*s and *PALM*s are equivalent because ${}_{PGLM}\mathbf{\Phi} = {}_{PALM}\mathbf{\Phi}$, so *PGLM*s characterize ${}_{PALM}\mathbf{\Phi}$.

For brevity, *language-generating models* are often called *grammars* in this book; for instance, in Example 2.6, the PGLM could be referred to as a *parenthesis grammar* for brevity.

Let $G = ({}_G\Sigma, {}_GR)$ be a grammar. The symbols occurring in $L(G)$ are referred to as *terminal symbols* or, briefly, *terminals*, denoted by ${}_G\Delta$ in this book, so ${}_G\Delta \subseteq {}_G\Sigma$. We set ${}_GN = {}_G\Sigma - {}_G\Delta$,

whose members are called *nonterminal symbols* or *nonterminals*. The start language of G always consists of a single symbol; more precisely, ${}_GN$ contains a special *start symbol*, denoted by ${}_GS$, so we always have ${}_G\Delta \subset {}_G\Sigma$. In ${}_GR$, at least one nonterminal occurs on the left-hand side of every rule. If $v \Rightarrow^* w$ in G, where $v, w \in \Sigma^*$, then we say that *G makes a derivation from v to w*. The *language generated by G*, $L(G)$, is defined as

$$L(G) = \{w \in {}_G\Delta^* \mid {}_GS \Rightarrow^* w\}$$

Example 2.7 discusses grammars in which the left-hand side of every rule consists of a single nonterminal. As a result, these grammars rewrite nonterminals regardless of the context in which they appear in the strings, hence their name *context-free grammars* (CFGs). Section III of this book is primarily dedicated to them and their languages, naturally referred to as *context-free languages*.

As illustrated in Example 2.7, in a CFG, $G = (\Sigma, R)$, there may exist many different derivations from v to w, where $v, w \in \Sigma^*$. Since a derivation multiplicity like this obviously complicates the discussion of G and $L(G)$, we often reduce it by considering only *leftmost derivations* in which G always rewrites the leftmost nonterminal occurring in the current rewritten string. Unfortunately, even if we reduce our attention only to these derivations, we may still face a derivation multiplicity of some sentences, and this undesirable phenomenon is then referred to as grammatical *ambiguity*.

Convention 2.5 Let $G = (\Sigma, R)$ be a grammar. By analogy with Convention 2.3, to explicitly express that Σ, Δ, N, S, and R represent the above-mentioned components in G, we write ${}_G\Sigma$, ${}_G\Delta$, ${}_GN$, ${}_GS$, and ${}_GR$ instead of plain Σ, Δ, N, S, and R, respectively. ■

Example 2.7 We define the notion of a CFG as a rewriting system, $G = ({}_G\Sigma, {}_GR)$, where ${}_G\Sigma = {}_GN \cup {}_G\Delta$, which satisfy the properties stated previously. Specifically, ${}_GN$ always contains a special start symbol, denoted by ${}_GS$, shortened to S throughout this example. ${}_GR$ is a finite set of rules of the form $A \to x$ with $A \in {}_GN$ and $x \in {}_G\Sigma^*$. We define the language generated by G as

$$L(G) = \{w \in {}_G\Delta^* \mid S \Rightarrow^* w\}$$

Set ${}_{CFG}\Psi = \{G \mid G \text{ is a CFG}\}$ (see Convention 2.4). Next, we consider two specific instances, U and V, from ${}_{CFG}\Psi$.

Let $U \in {}_{CFG}\Psi$ be defined as $U = ({}_U\Sigma, {}_UR)$, ${}_U\Delta = \{a, b\}$, ${}_UN = \{S, A\}$, and ${}_UR = \{S \to SS, S \to A, A \to aAb, A \to \varepsilon\}$. For example, U makes

$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \underline{A}S \Rightarrow aAb\underline{S} \Rightarrow aAb\underline{A} \Rightarrow aAba\underline{A}b \Rightarrow a\underline{A}bab \Rightarrow aa\underline{A}bbab \Rightarrow aabbab$$

in U; recall that we specify the rewritten symbols by underlining (see Convention 2.3). Thus, $S \Rightarrow^* aabbab$, so $aabbab \in L(U)$. As an exercise, prove that

$$L(U) = K^*$$

with

$$K = \{a^i b^i \mid i \geq 0\}$$

In words, the language generated by U is the closure of K (see Section 2.1 for the definition of closure).

As already stated, a leftmost derivation is a derivation in which G always rewrites the leftmost nonterminal occurring in every string. While the above-mentioned derivation is not leftmost, the next derivation represents a leftmost derivation from S to *aabbab*:

$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \underline{A}S \Rightarrow a\underline{A}bS \Rightarrow aa\underline{A}bbS \Rightarrow aabb\underline{S} \Rightarrow aabb\underline{A} \Rightarrow aabba\underline{A}b \Rightarrow aabbab$$

Notice, however, that there exist infinitely many other leftmost derivations from S to *aabbab*, including these two leftmost derivations:

$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \underline{S}SS \Rightarrow \underline{A}SS \Rightarrow \underline{S}S \Rightarrow \underline{A}S \Rightarrow a\underline{A}bS \Rightarrow aa\underline{A}bbS \Rightarrow aabb\underline{S} \Rightarrow aabb\underline{A} \Rightarrow aabba\underline{A}b \Rightarrow aabbab$$

$$\underline{S} \Rightarrow \underline{S}S \Rightarrow \underline{A}S \Rightarrow a\underline{A}bS \Rightarrow aa\underline{A}bbS \Rightarrow aabb\underline{S} \Rightarrow aabb\underline{S}S \Rightarrow aabb\underline{A}S \Rightarrow aabba\underline{A}bS \Rightarrow aabbab\underline{S} \Rightarrow aabbab\underline{A} \Rightarrow aabbab$$

Thus, U is ambiguous, which usually represents a highly undesirable property in practice. Therefore, we may prefer using an equivalent unambiguous CFG, such as $V \in {}_{CFG}\Psi$ defined next.

Let $V = ({}_V\Sigma, {}_VR)$, where ${}_V\Delta = \{a, b\}$, ${}_VN = \{S, A\}$, and ${}_VR = \{S \rightarrow AS, S \rightarrow \varepsilon, A \rightarrow aAb, A \rightarrow ab\}$. For example, V generates *aabbab* by this unique leftmost derivation:

$$\underline{S} \Rightarrow \underline{A}S \Rightarrow a\underline{A}bS \Rightarrow aabb\underline{S} \Rightarrow aabb\underline{A}S \Rightarrow aabbab\underline{S} \Rightarrow aabbab$$

Clearly, U and V are equivalent; however, U is ambiguous while V is not.

In this book, we often introduce a new notion of a grammar as a special case of the notion of a grammar that has already been defined. To illustrate, we define the notion of a *linear grammar* (*LG*) as a CFG, $H = ({}_H\Sigma, {}_HR)$, in which the right-hand side of every rule contains no more than one occurrence of a nonterminal. In other words, ${}_HR$ is a finite set of rules of the form $A \rightarrow uBv$ with $A \in {}_HN$, $B \in {}_HN \cup \{\varepsilon\}$, and $u, v \in {}_H\Delta^*$. We set ${}_{LG}\Psi = \{G |\ G \text{ is an LG}\}$ (see Convention 2.4).

For instance, consider $W \in {}_{LG}\Psi$ defined as $W = ({}_W\Sigma, {}_WR)$, ${}_W\Delta = \{a, b\}$, ${}_WN = \{S\}$, and ${}_WR = \{S \rightarrow aSb, S \rightarrow \varepsilon\}$. For example,

$$S \Rightarrow aSb \Rightarrow aaSbb \Rightarrow aabb$$

in W. Thus, $aabb \in L(W)$. Observe that $L(W) = K$, where $K = \{a^ib^i |\ i \geq 0\}$ as explained earlier.

We close this example by taking a more general view at CFGs, LGs, and the language families they generate. Following Convention 2.4, set

$${}_{CFG}\Phi = \{L(C) |\ C \in {}_{CFG}\Psi\} \text{ and } {}_{LG}\Phi = \{L(D) |\ D \in {}_{LG}\Psi\}$$

LGs are special cases of CFGs, so ${}_{LG}\Phi \subseteq {}_{CFG}\Phi$. To prove that this inclusion is proper, take

$$K^2 = KK = \{a^ib^ia^jb^j |\ i, j \geq 0\}$$

Observe that no more than one occurrence of a nonterminal appears in any string generated by an LG. As an exercise, make use of this property to prove that no LG generates K^2, so $K^2 \notin {}_{LG}\Phi$. Clearly, $K^2 \in {}_{CFG}\Phi$. Since $K^2 \in {}_{CFG}\Phi - {}_{LG}\Phi$ and ${}_{LG}\Phi \subseteq {}_{CFG}\Phi$, ${}_{LG}\Phi \subset {}_{CFG}\Phi$.

Let M and N be two language models. If ${}_M\Phi \subset {}_N\Phi$ (see Convention 2.4), then we say that Ns are *stronger* or, equivalently speaking, *more powerful* than Ms. For instance, in Example 2.7, we have proved that CFGs are stronger than LGs because ${}_{LG}\Phi \subset {}_{CFG}\Phi$.

Regarding properties of language families, this book pays its principal attention to *closure properties*, which are helpful to answer some important questions concerning these families, such as whether a given language belongs to a language family. To give an insight into them, let o be an n-ary operation. Let Ξ be a language family. We say that Ξ *is closed under* o or,

synonymously, *o preserves* Ξ if the application of *o* to any *n* languages in Ξ results into a language that is also in Ξ.

We demonstrate most closure properties *effectively* in this book. To explain what this means, let *M* be a language-defining model, ${}_M\Psi = \{X | X \text{ is an instance of } M\}$, and ${}_M\Phi = \{L(X) | X \in {}_M\Psi\}$. Giving an effective proof that ${}_M\Phi$ is closed under *o* consists in constructing an algorithm that converts any *n* models in ${}_M\Psi$ to a model in ${}_M\Psi$ so that the resulting model defines the language resulting from *o* applied to the languages defined by the *n* models.

In this introductory book, closure properties are discussed only in terms of *n*-ary operation for $n = 1$ and $n = 2$—that is, only unary and binary operations are considered.

Example 2.8 Let ${}_{CFG}\Phi$ and ${}_{LG}\Phi$ have the same meaning as in Example 2.7. As operation, consider closure (see Section 2.1). Recall that the closure of a language *L* is denoted by L^*.

First, we effectively prove that ${}_{CFG}\Phi$ is closed under this operation. Let *L* be any language in ${}_{CFG}\Phi$, and let $X = ({}_X\Sigma, {}_XR)$ be a CFG that generates *L*, mathematically written as $L = L(X)$. Next, we give an algorithm that converts *X* to a CFG $Y = ({}_Y\Sigma, {}_YR)$ so *Y* generates the closure of $L(X)$, symbolically written as $L(Y) = L(X)^*$. Introduce a new symbol, *A*. Turn *X* to $Y = ({}_Y\Sigma, {}_YR)$ so ${}_Y\Sigma = {}_X\Sigma \cup \{A\}$, ${}_Y\Delta = {}_X\Delta$, ${}_YR = \{A \rightarrow {}_XSA, A \rightarrow \varepsilon\} \cup {}_XR$, where ${}_XS$ is the start symbol of *X*. Define *A* as the start symbol of *Y*—that is, $A = {}_YS$ (see Convention 2.5). As an exercise, prove that $L(Y) = L(X)^*$. Since $L = L(X)$, $L^* \in {}_{CFG}\Phi$. Hence, ${}_{CFG}\Phi$ is closed under this operation.

Second, we prove that ${}_{LG}\Phi$ is not closed under this operation. Take $K = \{(^i)^i | \ i \geq 0\}$. In Example 2.7, we give an LG *W* such that $L(W) = K$, so $K \in {}_{LG}\Phi$. Just like in the conclusion of Example 2.7, prove that $K^j \notin {}_{LG}\Phi$, for any $j \geq 2$. By using this result, show that $K^* \notin {}_{LG}\Phi$. Hence, ${}_{LG}\Phi$ is not closed under this operation.

Finally, in I and II given next, we illustrate how to use closure properties to establish results concerning ${}_{CFG}\Phi$ and ${}_{LG}\Phi$.

I. In Example 2.7, we have constructed CFGs that generate K^*. As a consequence, we have actually proved that $K^* \in {}_{CFG}\Phi$ in a constructive way. Now, we demonstrate that closure properties may free us from grammatical constructions in proofs like this. Indeed, as $K \in {}_{LG}\Phi$ and every LG is a special case of a CFG, $K \in {}_{CFG}\Phi$. Since ${}_{CFG}\Phi$ is closed with respect to closure, $K^* \in {}_{CFG}\Phi$, which completes the proof.
II. In Example 2.7, we have also proved that ${}_{LG}\Phi \subset {}_{CFG}\Phi$. By using closure properties, we can establish this proper inclusion in an alternative way. Indeed, by definition, ${}_{LG}\Phi \subseteq {}_{CFG}\Phi$. Since ${}_{CFG}\Phi$ is closed under closure while ${}_{LG}\Phi$ is not, ${}_{LG}\Phi \subset {}_{CFG}\Phi$, and we are done.

2.2.3 Rewriting Systems as Computational Models

While Section 2.2.2 has primarily dealt with language-generating models, this section bases its discussion upon language-accepting models, frequently referred to as *automata* or *machines* for brevity. More specifically, in this section, automata are primarily seen as models of computation. To introduce this important topic, Example 2.9 gives an automaton, which accepts strings just like any other language-accepting model. In addition, however, it acts as a computational model, too.

Example 2.9 We introduce a rewriting system, $M = (\Sigma, R)$, which acts as a computer of well-written postfix Polish expressions, defined in Example 1.1. Set

- $\Sigma = \{\mathbf{0}, \mathbf{1}, \vee, \wedge\}$;
- $R = \{\mathbf{11}\vee \rightarrow \mathbf{1}, \mathbf{10}\vee \rightarrow \mathbf{1}, \mathbf{01}\vee \rightarrow \mathbf{1}, \mathbf{00}\vee \rightarrow \mathbf{0}, \mathbf{11}\wedge \rightarrow \mathbf{1}, \mathbf{10}\wedge \rightarrow \mathbf{0}, \mathbf{01}\wedge \rightarrow \mathbf{0}, \mathbf{00}\wedge \rightarrow \mathbf{0}\}$.

Observe that for all $x \in \Sigma^*$ and $i \in \{\mathbf{0}, \mathbf{1}\}$,

$$x \Rightarrow^* i \text{ iff } x \text{ is a postfix polish expression whose logical value is } i$$

For instance, $\mathbf{10}\vee\mathbf{0}\wedge \Rightarrow \mathbf{10}\wedge \Rightarrow \mathbf{0}$, so $\mathbf{10}\vee\mathbf{0}\wedge \Rightarrow^* \mathbf{0}$, and observe the logical value of $\mathbf{10}\vee\mathbf{0}\wedge$ is indeed $\mathbf{0}$. On the other hand, $\mathbf{101}\wedge \Rightarrow \mathbf{10}$, and from $\mathbf{10}$, M can make no further rewriting step; notice that $\mathbf{101}\wedge$ is no postfix polish expression.

In Example 2.9, M can be viewed as a highly stylized procedure, which evaluates all well-constructed postfix logical expressions over $\{\mathbf{0}, \mathbf{1}, \vee, \wedge\}$. This view brings us to considering rewriting systems as computational models that formalize the intuitive notion of an *effective procedure* or, briefly, a *procedure*—the central notion of computation as a whole. We surely agree that each procedure is finitely describable and consists of discrete steps, each of which can be executed mechanically. That is, throughout this book, we understand the *intuitive notion of a procedure* as a finite set of instructions, each of which can be executed in a fixed amount of time. When executed, a procedure reads input data, executes its instructions, and produces output data. Of course, both the input data and the output data may be nil. An *algorithm* is a special case of a procedure that halts on all inputs. For instance, every computer program represents a procedure, and if the program never enters an endless loop, then it is an algorithm.

In this introductory book, we restrict our attention only to rewriting systems that act as computational models of nonnegative integer functions. To use rewriting systems to compute the numeric functions, we obviously need to represent all nonnegative integers by strings. Traditionally, they are represented in unary. More specifically, every $i \in {}_0\mathbb{N}$ is represented as $unary(i)$, where *unary* is defined in Example 2.4. Consequently, $unary(j) = a^j$ for all $j \geq 0$; for instance, $unary(0)$, $unary(2)$, and $unary(999)$ are equal to ε, aa, and a^{999}, respectively.

Example 2.10 describes rewriting systems that act as integer function computers. The way by which they perform their computation strongly resembles the way by which *Turing machines* (TMs) compute integer functions in Section IV of this book. The example also illustrates that just like grammars use some auxiliary symbols, so do automata, whose alphabets often contain some *delimiter* and *state* symbols.

Example 2.10 Let $M = (\Sigma, R)$ be a rewriting system, where $\Sigma = \{\triangleright, \triangleleft, \blacktriangleright, \blacksquare, a\}$, and R is a finite set of rules of the form $x \to y$, where either $x, y \in \{\triangleright\}W$ or $x, y \in W$ or $x, y \in W\{\triangleleft\}$ with $W = \{a\}^*\{\blacktriangleright, \blacksquare\}\{a\}^*$. Let f be a function over ${}_0\mathbb{N}$. *M computes f* iff the next equivalence holds true:

$$f(i) = j \text{ iff } \triangleright\blacktriangleright unary(i)\triangleleft \Rightarrow^* \triangleright\blacksquare unary(j)\triangleleft \text{ in } M$$

where $i, j \in {}_0\mathbb{N}$. Considering the definition of *unary* (see Example 2.4), we can rephrase this equivalence as

$$f(i) = j \text{ iff } \triangleright\blacktriangleright a^i\triangleleft \Rightarrow^* \triangleright\blacksquare a^j\triangleleft \text{ in } M$$

Notice that the string of as is delimited by $\triangleright$ and $\triangleleft$. M always starts its computation from $\blacktriangleright$, referred to as the *start state*, and finalizes it in $\blacksquare$, referred to as a *final state*.

Let g be the *successor function* defined as $g(i) = i + 1$, for all $i \geq 0$. Informally, we construct a rewriting system, $X = ({}_X\Sigma, {}_XR)$, that computes g so $\triangleright\blacktriangleright a^i\triangleleft \Rightarrow^* \triangleright\blacksquare a^{i+1}\triangleleft$ so it moves $\blacktriangleright$ across a^i to the $\triangleleft$ and replaces it with $a\triangleleft$. As a result, in between $\triangleright$ and $\triangleleft$, X increases the number of as by one, so X constructed in this way computes g. Formally, ${}_X\Sigma = \{\triangleright, \triangleleft, \blacktriangleright, \blacksquare, a\}$ and

$${}_XR = \{\blacktriangleright a \to a\blacktriangleright, \blacktriangleright\triangleleft \to \blacksquare a\triangleleft, a\blacksquare \to \blacksquare a\}$$

For instance, X computes $g(2) = 3$ as

$$\triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright aa\blacktriangleright\triangleleft \Rightarrow \triangleright aa\blacksquare a\triangleleft \Rightarrow \triangleright a\blacksquare aa\triangleleft \Rightarrow \triangleright\blacksquare aaa\triangleleft$$

Modify X by introducing a new rule of the form $a\blacktriangleright \rightarrow \blacktriangleright a$. Denote the rewriting system modified in this way by Y. Formally, $Y = ({}_Y\Sigma, {}_YR)$, ${}_Y\Sigma = \{\triangleright, \triangleleft, \blacktriangleright, \blacksquare, a\}$, and

$${}_YR = \{\blacktriangleright a \rightarrow a\blacktriangleright,\ a\blacktriangleright \rightarrow \blacktriangleright a,\ \blacktriangleright\triangleleft \rightarrow \blacksquare a\triangleleft,\ a\blacksquare \rightarrow \blacksquare a\}$$

As a simple exercise, prove that Y also computes g. However, Y can move $\blacktriangleright$ across as between $\triangleright$ and $\triangleleft$ in either direction while X cannot. For instance, Y computes $g(2) = 3$ by infinitely many sequences of rewriting steps, including these three sequences:

I. $\triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright aa\blacktriangleright\triangleleft \Rightarrow \triangleright aa\blacksquare a\triangleleft \Rightarrow \triangleright a\blacksquare aa\triangleleft \Rightarrow \triangleright\blacksquare aaa\triangleleft$

II. $\triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright aa\blacktriangleright\triangleleft \Rightarrow \triangleright aa\blacksquare a\triangleleft \Rightarrow \triangleright a\blacksquare aa\triangleleft \Rightarrow \triangleright\blacksquare aaa\triangleleft$

III. $\triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow$
$\triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow \triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright a\blacktriangleright a\triangleleft \Rightarrow$
$\triangleright aa\blacktriangleright\triangleleft \Rightarrow \triangleright aa\blacksquare a\triangleleft \Rightarrow \triangleright a\blacksquare aa\triangleleft \Rightarrow \triangleright\blacksquare aaa\triangleleft$

We say that X works deterministically while Y does not, and we return to the topic of *determinism* from a more general viewpoint after closing this example.

Let us now demonstrate that we can approach the discussion of rewriting systems acting as computers of functions in the other way around. That is, given a rewriting system, we can naturally ask what function the system computes. Consider, for instance, $Z = ({}_Z\Sigma, {}_ZR)$, ${}_Z\Sigma = \{\triangleright, \triangleleft, \blacktriangleright, \blacksquare, a\}$ and

$${}_ZR = \{\blacktriangleright a \rightarrow aa\blacktriangleright,\ \blacktriangleright\triangleleft \rightarrow \blacksquare\triangleleft,\ a\blacksquare \rightarrow \blacksquare a\}$$

For example, notice that

$$\triangleright\blacktriangleright aa\triangleleft \Rightarrow \triangleright aa\blacktriangleright a\triangleleft \Rightarrow \triangleright aaaa\blacktriangleright\triangleleft \Rightarrow \triangleright aaaa\blacksquare\triangleleft \Rightarrow \triangleright aaa\blacksquare a\triangleleft \Rightarrow \triangleright aa\blacksquare aa\triangleleft \Rightarrow$$
$$\triangleright a\blacksquare aaa\triangleleft \Rightarrow \triangleright\blacksquare aaaa\triangleleft$$

More generally, M deterministically computes the function $f(i) = 2i$ for all $i \geq 0$; for instance, $f(1) = 2$ and $f(9) = 18$. Indeed, starting from $\triangleright\blacktriangleright a^i\triangleleft$, it computes $\triangleright\blacktriangleright a^i\triangleleft \Rightarrow^* \triangleright\blacksquare a^{2i}\triangleleft$ for all $i \geq 0$.

Finally, let us point out that we can approach computational models in a more fundamental way. Return to the definition given in the beginning of this example, and assume that f may be, in a general case, a partial function. Under this assumption, literally every rewriting system with $\{\triangleright, \triangleleft, \blacktriangleright, \blacksquare, a\}$ as its alphabet computes a function. At a glance, this statement sounds almost incredible because we may have rewriting systems with rules by which they can never compute anything (for instance, this is the case whenever they have no rule containing $\blacksquare$); consequently, the computed function is undefined for all arguments. However, realize that functions that are everywhere undefined are mathematically legitimate instances of partial functions, too. When discussing models of computation later in Chapter 10, we will also approach them in such a general way—that is, we will consider the entire set of them all as well as the corresponding set of functions computed by them. Indeed, we will define functions computed by aforementioned TMs. We will investigate the set of all these functions in order to find out what is computable by computers and what is not, which is obviously a problem area of general mathematical as well as philosophical interest.

Consider X and Y in the previous example. They both compute the same function by rewriting strings over K, where $K = \triangleright\{a\}^*\{\blacktriangleright, \blacksquare\}\{a\}^*\triangleleft$. As already pointed out in the example, however, in terms of the way they work, there exists a fundamental difference between X and Y—the former

works deterministically over K while the latter does not. That is, X rewrites any string from K by no more than one rule while Y does not satisfy this property.

Definition 2.6 Let $M = (\Sigma, R)$ be a rewriting system and $K \subseteq \Sigma^*$. M is *deterministic over* K if for every $w \in K$, there is no more than one $r \in R$ such that $w \,{}_M\Rightarrow\, v\ [r]$ with $v \in K$ (when K is understood, we usually just say that M is *deterministic*). ■

Consequently, if M is deterministic over K, then ${}_M\Rightarrow$ represents a function over K—that is, for all $u, v, w \in K$, if $u \,{}_M\Rightarrow\, v$ and $u \,{}_M\Rightarrow\, w$, then $v = w$. In general, the basic versions of various types of rewriting systems are always introduced quite generally and, therefore, nondeterministically in the theory of computational and language models. That is also why we first define the basic versions of these models in a nondeterministic way throughout this book. In practice, however, we obviously prefer their deterministic versions because they are easier to implement. Therefore, we usually place a restriction on their rules so that the models with rules restricted in this way necessarily work deterministically. Of course, we always study whether all the nondeterministic versions can be converted to equivalent deterministic versions, and if so, we want to perform this conversion algorithmically. As determinism obviously represents such an important investigation area, we pay a special attention to this topic throughout this book.

As pointed out in the conclusion of Example 2.10, when discussing models of computation, we are inescapably lead to a certain *metaphysics of computation*, trying to find out what computers can compute and what they cannot. To narrow this investigation to mathematics, we obviously want to know whether there is a procedure that computes any function. Unfortunately, the answer is no. To justify this answer, we need a formalization of the intuitive notion of a procedure, and any formalization of this kind obviously has to satisfy the essential property that each of its instances is finitely describable. Suppose we have a general mathematical model Γ formalizing the intuitive notion of a procedure that satisfies this property. All the instances of Γ are countable because we can make a list of all their finite descriptions, for instance, according to length and alphabetic order, so the set of these descriptions is equal in cardinality to $\mathbb{N}$. However, we already know that the set of all functions is uncountable (see Example 1.3), so there necessarily exist functions that cannot be computed by any procedure. Simply put, the number of all functions is uncountable while the number of formalized procedures is countable.

More surprisingly, even if we narrow our attention to the set ϕ containing all total functions over $\mathbb{N}$, by using the diagonalization proof technique (see Example 1.3), the theory of *computability*, which studies questions of this kind, can easily demonstrate a specific function $g \in \phi$ that cannot be computed by any Γ-formalized procedure. Indeed, since each function $h \in \phi$ is total, it has to be computed by an algorithm, which always halts and produces $h(j)$ for all $j \in \mathbb{N}$. For the sake of contradiction, suppose that all functions in ϕ are computed by an algorithm formalized by an instance of Γ. Consider all the descriptions of the Γ instances that compute the functions in ϕ. Let ${}_1F, {}_2F, \ldots$ be an enumeration of these finite descriptions. By ${}_iF\text{-}f$, we denote the function computed by the algorithm formalized by the model described as ${}_iF$ in the enumeration. Define the function g as $g(k) = {}_kF\text{-}f(k) + 1$, for all $k \in \mathbb{N}$. As $g \in \phi$, the enumeration ${}_1F, {}_2F, \ldots, {}_jF, \ldots$ contains ${}_jF$ such that ${}_jF\text{-}f$ coincides with g, for some $j \geq 1$. Then, ${}_jF\text{-}f(j) = g(j) = {}_jF\text{-}f(j) + 1$, which is a contradiction. Thus, no Γ-formalized algorithm computes g.

Apart from uncomputable functions, there also exist undecidable problems, which cannot be decided by any algorithm. More regretfully and surprisingly, the theory of *decidability* has even proved that there will never exist algorithms that decide problems with genuine significance in

computer science as a whole. For instance, it is undecidable whether a program always halts; that is, the existence of a general algorithm that decides this problem is ruled out once and for all.

However, even if we restrict our attention only to decidable problems and take a closer look at them, we find out that they significantly differ in terms of their *time and space computational complexity*. Indeed, two decidable problems may differ so the computation of one problem takes reasonable amount of time while the computation of the other does not—that is, compared to the first problem, the other problem is considered as *intractable* because its solution requires an unmanageable amount of time. Thus, apart from theoretically oriented investigation, this study of computational complexity is obviously crucially important to most application-oriented areas of computer science as well.

2.3 Synopsis of the Book

In this section, we link all the terminology introduced in this chapter to the rest of this book and, thereby, make its synopsis. The book is divided into Sections I through V. Section I is concluded by this chapter. The others are outlined next.

In this book, the most important language-defining models are finite automata, CFGs, and TMs, which define the language families denoted by ${}_{FA}\Phi$, ${}_{CFG}\Phi$, and ${}_{TM}\Phi$, respectively. Accordingly, Sections II, III, and IV cover models, applications, and properties concerning ${}_{FA}\Phi$, ${}_{CFG}\Phi$, and ${}_{TM}\Phi$, respectively. Each of these sections consists of three chapters.

Section II

Section II consists of Chapters 3 through 5. Chapter 3 introduces finite automata and regular expressions as the basic language models that characterize ${}_{FA}\Phi$, whose languages are usually referred to as regular languages. Chapter 4 demonstrates how to apply these models to text processing. Specifically, based upon them, it builds up lexical analyzers. Finally, Chapter 5 establishes several properties, including closure properties, which are helpful to prove or disprove that some languages are regular.

Section III

Section III consists of Chapters 6 through 8. Chapter 6 defines already mentioned CFGs, which characterize ${}_{CFG}\Phi$—the family of context-free languages. In addition, it defines pushdown automata, which also characterize ${}_{CFG}\Phi$, so these grammars and automata are equivalent. Chapter 7 applies these models to syntax analysis. It explains how to describe programming language syntax by CFGs and, then, convert these grammars to efficient syntax analyzers that act as pushdown automata. In many respects, Chapter 8 parallels Chapter 5; however, Chapter 8 obviously studies language properties in terms of ${}_{CFG}\Phi$. That is, it establishes many properties concerning CFGs and explains how to use them to answer certain important questions concerning them.

Section IV

While Sections II and III make use of various rewriting systems as language-defining models, Section IV uses them primarily as computational models. It consists of Chapters 9 through 11. Chapter 9 defines already mentioned TMs. Based upon them, Chapter 10 outlines the theory of computability, decidability, and computational complexity. To link these machines to the theory of formal languages, Chapter 11 considers them as language-accepting models, defines their grammatical

counterparts, and establishes the relation between several subfamilies of ${}_{TM}\Phi$. Perhaps most importantly, it states that

$$ {}_{fin}\Phi \subset {}_{FA}\Phi \subset {}_{CFG}\Phi \subset {}_{TM}\Phi \subset {}_{all}\Phi $$

Section V

The purpose of the final one-chapter section is fourfold. First, it summarizes this book. Second, it places all its material into a historical and bibliographical context. Third, it selects several modern and advanced topics, omitted in this introductory book, and gives their overview. Finally, Section V suggests further reading for the serious student.

Exercises

1. Let Σ be an alphabet. Let $x = aaababababbb$, where $a, b \in \Sigma$. Determine *prefix*(x), *suffix*(x), and *substring*(x).
2. Give a nonempty string x such that $x^i = reversal(x)^i$, for all $i \geq 0$.

3 S. Let $L = \{a^n | \ n \geq 2\}$ be a language over an alphabet, Σ. Determine ~L with (i) $\Sigma = \{a\}$ and (ii) $\Sigma = \{a, b\}$.

4. Let Σ be an alphabet. Prove that every $x \in \Sigma^*$ satisfies (a) through (c), given next.
   a. *prefix*(x) $\subseteq$ *substring*(x)
   b. *suffix*(x) $\subseteq$ *substring*(x)
   c. $\{\varepsilon, x\} \subseteq$ *prefix*(x) $\cap$ *suffix*(x) $\cap$ *substring*(x)
5. Select some common components of your favorite programming language. Specify them by using the notions introduced in Section 2.1, such as various language operations. For instance, consider integers in C and specify them by using such simple operations as concatenation and closure.

6 S. Formalize the usual dictionary order as a *lexicographic order* based upon a linear order, defined in Exercise 8 in Chapter 1. Write a program that implements the lexicographic order. Test this program on a large file of English words.

7. Let Σ be an alphabet. Prove or disprove each of the following four statements.
   a. For all $i \geq 0$, $\varepsilon^i = \varepsilon$.
   b. For all $x \in \Sigma^*$, $x\varepsilon = \varepsilon x = x$.
   c. For all $x \in \Sigma^*$, $x^i x^j = x^j x^i = x^{i+j}$.
   d. For all $x, y \in \Sigma^*$, *reversal*(xy) = *reversal*(y)*reversal*(x).
8. Consider the language $L = \{011, 111, 110\}$. Determine *reversal*(L), *prefix*(L), *suffix*(L), L^2, L^*, and L^+.
9. Prove that every language L satisfies $L\{\varepsilon\} = \{\varepsilon\}L = L$, $L\varnothing = \varnothing L = \varnothing$, $L^+ = LL^* = L^*L$, and $L^* = L^+ \cup \{\varepsilon\}$.
10. Let Σ be an alphabet. Determine all languages L over Σ satisfying $L^* = L$; for instance, Σ^* is one of them.
11. Let Σ be an alphabet. Prove that the family of all finite languages over Σ is countable, but the family *power*(Σ^*)—that is, the family of all languages over Σ—is not countable.
12. Let K and L be two finite languages over $\{0, 1\}$ defined as $K = \{00, 11\}$ and $L = \{0, 00\}$. Determine KL, $K \cup L$, $K \cap L$, and $K - L$.

13 S. Prove or disprove that the following two equations hold for any two languages J and K.
   a. $(J \cup K)^* = (J^*K^*)^*$
   b. $(JK \cup K)^* = J(KJ \cup J)^*$
14. Consider each of the following equations. Prove or disprove that it holds for any three languages J, K, and L.
   a. $(JK)L = J(KL)$
   b. $(J \cup K)L = JL \cup KL$
   c. $L(J \cup K) = LJ \cup LK$

d. $(J \cap K)L = JL \cap KL$
e. $L(J \cap K) = LJ \cap LK$
f. $L(J - K) = LJ - LK$

15. Let Σ be an alphabet. In terms of formal languages, *DeMorgan's law* says that $\sim(\sim K \cup \sim L) = K \cap L$ for any two languages K and L over Σ. Prove this law.

16 S. Recall that a rewriting system is a pair, $M = (\Sigma, R)$, where Σ is an alphabet, and R is a finite relation on Σ^* (see Definition 2.2). Furthermore, the rewriting relation over Σ^* is denoted by $\Rightarrow$ and defined so that for every $u, v \in \Sigma^*$, $u \Rightarrow v$ in M iff there exist $x \to y \in R$ and $w, z \in \Sigma^*$ such that $u = wxz$ and $v = wyz$. For every $n \geq 0$, the n-fold product of $\Rightarrow$ is denoted by $\Rightarrow^n$. Determine $m \in {}_0\mathbb{N}$ satisfying for all $u, v \in \Sigma^*$ and $n \geq 0$, $u \Rightarrow^n v$ in M implies $|v| \leq nm|u|$.

17. Let $G = (\Sigma, R)$ be a rewriting system (see Definition 2.2), $v, w \in \Sigma^*$, and $v \Rightarrow^* w$. If w cannot be rewritten by any rule from R—in other words, $w \Rightarrow z$ is false for any $z \in \Sigma^*$, then $v \Rightarrow^* w$ is said to be *terminating*, symbolically written as $v \,{}_t\!\Rightarrow^* w$. Let $S \in \Sigma$ be a special start symbol. Set ${}_tL(G) = \{w \in \Sigma^* | \, S \,{}_t\!\Rightarrow^* w\}$.

Consider each of the following languages over $\{a, b, c\}$. Construct a rewriting system G such that ${}_tL(G)$ coincides with the language under consideration.

a. $\{a^i b a^i b a^j | \, i, j \geq 1\}$
b. $\{a^i b a^i | \, i \geq 0\}$
c. $\{a^i b^i a^{2i} | \, i \geq 0\}$
d. $\{a^i b^i c^i | \, i \geq 0 \text{ and } j \geq 1\}$
e. $\{b^i a^j b^i c^j | \, i, j \geq 0\}$
f. $\{b^i a^k b^k a^l b^k | \, i, j \geq 0, i = k \text{ or } l = k\}$
g. $\{x | \, x \in \{a, b, c\}^*, occur(x, a) > occur(x, b) > occur(x, c)\}$
h. $\{x | \, x \in \{a, b, c\}^*, occur(x, a) = occur(x, b) = occur(x, c)\}$
i. $\{a^i | \, i = 2^n, i \geq 0\}$
j. $\{xx | \, x \in \{a, b\}^*\}$

18. Return to Example 2.7.
a. Modify CFGs so S is a finite language, not a single symbol. Formalize this modification. Are the CFGs modified in this way as powerful as CFGs?
b. Rephrase and solve (a) in terms of LGs.

19 S. Is every formal language defined by a language-defining rewriting system? Justify your answer.

20 S. Define the notion of a rewriting system that acts as computers of functions over Σ^*, where Σ is any alphabet. Then, based on this definition, introduce a specific rewriting system that acts as a computer of the function f over $\{\mathbf{0}, \mathbf{1}, \vee, \wedge\}^*$ defined for all $x \in \Sigma^*$ and $i \in \{\mathbf{0}, \mathbf{1}\}$, by this equivalence: $f(x) = i$ iff x is a prefix Polish expression whose logical value is i.

21 S. Consider Example 2.5 and Definition 2.6.
a. Consider M in Example 2.5. Demonstrate that M rewrites strings nondeterministically in terms of Definition 2.6.
b. Modify Example 2.5 as follows. Redefine $M = (\Sigma, R)$ so $\Sigma = {}_{English}\Delta \cup \{\cdot, -, \#\}$ and $R = \{a \to \mu(a)\# | \, a \in {}_{English}\Delta\}$. Define the function ν from ${}_{English}\Delta^*$ to $\{\cdot, -, \#\}^*$ as $\nu = \{(s, t) | \, s \Rightarrow^* t \text{ in } M, s \in {}_{English}\Delta^*, t \in \{\cdot, -, \#\}^*\}$; for instance, ν contains $(SOS, \cdots \# - - - \# \cdots \#)$. Construct deterministic rewriting systems D and E that define ν and $inverse(\nu)$, respectively.

Solutions to Selected Exercises

3. If $\Sigma = \{a\}$, $\sim L = \{\varepsilon, a\}$. If $\Sigma = \{a, b\}$, $\sim L = \{a, b\}^* - \{a\}\{a\}^+$.
6. Let Σ be a set, and let β be a linear order on Σ. We extend β to Σ^* so that for any $x, y \in \Sigma^*$, $x \beta y$ if $x \in prefixes(y) - \{y\}$, or for some $k \geq 1$ such that $|x| > k$ and $|y| > k$, $prefix(x, k - 1) = prefix(y, k - 1)$ and $symbol(x, k) \beta symbol(y, k)$. This extended definition of β is referred to as the *lexicographic order* β on Σ^*. Take, for instance, Σ as the English alphabet and β as its alphabetical order. Then, the lexical order β extended in the above-mentioned way represents the usual dictionary order on Σ^*.

13. To disprove (ii), take J and K as any two languages such that $\varepsilon \notin J$ and $\varepsilon \in K$; then, $\varepsilon \in (JK \cup K)^*$, but $\varepsilon \notin J(KJ \cup J)^*$.
16. Take m as the minimal nonnegative integer satisfying $m \geq |x| - |y|$, for all $x \to y \in R$. By induction on $n \geq 0$, prove that for all $u, v \in \Sigma^*$, $u \Rightarrow^n v$ in M implies $|v| \leq nm|u|$.
19. In the conclusion of Section 2.2, there is an explanation why some functions cannot be computed by any procedure, which has a finite description. Make use of a similar argument to prove that some formal languages are not defined by any language-defining rewriting systems.
20. Let Σ be an alphabet. Without any loss of generality, suppose that Σ and $\{\triangleright, \triangleleft, \blacktriangleright, \blacksquare\}$ are disjoint. Let $M = (W, R)$ be a rewriting system, where $W = \Sigma \cup \{\triangleright, \triangleleft, \blacktriangleright, \blacksquare\}$, and R is a finite set of rules of the form $x \to y$, where either $x, y \in \{\triangleright\}X$ or $x, y \in X$ or $x, y \in X\{\triangleleft\}$ with $X = \Sigma^*\{\blacktriangleright, \blacksquare\}\Sigma^*$. Let f be a function over Σ^*. *M computes f* iff this equivalence holds true: $f(x) = y$ iff $\triangleright\blacktriangleright x\triangleleft \Rightarrow^* \triangleright\blacksquare y\triangleleft$ in M, where $x, y \in \Sigma^*$.

    Define the rewriting system $M = (W, R)$ with $W = \{\mathbf{0}, \mathbf{1}, \vee, \wedge, \triangleright, \triangleleft, \blacktriangleright, \blacksquare\}$ and

$$
\begin{aligned}
R = {} & \{\blacktriangleright \vee ij \to \blacktriangleright k \mid i, j, k \in \{\mathbf{0}, \mathbf{1}\}, k = \mathbf{0} \text{ iff } i = j = \mathbf{0}\} \\
& \cup \{\blacktriangleright \wedge ij \to \blacktriangleright k \mid i, j, k \in \{\mathbf{0}, \mathbf{1}\}, k = \mathbf{1} \text{ iff } i = j = \mathbf{1}\} \\
& \cup \{\blacktriangleright i \to i\blacktriangleright \mid i \in \{\mathbf{0}, \mathbf{1}, \vee, \wedge\}\} \\
& \cup \{i\blacktriangleright \to \blacktriangleright i \mid i \in \{\mathbf{0}, \mathbf{1}, \vee, \wedge\}\} \\
& \cup \{\blacktriangleright i\triangleleft \to \blacksquare i\triangleleft \mid i \in \{\mathbf{0}, \mathbf{1}\}\}
\end{aligned}
$$

    Observe that for all $x \in \Sigma^*$ and $i \in \{\mathbf{0}, \mathbf{1}\}$, $f(x) = i$ iff x is a prefix Polish expression whose logical value is i. For instance, $\triangleright\blacktriangleright\wedge\vee\mathbf{100}\triangleleft \Rightarrow \triangleright\wedge\blacktriangleright\vee\mathbf{100}\triangleleft \Rightarrow \triangleright\wedge\blacktriangleright\mathbf{10}\triangleleft \Rightarrow \triangleright\blacktriangleright\wedge\mathbf{10}\triangleleft \Rightarrow \triangleright\blacktriangleright\mathbf{0}\triangleleft \Rightarrow \triangleright\blacksquare\mathbf{0}\triangleleft$.
21. Consider (a). Notice that, for instance, $SO\underline{S} \Rightarrow SO \cdots$ and $\underline{S}OS \Rightarrow \cdots OS$ in M, so M rewrites strings nondeterministically in terms of Definition 2.6.

    Consider (b). Define $D = ({}_D\Sigma, {}_DR)$ with ${}_D\Sigma = {}_{English}\Delta \cup \{S, \cdot, -, \#, \$\}$ and ${}_DR = \{Sa \to \mu(a)S \mid a \in {}_{English}\Delta\} \cup \{S\$ \to \$\}$. Define $T(D) = \{(s, t) \mid Ss\$ \Rightarrow^* t\$, s \in {}_{English}\Delta^*, t \in \{\cdot, -, \#\}^*\}$. Prove that $T(D) = \nu$.

    Define $E = ({}_E\Sigma, {}_ER)$ with ${}_E\Sigma = {}_D\Sigma$ and ${}_ER = \{S\mu(a) \to aS \mid a \in {}_{English}\Delta\} \cup \{S\$ \to \$\}$. Define $T(E) = \{(s, t) \mid Ss\$ \Rightarrow^* t\$, s \in \{\cdot, -, \#\}^*, t \in {}_{English}\Delta^*\}$. Prove that $T(E) = inverse(\nu)$.