strings of a language: before considering a formal definition, it is useful to analyze the structure of a sentence. We used two classes of symbols: one, enclosed in brackets, to represent linguistic classes (grammatical units used as intermediate steps in the formal generation process); and another, composed of Roman letters, from which the generated sentence was eventually formed. Because the symbols of one set are in the sentence when generation terminates, while those of the other appear only in intermediate steps, they are referred to as *terminal* and *nonterminal* symbols, respectively. One nonterminal symbol, the *starting symbol* (in our example, ⟨sentence⟩) is distinguished as the symbol with which the generation process begins.

Definition 3: The *terminal symbols* are the symbols of the alphabet T. The *nonterminal symbols* are a set N of symbols not in T that represent intermediate states in the generation process. The *starting symbol* is a distinguished nonterminal symbol from which all strings of the language are derived.

The generation process itself consisted of applying, at each step, any one of the set of rewriting rules or productions. This process transforms the string into a new string; the process stops when there is no production that can be applied or when the string consists solely of terminal symbols.

Definition 4: A *production*[1] is a string transformation rule having a left-hand side that is a pattern to match a substring (possibly all) of the string to be transformed, and a right-hand side that indicates a replacement for the matched portion of the string.

It is important to realize that any substring of the current string may be replaced by an applicable production and that only that part of the string matched by the left-hand side of the production is affected. Productions can totally replace substrings, or they may merely rearrange the symbols of the matched substring.

Definition 5: A formal grammar G is a 4-tuple $G = (N,T,\Sigma,P)$ where

(1) N is the set of nonterminal symbols
(2) T is the set of terminal symbols
(3) Σ is the starting symbol; $\Sigma \in N$

[1]The term production is due to the mathematician Emil Post who first used similar rules to define languages. The formal grammars presented here are essentially the same as Post's canonical systems with some restrictions on the use of symbols in productions.

(4) P is the set of productions $\alpha \rightarrow \beta$ where $\alpha, \beta \in (N \cup T)^*$, $\alpha \neq \lambda$ (i.e., α is not null)
(5) $N \cap T$ is empty

Requirement (5) assures that it is always possible to distinguish nonterminals from terminals.

7.3.1 Examples of Formal Grammars

To clarify the notion of formal grammar we consider two examples. The nonterminals will be capital Roman letters (A,B,C) and Σ; the terminal symbols will be small Roman letters (a,b,c).

Example 1: $N = \{A,B,\Sigma\}$ $T = \{a,b\}$

$$P = \begin{cases} \Sigma \rightarrow AB & (1) \\ A \rightarrow aA & (2) \\ A \rightarrow a & (3) \\ B \rightarrow Bb & (4) \\ B \rightarrow b \end{cases} \quad (5)$$

Notice that in production (2) the nonterminal A occurs on both sides of the rule; (2) indicates that the class of strings corresponding to A is closed under prefixing with "a". Since the only other production indicating the structure of class A is production 3, it is easy to see that A is closed under concatenation and consists of the class of finite strings of "a"s: a,aa,aaa,aaaa, . . . Likewise, the class B consists of finite strings of "b"s: b,bb,bbb,bbbb, . . . The language produced by the grammar consists of all strings formed from a string of "a"s followed by a string of "b"s.

Example 2: $N = \{A,\Sigma\}$ $T = \{a,b\}$

$$P = \begin{cases} \Sigma \rightarrow A \\ A \rightarrow aAb \\ A \rightarrow ab \end{cases}$$

Since the terminal alphabets are the same in examples 1 and 2, both languages are subsets of the set of strings containing "a"s and "b"s. There is a difference, however. In the language of example 1 an arbitrary number of "a"s may precede an equally arbitrary number of "b"s. But in example 2 every time an "a" is generated, a "b" is also generated. Therefore, the legal sentences of example 2 consist of a string of "a"s followed by an equal number of "b"s.

7.3.2 The Derivation of Sentences

So far we have indicated that formal grammars provide a generative specification of a language, but we have not formally defined the process of generating a string.

Definition 6: A string γ is *immediately derived* from a string μ (write $\mu \Rightarrow \gamma$) in a grammar G if and only if
$\mu = \sigma\,\alpha\,\tau$, $\gamma = \sigma\,\beta\,\tau$, and
$\alpha \rightarrow \beta \in$ P of G
where σ and τ represent arbitrary (possibly empty) strings.

Referring to example 1 above, suppose that μ = aABb and γ = aaBb. aABb $\Rightarrow$ aaBb is then an immediate derivation with σ = a, α = A, τ = Bb, β = a, and the production A $\rightarrow$ a. We now define proof in a grammar.

Definition 7: A string γ is *derived* from a string μ (write $\mu \Rightarrow^* \gamma$) in a grammar G if and only if there is a sequence of strings $\omega_0, \omega_1, \ldots, \omega_{n-1}, \omega_n$, for $n \geq 0$, such that
$\mu = \omega_0$,
$\gamma = \omega_n$
and for every $0 \leq i < n$ $\quad \omega_i \Rightarrow \omega_{i+1}$
(i.e. $\mu = \omega_0 \Rightarrow \omega_1 \Rightarrow \ldots \Rightarrow \omega_{n-1} \Rightarrow \omega_n = \gamma$)
with $\omega_i \in (N \cup T)^* - \lambda$ for every i.
The list $\{\omega_i\}$ is a *proof* of γ in G.

This is a formal statement that each new string in the derivation process must come from some previously derived string by application of a production of G. The last condition rules out deriving a string from the *empty string* λ (i.e., a string without symbols). A typical derivation in the grammar of example 1 is $\Sigma \Rightarrow$ AB $\Rightarrow$ aAB $\Rightarrow$ aABb $\Rightarrow$ aaBb $\Rightarrow$ aabb hence $\Sigma \Rightarrow^*$ aabb (also aAB $\Rightarrow^*$ aaBb, etc).

7.3.3 Sentential Forms and Sentences

The above definitions specify formally the generation process. It is now necessary to designate which of the set of possible derivations terminate on strings of the language.

Definition 8: A *sentential form* is any string which can be derived from the starting symbol Σ.

Examples of sentential forms in the grammar of example 1 are aAB and aabb. The sentential or sentence-like forms include formulae which have nonterminal symbols in the final string.

Definition 9: A *sentence* is a sentential form containing only terminal symbols.

Therefore, aabb is a sentence, but aAB is not.

Definition 10: A language L defined by a grammar G (write L(G)) is the set of sentences that can be derived from Σ in G:
$L(G) = \{\omega \in T^* \mid \Sigma \Rightarrow^* \omega\}$.
L is called *ambiguous* if it contains a formula for which there is more than one distinct proof in which the left-most nonterminal is replaced at each step.

7.4 HIERARCHY OF LANGUAGES

The definition of production allows for a wide variety of string transformations. Certain restrictions on the form of productions give grammars producing subclasses of the class of formal languages – e.g., linear grammars, producing regular languages. Noam Chomsky has constructed a system of four language types that classify some languages according to such restrictions.

The most general type of grammar imposes no restrictions on the productions. In particular, productions that eliminate ("erase") symbols are permitted. This allows the intermediate strings to expand and contract. An example of an erasing production is aAB → aB, in which A is erased from the context aAB. A grammar (as we have defined it) without restrictions is called a type 0 grammar.

The simplest restriction which produces a strictly smaller class of languages is to require the right-hand side of every production to have at least as many symbols as the left-hand side. A grammar with this restriction is called a type 1 or noncontracting context-sensitive grammar because $\alpha \Rightarrow \beta$ is possible only if length $(\beta) \geq$ length (α). Examples of productions in a type 1 grammar are

$bB \rightarrow Bb$ (interchange symbols)
$\sigma\alpha\tau \rightarrow \sigma\beta\tau$ (where length $(\beta) \geq$ Length (α))

Definition 11: The *length* of a string is the number of symbols in the string. If α consists of a single symbol, length $(\alpha) = 1$; length $(\lambda) = 0$ (the null string). For strings α, β, length $(\alpha\beta)$ = length (α) + length (β).

In both type 0 and type 1 grammars α can be any string. "Context-sensitive" refers to the fact that some productions may recognize context – e.g., in the case of $\sigma\alpha\tau \rightarrow \sigma\beta\tau$, the transformation $\alpha \rightarrow \beta$ occurs only in the context σ_τ. An example of a type 1 grammar is:

Example 3: $N = \{A,B,\Sigma\}$ $T = \{a,b,c\}$

$P = \{\Sigma \rightarrow Abc$
$Ab \rightarrow aAbB$
$Bb \rightarrow bB$
$Bc \rightarrow bcc$
$A \rightarrow a\}$

This grammar generates strings of the form $a^n b^n c^n$ for $n \geq 1$.

If the left-hand side of the production is restricted to a single nonterminal symbol, its application cannot be dependent on the context in which the symbol occurs. Grammars with this restriction (and nonblank right-hand strings) are called *type 2, context-free* or *simple phrase-structure* grammars. The latter name comes from an analogy to the method that we used to generate the sentence "The student studies hard." Our grammar satisfied the single-symbol restriction, and every nonterminal symbol expanded into a word or phrase – for example, ⟨sentence⟩ became the concatenation of a Noun Phrase ⟨NP⟩ and a Verb Phrase ⟨VP⟩. Two more context-free grammars appeared in examples 1 and 2 above. A subclass of context-free languages called *bounded-context languages* has become important in practical compiling.

A third type of restriction on productions restricts the number of terminals and nonterminals that each step can create. When, at most, one nonterminal symbol is used in both the right- and left-hand sides of a production, the production is said to be *linear*. If the nonterminal occurs to the right of all other symbols on the right-hand side of a production, the production is called a *right-linear production*. Similarly, if the nonterminal occurs to the left of all other symbols the production is called a *left-linear production*. A grammar is called right- (left-) linear if each of its productions is right- (left-) linear; a language is called *regular* if it can be generated by a right- or left-linear grammar.

Each of the above restrictions includes those above it. These types form a hierarchy that is summarized in Figure 7.3. We remark without proof that no type 3 grammar can generate the language defined in example 2, so type 3 is a strict subset of type 2. Similarly, no type 2 grammar can generate the language of example 3, so type 2 is a strict subset of type 1. Finally, type 1 is a strict subset of type 0. We classify languages according to the type of grammar that can generate them: a type i language is a language that can be generated by a grammar of type i (but not by one of type i+1, for i=0,1, or 2).

Languages can also be defined in terms of the machines (e.g., translators, interpreters) that accept them. To each of these general language types there is a corresponding type of abstract machine: for example, regular languages are languages that can be recognized by finite-state machines; while type 0 languages are all languages that can be recognized by a class of machines called Turing machines. The table in Figure 7.3 indicates the type of abstract machine corresponding to each class of formal grammar.

Type	Type of language and recognizing automation	Production form and restrictions
0	Contracting context-sensitive (Post systems): Turing machines	$\alpha \to \beta$ $\alpha,\beta \in (N \cup T)^*$; $\alpha \neq \lambda$
1	Noncontracting context-sensitive: non-deterministic linear-bounded automata	$\sigma\, \alpha\, \tau \to \sigma\, \beta\, \tau$ $\sigma,\tau \in (N \cup T)^*$; $\alpha,\beta \in (N \cup T)^* - \lambda$ and length $(\alpha) \leq$ length (β)
2	Context-free: non-deterministic push-down storage automata	$A \to \beta$ $\beta \in (N \cup T)^* - \lambda$; $A \in N$
3	Regular or finite-state: finite-state automata	Right-linear: $A \to aB$, $A \to a$ Left-linear: $A \to Ba$, $A \to a$ $a \in T$; $A,B \in N$

Notation: By convention X^* consists of all finite strings of symbols from the set X including the empty string λ.

FIGURE 7.3 Basic formal grammars

7.5 BACKUS-NAUR FORM – BACKUS NORMAL FORM – BNF

BNF is a notation for writing grammars that is commonly used to specify the syntax of programming languages. In BNF nonterminals are written as names enclosed in corner-brackets '⟨ ⟩'. The sign → is written ::= (read "is replaced by"). Alternative ways of rewriting a given nonterminal are separated by a vertical bar, |, (read "or"). Using BNF notation, example 1 of the last section would be written

⟨Σ⟩::= ⟨A⟩ ⟨B⟩ — *read "the sign Σ is replaced by A followed by B"*
⟨A⟩::= a ⟨A⟩ | a — *read "A is replaced by 'a' followed by A or by 'a'"*
⟨B⟩::= ⟨B⟩ b | b

As an example of BNF, we specify in Figure 7.4 a FORTRAN-like language consisting only of GO TO statements. The GO TO statements have statement labels and reference labels of arbitrary length. It would of course be desirable to have all the reference labels appear among the list of statement labels. We will see that we cannot impose this restriction on a language using Backus Normal Form.

```
⟨letter⟩ ::= A | B | C | . . . | Z   — (reader reads "letter is replaced by A or B. . .)
⟨identifier⟩ ::= ⟨letter⟩ | ⟨letter⟩ ⟨identifier⟩
⟨go to stm⟩ ::= GO TO ⟨identifier⟩;
⟨program⟩ ::= ⟨identifier⟩ ⟨go to stm⟩ | ⟨identifier⟩ ⟨go to stm⟩ ⟨program⟩
```

FIGURE 7.4 A BNF specification of a subset of FORTRAN-like language

An example of a program in this language would be:

```
   AB    GO TO    XY;
   XY    GO TO    WXYZA;
WXYZA    GO TO    AB;
```

According to the specification it is possible to generate a program such as "A GO TO B; C GO TO D;" where no statements are labelled B or D. We would like to exclude such programs. However, there is no way to do this formally in a BNF specification. The set ⟨program⟩ is much larger than the set that we wish to call valid programs. BNF notation is equivalent to context-free (or phrase-structure) grammar, so class symbols expand without reference to surrounding context. We would like to distinguish a subset of programs in which every reference label occurs as a statement label. To express such a relationship or function, we need a more powerful formal system, one with the capability of cross-reference between elements of the sentence structure that it generates. Cross-reference is a context-sensitive feature.

7.6 CANONIC SYSTEMS

We present canonic systems as another more powerful method for defining languages. The author feels that canonic systems provide a useful vehicle for a theoretical approach to language. They also exemplify some of the theoretical work currently being done, and the same basic questions that we pose about them may be directed at any formal language specification.

A *canonic system* is a type of formal system that operates on several sets of strings over a finite alphabet. Canonic systems (equivalent to Smullyan's *ele-*