PART II

Grammars, Automata, and Languages

The syntax of a language specifies the permissible forms of the strings in the language. In Chapter 2, set-theoretic operations and recursive definitions were used to generate the strings of a language. These string-building tools, although primitive, were adequate for enforcing simple constraints on the order and the number of elements in a string. We now introduce a rule-based approach for defining and generating the strings of a language. This approach to language definition has its origin in both linguistics and computer science: linguistics in the attempt to formally describe natural language and computer science in the need to have precise and unambiguous definitions of high-level programming languages. Using terminology from the linguistic study, the string generation systems are called grammars.

In Chapter 3 we introduce two families of grammars, regular and context-free grammars. A family of grammars is defined by the form of the rules and the conditions under which they are applicable. A rule specifies a string transformation, and the strings of a language are generated by a sequence of rule applications. The flexibility provided by rules has proved to be well suited for defining the syntax of programming languages. The grammar that describes the programming language Java is used to demonstrate the context-free definition of several common programming language constructs.

After defining languages by the generation of strings, we turn our attention to the mechanical verification of whether a string satisfies a desired condition or matches a desired pattern. The family of deterministic finite automata is the first in a series of increasingly powerful abstract machines that we will use for pattern matching and language definition. We refer to the machines as abstract because we are not concerned with constructing hardware or software implementations of them. Instead, we are interested in determining the computational capability of the machines. The input to an abstract machine is a string, and the result of a computation indicates the acceptability of the input string. The language of a machine is the set of strings accepted by the computations of the machine.

A deterministic finite automaton is a read-once machine in which the instruction to be executed is determined by the state of the machine and the input symbol being processed. Finite automata have many applications including the lexical analysis of computer programs, digital circuit design, text searching, and pattern recognition. Kleene's theorem shows that the languages accepted by finite automata are precisely those that can be described by regular expressions and generated by regular grammars. A more powerful class of read-once machines, pushdown automata, is created by augmenting a finite automaton with a stack memory. The addition of the external memory permits pushdown automata to accept the context-free languages.

The correspondence between the generation of language by grammars and their acceptance by machines is a central theme of this book. The relationship between machines and grammars will continue with the families of unrestricted grammars and Turing machines introduced in Part III. The regular, context-free, and unrestricted grammars are members of the Chomsky hierarchy of grammars that will be examined in Chapter 10.

CHAPTER 3

Context-Free Grammars

In this chapter we present a rule-based approach for generating the strings of a language. Borrowing the terminology of natural languages, we call a syntactically correct string a **sentence** of the language. A small subset of the English language is used to illustrate the components of the string-generation process. The alphabet of our miniature language is the set {*a, the, John, Jill, hamburger, car, drives, eats, slowly, frequently, big, juicy, brown*}. The elements of the alphabet are called the **terminal symbols** of the language. Capitalization, punctuation, and other important features of written languages are ignored in this example.

The sentence-generation procedure should construct the strings *John eats a hamburger* and *Jill drives frequently.* Strings of the form *Jill* and *car John slowly* should not result from this process. Additional symbols are used during the construction of sentences to enforce the syntactic restrictions of the language. These intermediate symbols, known as **variables** or **nonterminals**, are represented by enclosing them in angle brackets ⟨ ⟩.

Since the generation procedure constructs sentences, the initial variable is named ⟨*sentence*⟩. The generation of a sentence consists of replacing variables by strings of a specific form. Syntactically correct replacements are given by a set of transformation rules. Two possible rules for the variable ⟨*sentence*⟩ are

1. ⟨*sentence*⟩ → ⟨*noun-phrase*⟩⟨*verb-phrase*⟩
2. ⟨*sentence*⟩ → ⟨*noun-phrase*⟩⟨*verb*⟩⟨*direct-object-phrase*⟩

An informal interpretation of rule 1 is that a sentence may be formed by a noun phrase followed by a verb phrase. At this point, of course, neither of the variables ⟨*noun-phrase*⟩ nor ⟨*verb-phrase*⟩ has been defined. The second rule gives an alternative definition of sentence, a noun phrase followed by a verb followed by a direct object phrase. The existence of multiple transformations indicates that syntactically correct sentences may have several different forms.

A noun phrase may contain either a proper or a common noun. A common noun is preceded by a determiner, while a proper noun stands alone. This feature of the syntax of the English language is represented by rules 3 and 4.

Rules for the variables that generate noun and verb phrases are given below. Rather than rewriting the left-hand side of alternative rules for the same variable, we list the right-hand sides of the rules sequentially. Numbering the rules is not a feature of the generation process, merely a notational convenience.

3. ⟨*noun-phrase*⟩ → ⟨*proper-noun*⟩
4. → ⟨*determiner*⟩⟨*common-noun*⟩
5. ⟨*proper-noun*⟩ → *John*
6. → *Jill*
7. ⟨*common-noun*⟩ → *car*
8. → *hamburger*
9. ⟨*determiner*⟩ → *a*
10. → *the*
11. ⟨*verb-phrase*⟩ → ⟨*verb*⟩⟨*adverb*⟩
12. → ⟨*verb*⟩
13. ⟨*verb*⟩ → *drives*
14. → *eats*
15. ⟨*adverb*⟩ → *slowly*
16. → *frequently*

With the exception of ⟨*direct-object-phrase*⟩, rules have been defined for each of the variables that have been introduced.

The application of a rule transforms one string to another. The transformation consists of replacing an occurrence of the variable on the left-hand side of the → with the string on the right-hand side. The generation of a sentence consists of repeated rule applications to transform the variable ⟨*sentence*⟩ into a string of terminal symbols.

For example, the sentence *Jill drives frequently* is generated by the following transformations:

Derivation	Rule Applied
⟨*sentence*⟩ ⇒ ⟨*noun-phrase*⟩⟨*verb-phrase*⟩	1
⇒ ⟨*proper-noun*⟩⟨*verb-phrase*⟩	3
⇒ *Jill* ⟨*verb-phrase*⟩	6
⇒ *Jill* ⟨*verb*⟩⟨*adverb*⟩	11
⇒ *Jill drives* ⟨*adverb*⟩	13
⇒ *Jill drives frequently*	16

The symbol ⇒, used to designate a rule application, is read "derives." The column on the right gives the number of the rule that was applied to achieve the transformation. The derivation terminates when all variables have been removed from the derived string. The resulting string, consisting solely of terminal symbols, is a sentence of the language. The set of terminal strings derivable from the variable ⟨*sentence*⟩ is the language generated by the rules of our example.

To complete the set of rules, the transformations for ⟨*direct-object-phrase*⟩ must be given. Before designing rules, we must decide upon the form of the strings that we wish to generate. In our language we will allow the possibility of any number of adjectives, including repetitions, to precede the direct object. This requires a set of rules capable of generating each of the following strings:

John eats a hamburger

John eats a big hamburger

John eats a big juicy hamburger

John eats a big brown juicy hamburger

John eats a big big brown juicy hamburger

As can be seen by the potential repetition of the adjectives, the rules of the grammar must be capable of generating strings of arbitrary length. The use of a recursive definition allows the elements of an infinite set to be generated by a finite specification. Following that example, recursion is introduced into the string-generation process, that is, into the rules.

17. ⟨*adjective-list*⟩ → ⟨*adjective*⟩⟨*adjective-list*⟩
18. → λ
19. ⟨*adjective*⟩ → *big*
20. → *juicy*
21. → *brown*

The definition of ⟨*adjective-list*⟩ follows the standard recursive pattern. Rule 17 defines ⟨*adjective-list*⟩ in terms of itself, while rule 18 provides the basis of the recursive definition. The λ on the right-hand side of rule 18 indicates that the application of this rule replaces ⟨*adjective-list*⟩ with the null string. Repeated applications of rule 17 generate a sequence of adjectives. Rules for ⟨*direct-object-phrase*⟩ are constructed using ⟨*adjective-list*⟩:

22. ⟨*direct-object-phrase*⟩ → ⟨*adjective-list*⟩⟨*proper-noun*⟩
23. → ⟨*determiner*⟩⟨*adjective-list*⟩⟨*common-noun*⟩

The sentence *John eats a big juicy hamburger* can be derived by the following sequence of rule applications:

Derivation	Rule Applied
⟨*sentence*⟩ ⇒ ⟨*noun-phrase*⟩ ⟨*verb*⟩ ⟨*direct-object-phrase*⟩	2
⇒ ⟨*proper-noun*⟩ ⟨*verb*⟩ ⟨*direct-object-phrase*⟩	3
⇒ *John* ⟨*verb*⟩ ⟨*direct-object-phrase*⟩	5
⇒ *John eats* ⟨*direct-object-phrase*⟩	14
⇒ *John eats* ⟨*determiner*⟩ ⟨*adjective-list*⟩ ⟨*common-noun*⟩	23
⇒ *John eats a* ⟨*adjective-list*⟩ ⟨*common-noun*⟩	9
⇒ *John eats a* ⟨*adjective*⟩ ⟨*adjective-list*⟩ ⟨*common-noun*⟩	17
⇒ *John eats a big* ⟨*adjective-list*⟩ ⟨*common-noun*⟩	19
⇒ *John eats a big* ⟨*adjective*⟩ ⟨*adjective-list*⟩ ⟨*common-noun*⟩	17
⇒ *John eats a big juicy* ⟨*adjective-list*⟩ ⟨*common-noun*⟩	20
⇒ *John eats a big juicy* ⟨*common-noun*⟩	18
⇒ *John eats a big juicy hamburger*	8

The generation of sentences is strictly a function of the rules. The string *the car eats slowly* is a sentence in the language since it has the form ⟨*noun-phrase*⟩ ⟨*verb-phrase*⟩ outlined by rule 1. This illustrates the important distinction between syntax and semantics; the generation of sentences is concerned with the form of the derived string without regard to any underlying meaning that may be associated with the terminal symbols.

By rules 3 and 4, a noun phrase consists of a proper noun or a common noun preceded by a determiner. The variable ⟨*adjective-list*⟩ may be incorporated into the ⟨*noun-phrase*⟩ rules, permitting adjectives to modify a noun:

3′. ⟨*noun-phrase*⟩ → ⟨*adjective-list*⟩ ⟨*proper-noun*⟩

4′. → ⟨*determiner*⟩ ⟨*adjective-list*⟩ ⟨*common-noun*⟩

With this modification, the string *big John eats frequently* can be derived from the variable ⟨*sentence*⟩.

3.1 Context-Free Grammars and Languages

We will now define a formal system, the context-free grammar, that is used to generate the strings of a language. The natural language example was presented to motivate the components and features of string generation using a context-free grammar.

Definition 3.1.1

A **context-free grammar** is a quadruple (V, Σ, P, S) where V is a finite set of variables, Σ (the alphabet) is a finite set of terminal symbols, P is a finite set of rules, and S is a distinguished element of V called the start symbol. The sets V and Σ are assumed to be disjoint.

A **rule** is written $A \to w$ where $A \in \mathrm{V}$ and $w \in (\mathrm{V} \cup \Sigma)^*$. A rule of this form is called an A rule, referring to the variable on the left-hand side. Since the null string is in $(\mathrm{V} \cup \Sigma)^*$, λ may occur on the right-hand side of a rule. A rule of the form $A \to \lambda$ is called a **null** or **λ-rule**.

Italics are used to denote the variables and terminals of a context-free grammar. Terminals are represented by lowercase letters occurring at the beginning of the alphabet, that is, a, b, c, ... Following the conventions introduced for strings, the letters p, q, u, v, w, x, y, z, with or without subscripts, represent arbitrary members of $(\mathrm{V} \cup \Sigma)^*$. Variables will be denoted by capital letters. As in the natural language example, variables are referred to as the *nonterminal symbols* of the grammar.

Grammars are used to generate properly formed strings over the prescribed alphabet. The fundamental step in the generation process consists of transforming a string by the application of a rule. The application of $A \to w$ to the variable A in uAv produces the string uwv. This is denoted $uAv \Rightarrow uwv$. The prefix u and suffix v define the *context* in which the variable A occurs. The grammars introduced in this chapter are called context-free because of the general applicability of the rules. An A rule can be applied to the variable A whenever and wherever it occurs; the context places no limitations on the applicability of a rule.

A string w is derivable from v if there is a finite sequence of rule applications that transforms v to w; that is, if a sequence of transformations

$$v \Rightarrow w_1 \Rightarrow w_2 \Rightarrow \cdots \Rightarrow w_n = w$$

can be constructed from the rules of the grammar. The derivability of w from v is denoted $v \overset{*}{\Rightarrow} w$. The set of strings derivable from v, being constructed by a finite but unbounded number of rule applications, can be defined recursively.

Definition 3.1.2

Let $\mathrm{G} = (\mathrm{V}, \Sigma, \mathrm{P}, S)$ be a context-free grammar and $v \in (\mathrm{V} \cup \Sigma)^*$. The set of strings **derivable** from v is defined recursively as follows:

i) Basis: v is derivable from v.

ii) Recursive step: If $u = xAy$ is derivable from v and $A \to w \in \mathrm{P}$, then xwy is derivable from v.

iii) Closure: A string is derivable from v only if it can be generated from v by a finite number of applications of the recursive step.

Note that the definition of a rule uses the → notation, while its application uses ⇒. The symbol $\overset{*}{\Rightarrow}$ denotes derivability and $\overset{+}{\Rightarrow}$ designates derivability utilizing one or more rule applications. The length of a derivation is the number of rule applications employed. A derivation of w from v of length n is denoted $v \overset{n}{\Rightarrow} w$. When more than one grammar is

being considered, the notation $v \overset{*}{\underset{G}{\Rightarrow}} w$ will be used to explicitly indicate that the derivation utilizes rules of the grammar G.

A language has been defined as a set of strings over an alphabet. A grammar consists of an alphabet and a method of generating strings. These strings may contain both variables and terminals. The start symbol of the grammar, assuming the role of ⟨*sentence*⟩ in the natural language example, initiates the process of generating acceptable strings. The language of the grammar G is the set of terminal strings derivable from the start symbol. We now state this as a definition.

Definition 3.1.3

Let $G = (V, \Sigma, P, S)$ be a context-free grammar.

i) A string $w \in (V \cup \Sigma)^*$ is a **sentential form** of G if there is a derivation $S \overset{*}{\Rightarrow} w$ in G.

ii) A string $w \in \Sigma^*$ is a **sentence** of G if there is a derivation $S \overset{*}{\Rightarrow} w$ in G.

iii) The **language** of G, denoted L(G), is the set $\{w \in \Sigma^* \mid S \overset{*}{\Rightarrow} w\}$.

A sentential form is a string that is derivable from the start symbol of the grammar. Referring back to the natural language example, the derivation

$$\begin{aligned} \langle sentence \rangle &\Rightarrow \langle noun\text{-}phrase \rangle \langle verb\text{-}phrase \rangle \\ &\Rightarrow \langle proper\text{-}noun \rangle \langle verb\text{-}phrase \rangle \\ &\Rightarrow Jill\ \langle verb\text{-}phrase \rangle \end{aligned}$$

shows that *Jill* ⟨*verb-phrase*⟩ is a sentential form of that grammar. It is not yet a sentence, it still contains variables, but it has the form of a sentence. A sentence is a sentential form that contains only terminal symbols. The language of a grammar consists of the sentences generated by the grammar. A set of strings over an alphabet Σ is said to be a **context-free language** if it is generated by a context-free grammar.

The use of recursion is necessary for a finite set of rules to generate strings of arbitrary length and languages with infinitely many strings. Recursion is introduced into grammars through the rules. A rule of the form $A \to uAv$ is called **recursive** since it defines the variable A in terms of itself. Rules of the form $A \to Av$ and $A \to uA$ are called *left-recursive* and *right-recursive*, respectively, indicating the location of recursion in the rule.

Because of the importance of recursive rules, we examine the form of strings produced by repeated applications of the recursive rules $A \to aAb$, $A \to aA$, $A \to Ab$, and $A \to AA$:

$A \Rightarrow aAb$	$A \Rightarrow aA$	$A \Rightarrow Ab$	$A \Rightarrow AA$
$\Rightarrow aAb$	$\Rightarrow aA$	$\Rightarrow Ab$	$\Rightarrow AAA$
$\Rightarrow aaAbb$	$\Rightarrow aaA$	$\Rightarrow Abb$	$\Rightarrow AAAA$
$\Rightarrow aaaAbbb$	$\Rightarrow aaaA$	$\Rightarrow Abbb$	$\Rightarrow AAAAA$
$\vdots$	$\vdots$	$\vdots$	$\vdots$

A derivation employing the rule $A \to aAb$ generates any number of a's followed by the same number of b's. Rules of this form are necessary for producing strings that contain symbols in

$G = (V, \Sigma, P, S)$
$V = \{S, A\}$
$\Sigma = \{a, b\}$
P: $S \to AA$
$A \to AAA \mid bA \mid Ab \mid a$

$S \Rightarrow AA$	$S \Rightarrow AA$	$S \Rightarrow AA$	$S \Rightarrow AA$
$\Rightarrow aA$	$\Rightarrow AAAA$	$\Rightarrow Aa$	$\Rightarrow aA$
$\Rightarrow aAAA$	$\Rightarrow aAAA$	$\Rightarrow AAAa$	$\Rightarrow aAAA$
$\Rightarrow abAAA$	$\Rightarrow abAAA$	$\Rightarrow AAbAa$	$\Rightarrow aAAa$
$\Rightarrow abaAA$	$\Rightarrow abaAA$	$\Rightarrow AAbaa$	$\Rightarrow abAAa$
$\Rightarrow ababAA$	$\Rightarrow ababAA$	$\Rightarrow AbAbaa$	$\Rightarrow abAbAa$
$\Rightarrow ababaA$	$\Rightarrow ababaA$	$\Rightarrow Ababaa$	$\Rightarrow ababAa$
$\Rightarrow ababaa$	$\Rightarrow ababaa$	$\Rightarrow ababaa$	$\Rightarrow ababaa$
(a)	(b)	(c)	(d)

FIGURE 3.1 Sample derivations of *ababaa* in G.

matched pairs, such as left and right parentheses. The right recursive rule $A \to aA$ generates any number of a's preceding the variable A, and the left recursive $A \to Ab$ generates any number of b's following A. Each application of the rule $A \to AA$, which is both left- and right-recursive, produces an additional A. The repetitive application of a recursive rule can be terminated at any time by the application of a different A rule.

A variable A is called *recursive* if there is a derivation $A \overset{+}{\Rightarrow} uAv$. A derivation of the form $A \Rightarrow w \overset{+}{\Rightarrow} uAv$, where A is not in w, is said to be *indirectly recursive*. Note that, due to indirect recursion, a variable A may be recursive even if there are no recursive A rules.

A grammar G that generates the language consisting of strings with a positive, even number of a's is given in Figure 3.1. The rules are written using the shorthand $A \to u \mid v$ to abbreviate $A \to u$ and $A \to v$. The vertical bar | is read "or." Four distinct derivations of the terminal string *ababaa* are shown in Figure 3.1. The definition of derivation permits the transformation of any variable in the string. Each rule application in derivations (a) and (b) in the figure transforms the first variable occurring in a left-to-right reading of the string. Derivations with this property are called *leftmost*. Derivation (c) is *rightmost*, since the rightmost variable has a rule applied to it. These derivations demonstrate that there may be more than one derivation of a string in a context-free grammar.

Figure 3.1 exhibits the flexibility of derivations in a context-free grammar. The essential feature of a derivation is not the order in which the rules are applied, but the manner in which each variable is transformed into a terminal string. The transformation is graphically depicted by a derivation or parse tree. The tree structure indicates the rule applied to each variable but does not designate the order of the rule applications. The leaves of the derivation tree can be ordered to yield the result of a derivation represented by the tree.

Definition 3.1.4

Let $G = (V, \Sigma, P, S)$ be a context-free grammar and let $S \overset{*}{\Rightarrow} w$ be a derivation in G. The **derivation tree**, DT, of $S \overset{*}{\Rightarrow} w$ is an ordered tree that can be built iteratively as follows:

i) Initialize DT with root S.

ii) If $A \to x_1x_2 \ldots x_n$ with $x_i \in (V \cup \Sigma)$ is the rule in the derivation applied to the string uAv, then add $x_1, x_2, \ldots, x_n$ as the children of A in the tree.

iii) If $A \to \lambda$ is the rule in the derivation applied to the string uAv, then add λ as the only child of A in the tree.

The ordering of the leaves also follows this iterative process. Initially, the only leaf is S and the ordering is obvious. When the rule $A \to x_1x_2 \ldots x_n$ is used to generate the children of A, each x_i becomes a leaf and A is replaced in the ordering of the leaves by the sequence $x_1, x_2, \ldots, x_n$. The application of a rule $A \to \lambda$ simply replaces A by the null string. Figure 3.2 traces the construction of the tree corresponding to derivation (a) of Figure 3.1. The ordering of the leaves is given along with each of the trees.

The order of the leaves in a derivation tree is independent of the derivation from which the tree was generated. The ordering provided by the iterative process is identical to the ordering of the leaves given by the relation LEFTOF in Section 1.8. The frontier of the derivation tree is the string generated by the derivation.

Figure 3.3 gives the derivation trees for each of the derivations in Figure 3.1. The trees generated by derivations (a) and (d) are identical, indicating that each variable is transformed into a terminal string in the same manner. The only difference between these derivations is the order of the rule applications.

A derivation tree can be used to produce several derivations that generate the same string. The rule applied to a variable A can be reconstructed from the children of A in the tree. The rightmost derivation

$$\begin{aligned} S &\Rightarrow AA \\ &\Rightarrow AAAA \\ &\Rightarrow AAAa \\ &\Rightarrow AAbAa \\ &\Rightarrow AAbaa \\ &\Rightarrow AbAbaa \\ &\Rightarrow Ababaa \\ &\Rightarrow ababaa \end{aligned}$$

is obtained from the derivation tree (a) in Figure 3.3. Notice that this derivation is different from the rightmost derivation (c) in Figure 3.1. In the latter derivation, the second variable in the string AA is transformed using the rule $A \to a$, while $A \to AAA$ is used in the preceding derivation. The two trees graphically illustrate the distinct transformations.

As we have seen, the context-free applicability of rules allows a great deal of flexibility in the constructions of derivations. Lemma 3.1.5 shows that a derivation may be broken into subderivations from each variable in the string. Derivability was defined recursively, the length of derivations being finite but unbounded. Consequently, we may use mathematical induction to establish that a property holds for all derivations from a given string.

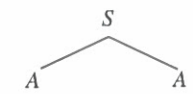
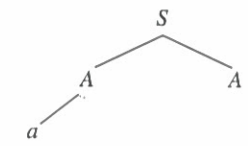
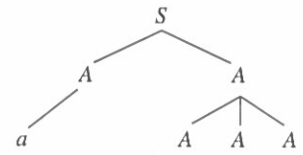
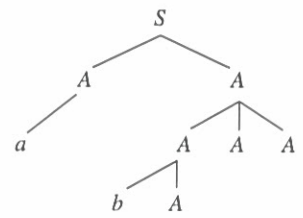
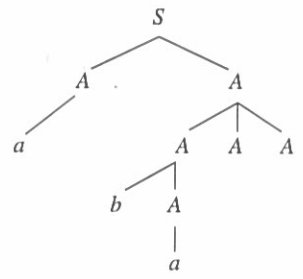
Derivation	Tree	Ordering of Leaves
S	S	S
$\Rightarrow AA$		A, A
$\Rightarrow aA$		a, A
$\Rightarrow aAAA$		a, A, A, A
$\Rightarrow abAAA$		a, b, A, A, A
$\Rightarrow abaAA$		a, b, a, A, A

FIGURE 3.2 Construction of derivation tree. *(continued on next page)*

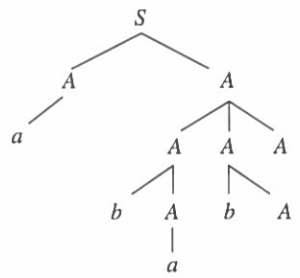
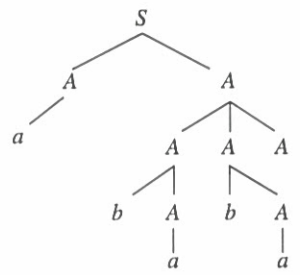
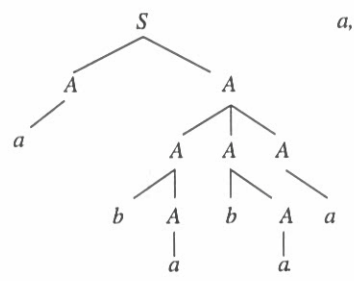
Derivation	Tree	Ordering of Leaves
$\Rightarrow ababAA$		a, b, a, b, A, A
$\Rightarrow ababaA$		a, b, a, b, a, A
$\Rightarrow ababaa$		a, b, a, b, a, a

FIGURE 3.2 *(continued)*

Lemma 3.1.5

Let G be a context-free grammar and $v \overset{n}{\Rightarrow} w$ be a derivation in G where v can be written

$$v = w_1 A_1 w_2 A_2 \dots w_k A_k w_{k+1},$$

with $w_i \in \Sigma^*$. Then there are strings $p_i \in (\Sigma \cup V)^*$ that satisfy

i) $A_i \overset{t_i}{\Rightarrow} p_i$

ii) $w = w_1 p_1 w_2 p_2 \dots w_k p_k w_{k+1}$

iii) $\sum_{i=1}^{k} t_i = n.$

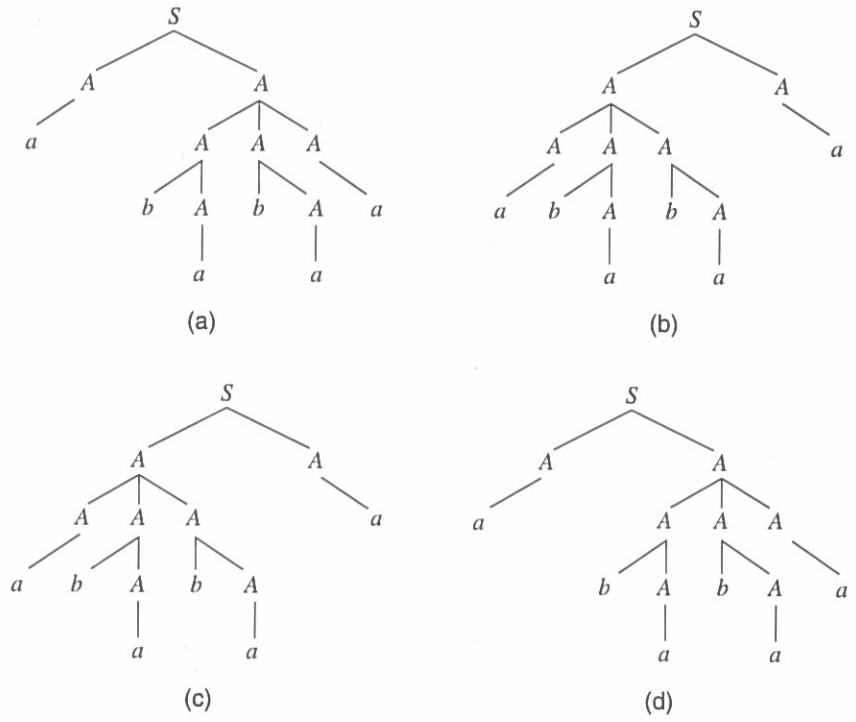


FIGURE 3.3 Trees corresponding to the derivations in Figure 3.1.

Proof. The proof is by induction on the length of the derivation of w from v.

Basis: The basis consists of derivations of the form $v \overset{0}{\Rightarrow} w$. In this case, $w = v$ and each A_i is equal to the corresponding p_i. The desired derivations have the form $A_i \overset{0}{\Rightarrow} p_i$.

Inductive Hypothesis: Assume that all derivations $v \overset{n}{\Rightarrow} w$ can be decomposed into derivations from the A_i's, the variables of v, which together form a derivation of w from v of length n.

Inductive Step: Let $v \overset{n+1}{\Longrightarrow} w$ be a derivation in G with

$$v = w_1 A_1 w_2 A_2 \dots w_k A_k w_{k+1},$$

where $w_i \in \Sigma^*$. The derivation can be written $v \Rightarrow u \overset{n}{\Rightarrow} w$. This reduces the original derivation to the application of a single rule and derivation of length n, the latter of which is suitable for the invocation of the inductive hypothesis.

The first rule application in the derivation, $v \Rightarrow u$, transforms one of the variables in v, call it A_j, with a rule of the form

$$A_j \to u_1 B_1 u_2 B_2 \dots u_m B_m u_{m+1},$$

where each $u_i \in \Sigma^*$. The string u is obtained from v by replacing A_j by the right-hand side of the A_j rule. Making this substitution, u can be written as

$$w_1 A_1 \dots A_{j-1} w_j u_1 B_1 u_2 B_2 \dots u_m B_m u_{m+1} w_{j+1} A_{j+1} \dots w_k A_k w_{k+1}.$$

Since w is derivable from u using n rule applications, the inductive hypothesis asserts that there are strings $p_1, \dots, p_{j-1}$, $q_1, \dots, q_m$, and $p_{j+1}, \dots, p_k$ that satisfy

i) $A_i \overset{t_i}{\Rightarrow} p_i$ for $i = 1, \dots, j-1, j+1, \dots, k$
$B_i \overset{s_i}{\Rightarrow} q_i$ for $i = 1, \dots, m$;

ii) $w = w_1 p_1 w_2 \dots p_{j-1} w_j u_1 q_1 u_2 \dots u_m q_m u_{m+1} w_{j+1} p_{j+1} \dots w_k p_k w_{k+1}$; and

iii) $\sum_{i=1}^{j-1} t_i + \sum_{i=j+1}^{k} t_i + \sum_{i=1}^{m} s_i = n$.

Combining the rule $A_j \to u_1 B_1 u_2 B_2 \dots u_m B_m u_{m+1}$ with the derivations $B_i \overset{*}{\Rightarrow} q_i$, we obtain a derivation

$$A_j \overset{*}{\Rightarrow} u_1 q_1 u_2 q_2 \dots u_m q_m u_{m+1} = p_j$$

whose length is the sum of lengths of the derivations from the B_i's plus one. The derivations $A_i \overset{*}{\Rightarrow} p_i$, $i = 1, \dots, k$, provide the desired decomposition of the derivation of w from v. ■

Lemma 3.1.5 demonstrates the flexibility and modularity of derivations in context-free grammars. Every complex derivation can be broken down into subderivations of the constituent variables. This modularity will be exploited in the design of complex languages by using variables to define smaller and more manageable subsets of the language. These independently defined sublanguages are then combined by additional rules to produce the syntax of the entire language.

3.2 Examples of Grammars and Languages

Context-free grammars have been introduced to generate languages. Formal languages, like computer languages and natural languages, have requirements that the strings must satisfy in order to be syntactically correct. Grammars for these languages must generate precisely the desired strings and no others. There are two natural approaches that we may take to help develop our understanding of the relationship between grammars and languages. One is to begin with an informal specification of a language and then construct a grammar that generates it. This is the approach followed in the design of programming languages—the syntax is selected and the language designer produces a set of rules that defines the correctly formed strings. Conversely, we may begin with the rules of a grammar and analyze them to determine the form of the strings of the language. This is the approach frequently taken when checking the syntax of the source code of a computer program. The syntax of the programming is specified by a set of grammatical rules, such as the definition of

the programming language Java given in Appendix IV. The syntax of constants, identifiers, statements, and entire programs is correct if the source code is derivable from the appropriate variables in the grammar.

Initially, determining the relationship between strings and rules may seem difficult. With experience, you will recognize frequently occurring patterns in strings and the rules that produce them. The goal of this section is to analyze examples to help you develop an intuitive understanding of language definition using context-free grammars.

In each of the examples a grammar is defined by listing its rules. The variables and terminals of the grammar are those occurring in the rules. The variable S is the start symbol of each grammar.

Example 3.2.1

Let G be the grammar given by the rules

$$S \to aSa \mid aBa$$
$$B \to bB \mid b.$$

Then $L(G) = \{a^n b^m a^n \mid n > 0,\ m > 0\}$. The rule $S \to aSa$ recursively builds an equal number of a's on each end of the string. The recursion is terminated by the application of the rule $S \to aBa$, ensuring at least one leading and one trailing a. The recursive B rule then generates any number of b's. To remove the variable B from the string and obtain a sentence of the language, the rule $B \to b$ must be applied, forcing the presence of at least one b. □

Example 3.2.2

The relationship between the number of leading a's and trailing d's in the language $\{a^n b^m c^m d^{2n} \mid n \geq 0,\ m > 0\}$ indicates that a recursive rule is needed to generate them. The same is true of the b's and c's. Derivations in the grammar

$$S \to aSdd \mid A$$
$$A \to bAc \mid bc$$

generate strings in an outside-to-inside manner. The S rules produce the a's and d's while the A rules generate the b's and c's. The rule $A \to bc$, whose application terminates the recursion, ensures the presence of the substring bc in every string in the language. □

Example 3.2.3

Recall that a string w is a palindrome if $w = w^R$. A grammar is constructed to generate the set of palindromes over $\{a, b\}$. The rules of the grammar mimic the recursive definition of palindromes given in Exercise 2.12. The basis of the set of palindromes consists of the strings λ, a, and b. The S rules

$$S \to a \mid b \mid \lambda$$

immediately generate these strings. The recursive part of the definition consists of adding the same symbol to each side of an existing palindrome. The rules

$$S \to aSa \mid bSb$$

capture the recursive generation process. □

Example 3.2.4

The first recursive rule of

$$S \to aSb \mid aSbb \mid \lambda$$

generates a trailing b for every a, while the second generates two b's for each a. Thus there is at least one b for every a and at most two. The language of the grammar is $\{a^n b^m \mid 0 \leq n \leq m \leq 2n\}$. □

Example 3.2.5

Consider the grammar

$$S \to abScB \mid \lambda$$
$$B \to bB \mid b.$$

The recursive S rule generates an equal number of ab's and cB's. The B rules generate b^+. In a derivation each occurrence of B may produce a different number of b's. For example, in the derivation

$$\begin{aligned} S &\Rightarrow abScB \\ &\Rightarrow ababScBcB \\ &\Rightarrow ababcBcB \\ &\Rightarrow ababcbcB \\ &\Rightarrow ababcbcbB \\ &\Rightarrow ababcbcbb, \end{aligned}$$

the first occurrence of B generates a single b and the second occurrence produces bb. The language of the grammar is the set $\{(ab)^n (cb^{m_n})^n \mid n \geq 0,\ m_n > 0\}$. The superscript m_n indicates that the number of b's produced by each occurrence of B may be different since b^{m_i} need not equal b^{m_j} when $i \neq j$. □

Example 3.2.6

Let G_1 and G_2 be the grammars

G_1: $S \to AB$, $A \to aA \mid a$, $B \to bB \mid \lambda$

G_2: $S \to aS \mid aA$, $A \to bA \mid \lambda$.

Both of these grammars generate the language a^+b^*. The A rules in G_1 provide the standard method of generating a nonnull string of a's. The use of the λ-rule to terminate the derivation allows the possibility of having no b's. The rules in grammar G_2 build the strings of a^+b^* in a left-to-right manner. □

Example 3.2.7

The grammars G_1 and G_2 generate the strings over $\{a, b\}$ that contain exactly two b's. That is, the language of the grammars is $a^*ba^*ba^*$.

G_1: $S \to AbAbA$, $A \to aA \mid \lambda$

G_2: $S \to aS \mid bA$, $A \to aA \mid bC$, $C \to aC \mid \lambda$

G_1 requires only two variables since the three instances of a^* are generated by the same A rules. The second builds the strings in a left-to-right manner, requiring a distinct variable for the generation of each sequence of a's. □

Example 3.2.8

The grammars from Example 3.2.7 can be modified to generate strings with at least two b's.

G_1: $S \to AbAbA$, $A \to aA \mid bA \mid \lambda$

G_2: $S \to aS \mid bA$, $A \to aA \mid bC$, $C \to aC \mid bC \mid \lambda$

In G_1, any string can be generated before, between, and after the two b's produced by the S rule. A derivation in G_2 produces the first b using the rule $S \to bA$ and the second b with $A \to bC$. The derivation finishes using applications of the C rules, which can generate any string of a's and b's. □

Two grammars that generate the same language are said to be *equivalent*. Examples 3.2.6, 3.2.7, and 3.2.8 show that equivalent grammars may produce the strings of a language by significantly different derivations. In later chapters we will see that rules having particular forms may facilitate the mechanical determination of the syntactic correctness of strings.

Example 3.2.9

A grammar is given that generates the language consisting of even-length strings over $\{a, b\}$. The strategy can be generalized to construct strings of length divisible by three, by four, and so forth. The variables S and O serve as counters. An S occurs in a sentential form when an

even number of terminals has been generated. An O records the presence of an odd number of terminals.

$$S \to aO \mid bO \mid \lambda$$
$$O \to aS \mid bS$$

The application of $S \to \lambda$ completes the derivation of a terminal string. Until this occurs, a derivation alternates between applications of S and O rules. □

Example 3.2.10

Let L be the language over $\{a, b\}$ consisting of all strings with an even number of b's. The grammar

$$S \to aS \mid bB \mid \lambda$$
$$B \to aB \mid bS \mid bC$$
$$C \to aC \mid \lambda$$

that generates L combines the techniques presented in the previous examples, Example 3.2.9 for the even number of b's and Example 3.2.7 for the arbitrary number of a's. Deleting all rules containing C yields another grammar that generates L. □

Example 3.2.11

Exercise 2.38 requested a regular expression for the language over $\{a, b\}$ consisting of strings with an even number of a's and an even number of b's. It was noted at the time that a regular expression for this language was quite complex. The flexibility provided by string generation with rules makes the construction of a context-free grammar for this language straightforward. The variables are chosen to represent the parities of the number of a's and b's in the derived string. The variables of the grammar with their interpretations are

Variable	Interpretation
S	Even number of a's and even number of b's
A	Even number of a's and odd number of b's
B	Odd number of a's and even number of b's
C	Odd number of a's and odd number of b's

The application of a rule adds one terminal symbol to the derived string and updates the variable to reflect the new status. The rules of the grammar are

$$S \to aB \mid bA \mid \lambda$$
$$A \to aC \mid bS$$
$$B \to aS \mid bC$$
$$C \to aA \mid bB.$$

When the variable S is present, the derived string has an even number of a's and an even number of b's. The application of $S \to \lambda$ removes the variable from the sentential form, producing a string that satisfies the language specification. □

Example 3.2.12

The rules of a grammar are designed to impose a structure on the strings in the language. This structure may consist of ensuring the presence or absence of certain combinations of elements of the alphabet. We construct a grammar with alphabet $\{a, b, c\}$ whose language consists of all strings that do not contain the substring abc. The variables are used to determine how far the derivation has progressed toward generating the string abc.

$$S \to bS \mid cS \mid aB \mid \lambda$$
$$B \to aB \mid cS \mid bC \mid \lambda$$
$$C \to aB \mid bS \mid \lambda$$

The strings are built in a left-to-right manner. At most one variable is present in a sentential form. If an S is present, no progress has been made toward deriving abc. The variable B occurs when the previous terminal is an a. The variable C is present only when preceded by ab. Thus, the C rules cannot generate the terminal c. □

3.3 Regular Grammars

Regular grammars are an important subclass of context-free grammars that play a prominent role in the lexical analysis and parsing of programming languages. Regular grammars are obtained by placing restrictions on the form of the right-hand side of the rules. In Chapter 6 we will show that regular grammars generate precisely the languages that are defined by regular expressions or accepted by finite-state machines.

Definition 3.3.1

A **regular grammar** is a context-free grammar in which each rule has one of the following forms:

i) $A \to a$,
ii) $A \to aB$, or
iii) $A \to \lambda$,

where $A, B \in V$, and $a \in \Sigma$.

Derivations in regular grammars have a particularly nice form; there is at most one variable present in a sentential form and that variable, if present, is the rightmost symbol in the string. Each rule application adds a terminal to the derived string until a rule of the

form $A \to a$ or $A \to \lambda$ terminates the derivation. These properties are illustrated using the regular grammar G_1

$$S \to aS \mid aA$$
$$A \to bA \mid \lambda$$

from Example 3.2.6 that generates the language a^+b^*. The derivation of $aabb$,

$$\begin{aligned} S &\Rightarrow aS \\ &\Rightarrow aaA \\ &\Rightarrow aabA \\ &\Rightarrow aabbA \\ &\Rightarrow aabb, \end{aligned}$$

shows the left-to-right generation of the prefix of terminal symbols. The derivation ends with the application of the rule $A \to \lambda$.

A language generated by a regular grammar is called a *regular* language. You may recall that the family of regular languages was introduced in Chapter 2 as the set of languages described by regular expressions. There is no conflict with what might appear to be two different definitions of the same term, since we will show that regular expressions and regular grammars define the same family of languages.

A regular language may be generated by both regular and nonregular grammars. The grammars G_1 and G_2 from Example 3.2.6 both generate the language a^+b^*. The grammar G_1 is not regular because the rule $S \to AB$ does not have the specified form. A language is regular if it is generated by some regular grammar; the existence of nonregular grammars that also generate the language is irrelevant. The grammars constructed in Examples 3.2.9, 3.2.10, 3.2.11, and 3.2.12 provide additional examples of regular grammars.

Example 3.3.1

We will construct a regular grammar that generates the same language as the context-free grammar

$$\begin{aligned} \text{G: } & S \to abSA \mid \lambda \\ & A \to Aa \mid \lambda. \end{aligned}$$

The language of G is $\lambda \cup (ab)^+a^*$. The equivalent regular grammar

$$S \to aB \mid \lambda$$
$$B \to bS \mid bA$$
$$A \to aA \mid \lambda$$

generates the strings in a left-to-right manner. The S and B rules generate a prefix from the set $(ab)^*$. If a string has a suffix of a's, the rule $B \to bA$ is applied. The A rules are used to generate the remainder of the string. □

3.4 Verifying Grammars

The grammars in the previous sections were built to generate specific languages. An intuitive argument was given to show that the grammar did indeed generate the correct set of strings. No matter how convincing the argument, the possibility of error exists. A proof is required to guarantee that a grammar generates precisely the desired strings.

To prove that the language of a grammar G is identical to a given language L, the inclusions $L \subseteq L(G)$ and $L(G) \subseteq L$ must be established. To demonstrate the techniques involved, we will prove that the language of the grammar

$$\begin{aligned} \text{G: } & S \to AASB \mid AAB \\ & A \to a \\ & B \to bbb \end{aligned}$$

is the set $L = \{a^{2n}b^{3n} \mid n > 0\}$.

A terminal string is in the language of a grammar if it can be derived from the start symbol using the rules of the grammar. The inclusion $\{a^{2n}b^{3n} \mid n > 0\} \subseteq L(G)$ is established by showing that every string in L is derivable in G. Since L contains an infinite number of strings, we cannot construct a derivation for every string in L. Unfortunately, this is precisely what is required. The apparent dilemma is solved by providing a derivation schema. The schema consists of a pattern that can be followed to construct a derivation for any string in L. A string of the form $a^{2n}b^{3n}$, for $n > 0$, can be derived by the following sequence of rule applications:

Derivation	Rule Applied
$S \overset{n-1}{\Longrightarrow} (AA)^{n-1}SB^{n-1}$	$S \to AASB$
$\Rightarrow (AA)^nB^n$	$S \to AAB$
$\overset{2n}{\Longrightarrow} (aa)^nB^n$	$A \to a$
$\overset{n}{\Longrightarrow} (aa)^n(bbb)^n$	$B \to bbb$
$= a^{2n}b^{3n}$	

where the superscripts on the $\Rightarrow$ specify the number of applications of the rule. The preceding schema provides a "recipe," that, when followed, can produce a derivation for any string in L.

The opposite inclusion, $L(G) \subseteq \{a^{2n}b^{3n} \mid n > 0\}$, requires each terminal string derivable in G to have the form specified by the set L. The derivation of a string in the language

consists of a finite number of rule applications, indicating the suitability of a proof by induction. The first difficulty is to determine exactly what we need to prove. We wish to establish a relationship between the a's and b's in all terminal strings derivable in G. A necessary condition for a string w to be a member of L is that three times the number of a's in the string be equal to twice the number of b's. Letting $n_x(u)$ be the number of occurrences of the symbol x in the string u, this relationship can be expressed by $3n_a(u) = 2n_b(u)$.

This numeric relationship between the symbols in a terminal string clearly is not true for every string derivable from S. Consider the derivation

$$\begin{aligned} S &\Rightarrow AASB \\ &\Rightarrow aASB. \end{aligned}$$

The string $aASB$, which is derivable in G, contains one a and no b's.

To account for the intermediate sentential forms that occur in a derivation, relationships between the variables and terminals that hold for all steps in the derivation must be determined. When a terminal string is derived, no variables will remain and the relationships should yield the required structure of the string.

The interactions of the variables and the terminals in the rules of G must be examined to determine their effect on the derivations of terminal strings. The rule $A \rightarrow a$ guarantees that every A will eventually be replaced by a single a. The number of a's present at the termination of a derivation consists of those already in the string and the number of A's in the string. The sum $n_a(u) + n_A(u)$ represents the number of a's that must be generated in deriving a terminal string from u. Similarly, every B will be replaced by the string bbb. The number of b's in a terminal string derivable from u is $n_b(u) + 3n_B(u)$. These observations are used to construct condition (i), establishing the correspondence of variables and terminals that holds for each step in the derivation.

i) $3(n_a(u) + n_A(u)) = 2(n_b(u) + 3n_B(u))$.

The string $aASB$, which we have seen is derivable in G, satisfies this condition since $n_a(aASB) + n_A(aASB) = 2$ and $n_b(aASB) + 3n_B(aASB) = 3$.

Conditions (ii) and (iii) are

ii) $n_A(u) + n_a(u) > 1$, and

iii) the a's and A's in a sentential form precede the S, which precedes the b's and B's.

All strings in $\{a^{2n}b^{3n} \mid n > 0\}$ contain at least two a's and three b's. Conditions (i) and (ii) combine to yield this property. Condition (iii) prescribes the order of the symbols in a derivable string. Not all of the symbols must be present in each string; strings derivable from S by one rule application do not contain any terminal symbols.

After the appropriate relationships have been determined, we must prove that they hold for every string derivable from S. The basis of the induction consists of all strings that can be obtained by derivations of length one (the S rules). The inductive hypothesis asserts that the conditions are satisfied for all strings derivable by n or fewer rule applications. The

inductive step consists of showing that the application of an additional rule preserves the relationships.

There are two derivations of length one, $S \Rightarrow AASB$ and $S \Rightarrow AAB$. For each of these strings, $3(n_a(u) + n_A(u)) = 2(n_b(u) + 3n_B(u)) = 6$. By observation, conditions (ii) and (iii) hold for the two strings.

The inductive hypothesis asserts that (i), (ii), and (iii) are satisfied by all strings derivable by n or fewer rule applications. We now use the inductive hypothesis to show that the three properties hold for all strings generated by derivations of $n + 1$ rule applications.

Let w be a string derivable from S by a derivation $S \overset{n+1}{\Longrightarrow} w$ of length $n + 1$. To use the inductive hypothesis, we write the derivation of length $n + 1$ as a derivation of length n followed by a single rule application:

$$S \overset{n}{\Rightarrow} u \Rightarrow w.$$

Written in this form, it is clear that the string u is derivable by n rule applications. The inductive hypothesis asserts that properties (i), (ii), and (iii) hold for u. The inductive step requires that we show that the application of one rule to u preserves these properties.

For any sentential form v, we let $j(v) = 3(n_a(v) + n_A(v))$ and $k(v) = 2(n_b(v) + 3n_B(v))$. By the inductive hypothesis, $j(u) = k(u)$ and $j(u)/3 > 1$. The effects of the application of an additional rule on the constituents of the string u are given in the following table.

Rule	$j(w)$	$k(w)$	$j(w)/3$
$S \rightarrow AASB$	$j(u) + 6$	$k(u) + 6$	$j(u)/3 + 2$
$S \rightarrow AAB$	$j(u) + 6$	$k(u) + 6$	$j(u)/3 + 2$
$A \rightarrow a$	$j(u)$	$k(u)$	$j(u)/3$
$B \rightarrow bbb$	$j(u)$	$k(u)$	$j(u)/3$

Since $j(u) = k(u)$, we conclude that $j(w) = k(w)$. Similarly, $j(w)/3 > 1$ follows from the inductive hypothesis that $j(u)/3 > 1$. The ordering of the symbols is preserved by noting that each rule application either replaces S by an appropriately ordered sequence of variables or transforms a variable to the corresponding terminal.

We have shown that the three conditions hold for every string derivable in G. Since there are no variables in a string $w \in \mathrm{L(G)}$, condition (i) implies $3n_a(w) = 2n_b(w)$. Condition (ii) guarantees the existence of a's and b's, while (iii) prescribes the order. Thus $\mathrm{L(G)} \subseteq \{a^{2n}b^{3n} \mid n > 0\}$. Having established the opposite inclusions, we conclude that the language of G is $\{a^{2n}b^{3n} \mid n > 0\}$.

As illustrated by the preceding argument, proving that a grammar generates a certain language is a complicated process. This, of course, was an extremely simple grammar with only a few rules. The inductive process is straightforward after the correct relationships have been determined. The most challenging part of the inductive proof is determining the

relationships between the variables and the terminals that must hold in the intermediate sentential forms. The relationships are sufficient if, when all references to the variables are removed, they yield the desired structure of the terminal strings.

As seen in the preceding argument, establishing that a grammar G generates a language L requires two distinct arguments:

i) that all strings of L are derivable in G, and

ii) that all strings generated by G are in L.

The former is accomplished by providing a derivation schema that can be used to produce a derivation for any sting in L. The latter uses induction to show that each sentential form satisfies conditions that lead to the generation of a string in L. The following examples further illustrate the steps involved in these proofs.

Example 3.4.1

Let G be the grammar

$$S \to aS \mid bB \mid \lambda$$
$$B \to aB \mid bS \mid bC$$
$$C \to aC \mid \lambda$$

given in Example 3.2.10. We will prove that $L(G) = a^*(a^*ba^*ba^*)^*$, the set of all strings over $\{a, b\}$ with an even number of b's. It is not true that every string derivable from S has an even number of b's. The derivation $S \Rightarrow bB$ produces a single b. To derive a terminal string, every B must eventually be transformed into a b. Consequently, we conclude that the desired relationship asserts that $n_b(u) + n_B(u)$ is even. When a terminal string w is derived, $n_B(w) = 0$ and $n_b(w)$ is even.

We will prove that $n_b(u) + n_B(u)$ is even for all strings derivable from S. The proof is by induction on the length of the derivations.

Basis: Derivations of length one. There are three such derivations:

$$S \Rightarrow aS$$
$$S \Rightarrow bB$$
$$S \Rightarrow \lambda.$$

By inspection, $n_b(u) + n_B(u)$ is even for these strings.

Inductive Hypothesis: Assume that $n_b(u) + n_B(u)$ is even for all strings u that can be derived with n rule applications.

Inductive Step: To complete the proof, we need to show that $n_b(w) + n_B(w)$ is even whenever w can be obtained by a derivation of the form $S \overset{n+1}{\Longrightarrow} w$. The key step is to reformulate the derivation to apply the inductive hypothesis. A derivation of w of length $n + 1$ can be written $S \overset{n}{\Rightarrow} u \Rightarrow w$.

By the inductive hypothesis, $n_b(u) + n_B(u)$ is even. We show that the result of the application of any rule to u preserves the parity of $n_b(u) + n_B(u)$. The table

Rule	$n_b(w) + n_B(w)$
$S \to aS$	$n_b(u) + n_B(u)$
$S \to bB$	$n_b(u) + n_B(u) + 2$
$S \to \lambda$	$n_b(u) + n_B(u)$
$B \to aB$	$n_b(u) + n_B(u)$
$B \to bS$	$n_b(u) + n_B(u)$
$B \to bC$	$n_b(u) + n_B(u)$
$C \to aC$	$n_b(u) + n_B(u)$
$C \to \lambda$	$n_b(u) + n_B(u)$

gives the value of $n_b(w) + n_B(w)$ when the corresponding rule is applied to u. Each of the rules leaves the total number of B's and b's fixed except the second, which adds two to the total. Thus the sum of the b's and B's in a string obtained from u by the application of a rule is even. Since a terminal string contains no B's, we have shown that every string in L(G) has an even number of b's.

To complete the proof, the opposite inclusion, $L(G) \subseteq a^*(a^*ba^*ba^*)^*$, must also be established. To accomplish this, we show that every string in $a^*(a^*ba^*ba^*)^*$ is derivable in G. A string in $a^*(a^*ba^*ba^*)^*$ has the form

$$a^{n_1}ba^{n_2}ba^{n_3} \ldots a^{n_{2k}}ba^{n_{2k+1}},\ k \geq 0.$$

Any string in a^* can be derived using the rules $S \to aS$ and $S \to \lambda$. All other strings in L(G) can be generated by a derivation of the form

Derivation	Rule Applied
$S \overset{n_1}{\Longrightarrow} a^{n_1}S$	$S \to aS$
$\Longrightarrow a^{n_1}bB$	$S \to bB$
$\overset{n_2}{\Longrightarrow} a^{n_1}ba^{n_2}B$	$B \to aB$
$\Longrightarrow a^{n_1}ba^{n_2}bS$	$B \to bS$
$\vdots$	
$\overset{n_{2k}}{\Longrightarrow} a^{n_1}ba^{n_2}ba^{n_3} \ldots a^{n_{2k}}B$	$B \to aB$
$\Longrightarrow a^{n_1}ba^{n_2}ba^{n_3} \ldots a^{n_{2k}}bC$	$B \to bC$
$\overset{n_{2k+1}}{\Longrightarrow} a^{n_1}ba^{n_2}ba^{n_3} \ldots a^{n_{2k}}ba^{n_{2k+1}}C$	$C \to aC$
$\Longrightarrow a^{n_1}ba^{n_2}ba^{n_3} \ldots a^{n_{2k}}ba^{n_{2k+1}}$	$C \to \lambda$

□

Example 3.4.2

Let G be the grammar

$$S \to aASB \mid \lambda$$
$$A \to ad \mid d$$
$$B \to bb.$$

We show that every string in L(G) has at least as many b's as a's. The number of b's in a terminal string depends upon the b's and B's in the intermediate steps of the derivation. Each B generates two b's, while an A generates at most one a. We will prove, for every sentential form u of G, that $n_a(u) + n_A(u) \leq n_b(u) + 2n_B(u)$. Let $j(u) = n_a(u) + n_A(u)$ and $k(u) = n_b(u) + 2n_B(u)$.

Basis: There are two derivations of length one

Rule	$j(u)$	$k(u)$
$S \Rightarrow aASB$	2	2
$S \Rightarrow \lambda$	0	0

and $j(u) \leq k(u)$ for both of the derivable strings.

Inductive Hypothesis: Assume that $j(u) \leq k(u)$ for all strings u derivable from S in n or fewer rule applications.

Inductive Step: We need to prove that $j(w) \leq k(w)$ whenever $S \stackrel{n+1}{\Longrightarrow} w$. The derivation of w can be rewritten $S \stackrel{n}{\Rightarrow} u \Rightarrow w$ and, by the inductive hypothesis, $j(u) \leq k(u)$. We must show that the inequality is preserved by an additional rule application. The effect of each rule application on j and k is indicated in the following table.

Rule	$j(w)$	$k(w)$
$S \to aASB$	$j(u) + 2$	$k(u) + 2$
$S \to \lambda$	$j(u)$	$k(u)$
$B \to bb$	$j(u)$	$k(u)$
$A \to ad$	$j(u)$	$k(u)$
$A \to d$	$j(u) - 1$	$k(u)$

The first rule adds 2 to each side of an inequality, maintaining the inequality. The final rule subtracts 1 from the smaller side, reinforcing the inequality. For a string $w \in$ L(G), the inequality yields $n_a(w) \leq n_b(w)$ as desired. □

Example 3.4.3

In Example 3.2.2 the grammar

$$\text{G: } S \to aSdd \mid A$$
$$A \to bAc \mid bc$$

was constructed to generate the language $L = \{a^n b^m c^m d^{2n} \mid n \geq 0,\ m > 0\}$. We develop relationships among the variables and terminals that are sufficient to prove that L(G) ⊆ L. The S and the A rules enforce the numeric relationships between the a's and d's and the b's and c's. In a derivation of G, the start symbol is removed by an application of the rule $S \to A$. The presence of an A guarantees that a b will eventually be generated. These observations lead to the following four conditions for every sentential form u of G:

i) $2n_a(u) = n_d(u)$.
ii) $n_b(u) = n_c(u)$.
iii) $n_S(u) + n_A(u) + n_b(u) > 0$.
iv) The a's precede the b's, which precede the S or A, which precede the c's, which precede the d's.

The equalities guarantee that the terminals occur in correct numerical relationships. The description of the language also demands that the terminals occur in a specified order. The final condition ensures that the order is maintained at each step in the derivation. □

3.5 Leftmost Derivations and Ambiguity

The language of a grammar is the set of terminal strings that can be derived, in any manner, from the start symbol. A terminal string may be generated by a number of different derivations. For example, Figure 3.1 gave a grammar and four derivations of the string *ababaa* using the rules of the grammar. Any one of the derivations is sufficient to exhibit the syntactic correctness of the string.

The derivations using the natural language example that introduced this chapter were all given as leftmost derivations. This is a natural technique for readers of English since the leftmost variable is the first encountered when reading a string. To reduce the number of derivations that must be considered in determining whether a string is in the language of a grammar, we now prove that every string in the language is derivable in a leftmost manner.

Theorem 3.5.1

Let G = (V, Σ, P, S) be a context-free grammar. A string w is in L(G) if, and only if, there is a leftmost derivation of w from S.

Proof. Clearly, $w \in$ L(G) whenever there is a leftmost derivation of w from S. We must establish the "only if" clause of the equivalence, that is, that every string in the L(G) is derivable in a leftmost manner. Let

$$S \Rightarrow w_1 \Rightarrow w_2 \Rightarrow w_3 \Rightarrow \cdots \Rightarrow w_n = w$$

be a, not necessarily leftmost, derivation of w in G. The independence of rule applications in a context-free grammar is used to build a leftmost derivation of w. Let w_k be the first sentential form in the derivation to which the rule application is not leftmost. If there is no such k, the derivation is already leftmost and there is nothing to show. We will show that

the rule applications can be reordered so that the first $k+1$ rule applications are leftmost. This procedure can be repeated, $n-k$ times if necessary, to produce a leftmost derivation.

By the choice of w_k, the derivation $S \overset{k}{\Rightarrow} w_k$ is leftmost. Assume that A is the leftmost variable in w_k and B is the variable transformed in the $k+1$st step of the derivation. Then w_k can be written $u_1Au_2Bu_3$ with $u_1 \in \Sigma^*$. The application of a rule $B \to v$ to w_k has the form

$$w_k = u_1Au_2Bu_3 \Rightarrow u_1Au_2vu_3 = w_{k+1}.$$

Since w is a terminal string, an A rule must eventually be applied to the leftmost variable in w_k. Let the first rule application that transforms the variable A occur at the $j+1$st step in the original derivation. Then the application of the rule $A \to p$ can be written

$$w_j = u_1Aq \Rightarrow u_1pq = w_{j+1}.$$

The rules applied in steps $k+2$ to j transform the string u_2vu_3 into q. The derivation is completed by the subderivation

$$w_{j+1} \overset{*}{\Rightarrow} w_n = w.$$

The original derivation has been divided into five distinct subderivations. The first k rule applications are already leftmost, so they are left intact. To construct a leftmost derivation, the rule $A \to p$ is applied to the leftmost variable at step $k+1$. The context-free nature of rule applications permits this rearrangement. A derivation of w that is leftmost for the first $k+1$ rule applications is obtained as follows:

$$\begin{aligned}
S \overset{k}{\Rightarrow} w_k &= u_1Au_2Bu_3 \\
&\Rightarrow u_1pu_2Bu_3 && \text{(applying } A \to p\text{)} \\
&\Rightarrow u_1pu_2vu_3 && \text{(applying } B \to v\text{)} \\
&\overset{j-k-1}{\Longrightarrow} u_1pq = w_{j+1} && \text{(using the derivation } u_2vu_3 \overset{*}{\Rightarrow} q\text{)} \\
&\overset{n-j-1}{\Longrightarrow} w_n. && \text{(using the derivation } w_{j+1} \overset{*}{\Rightarrow} w_n\text{).}
\end{aligned}$$

Every time this procedure is repeated, the derivation becomes "more" leftmost. If the length of a derivation is n, then at most n iterations are needed to produce a leftmost derivation of w. ■

Theorem 3.5.1 does not guarantee that all sentential forms of the grammar can be generated by a leftmost derivation. Only leftmost derivations of terminal strings are assured. Consider the grammar

$$\begin{aligned}
S &\to AB \\
A &\to aA \mid \lambda \\
B &\to bB \mid \lambda
\end{aligned}$$

that generates a^*b^*. The sentential form A can be obtained by the rightmost derivation $S \Rightarrow AB \Rightarrow A$. It is easy to see that there is no leftmost derivation of A.

A similar result (Exercise 31) establishes the sufficiency of using rightmost derivations for the generation of terminal strings. Leftmost and rightmost derivations of w from v are explicitly denoted $v \overset{*}{\underset{L}{\Rightarrow}} w$ and $v \overset{*}{\underset{R}{\Rightarrow}} w$.

Restricting our attention to leftmost derivations eliminates many of the possible derivations of a string. Is this reduction sufficient to establish a canonical derivation? That is, is there a unique leftmost derivation of every string in the language of a grammar? Unfortunately, the answer is no. Two distinct leftmost derivations of the string *ababaa* were given in Figure 3.1.

The possibility of a string having several leftmost derivations introduces the notion of ambiguity. Ambiguity in formal languages is similar to ambiguity encountered frequently in natural languages. The sentence *Jack was given a book by Hemingway* has two distinct structural decompositions. The prepositional phrase *by Hemingway* can modify either the verb *was given* or the noun *book*. Each of these structural decompositions represents a syntactically correct sentence.

The compilation of a computer program utilizes the derivation produced by the parser to generate machine-language code. The compilation of a program that has two derivations uses only one of the possible interpretations to produce the executable code. An unfortunate programmer may then be faced with debugging a program that is completely correct according to the language definition but does not perform as expected. To avoid this possibility—and help maintain the sanity of programmers everywhere—the definitions of computer languages should be constructed so that no ambiguity can occur. The preceding discussion of ambiguity leads to the following definition.

Definition 3.5.2

A context-free grammar G is **ambiguous** if there is a string $w \in L(G)$ that can be derived by two distinct leftmost derivations. A grammar that is not ambiguous is called **unambiguous.**

Example 3.5.1

Let G be the grammar

$$S \to aS \mid Sa \mid a$$

that generates a^+. G is ambiguous since the string aa has two distinct leftmost derivations:

$$\begin{aligned}
S &\Rightarrow aS & S &\Rightarrow Sa \\
&\Rightarrow aa & &\Rightarrow aa.
\end{aligned}$$

The language a^+ is also generated by the unambiguous grammar

$$S \to aS \mid a.$$

This grammar, being regular, has the property that all strings are generated in a left-to-right manner. The variable S remains as the rightmost symbol of the string until the recursion is halted by the application of the rule $S \to a$. □

The previous example demonstrates that ambiguity is a property of grammars, not of languages. When a grammar is shown to be ambiguous, it is often possible to construct an equivalent unambiguous grammar. This is not always the case. There are some context-free languages that cannot be generated by any unambiguous grammar. Such languages are called **inherently ambiguous**. The syntax of most programming languages, which require unambiguous derivations, is sufficiently restrictive to avoid inherent ambiguity.

Example 3.5.2

Let G be the grammar

$$S \to bS \mid Sb \mid a$$

with language b^*ab^*. The leftmost derivations

$$\begin{aligned} S &\Rightarrow bS & S &\Rightarrow Sb \\ &\Rightarrow bSb & &\Rightarrow bSb \\ &\Rightarrow bab & &\Rightarrow bab \end{aligned}$$

exhibit the ambiguity of G. The ability to generate the b's in either order must be eliminated to obtain an unambiguous grammar. L(G) is also generated by the unambiguous grammars

$$\begin{aligned} G_1&: \ S \to bS \mid aA & G_2&: \ S \to bS \mid A \\ &\quad A \to bA \mid \lambda & &\quad A \to Ab \mid a. \end{aligned}$$

In G_1, the sequence of rule applications in a leftmost derivation is completely determined by the string being derived. The only leftmost derivation of the string b^nab^m has the form

$$\begin{aligned} S &\overset{n}{\Rightarrow} b^nS \\ &\Rightarrow b^naA \\ &\overset{m}{\Rightarrow} b^nab^mA \\ &\Rightarrow b^nab^m. \end{aligned}$$

A derivation in G_2 initially generates the leading b's, followed by the trailing b's, and finally the a. □

A grammar is unambiguous if, at each step in a leftmost derivation, there is only one rule whose application can lead to a derivation of the desired string. This does not mean that there is only one applicable rule, but rather that the application of any other rule makes it impossible to complete a derivation of the string.

Consider the possibilities encountered in constructing a leftmost derivation of the string $bbabb$ using the grammar G_2 from Example 3.5.2. There are two S rules that can initiate a derivation. Derivations initiated with the rule $S \to A$ generate strings beginning with a. Consequently, a derivation of $bbabb$ must begin with the application of the rule $S \to bS$. The second b is generated by another application of the same rule. At this point, the derivation continues using $S \to A$. Another application of $S \to bS$ would generate the prefix bbb. The suffix bb is generated by two applications of $A \to Ab$. The derivation is successfully completed with an application of $A \to a$. Since the terminal string specifies the exact sequence of rule applications, the grammar is unambiguous.

Example 3.5.3

The grammar from Example 3.2.4 that generates the language $L = \{a^nb^m \mid 0 \le n \le m \le 2n\}$ is ambiguous. The string $aabbb$ can be generated by the derivations

$$\begin{aligned} S &\Rightarrow aSb & S &\Rightarrow aSbb \\ &\Rightarrow aaSbbb & &\Rightarrow aaSbbb \\ &\Rightarrow aabbb & &\Rightarrow aabbb. \end{aligned}$$

A strategy for unambiguously generating the strings of L is to initially produce a's with a single matching b. This is followed by generating a's with two b's. An unambiguous grammar that produces the strings of L in this manner is

$$\begin{aligned} &S \to aSb \mid A \mid \lambda \\ &A \to aAbb \mid abb. \end{aligned}$$

□

A derivation tree depicts the transformation of the variables in a derivation. There is a natural one-to-one correspondence between leftmost (rightmost) derivations and derivation trees. Definition 3.1.4 outlines the construction of a derivation tree directly from a leftmost derivation. Conversely, a unique leftmost derivation of a string w can be extracted from a derivation tree with frontier w. Because of this correspondence, ambiguity is often defined in terms of derivation trees. A grammar G is ambiguous if there is a string in L(G) that is the frontier of two distinct derivation trees. Figure 3.3 shows that the two leftmost derivations of the string $ababaa$ given in Figure 3.1 generate distinct derivation trees.

3.6 Context-Free Grammars and Programming Language Definition

In the preceding sections we used context-free grammars to generate "toy" languages using an alphabet with only a few elements and a small number of rules. These examples demonstrated the ability of context-free rules to produce strings that satisfy particular syntactic requirements. A programming language has a larger alphabet and more complicated syntax, increasing the number and complexity of the rules needed to define the language.

The first formal specification of a high-level programming language was given for the language ALGOL 60 by John Backus [1959] and Peter Naur [1963]. The system employed by Backus and Naur is now referred to as *Backus-Naur form,* or *BNF.* The programming language Java, whose specification was given in BNF, will be used to illustrate principles of the syntactic definition of a programming language. A complete formal definition of Java is given in Appendix IV.

A BNF description of a language is a context-free grammar; the only difference is the notation used to define the rules. We will give the rules using the context-free notation, with one exception. The subscript *opt* after a variable or a terminal indicates that it is optional. This notation reduces the number of rules that need to be written, but rules with optional components can easily be transformed into equivalent context-free rules. For example, $A \rightarrow B_{opt}$ and $A \rightarrow B_{opt}C$ can be replaced by the rules $A \rightarrow B \mid \lambda$ and $A \rightarrow BC \mid C$, respectively.

The notational conventions used in the Java rules are the same as the natural language example at the beginning of the chapter. The names of the variables indicate the components of the language that they generate and are enclosed in ⟨ ⟩. Java keywords are given in bold, and other terminal symbols are represented by character strings delimited by blanks.

The design of a programming language, like the design of a complex program, is greatly simplified utilizing modularity to develop subsets of the grammar independently. The techniques you have used in building small rule sets provide the skills needed to design a grammar for larger languages with more complicated syntaxes. These techniques include using rules to ensure the presence or relative position of elements and using recursion to generate sequences and to nest parentheses.

To illustrate the principles of language design, we will examine rules that define literals, identifiers, and arithmetic expressions in Java. Literals, strings that have a fixed type and value, are frequently used to initialize variables, to set the bounds on repetitive statements, and to store standard messages to be output. The rule for the variable ⟨*Literal*⟩ defines the types of Java literals. The Java literals, along with the variables that generate them, are

Literal	Variable	Examples
Boolean	< *BooleanLiteral* >	**true**, **false**
Character	< *CharacterLiteral* >	'a', '\n' (linefeed escape sequence), 'π',
String	< *StringLiteral* >	"" (empty string), "This is a nonempty string"
Integer	< *IntegerLiteral* >	0, 356, 1234L (long), 077 (octal), 0x1ab2 (hex)
Floating point	< *FloatingPointLiteral* >	2., .2, 2.0, 12.34, 2e3, 6.2e-5
Null	< *NullLiteral* >	**null**

Each floating point literal can have an f, F, d, or D as a suffix to indicate its precision. The definitions for the complete set of Java literals are given in rules 143–167 in Appendix IV.

We will consider the rules that define the floating point literals, since they have the most interesting syntactic variations. The four ⟨*FloatingPointLiteral*⟩ rules specify the general form of floating point literals.

$$\langle FloatingPointLiteral\rangle \rightarrow \langle Digits\rangle\ .\ \langle Digits\rangle_{opt}\langle ExponentPart\rangle_{opt}\langle FloatTypeSuffix\rangle_{opt} \mid$$
$$.\ \langle Digits\rangle\ \langle ExponentPart\rangle_{opt}\langle FloatTypeSuffix\rangle_{opt} \mid$$
$$\langle Digits\rangle\ \langle ExponentPart\rangle\langle FloatTypeSuffix\rangle_{opt} \mid$$
$$\langle Digits\rangle\ \langle ExponentPart\rangle_{opt}\langle FloatTypeSuffix\rangle$$

The variables ⟨*Digits*⟩, ⟨*ExponentPart*⟩, and ⟨*FloatTypeSuffix*⟩ generate the components that make up the literal. The variable ⟨*Digits*⟩ generates a string of digits using recursion. The nonrecursive rule ensures the presence of at least one digit.

$$\langle Digits\rangle \rightarrow \langle Digit\rangle \mid \langle Digits\rangle\langle Digit\rangle$$
$$\langle Digit\rangle \rightarrow 0 \mid \langle NonZeroDigit\rangle$$
$$\langle NonZeroDigit\rangle \rightarrow 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$
$$\langle ExponentPart\rangle \rightarrow \langle ExponentIndicator\rangle\langle SignedInteger\rangle$$
$$\langle ExponentIndicator\rangle \rightarrow e \mid E$$
$$\langle SignedInteger\rangle \rightarrow \langle Sign\rangle_{opt}\langle Digits\rangle$$
$$\langle Sign\rangle \rightarrow + \mid -$$
$$\langle FloatTypeSuffix\rangle \rightarrow f \mid F \mid d \mid D$$

The subscript *opt* in the rule ⟨*SignedInteger*⟩ → ⟨*Sign*⟩$_{opt}$⟨*Digits*⟩ indicates that a signed integer may begin with + or −, but the sign is not necessary.

The first ⟨*FloatingPointLiteral*⟩ rule generates literals of the form 1., 1.1, 1.1e, 1.e, 1.1ef, 1.f, 1.1f, and 1.ef. The leading string of digits and decimal point are required; all other components are optional. The second rule generates literals that begin with a decimal point, and the last two rules define the floating point literals without decimal points.

Identifiers are used as names of variables, types, methods, and so forth. Identifiers are defined by the rules

$$\langle Identifier\rangle \rightarrow \langle IdentifierChars\rangle$$
$$\langle IdentifierChars\rangle \rightarrow \langle JavaLetter\rangle \mid \langle JavaLetter\rangle\langle JavaLetterOrDigit\rangle$$

where the Java letters include the letters A to Z and a to z, the underscore _, and the dollar sign $, along with other characters represented in the Unicode encoding.

The definition of statements in Java begins with the variable ⟨*Statement*⟩:

$$\langle Statement\rangle \rightarrow \langle StatementWithoutTrailing\ Substatement\rangle \mid \langle LabeledStatement\rangle \mid$$
$$\langle IfThenStatement\rangle \mid \langle IfThenElseStatement\rangle \mid$$
$$\langle WhileStatement\rangle \mid \langle ForStatement\rangle .$$

Statements without trailing substatements include blocks and the **do** and **switch** statements. The entire set of statements is given in rules 73–75 in Appendix IV. Like the rules for the literals, the statement rules define the high-level structure of a statement. For example, **if-then** and **do** statements are defined by

$$\langle \textit{IfThenStatement} \rangle \rightarrow \textbf{if } (\langle \textit{Expression} \rangle) \langle \textit{Statement} \rangle$$
$$\langle \textit{DoStatement} \rangle \rightarrow \textbf{do } \langle \textit{Statement} \rangle \textbf{ while } (\langle \textit{Expression} \rangle).$$

The occurrence of the variable ⟨*Statement*⟩ on the right-hand side of the preceding rules generates the statements to be executed after the condition in the **if-then** statement and in the loop in the **do** loop.

The evaluation of expressions is the key to numeric computation and checking the conditions in **if-then**, **do**, **while**, and **switch** statements. The syntax of expressions is defined by the rules 118–142 in Appendix IV. The syntax is complicated because Java has numeric and Boolean expressions that may utilize postfix, prefix, or infix operators. Rather than describing individual rules, we will look at several subderivations that occur in the derivation of a simple arithmetic assignment.

The first steps transform the variable ⟨*Expression*⟩ to an assignment:

$$\begin{aligned}
\langle \textit{Expression} \rangle &\Rightarrow \langle \textit{AssignmentExpression} \rangle \\
&\Rightarrow \langle \textit{Assignment} \rangle \\
&\Rightarrow \langle \textit{LeftHandSide} \rangle \langle \textit{AssignmentOperator} \rangle \langle \textit{AssignmentExpression} \rangle \\
&\Rightarrow \langle \textit{ExpressionName} \rangle \langle \textit{AssignmentOperator} \rangle \langle \textit{AssignmentExpression} \rangle \\
&\Rightarrow \langle \textit{Identifier} \rangle \langle \textit{AssignmentOperator} \rangle \langle \textit{AssignmentExpression} \rangle \\
&\Rightarrow \langle \textit{Identifier} \rangle = \langle \textit{AssignmentExpression} \rangle.
\end{aligned}$$

The next step is to derive ⟨*AdditiveExpression*⟩ from ⟨*AssignmentExpression*⟩.

$$\begin{aligned}
\langle \textit{AssignmentExpression} \rangle &\Rightarrow \langle \textit{ConditionalExpression} \rangle \\
&\Rightarrow \langle \textit{ConditionalOrExpression} \rangle \\
&\Rightarrow \langle \textit{ConditionalAndExpression} \rangle \\
&\Rightarrow \langle \textit{InclusiveOrExpression} \rangle \\
&\Rightarrow \langle \textit{ExclusionOrExpression} \rangle \\
&\Rightarrow \langle \textit{AndExpression} \rangle \\
&\Rightarrow \langle \textit{EqualityExpression} \rangle \\
&\Rightarrow \langle \textit{RelationalExpression} \rangle \\
&\Rightarrow \langle \textit{ShiftExpression} \rangle \\
&\Rightarrow \langle \textit{AdditiveExpression} \rangle.
\end{aligned}$$

Derivations beginning with ⟨*AdditiveExpression*⟩ produce correctly formed expressions with additive operators, multiplicative operators, and parentheses. For example,

$$\begin{aligned}
\langle \textit{AdditiveExpression} \rangle &\Rightarrow \langle \textit{AdditiveExpression} \rangle + \langle \textit{MultiplicativeExpression} \rangle \\
&\Rightarrow \langle \textit{MultiplicativeExpression} \rangle + \langle \textit{MultiplicativeExpression} \rangle \\
&\Rightarrow \langle \textit{UnaryExpression} \rangle + \langle \textit{MultiplicativeExpression} \rangle \\
&\stackrel{*}{\Rightarrow} \langle \textit{Identifier} \rangle + \langle \textit{MultiplicativeExpression} \rangle \\
&\Rightarrow \langle \textit{Identifier} \rangle + \\
&\quad \langle \textit{MultiplicativeExpression} \rangle * \langle \textit{MultiplicativeExpression} \rangle
\end{aligned}$$

begins such a derivation. Derivations from ⟨*UnaryExpression*⟩ can produce literals, variables, or (⟨*Expression*⟩) to obtain nested parentheses.

The rules that define identifiers, literals, and expressions show how the design of a large language is decomposed into creating rules for frequently recurring subsets of the language. The resulting variables ⟨*Identifier*⟩, ⟨*Literal*⟩, and ⟨*Expression*⟩ become the building blocks for higher-level rules.

The start symbol of the grammar is ⟨*CompilationUnit*⟩ and the derivation of a Java program begins with the rule

$$\langle \textit{CompulationUnit} \rangle \rightarrow \langle \textit{PackageDeclaration} \rangle_{opt} \langle \textit{ImportDeclarations} \rangle_{opt} \langle \textit{TypeDeclarations} \rangle_{opt}.$$

A string of terminal symbols derivable from this rule is a syntactically correct Java program.

Exercises

1. Let G be the grammar

$$S \rightarrow abSc \mid A$$
$$A \rightarrow cAd \mid cd.$$

   a) Give a derivation of $ababccddcc$.
   b) Build the derivation tree for the derivation in part (a).
   c) Use set notation to define L(G).

2. Let G be the grammar

$$S \rightarrow ASB \mid \lambda$$
$$A \rightarrow aAb \mid \lambda$$
$$B \rightarrow bBa \mid ba.$$

   a) Give a leftmost derivation of $aabbba$.
   b) Give a rightmost derivation of $abaabbbabbaa$.

c) Build the derivation tree for the derivations in parts (a) and (b).

d) Use set notation to define L(G).

3. Let G be the grammar

$$S \to SAB \mid \lambda$$
$$A \to aA \mid a$$
$$B \to bB \mid \lambda.$$

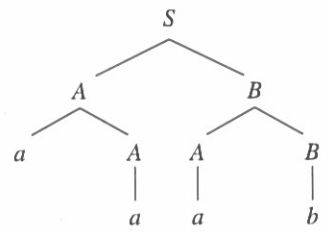
a) Give a leftmost derivation of $abbaab$.

b) Give two leftmost derivations of aa.

c) Build the derivation tree for the derivations in part (b).

d) Give a regular expression for L(G).

4. Let DT be the derivation tree



a) Give a leftmost derivation that generates the tree DT.

b) Give a rightmost derivation that generates the tree DT.

c) How many different derivations are there that generate DT?

5. Give the leftmost and rightmost derivations corresponding to each of the derivation trees given in Figure 3.3.

6. For each of the following context-free grammars, use set notation to define the language generated by the grammar.

a) $S \to aaSB \mid \lambda$
$B \to bB \mid b$

b) $S \to aSbb \mid A$
$A \to cA \mid c$

c) $S \to abSdc \mid A$
$A \to cdAba \mid \lambda$

d) $S \to aSb \mid A$
$A \to cAd \mid cBd$
$B \to aBb \mid ab$

e) $S \to aSB \mid aB$
$B \to bb \mid b$

7. Construct a grammar over $\{a, b, c\}$ whose language is $\{a^n b^{2n} c^m \mid n, m > 0\}$.

8. Construct a grammar over $\{a, b, c\}$ whose language is $\{a^n b^m c^{2n+m} \mid n, m > 0\}$.

9. Construct a grammar over $\{a, b, c\}$ whose language is $\{a^n b^m c^i \mid 0 \le n + m \le i\}$.

10. Construct a grammar over $\{a, b\}$ whose language is $\{a^m b^n \mid 0 \le n \le m \le 3n\}$.

11. Construct a grammar over $\{a, b\}$ whose language is $\{a^m b^i a^n \mid i = m + n\}$.

12. Construct a grammar over $\{a, b\}$ whose language contains precisely the strings with the same number of a's and b's.

***13.** Construct a grammar over $\{a, b\}$ whose language contains precisely the strings of odd length that have the same symbol in the first and middle positions.

14. For each of the following regular grammars, give a regular expression for the language generated by the grammar.

a) $S \to aA$
$A \to aA \mid bA \mid b$

b) $S \to aA$
$A \to aA \mid bB$
$B \to bB \mid \lambda$

c) $S \to aS \mid bA$
$A \to bB$
$B \to aB \mid \lambda$

d) $S \to aS \mid bA \mid \lambda$
$A \to aA \mid bS$

For Exercises 15 through 25, give a regular grammar that generates the described language.

15. The set of strings over $\{a, b, c\}$ in which all the a's precede the b's, which in turn precede the c's. It is possible that there are no a's, b's, or c's.

16. The set of strings over $\{a, b\}$ that contain the substring aa and the substring bb.

17. The set of strings over $\{a, b\}$ in which the substring aa occurs at least twice. (*Hint:* Beware of the substring aaa.)

18. The set of strings over $\{a, b\}$ that contain the substring ab and the substring ba.

19. The set of strings over $\{a, b\}$ in which the number of a's is divisible by three.

20. The set of strings over $\{a, b\}$ in which every a is either immediately preceded or immediately followed by b, for example, $baab$, aba, and b.

21. The set of strings over $\{a, b\}$ that do not contain the substring aba.

22. The set of strings over $\{a, b\}$ in which the substring aa occurs exactly once.

23. The set of strings of odd length over $\{a, b\}$ that contain exactly two b's.

***24.** The set of strings over $\{a, b, c\}$ with an odd number of occurrences of the substring ab.

25. The set of strings over $\{a, b\}$ with an even number of a's or an odd number of b's.

26. The grammar in Figure 3.1 generates $(b^*ab^*ab^*)^+$, the set of all strings with a positive, even number of a's. Prove this.

27. Prove that the grammar given in Example 3.2.2 generates the prescribed language.

28. Let G be the grammar

$$S \to aSb \mid B$$
$$B \to bB \mid b.$$

Prove that $L(G) = \{a^n b^m \mid 0 \le n < m\}$.

29. Let G be the grammar

$$S \to aSaa \mid B$$
$$B \to bbBdd \mid C$$
$$C \to bd.$$

a) What is L(G)?

b) Prove that L(G) is the set given in part (a).

*** 30.** Let G be the grammar

$$S \to aSbS \mid aS \mid \lambda.$$

Prove that every prefix of a string in L(G) has at least as many a's as b's.

31. Let G be a context-free grammar and $w \in L(G)$. Prove that there is a rightmost derivation of w in G.

32. Let G be the grammar

$$S \to aS \mid Sb \mid ab.$$

a) Give a regular expression for L(G).

b) Construct two leftmost derivations of the string $aabb$.

c) Build the derivation trees for the derivations from part (b).

d) Construct an unambiguous grammar equivalent to G.

33. For each of the following grammars, give a regular expression or set-theoretic definition for the language of the grammar. Show that the grammar is ambiguous and construct an equivalent unambiguous grammar.

a) $S \to aaS \mid aaaaaS \mid \lambda$

b) $S \to aSA \mid \lambda$
$A \to bA \mid \lambda$

c) $S \to aSb \mid aAb$
$A \to cAd \mid B$
$B \to aBb \mid \lambda$

d) $S \to AaSbB \mid \lambda$
$A \to aA \mid a$
$B \to bB \mid \lambda$

* e) $S \to A \mid B$
$A \to abA \mid \lambda$
$B \to aBb \mid \lambda$

34. Let G be the grammar

$$S \to aA \mid \lambda$$
$$A \to aA \mid bB$$
$$B \to bB \mid b.$$

a) Give a regular expression for L(G).

b) Prove that G is unambiguous.

35. Let G be the grammar

$$S \to aS \mid aA \mid a$$
$$A \to aAb \mid ab.$$

a) Give a set-theoretic definition of L(G).

b) Prove that G is unambiguous.

36. Let G be the grammar

$$S \to aS \mid bA \mid \lambda$$
$$A \to bA \mid aS \mid \lambda.$$

Give a regular expression for L(G). Is G ambiguous? If so, give an unambiguous grammar that generates L(G). If not, prove it.

37. Construct unambiguous grammars for the languages $L_1 = \{a^n b^n c^m \mid n, m > 0\}$ and $L_2 = \{a^n b^m c^m \mid n, m > 0\}$. Construct a grammar G that generates $L_1 \cup L_2$. Prove that G is ambiguous. This is an example of an inherently ambiguous language. Explain, intuitively, why every grammar generating $L_1 \cup L_2$ must be ambiguous.

38. Use the definition of Java in Appendix IV to construct a derivation of the string 1.3*e*2 from the variable ⟨*Literal*⟩.

*** 39.** Let G_1 and G_2 be the following grammars:

G_1: $S \to aABb$
$A \to aA \mid a$
$B \to bB \mid b$

G_2: $S \to AABB$
$A \to AA \mid a$
$B \to BB \mid b.$

a) For each variable X, show that the right-hand side of every X rule of G_1 is derivable from the corresponding variable X using the rules of G_2. Use this to conclude that $L(G_1) \subseteq L(G_2)$.

b) Prove that $L(G_1) = L(G_2)$.

* **40.** A **right-linear grammar** is a context-free grammar, each of whose rules has one of the following forms:

i) $A \to w$, or

ii) $A \to wB$,

where $w \in \Sigma^*$. Prove that a language L is generated by a right-linear grammar if, and only if, L is generated by a regular grammar.

41. Try to construct a regular grammar that generates the language $\{a^n b^n \mid n \geq 0\}$. Explain why none of your attempts succeed.

42. Try to construct a context-free grammar that generates the language $\{a^n b^n c^n \mid n \geq 0\}$. Explain why none of your attempts succeed.

Bibliographic Notes

Context-free grammars were introduced by Chomsky [1956], [1959]. Backus-Naur form was developed by Backus [1959]. This formalism was used to define the programming language ALGOL; see Naur [1963]. The BNF definition of Java is given in Appendix IV. The equivalence of context-free languages and the languages generated by BNF definitions was noted by Ginsburg and Rice [1962].

Properties of ambiguity are examined in Floyd [1962], Cantor [1962], and Chomsky and Schutzenberger [1963]. Inherent ambiguity was first noted in Parikh [1966]. A proof that the language in Exercise 37 is inherently ambiguous can be found in Harrison [1978]. Closure properties for ambiguous and inherently ambiguous languages were established by Ginsburg and Ullian [1966a, 1966b].

CHAPTER 4

Normal Forms for Context-Free Grammars

The definition of a context-free grammar permits unlimited flexibility in the form of the right-hand side of a rule. This flexibility is advantageous for designing grammars, but the lack of structure makes it difficult to establish general relationships about grammars, derivations, and languages. Normal forms for context-free grammars impose restrictions on the form of the rules to facilitate the analysis of context-free grammars and languages. Two properties characterize a normal form:

i) The grammars that satisfy the normal form requirements should generate the entire set of context-free languages.

ii) There should be an algorithmic transformation of an arbitrary context-free grammar into an equivalent grammar in the normal form.

In this chapter we introduce two important normal forms for context-free grammars, the Chomsky and Greibach normal forms. Transformations are developed to convert an arbitrary context-free grammar into an equivalent grammar that satisfies the conditions of the normal form. The transformations consist of a series of rule modifications, additions, and deletions, each of which preserves the language of the original grammar.

The restrictions imposed on the rules by a normal form ensure that derivations of the grammar have certain desirable properties. The derivation trees for derivations in a Chomsky normal form grammar are binary trees. In Chapter 7 we will use the relationship between the depth and number of leaves of a binary tree to guarantee the existence of repetitive patterns in strings in a context-free language. We will also use the properties of derivations