Chapter 16

Basic Concepts

16.1 Languages, grammars and automata

At one level of description, a natural language is simply a set of strings—finite sequences of words, morpheme, phonemes, or whatever. Not every possible sequence is in the language: we distinguish the *grammatical* strings from those that are *ungrammatical*. A *grammar*, then, is some explicit device for making this distinction; it is, in other words, a means for selecting a subset of strings, those that are grammatical, from the set of all possible strings formed from an initially given alphabet or vocabulary.

In this chapter we will consider two classes of formal devices which can function as grammars in this very general sense: (1) automata, which are abstract computing machines, and (2) string rewriting systems, which generally bear the name "grammar" or "formal grammar". The latter will be familiar to linguists inasmuch as grammars in this sense have formed the basis of much of the work in generative transformational theory.

We begin by considering certain properties of strings and sets of strings. Given a finite set A, a *string on* (or *over*) A is a finite sequence of occurrences of elements from A. For example, if $A = \{a, b, c\}$, then *acbaab* is a string on A. Strings are by definition finite in length. (Infinite sequences of symbols are also perfectly reasonable objects of study, but they are not suitable as models for natural language strings.) The set from which strings are formed is often called the *vocabulary* or *alphabet*, and this too is always assumed to be finite. The length of a string is, or course, the number of occurrences of symbols in it (i.e., the number of tokens, not the number of types). The string *acbaab* thus is of length 6.

Because we are dealing with tokens of an alphabet, there is an important difference between the linearly ordered sequences we call strings and a linearly ordered set. If the set $A = \{a, b, c\}$ were linearly ordered, say, as $b \rightarrow a \rightarrow c$, each element of A would occupy a unique place in the ordering. In a string, e.g., *acbaab*, tokens of a, occur in the first, fourth, and fifth positions.

To be formal, one could define a string of length n over the alphabet A to be a function mapping the first n positive integers into A. For example, *acbaab* would be the function $\{\langle 1, a\rangle, \langle 2, c\rangle, \langle 3, b\rangle, \langle 4, a\rangle, \langle 5, a\rangle, \langle 6, b\rangle\}$. There is little to be gained in this case by the reduction to the primitives of set theory, however, so we will continue to think of strings simply as finite sequences of symbols. A string may be of length 1, and so we distinguish the string b of length 1 from the symbol b itself. We also recognize the (unique) string of length 0, the *empty string*, which we will denote e (some authors use Λ). Two strings are identical if they have the same symbol occurrences in the same order; thus, *acb* is distinct from *abc*, and strings of different length are always distinct.

An important binary operation on strings is concatenation, which amounts simply to juxtaposition. For example, the strings *abca* and *bac* can be concatenated, in the order mentioned, to give the string *abcabac*. Sometimes concatenation is denoted with the symbol "$\frown$" thus, $abca^\frown bac$. Concatenation is associative since for any strings $\alpha, \beta, \gamma, (\alpha^\frown\beta)^\frown\gamma = \alpha^\frown(\beta^\frown\gamma)$, but it is not commutative, since in general $\alpha^\frown\beta \neq \beta^\frown\alpha$. The empty string is the identity element for concatenation; i.e., for any string α, $\alpha^\frown e = e^\frown\alpha = \alpha$.

Given a finite set A, the set of all strings over A, denoted A^*, together with the operation of concatenation constitutes a monoid. Concatenation is well-defined for any pair of strings in A^* and the result is a string in A^*; the operation is associative; and there is an identity element. $\langle A^*, {}^\frown \rangle$ fails to be a group since no element other than e has an inverse: no string concatenated with a non-empty string x will yield the empty string. Since concatenation is not commutative, $\langle A^*, {}^\frown \rangle$ is not an Abelian monoid.

A frequently encountered unary operation on strings is reversal. The reversal of a string x, denoted x^R, is simply the string formed by writing the symbols of x in the reverse order. Thus $(acbab)^R = babca$. The reversal of e is just e itself. To be formal, we could define reversal by induction on the length of a string:

DEFINITION 16.1 *Given an alphabet A:*

(1) *If x is a string of length 0, then $x^R = x$ (i.e., $e^R = e$)*

(2) *If x is a string of length $k+1$, then it is of the form wa, where $a \in A$ and $w \in A^*$; then $x^R = (wa)^R = aw^R$.*

■

Concatenation and reversal are connected in the following way: For all strings x and y, $(x^\frown y)^R = y^{R\frown} x^R$. For example,

(16–1) $(bca^\frown ca)^R = (ca)^{R\frown}(bca)^R = ac^\frown acb = acacb$

Given a string x, a *substring of x* is any string formed from continguous occurrences of symbols in x taken in the same order in which they occur in x. For example, bac is a substring of $abacca$, but neither bcc nor cb is a substring. Formally, y is a substring of x iff there exist strings z and w such that $x = z^\frown y^\frown w$. In general, z or w (or both) may be empty, so every string is trivially a substring of itself. (Non-identical substrings can be called *proper* substrings.) The empty string is a substring of every string; i.e., given x we can choose z in the definition as e and w as x so that $x = e^\frown e^\frown x$.

An initial substring is called a *prefix*, and a final substring, a *suffix*. Thus, ab is a (proper) prefix of $abacca$, and cca is a (proper) suffix of this string.

We may now define a *language* (over a vocabulary A) as any subset of A^*. Since A^* is a denumerably infinite set, it has cardinality $\aleph_o$; its power set, i.e., the set of all languages over A, has cardinality $2^{\aleph_o}$ and is thus non-denumerably infinite. Since the devices for characterizing languages which we will consider, *viz.*, formal grammars and automata, form denumerably infinite classes, it follows that there are infinitely many languages–in fact, non-denumerably infinitely many–which have no grammar. What this means in intuitive terms is that there are languages which are such motley collections of strings that they cannot be completely characterized by any finite device. The languages which *are* so characterizable exhibit a certain amount of order or pattern in their strings which allows these strings to be distinguished from others in A^* by a grammar or automaton with finite resources. The study of formal languages is essentially the investigation of a scale of complexity in this patterning in strings. For example, we might define a language over the alphabet $\{a, b\}$ in the following way:

(16–2) $L = \{x \mid x$ contains equal numbers of a's and b's (in any order)$\}$

We might then compare this language with the following:

(16–3) $L_1 = \{x \in \{a, b\}^* \mid x = a^n b^n (n \geq 0)\}$, i.e., strings consisting of some number of a's followed by the same number of b's
$L_2 = \{x \in \{a, b\}^* \mid x$ contains a number of a's which is the square of the number of b's$\}$

Is L_1 or L_2 in some intuitive sense more complex than L? Most would probably agree that L_2 is a more complex language than L in that greater effort would be required to determine that the members of a's and b's stood the "square" relation than to determine merely that they were equal. In other words, a device which could discriminate strings from non-strings of L_2 would have to be more powerful or more "intelligent" than a device for making the comparable discrimination for L.

What of L_1 and L? Here our intuitions are much less clear. Some might think that it would require a less powerful device to recognize strings in L reliably than to recognize strings in L_1; others might think it is the other way around or see no difference. As it happens, the particular scale of complexity we will investigate (the so-called Chomsky Hierarchy) does regard L_2 as more complex than L but puts L_1 and L in the same complexity class. At least this is so for the overall complexity measure. Finer divisions could be established which might distinguish L_1 from L

One linguistic application of these investigations is to try to locate natural languages on this complexity scale. This is part of the overall task of linguistics to characterize as precisely as possible the class of (potential and actual) natural languages and to distinguish this class from the class of all language-like systems which could not be natural languages. One must keep clearly in mind the limitations of this enterprise, however, the principal one being that languages are regarded here simply as string sets. It is clear that sentences of any natural language have a great deal more structure than simply the concatenation of one element with another. Thus, to establish a complexity scale for string sets and to place natural languages on this scale may, because of the neglect of other important structural properties, be to classify natural language along an ultimately irrelevant dimension. Extending results from the study of formal languages into linguistic theory must therefore be done with great caution.

16.2 Grammars

A formal grammar (or simply, grammar) is essentially a deductive system of axioms and rules of inference (see Chapter 8), which generates the sentences of a language as its theorems. By the usual definitions, a grammar contains just one axiom, the string consisting of the *initial symbol* (usually S), and a finite number of rules of the form $\psi \to \omega$, where ψ and ω are strings, and the interpretation of a rule is the following: whenever ψ occurs as a substring of any given string, that occurrence may be replaced by ω to yield a new string. Thus if a grammar contained the rule $AB \to CDA$, we could derive from the string $EBABCC$ the string $EBCDACC$.

Grammars use two alphabets: a *terminal alphabet* and a *non-terminal alphabet*, which are assumed to be disjoint. The strings we are interested in deriving, i.e., the sentences of the language, are strings over the terminal alphabet, but intermediate strings in derivations (proofs) by the grammar may contain symbols from both alphabets. We also require in the rules of the grammar that the string on the left side not consist entirely of terminal symbols. Here is an example of a grammar meeting these requirements:

(16–4) V_T (the terminal alphabet) $= \{a, b\}$
V_N (the non-terminal alphabet) $= \{S, A, B\}$
S (the initial symbol—a member of V_N)

$$R \text{ (the set of rules)} = \left\{ \begin{array}{lll} S & \to & ABS \\ S & \to & e \\ AB & \to & BA \\ BA & \to & AB \\ A & \to & a \\ B & \to & b \end{array} \right\}$$

A common notational convention is to use lower case letters for the terminal alphabet and upper case letters for the non-terminal alphabet.

A derivation of the string *abba* by this grammar could proceed as follows:

(16–5) $S \Longrightarrow ABS \Longrightarrow ABABS \Longrightarrow ABAB \Longrightarrow ABBA \Longrightarrow ABbA \Longrightarrow aBbA \Longrightarrow abbA \Longrightarrow abba$

Here we have used the symbol "$\Rightarrow$" to mean "yields in one rule application." Note that *abba* is not subject to further rewriting inasmuch as it consists entirely of terminal symbols and no rule licenses rewriting strings of terminals. The sequence (16-5) is said to be a *derivation (of abba from S)*, and the string *abba* is said to be *generated by* the grammar. The *language generated by* the grammar is the set of all strings generated. Here are the formal definitions:

DEFINITION 16.2 *Let* $\Sigma = V_T \cup V_N$. *A (*formal grammar G *is a quadruple* $\langle V_T, V_N, S, R\rangle$, *where* V_T *and* V_N *are finite disjoint sets,* S *is a distinguished member of* V_N, *and* R *is a finite set of ordered pairs in* $\Sigma^* V_N \Sigma^* \times \Sigma^*$. ∎

We have written $\psi \rightarrow \omega$ above for clarity instead of $\langle\psi, \omega\rangle$. The last condition simply says that a rule rewrites a string containing at least one non-terminal as some (possibly empty) string.

DEFINITION 16.3 *Given a grammar* $G = \langle V_T, V_N, S, R\rangle$, *a derivation is a sequence of strings* $x_1, x_2, \ldots, x_n$ $(n \geq 1)$ *such that* $x_1 = S$ *and for each* x_i $(2 \leq i \leq n)$, x_i *is obtained from* x_{i-1} *by one application of some rule in* R. ∎

To be completely formal, we would spell out in detail what it means to apply a rule of R to a string. The reader may want to do this as an exercise.

DEFINITION 16.4 *A grammar* G *generates a string* $x \in V_T^*$ *if there is a derivation* $x_1, \ldots, x_n$ *by* G *such that* $x_n = x$. ∎

Note that by this definition only strings of terminal symbols are said to be generated.

DEFINITION 16.5 *The language generated by a grammar* G, *denoted* $L(G)$, *is the set of all strings generated by* G. ∎

The language generated by the grammar in the example of (16-4) is $\{x \in \{a, b\}^* \mid x$ contains equal numbers of a's and b's $\}$.

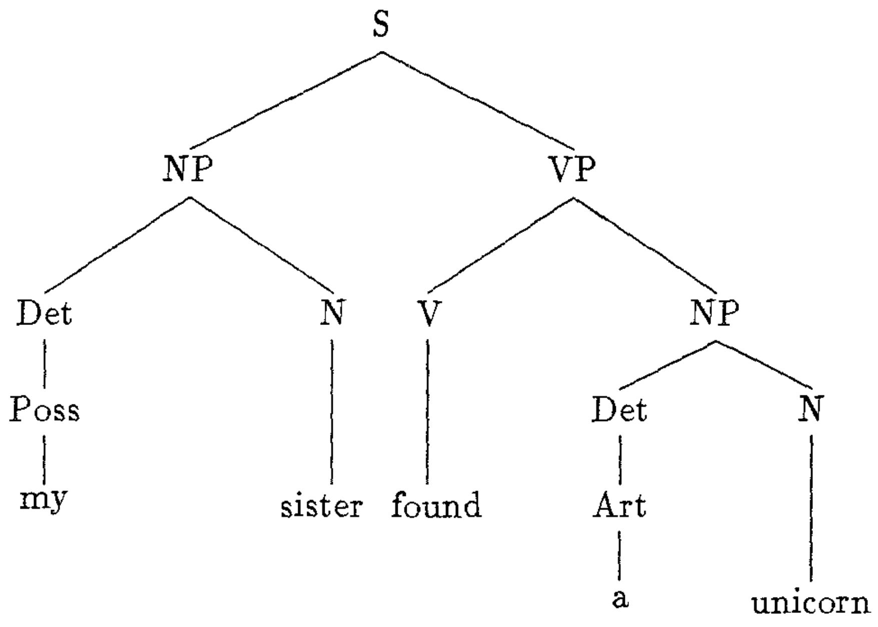


Figure 16–1: A typical constitutent structure tree

16.3 Trees

When the rules of a grammar are restricted to rewriting only a single non-terminal symbol, it is possible to contrue grammars as generating *constituent structure trees* rather than simply strings. An example of such a tree is shown in Fig. 16-1.

Such diagrams represent three sorts of information about the syntactic structure of a sentence:

1. The hierarchical grouping of the parts of the sentence into constituents
2. The grammatical type of each constituent
3. The left-to-right order of the constituents

For example, Fig. 16-1 indicates that the largest constitutent, which is labeled by S (for Sentence), is made up of a constituent which is a N(oun) P(hrase) and one which is a V(erb) P(hrase) and that the noun phrase is composed of two constitutents: a Det(erminer) and a N(oun), etc. Further, in the sentence constituent the noun phrase precedes the verb phrase, the determiner precedes the noun in the noun phrase constituents, and so on. The tree diagram itself is said to be composed of *nodes*, or points, some of which are connected by lines called *branches*. Each node has associated with it a *label* chosen from a specified finite set of grammatical categories (S, NP, VP, etc.) and formatives (*my, sister,* etc.). As they are customarily drawn, a tree diagram has a vertical orientation on the page with the nodes labeled by the formatives at the bottom. Because a branch always connects a higher node to a lower one, it is an inherently directional connection. This directionality is ordinarily not indicated by an arrow, as in the usual diagrams of relations, but only by the vertical orientation of the tree taken together with the convention that a branch extends *from* a higher node *to* a lower node.

16.4 Grammars and trees

As we have said, if a grammar has only rules of the form $A \rightarrow \psi$, where A is a nonterminal symbol, there is a natural way to associate applications of such rules with the generation of a tree. For example, if the grammar contains the rule $A \rightarrow aBc$, we can associate this with the (sub)tree in Fig. 16-5.

in which A immediately dominates a,B, and c, and the latter three elements stand in the precedence relation in the order given. Further, if the grammar

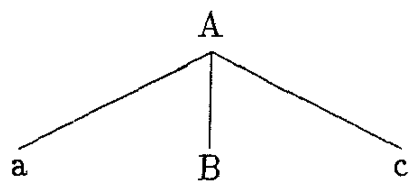


Figure 16–5.

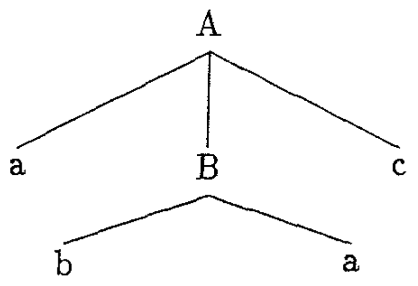


Figure 16–6

also contains the rule $B \rightarrow ba$, we can apply this rule at the node labelled B in the preceding tree to produce the tree shown in Fig. 16-6.

Let us define the *yield* of a tree as the string formed by its leaves ordered according to the precedence relation. The yield of the tree in Fig. 16-6, for example, is *abac*; that of Fig. 16-5 is aBc. We can now say:

DEFINITION 16 10 *A grammar (having all rules of the form $A \rightarrow \psi$)* generates *a tree iff all the following hold:*

(i) *the root is labelled with the initial symbol of the grammar*

(ii) *the yield is a string of terminal symbols*

(iii) *for each subtree of the form* $\underset{\alpha_1 \cdots \alpha_n}{\overset{A}{\triangle}}$ *in the tree, where A immediately dominates $\alpha_1 \dots \alpha_n$, there is a rule in the grammar $A \rightarrow \alpha_1 \dots \alpha_n$.*

■

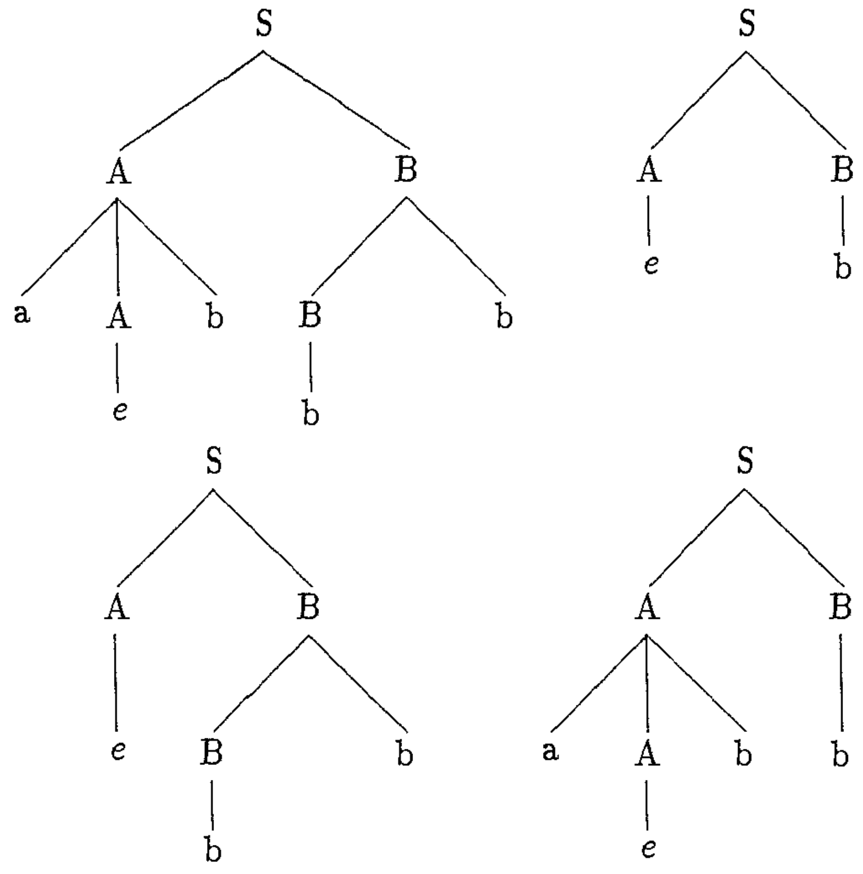


Figure 16–7

Thus the grammar $G = \langle \{a, b\}, \{S, A, B\}, S, R\rangle$ where

$$R = \left\{ \begin{array}{ll} S \rightarrow AB & B \rightarrow Bb \\ A \rightarrow aAb & B \rightarrow b \\ A \rightarrow e & \end{array} \right\}$$

generates trees such as those in Fig. 16-7. We can further say that a string is generated by such a grammar iff it is the yield of some tree which is generated. The language generated is, as usual, the set of all strings generated. For grammars in which there is only a single symbol on the left side of each rule, this definition and the earlier definition of generation of a string turn out to be equivalent: a string is generated (by the earlier definition) iff it is the yield of some generated tree.

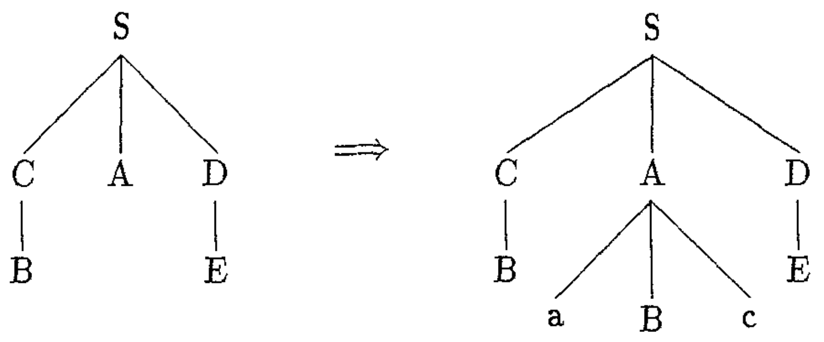
Problem: What language is generated by the above grammar?

Such grammars have interested linguists precisely because of the possibility of specifying a constituent structure tree for each string generated In attempting to write such grammars for natural languages, however, linguists have noted that often such rules are not universally applicable but may be allowed only in certain contexts. For example, a rule rewriting Det(erminer) as *many* might be applied only if the following noun were a plural form. Such considerations led to the investigation of formal grammar rules of the form $A \rightarrow \psi/\alpha\_\beta$, where the "/" is read "in the context", and where "_" marks the position of the A. The interpretation of such a rule is that the symbol A can be replaced by the string ψ in a derivation only when the string α immediately precedes A and the string β immediately follows A The context specifications are not necessarily exhaustive: additional symbols may occur to the left of the α and to the right of β. For example, if the rule were $A \rightarrow aBc/C\_Dc$, then the string $BECADcbA$ could be rewritten as $BECaBcDcbA$

Such rules are called *context sensitive* in contrast to rules of the form $A \rightarrow \psi$, which are called *context free* A context free rule, thus, is a context sensitive rule in which the context is null

A context sensitive rule $A \rightarrow \psi/\alpha\_\beta$ can also be written as $\alpha A\beta \rightarrow \alpha\psi\beta$ in conformity with the schema for grammar rules generally. So long as we regard these grammars as string rewriting systems the notations are interchangeable: in either case we may replace A by ψ when we find the substring $\alpha A\beta$. However, if we want to think of context sensitive rules as generating trees, the two representations may not be equivalent. For example, the rule $CABD \rightarrow CAaBD$ could be construed either as $A \rightarrow Aa/C\_BD$ or as $B \rightarrow aB/CA\_D$, and the associated trees would obviously differ depending on whether an A node or a B node was expanded.

Another problem which arises is how the context restriction is to be satisfied by the tree. If we think of the rules as specifying how one tree is to be converted into the next in a derivation, then does a rule such as $A \rightarrow aBc/C\_D$ mean that the C and D must be *leaves* immediately to the left and right, respectively, of A when the rule is applied, or is it sufficient that the C *immediately precede* the A and the D *immediately follow*, without necessarily being leaves along with A? Under the latter interpretation, the following derivational step would be allowed, but by the former it would not.



Note also that in the definition of tree derivation by means of context free rules in Def 16-10 above, we essentially thought of the trees being somehow given in advance and then checked for well-formedness by the grammar rules. That is, the rules served as so-called "node admissibility conditions" rather than as directions for converting one tree into another In the context free case, the two points of view are equivalent, but this is not the case for context sensitive rules. For example, the grammar

(16–6) $S \rightarrow AB$
$A \rightarrow a/\_b$
$B \rightarrow b/a\_$

will generate the tree

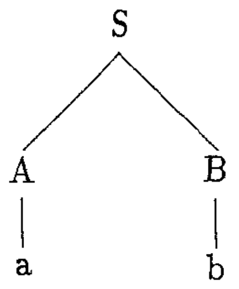


Figure 16–8.

if the rules are interpreted as node admissibility conditions but not if they are interpreted as tree generating rules (the problem being that the A cannot be rewritten until the B has, and vice versa.

16.5 The Chomsky Hierarchy

By putting increasingly stringent restrictions on the allowed forms of rules we can establish a series of grammars of decreasing generative power. Many such series are imaginable, but the one which has received the most attention is due to Chomsky and has come to be known as the Chomsky Hierarchy. At the top are the most general grammars of the sort we defined above in Section 16.2. There are no restrictions on the form of the rules except that the left side must contain at least one non-terminal symbol (Actually, even this restriction could be eliminated in favor of one which says simply that the left side cannot be the empty string. The formulation we have chosen is essentially a technical convenience) Chomsky dubbed such grammars 'Type 0,' and they are also sometimes called unrestricted rewriting systems (urs) The succeeding three types are as follows:

Type 1: each rule is of the form $\alpha A\beta \rightarrow \alpha\psi\beta$, where $\psi \neq e$.

Type 2: each rule is of the form $A \rightarrow \psi$.

Type 3: each rule is of the form $A \rightarrow xB$ or $A \rightarrow x$

In the above α, β, and ψ are arbitrary strings (possibly empty unless otherwise specified) over the union of the terminal and non-terminal alphabets; A and B are non-terminals, and x is a string of terminal symbols

Type 1 grammars are also called *context sensitive*; an equivalent formulation is to say that each rule is of the form $\psi \rightarrow \omega$, where ω is at least as long as ψ (i.e., the rules are "non-shrinking"). Type 2 grammars are called *context free*, and Type 3 grammars are called *regular* or *right linear* for reasons which will become apparent in the next section.

Note that these classes of grammars do not form a strict hierarchy in the sense that each type is a subclass of the one with the next lower number. Every Type 1 grammar is also a Type 0 grammar, but because rules of the form $A \rightarrow e$ are allowed in Type 2 grammars, these are not properly contained in Type 1. Type 3 grammars, however, are properly contained in the Type 2 grammars. It is nonetheless apparent, technical details concerning the empty string aside, that the hierarchy represents a series of generally increasing restrictions on the allowed form of rules.

The question then arises whether the languages generated by such grammars stand in an analogous relationship. We say that a language is of Type

n ($n = 0$, 1, 2, or 3) iff it is generated by some grammar of Type n. For example, we saw in Section 16.2 that $L = \{x \in \{a,b\}^* \mid x$ contains equal numbers of a's and b's$\}$ is of Type 0 inasmuch as it is generated by the grammar given in 16-4 But one might wonder whether it could also be generated by a grammar of some other type—say of Type 2 This is indeed the case; this language is generated by the following Type 2 grammar:

(16-7) $G = \langle\{a,b\},\{S,A,B\},S,R\rangle$ where

$$R = \left\{\begin{array}{ll} S \rightarrow e & A \rightarrow a \\ S \rightarrow aB & A \rightarrow aS \\ S \rightarrow bA & A \rightarrow bAA \\ B \rightarrow b & B \rightarrow aBB \\ B \rightarrow bS & \end{array}\right\}$$

This fact immediately establishes this language as Type 0 also, since every Type 2 grammar is perforce a Type 0 grammar. (It does not at the same time establish it as a Type 1 language since the given grammar is not Type 1, because of the rule $S \rightarrow e$ In fact, this language could not be Type 1 since Type 1 languages can never contain e.)

Is this language also Type 3? It turns out that it is not, but to prove this is not a simple matter. One must show somehow that *no* Type 3 grammar, however elaborate, can generate this language. We will consider techniques for proving such results in later sections.

Note that if one has two classes of grammars G_i and G_j such that G_i is properly contained in G_j, it does not necessarily follow that the corresponding classes of languages stand in the proper subset relation. Because every Type i grammar is also a Type $i+1$ grammar it *does* follow that every Type i language is also a Type $i+1$ language, i e., $L_i \subseteq L_{i+1}$. But it might also be the case that every Type $i+1$ language happens to have some Type i grammar which generates it. In such a case L_i is a subset of L_{i+1} but not a proper subset. Among the earliest results achieved in the study of formal grammars and languages were proofs that the inclusions among the languages of the Chomsky hierarchy are in fact proper inclusions. Specifically,

(i) the Type 3 languages are properly included in the Type 2 languages;

(ii) the Type 2 languages not containing the empty string are properly included in the Type 1 languages;

(iii) the Type 1 languages are properly included in the Type 0 languages.

Some of the proofs will be sketched in the following chapters

16.6 Languages and automata

As we mentioned at the beginning of this section, languages can also be characterized by abstract computing devices called automata. Ultimately we will define a hierarchy of automata and establish correspondences between them and the grammars of the Chomsky Hierarchy. This gives us yet another point of view from which to examine the notion of 'complexity of a language' which we hope eventually to put to use in characterizing natural language.

Before turning to the detailed study of the various classes of automata, it would be well to make a few general remarks about these devices.

An automaton is an idealized abstract computing machine—that is, it is a mathematical object rather than a physical one An automaton is characterized by the manner in which it performs computations: for any automaton there is a class of *inputs* to which it reacts, and a class of *outputs* which it produces, the relation between these being determined by the *structure*, or internal organization of the automaton We will consider only automata whose inputs and outputs are discrete (e.g., strings over an alphabet) rather than continuous (e.g , readings on a dial), and we will not deal with automata whose behavior is probabilistic

Central to the notion of the structure of an automaton is the concept of a *state*. A state of an automaton is analogous to the arrangement of bits in the memory banks and registers of an actual computer, but since we are abstracting away from physical realizations here, we can think of a state as a characteristic of an automaton which in general changes during the course of a computation and which serves to determine the relationship between inputs and outputs. We will consider only automata which have a finite number of states (cf a computer whose internal hardware at any given moment can be in only one of a finite number of different arrangements of 1's and 0's.)

An automaton may also have a *memory*. For the simplest automata, the memory consists simply of the states themselves. More powerful automata may be outfitted with additional devices, generally "tapes" on which the machine can read and write symbols and do "scratch work." Since the amount of memory available on such tapes is potentially unlimited, these machines can in effect overcome the limitations inherent in having only a finite number of states. We will see that the most powerful automata, Turing machines, are capable in principle of performing any computation for which an explicit set of instructions can be given

Automata may be regarded as devices for computing functions, i.e., for pairing inputs with outputs, but we will normally view them as *acceptors*, i.e., devices which, when given an input, either accept or reject it after some finite amount of computation. In particular, if the input is a string over some alphabet A, then an automaton can be thought of as the acceptor of some language over A and the rejector of its complement. As we will see, it is also possible to regard automata as *generators* of strings and languages in a manner similar to grammars