4.9 Formal Languages

Languages are introduced in Section 1.2.3. A **language** is a set of strings over a finite set Σ called an **alphabet**. Σ^* is the language of all strings over Σ including the **empty string** ϵ, which has zero length. The empty string has the property that for an arbitrary string w, $\epsilon w = w = w\epsilon$. Σ^+ is the set Σ^* without the empty string.

In this section we introduce grammars for languages, rules for **rewriting strings** through the substitution of substrings. A **grammar** consists of alphabets $\mathcal{T}$ and $\mathcal{N}$ of terminal and non-terminal symbols, respectively, plus a set of rules $\mathcal{R}$ for rewriting strings. Below we define four types of language in terms of their grammars: the phrase-structure, context-sensitive, context-free, and regular grammars.

The role of grammars is best illustrated with an example for a small fragment of English. Consider a grammar G whose non-terminals $\mathcal{N}$ contain a start symbol S denoting a generic sentence and NP and VP denoting generic noun and verb phrases, respectively. In turn, assume that $\mathcal{N}$ also contains non-terminals for adjectives and adverbs, namely AJ and AV. Thus, $\mathcal{N} = \{\text{S}, \text{NP}, \text{VP}, \text{AJ}, \text{AV}, \text{N}, \text{V}\}$. We allow the grammar to have the following words as terminals: $\mathcal{T} = \{bob, alice, duck, big, smiles, quacks, loudly\}$. Here bob, $alice$, and $duck$ are nouns, big is an adjective, $smiles$ and $quacks$ are verbs, and $loudly$ is an adverb. In our fragment of English a sentence consists of a noun phrase followed by a verb phrase, which we denote by the rule S $\rightarrow$ NP VP. This and the other rules $\mathcal{R}$ of the grammar are shown below. They include rules to map non-terminals to terminals, such as N $\rightarrow$ bob

$$
\begin{array}{lllllllll}
\text{S} & \rightarrow & \text{NP VP} & \text{N} & \rightarrow & bob & \text{V} & \rightarrow & smiles \\
\text{NP} & \rightarrow & \text{N} & \text{N} & \rightarrow & alice & \text{V} & \rightarrow & quacks \\
\text{NP} & \rightarrow & \text{AJ N} & \text{N} & \rightarrow & duck & \text{AV} & \rightarrow & loudly \\
\text{VP} & \rightarrow & \text{V} & \text{AJ} & \rightarrow & big & & & \\
\text{VP} & \rightarrow & \text{V AV} & & & & & &
\end{array}
$$

With these rules the following strings (sentences) can be generated: $bob\ smiles$; $big\ duck\ quacks\ loudly$; and $alice\ quacks$. The first two sentences are acceptable English sentences, but the third is not if we interpret $alice$ as a person. This example illustrates the need for rules that limit the rewriting of non-terminals to an appropriate context of surrounding symbols.

Grammars for formal languages generalize these ideas. Grammars are used to interpret programming languages. A language is translated and given meaning through a series of steps the first of which is **lexical analysis**. In lexical analysis symbols such as a, l, i, c, e are grouped into tokens such as $alice$, or some other string denoting $alice$. This task is typically done with a finite-state machine. The second step in translation is **parsing**, a process in which a tokenized string is associated with a series of **derivations** or applications of the rules of a grammar. For example, $big\ duck\ quacks\ loudly$, can be produced by the following sequence of derivations: S $\rightarrow$ NP VP; NP $\rightarrow$ AJ N; AJ $\rightarrow$ big; N $\rightarrow$ $duck$; VP $\rightarrow$ V AV; V $\rightarrow$ $quacks$; AV $\rightarrow$ $loudly$.

In his exploration of models for natural language, Noam Chomsky introduced four language types of decreasing expressibility, now called the **Chomsky hierarchy**, in which each language is described by the type of grammar generating it. These languages serve as a basis for the classification of programming languages. The four types are the phrase-structure languages, the context-sensitive languages, the context-free languages, and the regular languages.

There is an exact correspondence between each of these types of languages and particular machine architectures in the sense that for each language type T there is a machine architecture A recognizing languages of type T and for each architecture A there is a type T such that all languages recognized by A are of type T. The correspondence between language and architecture is shown in the following table, which also lists the section or problem where the result is established. Here the **linear bounded automaton** is a Turing machine in which the number of tape cells that are used is linear in the length of the input string.

Level	*Language Type*	*Machine Type*	*Proof Location*
0	phrase-structure	Turing machine	Section 5.4
1	context-sensitive	linear bounded automaton	Problem 4.36
2	context-free	nondet. pushdown automaton	Section 4.12
3	regular	finite-state machine	Section 4.10

We now give formal definitions of each of the grammar types under consideration.

4.9.1 Phrase-Structure Languages

In Section 5.4 we show that the phrase-structure grammars defined below are exactly the languages that can be recognized by Turing machines.

DEFINITION 4.9.1 *A* **phrase-structure grammar** *G is a four-tuple $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \textsc{s})$ where $\mathcal{N}$ and $\mathcal{T}$ are disjoint alphabets of* **non-terminals** *and* **terminals**, *respectively. Let $V = \mathcal{N} \cup \mathcal{T}$. The* **rules** *$\mathcal{R}$ form a finite subset of $V^+ \times V^*$ (denoted $\mathcal{R} \subseteq V^+ \times V^*$) where for every rule $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}$, $\boldsymbol{a}$ contains at least one non-terminal symbol. The symbol $\textsc{s} \in \mathcal{N}$ is the* **start symbol**.

If $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}$ we write $\boldsymbol{a} \rightarrow \boldsymbol{b}$. If $\boldsymbol{u} \in V^+$ and $\boldsymbol{a}$ is a contiguous substring of $\boldsymbol{u}$, then $\boldsymbol{u}$ can be replaced by the string $\boldsymbol{v}$ by substituting $\boldsymbol{b}$ for $\boldsymbol{a}$. If this holds, we write $\boldsymbol{u} \Rightarrow_G \boldsymbol{v}$ and call it an **immediate derivation**. *Extending this notation, if through a sequence of immediate derivations (called a* **derivation**) *$\boldsymbol{u} \Rightarrow_G \boldsymbol{x}_1, \boldsymbol{x}_1 \Rightarrow_G \boldsymbol{x}_2, \cdots, \boldsymbol{x}_n \Rightarrow_G \boldsymbol{v}$ we can transform $\boldsymbol{u}$ to $\boldsymbol{v}$, we write $\boldsymbol{u} \overset{*}{\Rightarrow}_G \boldsymbol{v}$ and say that $\boldsymbol{v}$* **derives** *from $\boldsymbol{u}$. If the rules $\mathcal{R}$ contain $(\boldsymbol{a}, \boldsymbol{a})$ for all $\boldsymbol{a} \in V^+$, the relation $\overset{*}{\Rightarrow}_G$ is called the* **transitive closure** *of the relation $\Rightarrow_G$ and $\boldsymbol{u} \overset{*}{\Rightarrow}_G \boldsymbol{u}$ for all $\boldsymbol{u} \in V^*$.*

The **language** *$L(G)$ defined by the grammar G is the set of all terminal strings that can be derived from the start symbol $\textsc{s}$; that is,*

$$L(G) = \{\boldsymbol{u} \in \mathcal{T}^* \mid \textsc{s} \overset{*}{\Rightarrow}_G \boldsymbol{u}\}$$

When the context is clear we drop the subscript G in $\Rightarrow_G$ and $\overset{*}{\Rightarrow}_G$. These definitions are best understood from an example. In all our examples we use letters in SMALL CAPS to denote non-terminals and letters in *italics* to denote terminals, except that ϵ, the empty letter, may

$$
\begin{array}{llllllllllll}
a) & \text{S} & \to & a\text{SBC} & d) & a\text{B} & \to & ab & g) & c\text{C} & \to & cc \\
b) & \text{S} & \to & a\text{BC} & e) & b\text{B} & \to & bb & & & & \\
c) & \text{CB} & \to & \text{BC} & f) & b\text{C} & \to & bc & & & &
\end{array}
$$

Clearly the string $aa\text{BCBC}$ can be rewritten as $aa\text{BBCC}$ using rule (c), that is, $aa\text{BCBC} \Rightarrow aa\text{BBCC}$. One application of (d), one of (e), one of (f), and one of (g) reduces it to the string $aabbcc$. Since one application of (a) and one of (b) produces the string $aa\text{BBCC}$, it follows that the language $L(G_1)$ contains $aabbcc$.

Similarly, two applications of (a) and one of (b) produce $aaa\text{BCBCBC}$, after which three applications of (c) produce the string $aaa\text{BBBCCC}$. One application of (d) and two of (e) produce $aaabbb\text{CCC}$, after which one application of (f) and two of (g) produces $aaabbbccc$. In general, one can show that $L(G_1) = \{a^n b^n c^n \mid n \geq 1\}$. (See Problem 4.38.)

4.9.2 Context-Sensitive Languages

The context-sensitive languages are exactly the languages accepted by linear bounded automata, nondeterministic Turing machines whose tape heads visit a number of cells that is a constant multiple of the length of an input string. (See Problem 4.36.)

DEFINITION 4.9.2 *A* **context-sensitive grammar** G *is a phrase structure grammar* $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ *in which each rule* $(\boldsymbol{a}, \boldsymbol{b}) \in \mathcal{R}$ *satisfies the condition that* $\mathbf{b}$ *has no fewer characters than does* $\boldsymbol{a}$, *namely,* $|\boldsymbol{a}| \leq |\boldsymbol{b}|$. *The languages defined by context-sensitive grammars are called* **context-sensitive languages** *(CSL).*

Each rule of a context-sensitive grammar maps a string to one that is no shorter. Since the left-hand side of a rule may have more than one character, it may make replacements based on the context in which a non-terminal is found. Examples of context-sensitive languages are given in Problems 4.38 and 4.39.

4.9.3 Context-Free Languages

As shown in Section 4.12, the context-free languages are exactly the languages accepted by pushdown automata.

DEFINITION 4.9.3 *A* **context-free grammar** $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$ *is a context-sensitive grammar in which each rule in* $\mathcal{R} \subseteq \mathcal{N} \times V^*$ *has a single non-terminal on the left-hand side. The languages defined by context-free grammars are called* **context-free languages** *(CFL).*

Each rule of a context-free grammar maps a non-terminal to a string over V^* without regard to the context in which the non-terminal is found because the left-hand side of each rule consists of a single non-terminal.

EXAMPLE 4.9.2 *Let* $\mathcal{N}_2 = \{\text{S}, \text{A}\}$, $\mathcal{T}_2 = \{\epsilon, a, b\}$, *and* $\mathcal{R}_2 = \{\text{S} \to a\text{S}b, \text{S} \to \epsilon\}$. *Then the*

EXAMPLE 4.9.3 *Consider the grammar G_3 with the following rules and the implied terminal and non-terminal alphabets:*

$a)$	$\text{S} \rightarrow$	$c\text{MN}c$		$d)$	$\text{N} \rightarrow$	$b\text{N}b$
$b)$	$\text{M} \rightarrow$	$a\text{M}a$		$e)$	$\text{N} \rightarrow$	c
$c)$	$\text{M} \rightarrow$	c				

G_3 is context-free and generates the language $L(G_3) = \{ca^n ca^n cb^m cb^m c \mid n, m \geq 0\}$, as is easily shown.

Context-free languages capture important aspects of many programming languages. As a consequence, the parsing of context-free languages is an important step in the parsing of programming languages. This topic is discussed in Section 4.11.

4.9.4 Regular Languages

DEFINITION 4.9.4 *A* **regular grammar** *G is a context-free grammar $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \text{S})$, where the right-hand side is either a terminal or a terminal followed by a non-terminal. That is, its rules are of the form $\text{A} \rightarrow a$ or $\text{A} \rightarrow b\text{C}$. The languages defined by regular grammars are called* **regular languages**.

Some authors define a regular grammar to be one whose rules are of the form $\text{A} \rightarrow a$ or $\text{A} \rightarrow b_1 b_2 \cdots b_k \text{C}$. It is straightforward to show that any language generated by such a grammar can be generated by a grammar of the type defined above.

The following grammar is regular.

EXAMPLE 4.9.4 *Consider the grammar $G_4 = (\mathcal{N}_4, \mathcal{T}_4, \mathcal{R}_4, \text{S})$ where $\mathcal{N}_4 = \{\text{S}, \text{A}, \text{B}\}$, $\mathcal{T}_4 = \{0,1\}$ and $\mathcal{R}_4$ consists of the rules given below.*

$a)$	$\text{S} \rightarrow$	0A		$d)$	$\text{B} \rightarrow$	0A
$b)$	$\text{S} \rightarrow$	0		$e)$	$\text{B} \rightarrow$	0
$c)$	$\text{A} \rightarrow$	1B				

It is straightforward to see that the rules a) $\text{S} \rightarrow 0$, b) $\text{S} \rightarrow 01\text{B}$, c) $\text{B} \rightarrow 0$, and d) $\text{B} \rightarrow 01\text{B}$ generate the same strings as the rules given above. Thus, the language G_4 contains the strings $0, 010, 01010, 0101010, \dots$, that is, strings of the form $(01)^k 0$ for $k \geq 0$. Consequently $L(G_4) = (01)^* 0$. A formal proof of this result is left to the reader. (See Problem 4.44.)

4.10 Regular Language Recognition

As explained in Section 4.1, a deterministic finite-state machine (DFSM) M is a five-tuple $M = (\Sigma, Q, \delta, s, F)$, where Σ is the input alphabet, Q is the set of states, $\delta : Q \times \Sigma \mapsto Q$ is the next-state function, s is the initial state, and F is the set of final states. A nondeterministic FSM (NFSM) is similarly defined except that δ is a next-set function $\delta : Q \times \Sigma \mapsto 2^Q$. In other words, in an NFSM there may be more than one next state for a given state and input.

THEOREM 4.10.1 *The languages generated by regular grammars and recognized by finite-state machines are the same.*

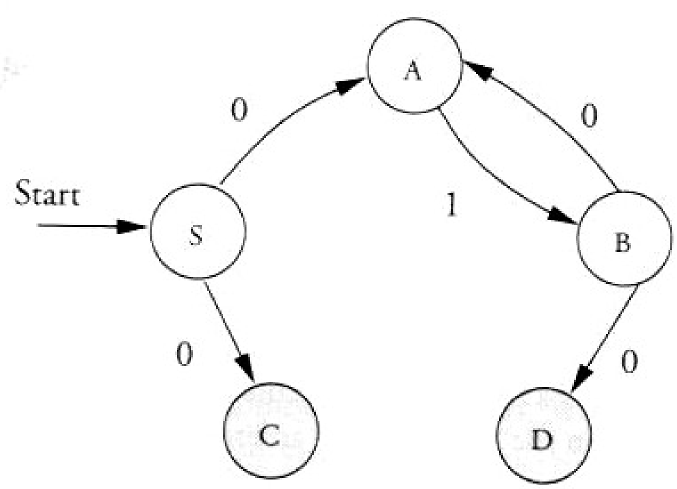
Proof Given a regular grammar G, we construct a corresponding NFSM M that accepts exactly the strings generated by G. Similarly, given a DFSM M we construct a regular grammar G that generates the strings recognized by M.

Let $G = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathrm{S})$ be a regular grammar. Then, its rules $\mathcal{R}$ are of the form $\mathrm{A} \rightarrow a$ or $\mathrm{A} \rightarrow b\mathrm{C}$. We first produce a grammar G' that generates the same language by replacing each rule of the form $\mathrm{A} \rightarrow a$ with two rules of the form $\mathrm{A} \rightarrow a\mathrm{B}$ and $\mathrm{B} \rightarrow \epsilon$. Here B is a new non-terminal particular to the rule $\mathrm{A} \rightarrow a\mathrm{B}$.

Every derivation $\mathrm{S} \overset{*}{\Rightarrow}_G \boldsymbol{w}$, $\boldsymbol{w} \in \mathcal{T}^*$, corresponds to a derivation $\mathrm{S} \overset{*}{\Rightarrow}_{G'} \boldsymbol{w}\mathrm{B}$ in which B is a new non-terminal added to G along with the new rule $\mathrm{B} \rightarrow \epsilon$ to form G'. Hence, the strings generated by G and G' are the same.

Now construct an NFSM $M_{G'}$ whose states correspond to the non-terminals of this new regular grammar and whose input alphabet is its set of terminals. Let the state associated with S be the start state of $M_{G'}$. Let there be a transition from state A to state B on input a if there is a rule $\mathrm{A} \rightarrow a\mathrm{B}$ in G'. Let a state B be a final state if there is a rule of the form $\mathrm{B} \rightarrow \epsilon$ in G'. Clearly, every derivation of a string $\boldsymbol{w}$ in $L(G')$ corresponds to a path in M that begins in the start state and ends on a final state. Hence, $\boldsymbol{w}$ is accepted by $M_{G'}$. On the other hand, if a string $\boldsymbol{w}$ is accepted by $M_{G'}$, given the one-to-one correspondence between edges and rules, there is a derivation of $\boldsymbol{w}$ from S in G'. Thus, the strings generated by G and the strings accepted by $M_{G'}$ are the same.

Now assume we are given a DFSM M that accepts a language L_M. Create a grammar G_M whose non-terminals correspond to the states of M and whose start symbol is the start state of M. G_M has a rule of the form $q_1 \rightarrow aq_2$ if M makes a transition from state q_1 to q_2 on input a. If state q is a final state of M, also add the rule $q \rightarrow \epsilon$. If a string is accepted by M, that is, it causes M to move to a final state, then G_M generates the same string. Since G_M generates only strings of this kind, every language accepted by M is generated by a regular grammar. ■



A simple example illustrates the construction of an NFSM from a regular grammar. Consider the grammar G_4 of Example 4.9.4. A new grammar G_4' is constructed with the following rules: a) $\mathrm{S} \to 0\mathrm{A}$, b) $\mathrm{S} \to 0\mathrm{C}$, c) $\mathrm{C} \to \epsilon$, d) $\mathrm{A} \to 1\mathrm{B}$, e) $\mathrm{B} \to 0\mathrm{A}$, f) $\mathrm{B} \to 0\mathrm{D}$, and g) $\mathrm{D} \to \epsilon$. Figure 4.27 (page 185) shows an NFSM that accepts the language generated by this grammar. A DFSM recognizing the same language can be obtained by invoking the construction of Theorem 4.2.1.

4.11 Parsing Context-Free Languages

Parsing is the process of deducing those rules of a grammar G (a **derivation**) that generates a terminal string w. The first rule must have the start symbol S on the left-hand side. In this section we give a brief introduction to the parsing of context-free languages, a topic central to the parsing of programming languages. The reader is referred to a textbook on compilers for more detail on this subject. (See, for example, [11] and [100].) The concepts of Boolean matrix multiplication and transitive closure are used in this section, topics that are covered in Chapter 6.

Generally a string w has many derivations. This is illustrated by the context-free grammar G_3 defined in Example 4.9.3 and described below.

EXAMPLE 4.11.1 $G_3 = (\mathcal{N}_3, \mathcal{T}_3, \mathcal{R}_3, \mathrm{S})$*, where* $\mathcal{N}_3 = \{\mathrm{S}, \mathrm{M}, \mathrm{N}\}$, $\mathcal{T}_3 = \{\mathrm{A}, \mathrm{B}, \mathrm{C}\}$ *and* $\mathcal{R}_3$ *consists of the rules below:*

$a)$	$\mathrm{S} \to c\mathrm{MN}c$		$d)$	$\mathrm{N} \to b\mathrm{N}b$
$b)$	$\mathrm{M} \to a\mathrm{M}a$		$e)$	$\mathrm{N} \to c$
$c)$	$\mathrm{M} \to c$			

The string $caacaabcbc$ can be derived by applying rules (a), (b) twice, (c), (d) and (e) to produce the following derivation:

$$\begin{aligned} \mathrm{S} \;&\Rightarrow\; c\mathrm{MN}c \;\Rightarrow\; ca\mathrm{M}a\mathrm{N}c \;\Rightarrow\; ca^2\mathrm{M}a^2\mathrm{N}c \\ &\Rightarrow\; ca^2ca^2\mathrm{N}c \;\Rightarrow\; ca^2ca^2b\mathrm{N}bc \;\Rightarrow\; ca^2ca^2bcbc \end{aligned} \tag{4.2}$$

The same string can be obtained by applying the rules in the following order: (a), (d), (e), (b) twice, and (c). Both derivations are described by the **parse tree** of Fig. 4.28. In this tree each instance of a non-terminal is rewritten using one of the rules of the grammar. The order of the descendants of a non-terminal vertex in the parse tree is the order of the corresponding symbols in the string obtained by replacing this non-terminal. The string ca^2ca^2bcbc, the **yield** of this parse tree, is the terminal string obtained by visiting the leaves of this tree in a left-to-right order. The **height** of the parse tree is the number of edges on the longest path (having the most edges) from the root (associated with the start symbol) to a terminal symbol. A **parser** for a language $L(G)$ is a program or machine that examines a string and produces a derivation of the string if it is in the language and an error message if not.

Because every string generated by a context-free grammar has a derivation, it has a cor-

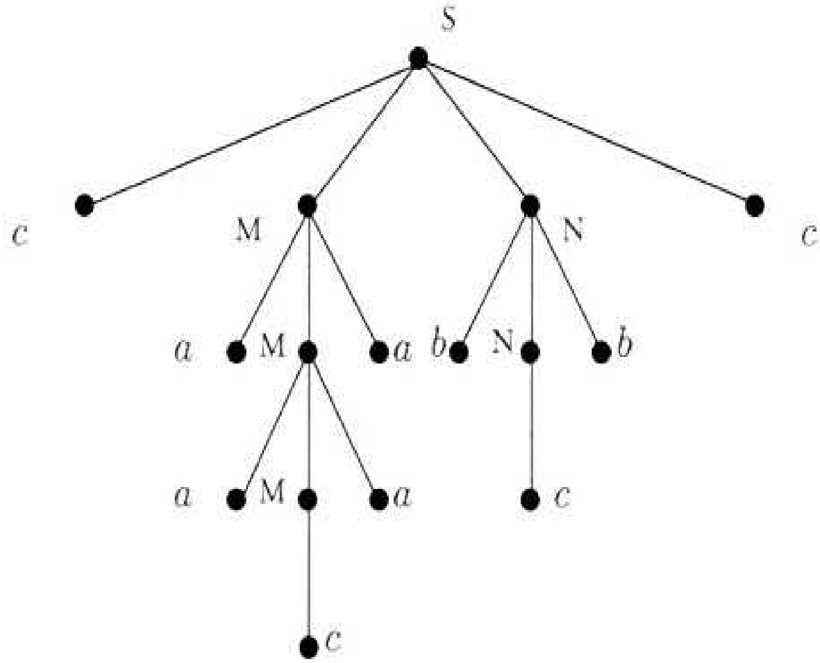


Figure 4.28 A parse tree for the grammar G_3.

associated with terminals and then traversing the remaining vertices in a depth-first manner (visit the first descendant of a vertex before visiting its siblings), assuming that descendants of a vertex are ordered from left to right. When a vertex is visited, apply the rule associated with that vertex in the tree. The derivation given in (4.2) is leftmost.

Not only can some strings in a context-free language have multiple derivations, but in some languages they have multiple parse trees. Languages containing strings with more than one parse tree are said to be **ambiguous languages**. Otherwise languages are **non-ambiguous**.

Given a string that is believed to be generated by a grammar, a **compiler** attempts to parse the string after first scanning the input to identify letters. If the attempt fails, an error message is produced. Given a string generated by a context-free grammar, can we guarantee that we can always find a derivation or parse tree for that string or determine that none exists? The answer is yes, as we now show.

To demonstrate that every CFL can be parsed, it is convenient first to convert the grammar for such a language to Chomsky normal form.

DEFINITION 4.11.1 *A context-free grammar G is in* **Chomsky normal form** *if every rule is of the form* $\mathrm{A} \to \mathrm{BC}$ *or* $\mathrm{A} \to u$, $u \in \mathcal{T}$ *except if* $\epsilon \in L(G)$, *in which case* $\mathrm{S} \to \epsilon$ *is also in the grammar.*

We now give a procedure to convert an arbitrary context-free grammar to Chomsky normal