Formal Grammars

The automatic translation of text from one language to another has long been a subject of interest both to linguists and computer scientists. The years following the introduction of high-level programming languages witnessed an active and fruitful search for precise formalizations of language to aid in understanding and simplifying the description of the translation process. Although the enticing goal of quality translation of natural language texts by computer has remained elusive, the theory has had profound influence on the design and processing of computer programming languages.

We present here the fundamental concepts of formal grammars and their relation to the languages that they describe. Although many classes of grammars and languages have been studied, our attention is focused on a hierarchy of four major classes of languages, the "Chomsky hierarchy." In subsequent chapters, we relate these classes of languages to important classes of abstract automata, and use the structure of the automata to study the structure of the languages. The formalism developed in this chapter is basic to this later work and provides insights not readily obtained from consideration of abstract machines alone.

3.1 Representations of Languages

According to the dictionary, a language is "the aggregate of words and of methods of combining them used by a particular community, nation, or race of people." This statement is hardly specific enough to serve as an
adequate mathematical description of the term "language." Linguists agree that any adequate formalism for natural language must cope with the infinity of possible sentences apparent in language use. One cannot hope to exhaustively list all possible sentences in a language, for each user of a language is able to speak sentences that have never previously been spoken and yet are intelligible to other users of the language. Similarly, one cannot hope to write down all possible Algol or Fortran computer programs, for any user is able to write programs never before written that will run perfectly well on a computer. Thus the central problem of describing a language is to provide a finite specification for an essentially infinite class of objects.

Given an alphabet \( V \), the set \( V^* \) is the totality of all possible strings on \( V \), and a language \( L \) on \( V \) is an arbitrary subset of \( V^* \). A description for \( L \) is sufficient for our purposes if it can be used to decide whether a given member of \( V^* \) is or is not a member of \( L \). This notion of sufficiency may appear to ignore questions concerning the "meaning" of sentences. However, we shall find that the structural descriptions of sentences provided by our representations of languages are closely related to the interpretations attached to those sentences by the language user.

For English, the alphabet might be the 26 letters in both upper- and lowercase, the ten digits, the space, and a few punctuation marks—the symbols on a typewriter keyboard. A formal grammar for English would be a finite set of rules with which one could decide whether any typed line is a legitimate sentence in English.

For a computer programming language, the alphabet is the collection of all nondivisible symbols that may appear in a program. A "sentence" in a programming language is usually considered to be any string of these symbols that represents a complete program. A satisfactory grammar for a programming language should permit one to determine by a mechanical procedure whether an arbitrary sequence of symbols is a "well-formed" program. We shall see how a grammar can provide a structural description of a well-formed program that is useful for assigning meaning to the program.

The requirement that a language representation be finite is astoundingly severe, for it means that infinitely many languages have no representations! If \( V \) is an alphabet, then each subset of \( V^* \) is a language on \( V \); that is, the set of languages on \( V \) is the power set of \( V^* \) and is thus uncountable. On the other hand, we shall find that the number of finite representations can be at best countably infinite. Hence uncountably many languages have no finite representation. Perhaps we should not be bothered by this fact, for languages that are not finitely representable have so little order to their structure that they are likely to be of no practical interest. However, one might wonder whether the natural languages belong to this class of unrepresentable languages.
One means of representing a language has been suggested in Chapter 1: an accepter machine for the language. If the accepter has a finite number of functionally relevant states, it constitutes a finite representation. We shall find, however, that the class of languages for which such finite-state machines provide structural descriptions is rather limited. Thus we shall be led to consider other, more powerful forms of abstract machines as possible finite representations of languages.

The definition of language by an accepter machine is an analytic approach to language representation, since the machine directly tests an arbitrary input sentence for membership in the language. Alternatively, we may use a generative representation of a language—a finite set of rules which, if followed in any valid order, will construct only strings that are sentences of the language. Formal grammars are such representations.

Can a generative representation of a language be used to determine whether an arbitrary string is a member of the language? In later chapters we shall study mechanical procedures that decide whether any particular string is generated by a given formal grammar. These procedures work for all but the most general form of grammar; for that form, we shall prove that no mechanical procedure exists that can decide whether a given string is generated by a grammar.

### 3.2 Basic Concepts of Grammars

#### 3.2.1 An Illustration from English

The essential concept in the grammatical description of language is familiar to every grammar school student of English: the construction of sentences by the successive application of grammatical rules. A rule specifies how a phrase may be rewritten as the concatenation of constituent phrases. For example, the sentence

```
Jack and Jill ran up the hill
```

consists of two major phrases:

```
<subject> <predicate>
```

```
Jack and Jill ran up the hill
```

This sentence construction can be described by the grammatical rule

```
1. <sentence> \rightarrow <subject> <predicate>
```

The rule states that a phrase of type `<sentence>` may be rewritten as a phrase of type `<subject>` followed by a phrase of type `<predicate>`. We have enclosed
the names of phrase types in angular brackets to prevent possible confusion
with words or symbols that may appear in sentences of the language being
described.

The \texttt{<subject>} of the illustrative sentence is a pair of nouns connected
by a conjunction. This construction could be described by the grammatical
rule

\[
\texttt{<subject>} \rightarrow \texttt{<noun> <conjunction> <noun>}
\]

However, we prefer to choose rules that explain the structure of as large a
collection of phrases as possible. Consider the rules

2. \texttt{<subject>} \rightarrow \texttt{<noun phrase>}
3. \texttt{<noun phrase>} \rightarrow \texttt{<noun>}
4. \texttt{<noun phrase>} \rightarrow \texttt{<noun> <conjunction> <noun phrase>}

Rule 2 states that a phrase of type \texttt{<subject>} may be a phrase of type \texttt{<noun phrase>}. Rules 3 and 4 state that a phrase of type \texttt{<noun phrase>}
may take either of two forms: (1) it may be a phrase of type \texttt{<noun> appearing alone,}
or (2) it may be rewritten as a phrase of type \texttt{<noun> followed by phrases of}
types \texttt{<conjunction> and <noun phrase> in succession. Thus a noun phrase}
may have a noun phrase as one of its constituents. It is this recursive property
of grammatical rules that permits the finite representation of infinite collec-
tions of sentences.

The \texttt{<subject>} of the sample sentence is generated by one application
each of rules 2 to 4 above, and one application of each of the following rules:

5. \texttt{<conjunction>} \rightarrow and
6. \texttt{<noun>} \rightarrow Jack
7. \texttt{<noun>} \rightarrow Jill

as follows:

\[
\begin{align*}
\texttt{<subject>} & \rightarrow \texttt{<noun phrase> (rule 2)} \\
\texttt{<noun phrase>} & \rightarrow \texttt{<noun> <conjunction> <noun phrase> (rule 4)} \\
\texttt{<noun> and <noun phrase>} & \rightarrow \texttt{<noun> <conjunction> <noun phrase> (rule 4)} \\
\texttt{Jack and <noun phrase>} & \rightarrow \texttt{<noun> <conjunction> <noun phrase> (rule 4)} \\
\texttt{Jack and <noun>} & \rightarrow \texttt{<noun> <conjunction> <noun phrase> (rule 4)} \\
\texttt{Jack and Jill} & \rightarrow \texttt{<noun> <conjunction> <noun phrase> (rule 4)}
\end{align*}
\]

These lines, known as sentential forms, make up what is called a derivation
of the subject phrase according to rules 2 through 7. Each line of a derivation
results from rewriting the preceding line as permitted by some rule. It is
often possible to represent a derivation by a tree diagram, as in Figure 3.1. The derivation above corresponds to the left part of the diagram.

The predicate of the sample sentence consists of a verb and a prepositional phrase:

8. \(<\text{predicate}> \rightarrow \langle\text{verb}\rangle \langle\text{prepositional phrase}\rangle\\
\quad \text{ran} \quad \text{up the hill}

9. \langle\text{verb}\rangle \rightarrow \text{ran}

The structure of the prepositional phrase is described by the following rule:

10. \langle\text{prepositional phrase}\rangle \rightarrow \langle\text{preposition}\rangle \langle\text{article}\rangle \langle\text{noun}\rangle

in which the constituents of the right side may be replaced as follows:

11. \langle\text{preposition}\rangle \rightarrow \text{up}
12. \langle\text{article}\rangle \rightarrow \text{the}
13. \langle\text{noun}\rangle \rightarrow \text{hill}

Rules 1 through 13 constitute a grammar. The given sentence is one member of the language generated by this grammar. Its structural description, the tree diagram in Figure 3.1, shows how the productions of the grammar

![Figure 3.1. Structural description of an English sentence.](image-url)
can be applied to generate the sentence. Each node of the tree diagram corresponds to the application of a particular grammatical rule, as indicated by the numbers in the figure. Since the productions specify how one type of phrase is formed by combining other phrase types, these grammars are often called phrase-structure grammars.

Any sentence for which a grammar provides a structural description is a valid sentence in the language generated by the grammar. For instance, our grammar also generates these sentences:

Jill and Jack ran up the hill

Jack and Jack and Jill ran up the hill

Jack and Jill and Jack and Jill ran up the hill

The reader should be able to construct the corresponding structural descriptions. These examples show how a finite set of rules, through repeated applications, can generate an arbitrarily large collection of sentences.

Other sentences belong to our language:

hill ran up the Jack and Jill

Jack and hill ran up the Jill and hill

We would normally regard these as nonsense sentences in English, but no constraints have been incorporated into our grammar to ensure that only sentences meaningful to a user of English are generated. Our grammar provides only that certain elementary rules concerning parts of speech be obeyed. The formulation of an adequate grammar for a natural language is a major unrealized goal of linguistic research.

3.2.2 An Illustration from Algol

For a second introductory example of a phrase-structure grammar, we consider a portion of the computer programming language Algol. Algol was the first practical programming language for which a complete set of grammatical rules was formulated. A program in Algol contains a sequence of phrases of type ⟨statement⟩. We consider a simplified form of Algol assignment statement, of which the phrase

\[ w := \text{if } x < y \text{ then } 0 \text{ else } x + y + 1 \]

is an example. This phrase has the following interpretation: if \( x < y \), assign the value 0 to \( w \); otherwise, assign the value of \( (x + y + 1) \) to \( w \).

This phrase is written with symbols from the alphabet

\[ \{w, x, y, 0, 1, +, -, :=, \text{if, then, else}\} \]

which is just a portion of the complete Algol alphabet. Note that if, then, and else are considered single, nondivisible symbols. The overall structure
of the phrase may be described by the following grammatical rule:

1. \[\langle \text{statement} \rangle \longrightarrow \langle \text{left part} \rangle \quad \langle \text{expression} \rangle\]
   \[
   w := \quad \text{if } x < y \text{ then } 0 \text{ else } x + y + 1
   \]

The meaning of an assignment statement to a programmer is that the phrase of type \(\langle \text{expression} \rangle\) represents a numerical value which is to be assigned to the variable defined by the \(\langle \text{left part} \rangle\) phrase. In the example, the phrase of type \(\langle \text{expression} \rangle\) is a conditional construction formed according to the following grammatical rule:

3. \[\langle \text{expression} \rangle \longrightarrow \text{if } \langle \text{Boolean} \rangle \text{ then } \langle \text{arithmetic} \rangle \text{ else } \langle \text{expression} \rangle\]
   \[
   x < y \quad 0 \quad x + y + 1
   \]

Here the phrase type \(\langle \text{Boolean} \rangle\) is interpreted as representing a value of either \textit{true} or \textit{false}. The conditional construction is to be interpreted as follows: its value is that of the \(\langle \text{arithmetic} \rangle\) phrase if the \(\langle \text{Boolean} \rangle\) phrase has value \textit{true}; otherwise, its value is that of the \(\langle \text{expression} \rangle\) phrase. The recurrence of the phrase type \(\langle \text{expression} \rangle\) in this rule permits conditional constructions to be nested to arbitrary depth.

The structure of the phrase
\[
x + y + 1
\]
is described by the rules

4. \[\langle \text{expression} \rangle \longrightarrow \langle \text{arithmetic} \rangle\]
7. \[\langle \text{arithmetic} \rangle \longrightarrow \langle \text{term} \rangle + \langle \text{arithmetic} \rangle\]
8. \[\langle \text{arithmetic} \rangle \longrightarrow \langle \text{term} \rangle\]

The phrase type \(\langle \text{term} \rangle\) is interpreted as being either a number or a variable-identifying letter; the value of a \(\langle \text{term} \rangle\) is either the number itself or the value of the variable. Rule 7, recursive with respect to the phrase type \(\langle \text{arithmetic} \rangle\), permits phrases containing an arbitrary number of \(\langle \text{term} \rangle\) phrases separated by the symbol \(+\).

The appearance of numbers and variable-denoting letters as \(\langle \text{term} \rangle\) phrases requires additional rules:

9. \[\langle \text{term} \rangle \longrightarrow \langle \text{identifier} \rangle\]
10. \[\langle \text{term} \rangle \longrightarrow 0\]
11. \[\langle \text{term} \rangle \longrightarrow 1\]
14. \[\langle \text{identifier} \rangle \longrightarrow x\]
15. \[\langle \text{identifier} \rangle \longrightarrow y\]

Figure 3.2 is a structural description of the phrase \(x + y + 1\).

The \(\langle \text{Boolean} \rangle\) phrase
\[
x < y
\]
is interpreted as a comparison of the values of two arithmetic expressions.
This construction is described by the single grammatical rule

5. \( \langle \text{Boolean} \rangle \rightarrow \langle \text{arithmetic} \rangle \cdot \langle \text{arithmetic} \rangle \)

\( x \)

\( y \)

That the letters \( x \) and \( y \) by themselves constitute phrases of type \( \langle \text{arithmetic} \rangle \) is already established by rules 8, 9, 14, and 15. Finally, the form of the \( \langle \text{left part} \rangle \) phrase is specified by

2. \( \langle \text{left part} \rangle \rightarrow \langle \text{identifier} \rangle := \)

and for the sample statement we need the additional rule

13. \( \langle \text{identifier} \rangle \rightarrow w \)

The complete structural description of the sample statement is given in Figure 3.3, and includes the number of the rewriting rule used in the expansion of each phrase.

To simplify further discussion, we introduce a notational convention used throughout our study of formal grammars: \textit{capital letters will be used to denote phrase types}. For the present example, let us use the following letters:

\[
\begin{align*}
S & \rightarrow \langle \text{statement} \rangle \\
L & \rightarrow \langle \text{left part} \rangle \\
E & \rightarrow \langle \text{expression} \rangle \\
I & \rightarrow \langle \text{identifier} \rangle \\
B & \rightarrow \langle \text{Boolean} \rangle \\
A & \rightarrow \langle \text{arithmetic} \rangle \\
T & \rightarrow \langle \text{term} \rangle \\
\end{align*}
\]
Figure 3.3. Structural description of the Algol statement \( w := \text{if } x < y \text{ then } 0 \text{ else } x + y + 1. \)

Expressed in terms of these letters, a more complete grammar for phrases of type \( \langle \text{statement} \rangle \) consists of the following rules:

1. \( S \rightarrow LE \)
2. \( L \rightarrow I := \)
3. \( E \rightarrow \text{if } B \text{ then } A \text{ else } E \)
4. \( E \rightarrow A \)
5. \( B \rightarrow A < A \)
6. \( B \rightarrow A = A \)
7. \( A \rightarrow T + A \)
8. \( A \rightarrow T \)
9. \( T \rightarrow I \)
10. \( T \rightarrow 0 \)
11. \( T \rightarrow 1 \)
12. \( T \rightarrow (E) \)
13. \( I \rightarrow w \)
14. \( I \rightarrow x \)
15. \( I \rightarrow y \)
Figure 3.4. Structural description of the Algol statement $x := (1 + (0 + x))$. 
This set of rewriting rules is a **formal grammar** in which the rules are known as *productions*. Productions 6 and 12 have been included so that forms of Algol phrases containing the symbols (,), and = may be generated. The interpretation of rule 6 should be clear. Rule 12 permits a phrase of type \langle\text{term}\rangle to be written as an expression enclosed within parentheses. The grammar now generates a wide variety of assignment statements. Figures 3.4 and 3.5 give structural descriptions of two further examples. In each case the tree diagram shows clearly how meaning should be assigned to each phrase according to our informal discussion.

![Tree Diagram]

*Figure 3.5. Structural description of the Algol statement* $y := \text{if } (x + y) = 1 \text{ then } 0 \text{ else } (\text{if } x < 0 \text{ then } x \text{ else } 0)$. 
Is the above grammar an adequate representation of the Algol assignment statement? Admitting that the grammar is incomplete, the answer still depends on one's objective. Each string generated by the grammar is a legal Algol phrase according to the official syntax of Algol. But the grammar generates such phrases as

\[ x ::= x \]

and

\[ x ::= \text{if} \ w > w \ \text{then} \ x + y \ \text{else} \ x \]

which would be of no value in a computer program. Yet it is true that most compilers for Algol accept such statements without comment.

We shall see that it may be possible to construct several structural descriptions of a phrase according to a given set of productions. Such a grammar is then an ambiguous representation of a language and can easily lead to incorrect translation by a compiler or misinterpretation by a person. Questions concerning ambiguity are therefore of basic interest to the language designer; they are studied in this and later chapters.

### 3.3 Formal Grammars

From the two examples of phrase-structure grammars, we see that a grammar has three major components:

1. The alphabet or collection of symbols from which sentences of the described language are constructed. These are called the *terminal letters* of the grammar, since the generation of a sentence through the application of grammatical rules must terminate in a string containing only these symbols.

2. A set of phrase-denoting symbols called the *nonterminal letters* of the grammar.

3. The collection of grammatical rules or *productions*.

To complete the specification of a language, a starting point for applying the productions is necessary. The sign \( \Sigma \), known as the *sentence symbol*, is reserved for this purpose in our work with formal grammars.

We are now ready to develop a formal characterization of grammars and begin a precise study of their properties. We start with a more general definition of formal grammar than is necessary for the illustrative examples given above. Our purpose is to show the full scope of rewriting rules as finite descriptions of sets of strings. The general treatment also provides background for comparison with the rewriting rules of Post systems studied in Chapter 14.

**Definition 3.1:** A *formal grammar* is a four-tuple

\[ G = (N, T, P, \Sigma) \]
where

- \( N \) is a finite set of nonterminal symbols
- \( T \) is a finite set of terminal symbols
- \( N \) and \( T \) are disjoint: \( N \cap T = \emptyset \)
- \( P \) is a finite set of productions
- \( \Sigma \) is the sentence symbol; \( \Sigma \notin (N \cup T) \)

Each production in \( P \) is an ordered pair of strings \((\alpha, \beta)\),

\[
\alpha = \phi A \psi \\
\beta = \phi \omega \psi
\]

in which \( \omega, \phi, \) and \( \psi \) are possibly empty strings in \((N \cup T)^*\) and \( A \) is \( \Sigma \) or a nonterminal letter. We usually write a production \((\alpha, \beta)\) as

\[
\alpha \rightarrow \beta
\]

Formal grammars as defined here are also known as phrase-structure grammars.

The process of generating a sentence according to a formal grammar is the successive rewriting of sentential forms through the use of the productions of the grammar, starting with the sentential form \( \Sigma \). The sequence of sentential forms required to generate a sentence constitutes a derivation of the sentence according to the grammar.

**Definition 3.2:** Let \( G \) be a formal grammar. A string of symbols in \( (N \cup T)^* \cup \{\Sigma\} \) is known as a sentential form. If \( \alpha \rightarrow \beta \) is a production of \( G \) and \( \omega = \phi \alpha \psi \) and \( \omega' = \phi \beta \psi \) are sentential forms, we say that \( \omega' \) is immediately derived from \( \omega \) in \( G \), and we indicate this relation by writing \( \omega \rightarrow \omega' \). If \( \omega_1, \omega_2, \ldots, \omega_n \) is a sequence of sentential forms such that \( \omega_1 \rightarrow \omega_2 \rightarrow \cdots \rightarrow \omega_n \), we say that \( \omega_n \) is derivable from \( \omega_1 \) and indicate this relation by writing \( \omega_1 \rightarrow^* \omega_n \). The sequence \( \omega_1, \omega_2, \ldots, \omega_n \) is called a derivation of \( \omega_n \) from \( \omega_1 \) according to \( G \).

**Definition 3.3:** The language \( L(G) \) generated by a formal grammar \( G \) is the set of terminal strings derivable from \( \Sigma \):

\[
L(G) = \{ \omega \in T^* | \Sigma \rightarrow^* \omega \}
\]

If \( \omega \in L(G) \), we say that \( \omega \) is a string, a sentence, or a word in the language generated by \( G \).

According to the definitions, each production of a formal grammar is a grammatical rule permitting modification of a sentential form by substitution of an arbitrary string for a particular phrase-denoting letter. The application of a production \( \phi A \psi \rightarrow \phi \omega \psi \) to rewriting the sentential form \( \omega \),
as $\omega_{i+1}$ is illustrated in Figure 3.6. The left side of the production is matched with a substring of $\omega_i$, and the phrase letter $A$ is replaced with the string $\omega$ to yield $\omega_{i+1}$.

Two generalizations have been made beyond the concepts used in the examples of Section 3.2. First, a substitution is conditioned on the phrase symbol appearing in a particular context specified by strings $\varphi$ and $\psi$ of the production. Second, the string $\omega$ substituted for a phrase letter may be the empty string $\lambda$, permitting arbitrary erasure of certain nonterminals in the course of a derivation. This second generalization permits a sentential form to decrease in length in a derivation step.

It would appear possible to define more general classes of grammars. For example, we could allow productions of the form $\alpha \longrightarrow \beta$ in which

$$\alpha \righthalfarrow \text{may be any nonempty string in } (N \cup T)^* \cup \{\Sigma\}$$

$$\beta \righthalfarrow \text{may be any (possibly empty) string in } (N \cup T)^*$$

Grammars in which such rules are permitted are sometimes called semi-Thue systems. They can be shown to be no more general that the grammars of

![Figure 3.6. Application of a production $\varphi A \psi \longrightarrow \varphi \omega \psi$ in a derivation step $\omega_i \longrightarrow \omega_{i+1}$.
]
Definition 3.1: unrestricted rules do not permit the representation of any language that cannot be represented by our grammars. (See Problem 3.18.)

In view of the equivalence between unrestricted forms of productions and those allowed by our definitions, we will occasionally use productions like

\[ AB \rightarrow BA \]

where convenient in formal discussions. Were we to adhere strictly to the forms of Definition 3.1, these four productions would be required to interchange \( A \) and \( B \):

\[ AB \rightarrow XB \]
\[ XB \rightarrow XY \]
\[ XY \rightarrow BY \]
\[ BY \rightarrow BA \]

where \( X \) and \( Y \) are not used elsewhere in the grammar.

**Example 3.1:** Let \( G_1 \) have \( N = \{A, B, C\}, T = \{a, b, c\} \) and the set of productions

\[ \Sigma \rightarrow A \quad CB \rightarrow BC \]
\[ A \rightarrow aABC \quad bB \rightarrow bb \]
\[ A \rightarrow abC \quad bC \rightarrow bc \]
\[ \quad cC \rightarrow cc \]

Figure 3.7 is a derivation of \( a^3b^3c^3 \) showing the production applied at each step. The reader should convince himself that the word \( a^k b^k c^k \) is in \( L(G_1) \) for all \( k \geq 1 \), and that only these words are in \( L(G_1) \). That is,

\[ L(G_1) = \{a^k b^k c^k | n \geq 1\} \]

Since by convention uppercase letters and lowercase letters always stand for nonterminal symbols and terminal symbols, respectively, the alphabets of a grammar are evident from the list of productions and will no longer be specified separately.

**Example 3.2:** Grammar \( G_2 \) is a modification of \( G_1 \):

\[ G_2: \quad \Sigma \rightarrow A \quad CB \rightarrow BC \]
\[ A \rightarrow aABC \quad bB \rightarrow bb \]
\[ A \rightarrow abC \quad bC \rightarrow b \]

A derivation of \( a^3b^3 \) is given in Figure 3.8. The reader may verify that \( L(G_2) = \{a^k b^k | k \geq 1\} \). Note that the last rule, \( bC \rightarrow b \), erases
Figure 3.7. Derivation of $a^3b^3c^3$. 
all the C's from the derivation, and that only this production removes the nonterminal C from sentential forms.

**Example 3.3:** A simpler grammar that generates \( \{a^k b^k | k \geq 1\} \) is the grammar \( G_3 \):

\[
G_3: \quad \Sigma \rightarrow S
\]
\[
S \rightarrow aSb
\]
\[
S \rightarrow ab
\]

A derivation of \( a^3 b^3 \) is

\[
\Sigma \rightarrow S \rightarrow aSb \rightarrow aaSbb \rightarrow aaabb
\]

The reader may verify that \( L(G_3) = \{a^k b^k | k \geq 1\} \).

It is not at all obvious from examining their productions that grammars \( G_2 \) and \( G_3 \) generate the same language. In fact, there is no general procedure
for determining whether arbitrary grammars generate the same language.
We return to this problem, and study related problems, in Chapter 12.

3.4 Types of Grammars

By restricting the forms of productions permitted a grammar, four
important types of formal grammars and corresponding classes of languages
can be specified. These restrictions are given in Table 3.1, together with the
names commonly used to identify the associated grammars and languages.

Proceeding down the table, the form of productions permitted is increas-
ingly restricted. It is obvious that a grammar of one type is also a grammar
of each type listed higher in the table. Using the notation $\mathcal{G}_i$ to mean a class
of grammars of type $i$, we have

$$
\mathcal{G}_0 \supset \mathcal{G}_1 \supset \mathcal{G}_2 \supset \mathcal{G}_3
$$

Consequently, each grammar type must generate languages that are a sub-
class of the languages generated by any grammar type listed higher in the
table. If we let $\mathcal{L}_i$ denote the class of languages generated by grammars in
$\mathcal{G}_i$, then

$$
\mathcal{L}_0 \supseteq \mathcal{L}_1 \supseteq \mathcal{L}_2 \supseteq \mathcal{L}_3
$$

We cannot conclude from our present knowledge that any of these contain-
ments of language classes are proper containments. That they are in fact
proper containments will be an important result of later chapters.

<table>
<thead>
<tr>
<th>Type</th>
<th>Format of Productions†</th>
<th>Remarks</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$\varphi A \psi \rightarrow \varphi \omega \psi$</td>
<td>Unrestricted Substitution Rules</td>
</tr>
<tr>
<td>1</td>
<td>$\varphi A \psi \rightarrow \varphi \omega \psi$, $\omega \neq \lambda$</td>
<td>Context Sensitive</td>
</tr>
<tr>
<td></td>
<td>$\Sigma \rightarrow \lambda$</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>$A \rightarrow \omega$, $\omega \neq \lambda$</td>
<td>Context Free</td>
</tr>
<tr>
<td></td>
<td>$\Sigma \rightarrow \lambda$</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>$A \rightarrow aB$</td>
<td>Right Linear</td>
</tr>
<tr>
<td></td>
<td>$A \rightarrow a$</td>
<td>Noncontracting Regular</td>
</tr>
<tr>
<td></td>
<td>$\Sigma \rightarrow \lambda$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$A \rightarrow Ba$</td>
<td>Left Linear</td>
</tr>
<tr>
<td></td>
<td>$A \rightarrow a$</td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\Sigma \rightarrow \lambda$</td>
<td></td>
</tr>
</tbody>
</table>

†$A \in N \cup \{\Sigma\}$, $\omega \in (N \cup T)^*$, $B \in N$, $a \in T$.  

|
In the next four sections we discuss briefly each of the four major classes of grammars and languages.

3.4.1 Unrestricted Grammars (Type 0)

With no restrictions, the class of formal grammars (Definition 3.1) is of surprising generality. (We have already noted that permitting a more general form of replacement rule adds nothing to the descriptive power of these formal grammars.) Sets of strings requiring this generality for their representation are not of great interest as artificial languages. Nevertheless, such systems of substitution rules are of considerable interest to logicians for studying the process of deduction. More will be said on this subject in Chapters 12 and 14.

One should note that the application of a grammatical rule \( \varphi A \psi \rightarrow \varphi \lambda \psi \) in a derivation step \( \omega_i \rightarrow \omega_{i+1} \) produces a new sentential form \( \omega_{i+1} \) shorter in length than \( \omega_i \). Such rules are known as contracting productions. The grammar \( G_2 \) of Example 3.2 has such a rule, the rule \( bC \rightarrow b \); its effect was displayed in Figure 3.8. A grammar containing at least one contracting production is said to be a contracting grammar. Only type 0 grammars may be contracting.

3.4.2 Context-Sensitive Grammars (Type 1)

**Definition 3.4:** A context-sensitive grammar \( G = (N, T, P, \Sigma) \) is a formal grammar in which all productions are of the form

\[
\varphi A \psi \rightarrow \varphi \omega \psi, \quad \omega \neq \lambda
\]

The grammar may also contain the production \( \Sigma \rightarrow \lambda \). If \( G \) is a context-sensitive (type 1) grammar, then \( L(G) \) is a context-sensitive (type 1) language.

A production \( \alpha \rightarrow \beta \) satisfying \( |\alpha| \leq |\beta| \) is known as a noncontracting production. Each production in a context-sensitive grammar is required to be noncontracting. As a consequence, sentential forms in any context-sensitive derivation

\[
\omega_1 \rightarrow \omega_2 \rightarrow \ldots \rightarrow \omega_n
\]

are nondecreasing in length:

\[
|\omega_1| \leq |\omega_2| \leq \ldots \leq |\omega_n|
\]

It follows that, if \( G \) is a context-sensitive grammar, \( \lambda \in L(G) \) if and only if \( G \) contains the production \( \Sigma \rightarrow \lambda \). Without this production, generation of the empty string would not be possible and a less elegant theory of formal languages would result.

The grammar in Example 3.1 is a context-sensitive grammar; the grammar in Example 3.2 is not because it has the contracting production \( bC \rightarrow b \).
3.4.3 Context-Free Grammars (Type 2)

In a context-free grammar, the context-denoting strings \( \varphi \) and \( \psi \) in a production \( \varphi \ A \ \psi \rightarrow \varphi \ \omega \ \psi \) are both required to be empty. Thus the possibility of replacing a nonterminal letter in a sentential form is independent of adjacent symbols; that is, it is independent of the context.

**Definition 3.5:** A *context-free grammar* \( G = (N, T, P, \Sigma) \) is a formal grammar in which all productions are of the form

\[
A \rightarrow \omega = \begin{cases}
A \in N \cup \{\Sigma\} \\
\omega \in (N \cup T)^* - \{\lambda\}
\end{cases}
\]

The grammar may also contain the production \( \Sigma \rightarrow \lambda \). If \( G \) is a context-free (type 2) grammar, then \( L(G) \) is a *context-free (type 2)* language.

Again, \( \lambda \in L(G) \) if and only if \( \Sigma \rightarrow \lambda \) is a production in the context-free grammar \( G \). The grammar \( G_3 \) of Example 3.3 is context free, as are the two grammars of Section 3.2.

In a context-free grammar, if \( A \) is \( \Sigma \) or a nonterminal letter, and \( \omega \in T^* \) is a string derivable from \( A \), we say that \( \omega \) is *denoted by* \( A \), or that \( \omega \) is a *phrase of type* \( A \). We use the notation

\[
L(G, A) = \{\omega \in T^* | A \rightarrow^* \omega\}
\]

for the set of phrases denoted by \( A \). If the sentential form \( A \) is derivable from \( \Sigma \), then \( L(G, A) \subseteq L(G) \), but it is not otherwise generally true that \( L(G, A) \subseteq L(G) \).

Context-free grammars are widely used to represent artificial languages, especially programming languages. We shall study these grammars in detail in Chapters 9 and 10.

3.4.4 Regular Grammars (Type 3)

Regular grammars are a highly restricted subclass of context-free grammars in which the right-hand side of a production may contain at most a single nonterminal symbol.

**Definition 3.6:** A production of the form

\[
A \rightarrow aB \quad \text{or} \quad A \rightarrow a
\]

\[
\begin{cases}
A \in N \cup \{\Sigma\} \\
B \in N \\
a \in T
\end{cases}
\]

is called a *right linear* production. A production of the form

\[
A \rightarrow Ba \quad \text{or} \quad A \rightarrow a
\]

\[
\begin{cases}
A \in N \cup \{\Sigma\} \\
B \in N \\
a \in T
\end{cases}
\]
is a left linear production. A formal grammar is right linear if it contains only right linear productions, and is left linear if it contains only left linear productions. Either form of linear grammar may contain the production \( \Sigma \rightarrow \lambda \). Left and right linear grammars are also known as regular grammars. If \( G \) is a regular (type 3) grammar, then \( L(G) \) is a regular (type 3) language.

The next example illustrates a general result, which will be established in Chapter 5: each regular language has both a left linear and a right linear grammar.

**Example 3.4:** A left linear grammar \( G_1 \) and a right linear grammar \( G_2 \) have productions as follows:

\[
\begin{align*}
G_1: & \quad \Sigma \rightarrow 1B & G_2: & \quad \Sigma \rightarrow B1 \\
& \quad \Sigma \rightarrow 1 & & \quad \Sigma \rightarrow 1 \\
& \quad A \rightarrow 1B & & \quad A \rightarrow B1 \\
& \quad B \rightarrow 0A & & \quad B \rightarrow A0 \\
& \quad A \rightarrow 1 & & \quad A \rightarrow 1
\end{align*}
\]

The reader may verify that 
\[
L(G_1) = (10)^*1 = 1(01)^* = L(G_2)
\]

The first two productions of grammar \( G_1 \) or \( G_2 \) in Example 3.4 could be replaced by the production \( \Sigma \rightarrow A \) without altering the language defined. Occasionally, it is convenient to use such a rule in a regular grammar even though it is not strictly permitted by Definition 3.6.

Regular grammars have been studied extensively since, as we shall see, their productions are related one to one with the transitions of finite-state accepter machines.

### 3.5 Derivation Trees and Diagrams

The examples developed in Section 3.2 have shown how a derivation may be displayed in the form of a tree diagram that clearly exhibits the phrase structure of a sentence. We are interested now in how such tree diagrams are related to derivations according to an arbitrary formal grammar.

By a tree we mean an acyclic directed graph (Figure 3.9) in which each node is connected by a unique directed path from a distinguished node called the root node of the tree. (The arcs in Figure 3.9 are directed downward by convention.) A node of a tree is said to be a descendant of any node from which it can be reached by a directed path. The root node is not the descendant of any other node, and the leaf nodes are those that have no descendants. A node that is neither the root node nor a leaf node is an interior node of the tree.
A tree diagram for a derivation

\[
\Sigma \rightarrow^* \omega, \quad \omega \in (N \cup T)^*
\]

in a context-free grammar is a tree in which the root node is labeled \( \Sigma \), the interior nodes are labeled with symbols in \( N \), and the leaf nodes are labeled with the letters of the sentential form \( \omega \). To construct the tree diagram, we start with a tree consisting just of a root node labeled \( \Sigma \) and extend the tree for each step of the derivation. At each stage in the construction, the leaf nodes of the tree (read from left to right) will be labeled with the letters of the corresponding sentential form. Consider any derivation step

\[
\phi \ A \ \psi \ \rightarrow \ \phi \ B_1 B_2 \ldots B_n \ \psi
\]

in which the production

\[
A \rightarrow B_1 B_2 \ldots B_n \quad \begin{cases} A \in (N \cup [\Sigma]) \\ B_i \in (N \cup T) \end{cases}
\]

is applied. The tree constructed to this point will have leaf nodes labeled with the letters of \( \phi \ A \ \psi \). We make the leaf node labeled \( A \) an interior node with descendent leaf nodes labeled \( B_1, B_2, \ldots, B_n \) as shown in Figure 3.10. The
resultant tree diagram will have leaf nodes labeled with the letters of the sentential form \( \varphi B_1B_2 \ldots B_n \psi \).

**Example 3.5:** Consider the context-free grammar

\[
G: \quad \Sigma \rightarrow S \\
S \rightarrow ST \\
S \rightarrow T \\
T \rightarrow (S) \\
T \rightarrow ( )
\]

The language generated by \( G \) is known as the *parenthesis language* \( L_p \), and consists of all well-formed strings of properly matching left and right parentheses that can occur in correctly written arithmetic expressions. Figure 3.11a gives a derivation of the string \( ( )(( )) \), and Figure 3.11b is the corresponding derivation tree. The grammar

![Derivation Diagram](image-url)

(a) Derivation

![Derivation Tree](image-url)

(b) Derivation Tree

**Figure 3.11.** Derivation and derivation tree for string \( ( )(( )) \).
actually provides nine distinct derivations of this terminal string. They all correspond to the same derivation tree, but represent different orders of performing the substitutions indicated in the tree diagram. The nine derivations are shown as a derivation diagram in Figure 3.12. Each path from top to bottom in the diagram is a valid derivation of the string ( )( ).

Derivation trees are a suitable means of representing the phrase structure of sentences according to a context-free grammar, because the association of a terminal string with each nonterminal letter of a sentential form is completely independent of the other letters in the form. That is, if

\[ A_1 A_2 \ldots A_i \ldots A_n, \quad A_i \in N \cup T \]

is a sentential form, applying a context-free production to rewrite a nonterminal letter \( A_i \) can neither eliminate nor create a possible substitution for other nonterminal letters in the form. Hence the order in which nonterminal letters are replaced in a context-free derivation is immaterial to the result.

This is not generally true for context-sensitive grammars, since the application of one production may modify the context required for application of another. For this reason, we prefer to use the derivation as our basic
tool in the formal study of grammars, and to use derivation trees only for illustration.

In the case of regular grammars, there is no way of deriving from $\Sigma$ a sentential form containing more than one nonterminal letter. Consequently, the derivation trees have a very simple form.

Example 3.6: Let $G$ be the right linear grammar

\[ G: \begin{align*}
\Sigma & \rightarrow A \\
A & \rightarrow 1A \\
A & \rightarrow 0B \\
B & \rightarrow 0B \\
B & \rightarrow 0
\end{align*} \]

A derivation of $10000$ is given in Figure 3.13. The reader may verify that $L(G) = 1^*000^*$. 

![Derivation Tree](image)

Figure 3.13. Derivation of 10000.

Indeed, a derivation according to a right linear grammar will always have the form shown in Figure 3.14. A derivation must start with an application of a production of the form $\Sigma \rightarrow \lambda$, $\Sigma \rightarrow a$, or $\Sigma \rightarrow aA$. In the first two cases the derivation is immediately complete. In the third case, the derivation must continue, with further applications of productions of the form $A \rightarrow aB$, and a single application of a production of the form $A \rightarrow a$. Similar
Figure 3.14. Right linear derivation tree.

Figure 3.15. Left linear derivation tree.

Remarks apply to the structure of derivations in a left linear grammar, as shown in Figure 3.15.

3.6 Ambiguity

The following example shows how a grammar may be "ambiguous" with respect to some sentence.
Example 3.7: Consider the context-free grammar

\[
G: \quad \Sigma \rightarrow S \\
S \rightarrow SS \\
S \rightarrow ab
\]

Figure 3.16 shows two derivations of the string \( ababab \). We see that the derivations correspond to different tree diagrams. The grammar \( G \) is ambiguous with respect to the sentence \( ababab \): if the tree diagrams were used as the basis for assigning meaning to the derived string, mistaken interpretation could result.

We shall be interested in whether or not a grammar generates ambiguous sentences, so we must precisely define what is meant by an ambiguous derivation and what is meant by an ambiguous grammar. (Although the notion of ambiguity is applicable to all classes of formal grammars, our treatment of ambiguity will be limited to the class of context-free grammars: such grammars have proved most useful for representing programming languages, and questions of ambiguity, therefore, are of greatest importance for these grammars.)

Definition 3.7: Let \( G \) be a context-free grammar, and let \( \omega \) be a sentence in the language generated by \( G \). Then \( \omega \) is ambiguous if there are derivations of \( \omega \) that correspond to different tree diagrams.

Figure 3.17 exhibits all possible derivations of the string \( ababab \) according to the grammar of Example 3.7 in the form of a derivation diagram. From the sentential form \( SS \), the diagram shows two paths in which the leftmost \( S \) is replaced first: path 1 corresponds to the tree of Figure 3.16a, and path 2 corresponds to the tree of Figure 3.16b. These two paths in Figure 3.17 have a unique property: in each derivation step, the leftmost nonterminal letter of the sentential form is rewritten. These derivations, known as leftmost derivations, permit a precise definition of ambiguous grammars.

Definition 3.8: Let \( G \) be a context-free grammar. A derivation

\[
\omega_0 \rightarrow \omega_1 \rightarrow \omega_2 \rightarrow \cdots \rightarrow \omega_n
\]

is a leftmost derivation if and only if the leftmost nonterminal letter of \( \omega_i \) is replaced to obtain \( \omega_{i+1} \). That is, for \( 0 \leq i < n \),

\[
\begin{align*}
\omega_i &= \alpha A \beta \\
\omega_{i+1} &= \alpha \omega \beta
\end{align*}
\]

\[\alpha \in T^* \quad A \in N \quad \beta \in (N \cup T)^*\]

and \( A \rightarrow \omega \) is a production of \( G \).
Figure 3.16. Ambiguous derivations in a context-free grammar.
It is easy to construct a unique leftmost derivation from a given derivation tree: Suppose that the final terminal string in the derivation is $\omega = t_n t_{n-1} \cdots t_1$ ($t_i \in T$). For $i = 1, 2, \ldots, n$, we follow the path from the root node to the leaf node labeled $t_i$, applying the production associated with each node on the path if it has not already been applied.

The interior nodes in the trees of Figure 3.16 have been numbered to indicate the order in which the corresponding productions are applied in the leftmost derivation.

Since it is easy to obtain a derivation tree from any derivation, the correspondence between trees and leftmost derivations is one-to-one: each sentence generated by a context-free grammar has a unique leftmost derivation.

**Definition 3.9:** A context-free grammar is *ambiguous* if and only if it generates some sentence by two or more distinct leftmost derivations.
The grammar of Example 3.7 is ambiguous, whereas the grammars of Examples 3.3 and 3.4 are not. There are two ways in which ambiguity can arise in a context-free grammar:

1. Some sentence has two structurally different derivation trees.
2. Some sentence has two structurally similar derivation trees with different labeling of their interior nodes.

The two cases are illustrated by the following examples.

**Example 3.8:** \( G: \Sigma \rightarrow A \)

\[
A \rightarrow A0A \\
A \rightarrow 1
\]

Two different trees for the string 10101 are shown in Figure 3.18.

![Figure 3.18. A structural ambiguity.](image)

**Example 3.9:** \( G: \Sigma \rightarrow A \)

\[
A \rightarrow B0 \\
A \rightarrow A0 \\
B \rightarrow B0 \\
A \rightarrow 1 \\
B \rightarrow 1
\]

Figure 3.19 shows two derivation trees for the string 100 that are identical apart from the nonterminal letters assigned to interior nodes.
Because each sentential form in a derivation may only contain a single nonterminal symbol, a regular grammar is ambiguous only if some string has derivation trees with different labeling of the interior nodes. We shall see how ambiguity may be removed from regular grammars in Chapter 5.

Ambiguity is important in language analysis because the analysis of a sentence with ambiguous derivations may lead to incorrect assignment of meaning. An illustration of ambiguity in English is shown in Figure 3.20.† It is this sort of ambiguity that makes natural language translation so difficult.

In programming languages, the algebraic phrase
\[ a \times b + c \]
presents a similar problem of interpretation (by a compiler, for example) if a carelessly designed grammar is used. Figure 3.21 presents a grammar and two derivation trees. The familiar interpretation \((a \times b) + c\) corresponds to the lefthand tree, but the righthand tree also represents a legitimate derivation according to the grammar.

Ambiguity in context-free grammars is the subject of Chapter 9.

Notes and References

Early descriptions of “formal grammars” for languages appear in Chomsky [1956, 1959], Chomsky and Miller [1958], and Bar-Hillel, Gaifman, and Shamir [1960]. The formalism presented in this chapter, as well as the notion of a language hierarchy, is from the 1959 publication of Chomsky.

Three classes of languages (and the corresponding grammars) are discussed more fully elsewhere in this book: type 3 languages in Chapter 5; type 2 languages in Chapters 8 to 10; and type 0 languages in Chapters 11, 12,

†This example is attributed to Noam Chomsky.
Figure 3.20. An ambiguity in English.

G: 

\[ A \rightarrow A \, B \, A \]
\[ A \rightarrow a \]
\[ A \rightarrow b \]
\[ A \rightarrow c \]
\[ B \rightarrow + \]
\[ B \rightarrow x \]

Figure 3.21. An ambiguity in defining arithmetic expressions.
and 14. References to supplementary material relating to these types of languages are found at the ends of those chapters.

Type 1 grammars have proved of little use in the specification either of programming languages or "natural" languages, partly because they possess certain undesirable properties (such as the ability to generate strings that cannot be readily analyzed to obtain structural descriptions), and partly because several important questions that have been resolved for other types of languages remain unresolved for type 1 languages. For this reason, such languages are not discussed in detail in this book; the reader who wishes to learn more about these languages may refer to Myhill [1960], Landweber [1963], and Kuroda [1964].

Questions of ambiguity in grammars have been studied by Cantor [1962], Floyd [1962], Parikh [1961], Ginsburg and Ullian [1966], and others. These questions are taken up again in later chapters of this book.

**Problems**

3.1. Consider the following grammars:

\[
\begin{align*}
G_1: & \quad \Sigma \rightarrow \lambda \\
    & \quad \Sigma \rightarrow S \\
    & \quad S \rightarrow SS \\
    & \quad S \rightarrow c \\
G_2: & \quad \Sigma \rightarrow \lambda \\
    & \quad \Sigma \rightarrow S \\
    & \quad S \rightarrow cSd \\
    & \quad S \rightarrow cd \\
G_3: & \quad \Sigma \rightarrow \lambda \\
    & \quad \Sigma \rightarrow S \\
    & \quad S \rightarrow Sd \\
    & \quad S \rightarrow cS \\
    & \quad S \rightarrow c \\
G_4: & \quad \Sigma \rightarrow \lambda \\
    & \quad \Sigma \rightarrow S \\
    & \quad S \rightarrow d \\
    & \quad S \rightarrow cS \\
    & \quad S \rightarrow Td \\
    & \quad S \rightarrow c \\
    & \quad T \rightarrow Td \\
    & \quad T \rightarrow d \\
G_5: & \quad \Sigma \rightarrow \lambda \\
    & \quad \Sigma \rightarrow S \\
    & \quad S \rightarrow ScS \\
    & \quad S \rightarrow c \\
\end{align*}
\]

a. Describe \( L(G_i) \) for \( i = 1, 2, 3, 4, 5 \).
b. Indicate any inclusions among the \( L(G_i) \).
c. For each language, give a derivation of a length 4 string.

3.2. Construct a grammar that generates each following language:

a. \( \{0^n1^m | m > n \geq 0\} \).
b. \( \{0^n1^m | m \text{ odd and } n \text{ even, or } n \text{ odd and } m \text{ even}\} \).
c. \( \{0^k1^m0^n | n = k + m\} \).
d. \( \{\omega \omega | \omega \in \{0, 1\}^*\} \).
3.3. Given a terminal alphabet $S = \{s_1, s_2, \ldots, s_k\}$, we can define a unary operator $R$ on $S^*$ as follows: (1) $R(\lambda) = \lambda$, (2) $R(s) = s$ for each symbol $s$ in $S$, and (3) $R(s\varphi) = R(\varphi)s$, for $s \in S$, $\varphi \in S^*$. For any word $\varphi \in S^*$, $R(\varphi)$ is called the reverse of $\varphi$ and is usually written $\varphi^R$.

A word $\omega$ is a palindrome if it is its own reverse, that is, if $\omega = \omega^R$.

A language $L$ is a palindrome language just if each of its elements is a palindrome.

Consider $G_1, G_2$ below:

$$G_1: \quad \Sigma \rightarrow S$$
$$S \rightarrow aSa$$
$$S \rightarrow aSb$$
$$S \rightarrow bSb$$
$$S \rightarrow bSa$$
$$S \rightarrow aa$$
$$S \rightarrow bb$$

$$G_2: \quad \Sigma \rightarrow S$$
$$S \rightarrow aS$$
$$S \rightarrow Sa$$
$$S \rightarrow bS$$
$$S \rightarrow Sb$$
$$S \rightarrow a$$
$$S \rightarrow b$$

a. Describe informally $L(G_1)$ and $L(G_2)$. Is either language a palindrome language?

b. The reverse of a language $L$, written $L^R$, is defined as

$$L^R = \{\omega^R | \omega \in L\}$$

Prove that if $L$ is a palindrome language, $L^R$ is a palindrome language. If $L = L^R$, must $L$ be a palindrome language?

c. Suppose that $L' = L \cap L^R$ for some language $L$. If $L$ is a palindrome language, must $L'$ be a palindrome language? If $L'$ is a palindrome language, must $L$ be a palindrome language?

d. Prove that if some string $\omega$ is a palindrome, then the language $L = \omega^*$ is a palindrome language.

3.4. a. Construct a left linear grammar $G$ such that $L(G) = 10^*.$

b. Construct a right linear grammar $G$ such that $L(G) = ab^* \cup c^*.$

3.5. Construct a context-free grammar generating the language $L = \{\omega \in (0 \cup 1)^* | N_0(\omega) = N_1(\omega)\}$, where $N_0(\omega)$ and $N_1(\omega)$ denote the number of occurrences of the symbols 0 and 1, respectively, in the word $\omega$.

3.6. Construct a context-free grammar $G$ generating the language $L$, where $L$ is

a. $\{a^nb^n | 1 \leq n \leq 2k\}$.

b. $\{\omega\omega^R | \omega \in (0 \cup 1)^*\}$.

c. The complement of the language $L'$, where $L'$ is $\{a^nb^kc^k | k > 0\}$. 
3.7. Describe the languages generated by the context-sensitive grammars \( G_1 \) and \( G_2 \) below:

\[
G_1: \quad \Sigma \rightarrow ACA \\
    AC \rightarrow AAC \Sigma \\
    AC \rightarrow ADc \\
    AC \rightarrow AcD \\
    D \rightarrow AD \\
    A \rightarrow 0 \\
    A \rightarrow 1 \\

G_2: \quad \Sigma \rightarrow AcB \\
    \Sigma \rightarrow BcA \\
    Ac \rightarrow XAcX \\
    Bc \rightarrow XBcX \\
    A \rightarrow 1Y \\
    B \rightarrow 0Y \\
    Y \rightarrow X \\
    Y \rightarrow XY \\
    A \rightarrow 1 \\
    B \rightarrow 0 \\
    X \rightarrow 1 \\
    X \rightarrow 0
\]

(a) Describe the language \( L = L(G_1) \cup L(G_2) \).

(b) Find a context-free grammar generating \( L \).

3.8. Construct a grammar generating the language \( L = \{a^n b^n c^n d^n | n > 0 \} \).
(The grammar constructed must be of type 0 or 1, since it can be shown that \( L \) is not context free.)

3.9. Construct a type 0 grammar generating the language \( L = \{1^x c^1 c^1 \alpha | \alpha = xy, x, y \geq 0 \} \).

3.10. Let \( N = \{A, B\} \) and \( T = \{a, b\} \). Construct a context-free grammar \( G \) that generates all possible sets of context-free productions on terminal and nonterminal alphabets \( T \) and \( N \), respectively, that is, generates all strings of the form \( \omega = \{p_1, p_2, \ldots, p_k\} \), where each substring \( p_k \) is of the form \( X \rightarrow \varphi \), with \( X \in N \) and \( \varphi \in (N \cup T)^* - \lambda \).

Show that a right linear grammar can be constructed that generates all right linear productions on \( T \) and \( N \).

3.11. Let \( G_1 = (N_1, T, P_1, \Sigma) \) and \( G_2 = (N_2, T, P_2, \Sigma) \) be arbitrary type \( i \) grammars, \( i \in \{0, 1, 2, 3\} \). Show how to construct, from \( G_1 \) and \( G_2 \), a type \( i \) grammar for the following:

(a) \( L(G_1) \cup L(G_2) \).

(b) \( L(G_1) L(G_2) \).

(c) \( L(G_1) \star \).

For each \( i \), prove that the grammars constructed in parts a through c are indeed type \( i \). (This problem shows that the class of type \( i \) lan-
guages, \( i = 0, \ldots, 3 \), is closed under the set operations of union, concatenation, and closure.)\)

3.12. Let \( G \) be a left linear grammar, let \( X \) and \( Y \) be symbols in its non-terminal alphabet, and let \( s \) be a symbol in its terminal alphabet.
   a. Show that if \( G \) contains the production \( X \rightarrow Y_s \), then \( L(G, Y)_s \subseteq L(G, X) \).
   b. Show that the converse of part a is false; that is, show that \( L(G, Y)_s \subseteq L(G, X) \) does not imply that \( G \) contains the production \( X \rightarrow Y_s \). Is the converse true if \( L(G, Y) \) is non-empty?
   c. Suppose that \( L(G, Y)_s \not\subseteq L(G, X) \). If \( G' \) is the grammar obtained from \( G \) by adding the production \( X \rightarrow Y_s \) to the productions of \( G \), must it be the case that \( L(G') \neq L(G) \)?

3.13. We say that a context-free grammar \( G \) is weakly \( k \)-generative if, whenever sentential form \( \omega' \) is derivable sentential from \( \omega \) in \( k \) or more steps, either \( \omega' \) is longer than \( \omega \) or \( \omega' \) contains more terminal symbols than \( \omega \). We say that \( G \) is \( k \)-generative if it is weakly \( k \)-generative but not weakly \((k - 1)\)-generative.
   a. Prove that for every context-free grammar \( G \) there exists a \( k \)-generative context-free grammar \( G' \) (for some \( k > 0 \)) such that \( L(G') = L(G) \). (Hint: Show that if \( G \) is a context-free grammar that is not \( k \)-generative for any \( k \), then \( G \) permits a derivation of the form

   \[
   A \xrightarrow{*} A
   \]

   for some nonterminal \( A \). In such a case, either \( G \) contains the production \( A \rightarrow A \), which can be discarded without altering the language generated, or \( G \) contains a series of productions

   \[
   A \rightarrow X_1 \\
   X_1 \rightarrow X_2 \\
   \vdots \\
   X_{n-1} \rightarrow X_n \\
   X_n \rightarrow A
   \]

   where \( X_1, \ldots, X_n \) are nonterminals distinct from \( A \). Show that in the latter case, each occurrence of a nonterminal \( X_i \) in \( G \), \( 1 \leq i \leq n \), can be replaced by an occurrence of \( A \) without altering the language generated by the grammar.)
   b. Prove that if \( G \) is a \( k \)-generative context-free grammar, \( k > 0 \), then there exists a 1-generative context-free grammar \( G' \) such that \( L(G') = L(G) \).
Parts a and b together prove that for every context-free grammar \( G \) there exists a 1-generative context-free grammar \( G' \) such that \( L(G') = L(G) \).

3.14 Describe a procedure for deciding, given an arbitrary context-free grammar \( G \) and an arbitrary string \( \omega \), whether \( \omega \) is in \( L(G) \). Does your procedure work if \( G \) is an arbitrary context-sensitive grammar? If not, modify the procedure so that it works for context-sensitive grammars. In Chapter 12, we show that no such procedure can be constructed for the class of type 0 grammars. Where does your procedure fail for that class of grammars?

3.15 Show that if we modify the definition of context-free grammars to allow productions of the form \( A \rightarrow \lambda \), where \( A \) is a nonterminal, the class of languages generable by context-free grammars does not change; that is, show that for any grammar \( G = (N, T, P, \Sigma) \) in which all productions are of the form

\[
A \rightarrow \alpha, \quad A \in N \cup \{\Sigma\}, \quad \alpha \in (N \cup T)^*
\]

there exists a grammar \( G' = (N, T, P', \Sigma) \) in which all productions are of the form

\[
A \rightarrow \alpha, \quad A \in N \cup \{\Sigma\}, \quad \alpha \in (N \cup T)^* - \lambda
\]

or of the form

\[
\Sigma \rightarrow \lambda
\]

such that \( L(G') = L(G) \).

3.16 Let \( h: S^* \rightarrow T^* \) be a homomorphism from the set \( S^* \) to the set \( T^* \), where \( S \) and \( T \) are arbitrary sets of terminal symbols. Let \( L \subseteq S^* \) be a type i language, \( i = 0, 1, 2, \) or 3.

a. Prove that if \( i = 0, 2, \) or 3, the language \( h(L) = \{h(\omega) | \omega \in L\} \) is a type i language.

*b. Prove that if \( L \) is a type 1 language, the language \( h(L) \) need not be type 1. (You may assume the existence of languages that are type 0 but not type 1.) Under what conditions on \( h \) will \( h(L) \) be type 1?

3.17 Suppose that \( G \) is a context-free grammar in which all productions are of the form

\[
X \rightarrow \alpha Y
\]

or

\[
X \rightarrow \alpha
\]

where \( X \) is \( \Sigma \) or a nonterminal, \( Y \) is a nonterminal, and \( \alpha \) is a nonempty terminal string.

a. Show that \( L(G) \) is a regular language.

*b. Suppose that we allow \( \alpha \) to be any (possibly empty) terminal string. Show that \( L(G) \) is still regular. (The results of Problem 3.16 may prove helpful.)
3.18. A semi-Thue system is a grammar \( G = (N, T, P, \Sigma) \) with productions of the form

\[
\alpha \rightarrow \beta, \quad \alpha \in ((T \cup N)^* - \lambda) \cup \Sigma, \quad \beta \in (T \cup N)^*
\]

That is, \( \beta \) can be any string and \( \alpha \) any nonempty string of symbols from \( T \cup N \); in addition, \( \alpha \) can be the starting symbol. If \( n \) is the length of the longest string appearing in either side of any production in \( P \), we say that \( G \) is of \textit{order} \( n \).

a. Let \( G \) be any semi-Thue system of order \( k > 2 \). Show that there exists a semi-Thue system \( G' \) such that \( L(G') = L(G) \) and \( G' \) is of order \( k - 1 \). Conclude that for any semi-Thue system there is a system of order 2 that generates the same language.

b. Let \( G \) be a semi-Thue system of order 2. Show that there exists a phrase-structure grammar \( G' \) of order 2 such that \( L(G) = L(G') \).

(The order of a phrase-structure grammar is defined in a manner analogous to that of semi-Thue systems.)

c. Prove that the class of languages generable by semi-Thue systems is precisely the type 0 languages.

3.19. In Problem 3.13, we defined \( k \)-generative context-free grammars. Suppose that \( G \) is a \( k \)-generative grammar for some \( k > 0 \), and \( \omega \) is a word in \( L(G) \) such that \( |\omega| = m \). What is the maximum number of derivation steps in any derivation of \( \omega \)? What is the maximum number of nodes in the corresponding derivation tree? (Note that if \( G \) is not \( k \)-generative for any \( k \), we can place no bound on either the number of derivation steps or the number of nodes in the tree.) Using the results of Problem 3.13, show that if \( L \) is a context-free language there is a grammar \( G \) for \( L \) such that each string \( \omega \in L(G) \) is derived in no more than \( 2|\omega| \) steps.

3.20. Which of the grammars in Problem 3.1 are ambiguous? For each ambiguous grammar, exhibit distinct derivation trees corresponding to the derivation of some word generated by the grammar.

3.21. Let \( T \) be a terminal set consisting of variable-denoting elements \( x, y, z \), operator-denoting elements \( **, *, /, +, - \), and parentheses. Construct a context-free grammar \( G \) generating the set of all valid Fortran arithmetic expressions composed of elements of \( T \). Show derivation trees for

a. \((x + y) * z / x\).

b. \(x ** y / (z - y * x - z)\).

Is the grammar \( G \) that you have constructed ambiguous? If so, show distinct derivation trees corresponding to the derivation of some expression.
3.22. Let \( G_1 = (N_1, T_1, P_1, \Sigma) \) and \( G_2 = (N_2, T_2, P_2, \Sigma) \) be unambiguous context-free grammars such that \( N_1 \cap N_2 = \emptyset \). Let \( G = (N_1 \cup N_2, T_1 \cup T_2, P_1 \cup P_2, \Sigma) \). Prove that \( G \) is ambiguous if and only if \( L(G_1) \) and \( L(G_2) \) have some nonempty word in common.

3.23. Let \( G = (N, T, P, \Sigma) \) be a context-free grammar, and let \( A \in N \). Suppose that there exist sentential forms \( \omega_1, \omega_2, \ldots, \omega_n \) such that \( A \xrightarrow{*} \omega_i \) for \( 1 \leq i \leq n \). Let \( G' = (N, T, P', \Sigma) \) be the context-free grammar obtained from \( G \) by adding to \( P \) the productions \( A \rightarrow \omega_i \), \( 1 \leq i \leq n \).

a. Show that \( L(G') = L(G) \).

b. Under what conditions is \( G' \) ambiguous?