1

Foundations of Language Theory—Strings and How to Generate Them

1.1 INTRODUCTION

Formal language theory concerns itself with sets of strings called "languages" and different mechanisms for generating and recognizing them. Certain finitary processes for generating these sets are called "grammars." A mathematical theory of such objects was proposed in the latter 1950's and has been extensively developed since that time for application both to natural languages and to computer languages. The aim of this book is to give a fairly complete account of the basic theory and to go beyond it somewhat into its applications to computer science.

In the present chapter, we begin in Section 1.2 by giving the basic string-theoretic notation and concepts that will be used throughout. Section 1.3 gives a number of useful results for dealing with strings. Many of those results will not be needed until later and so the section has an asterisk after its number. Sections which are starred in this manner may be read lightly. When references to a theorem from that section occur later, the reader is advised to return to the starred section and read it more thoroughly.

Section 1.4 introduces the family of phrase-structure grammars, which are fundamental objects of our study. A number of examples from natural language and programming languages help to motivate the study. Section 1.5 introduces many other families of grammars and indicates intuitively how their languages may also be characterized by different types of automata.

Section 1.6 is devoted to trees and their relation to context-free grammars. The intuition developed in that section will be important in subsequent chapters.
1.2 BASIC DEFINITIONS AND NOTATION

A formalism is required to deal with strings and sets of strings. Let us fix a finite nonempty set \( \Sigma \) called an alphabet. The elements of \( \Sigma \) are assumed to be indivisible symbols.

**Examples** \( \Sigma = \{0, 1\} \) or \( \Sigma = \{a, b\} \).

In certain applications to compiling, we might have an alphabet which contains begin, end. These are assumed to be indivisible symbols in the applications. A set \( X = \{1, 10, 100\} \) is not an alphabet since the elements of \( X \) are made up of other basic symbols.

**Definition** A word (or string) over an alphabet \( \Sigma \) is a finite-length \( \Sigma \)-sequence.

A typical word can be written as \( x = a_1 \cdots a_k, k \geq 0, a_i \in \Sigma \) for \( 1 \leq i \leq k \). We allow \( k = 0 \), which gives the null (or empty) word, which is denoted by \( \Lambda \). \( k \) is called the length of \( x \), written \( \lg(x) \), and is the number of occurrences of letters of \( \Sigma \) in \( x \).

Let \( \Sigma^* \) denote the set of all finite-length \( \Sigma \)-sequences.

Let \( x \) and \( y \) be in \( \Sigma^* \). Define a binary operation of concatenation, where we form the new word \( xy \).

**Example** Let \( \Sigma = \{0, 1\} \) and

\[
\begin{align*}
x &= 010 & y &= 1
\end{align*}
\]

Then

\[
\begin{align*}
xy &= 0101 & yx &= 1010
\end{align*}
\]

The following proposition summarizes the basic properties of concatenation.

**Proposition 1.2.1** Let \( \Sigma \) be an alphabet.

a) Concatenation is associative; i.e., for each \( x, y, z \) in \( \Sigma^* \),

\[
(xyz) = (xy)z
\]

b) \( \Lambda \) is a two-sided identity for \( \Sigma^* \); i.e., for each \( x \in \Sigma^* \),

\[
x\Lambda = \Lambda x = x
\]

c) \( \Sigma^* \) is a monoid under concatenation and \( \Lambda \) is the identity element.

\[\lg(xy) = \lg(x) + \lg(y)\]

It is necessary to extend our operations on strings to sets. Let \( X, Y \) be sets of words. Hereafter, one writes \( X \cup Y \). Then the product of two sets is:

\[XY = \{xy | x \in X, y \in Y\}\]

Exponent notation can be used in an advantageous fashion for sets. Let \( X \subseteq \Sigma^* \). Define

\[X^0 = \{\Lambda\}\]

and semigroup closure by

\[X^* = \bigcup_{i \geq 0} X^i\]

and

\[X^+ = \bigcup_{i > 0} X^i\]

**Examples** Let \( \Sigma = \{0, 1\} \). In general,

\[
\Sigma^i = \{x \in \Sigma^* | \lg(x) = i\}
\]

Thus

\[
\Sigma^0 = \{\Lambda\}
\]

Note that

\[
\Sigma^* = (\Sigma^*)^*
\]

Also, if

\[
X = \{\Lambda, 01\} \quad \text{and} \quad Y = \{0, 0\}
\]

then

\[XY = \{0, 01, 00, 010, 011, 11, 100, 1010\}\]

**Proposition 1.2.2** For each \( X \subseteq \Sigma^* \),

a) for each \( i \geq 0 \), \( X^i \subseteq X^* \);

b) for each \( i \geq 0 \), \( X^i \subseteq X^* \);

c) \( \Lambda \in X^* \);

d) \( \Lambda \in X^* \) if and only if \( \Lambda \in X \).

\[\text{A semigroup consists of a set } S \text{ with a binary associative operation } \cdot \text{ defined on } S. \text{ A monoid is a semigroup which possesses a two-sided identity. The set of functions on } S \text{ set is a monoid under the operation of functional composition.}\]

\[\text{The set of all } \Sigma \text{-words can be obtained by applying the star operation to } \Sigma.\]
Examples Let $\Sigma = \{a, b\}$ and $L = ab^* = \{ab^i \mid i \geq 0\}$. Then

$$L^T = b^*a = \{ba^i \mid i \geq 0\}.$$ 

Also, the following sequence of containments is a good test of familiarity with the notation:

$$\{ab^i \mid i \geq 0\} \subseteq a^*b^* = \{ab^i \mid 0 \leq i \leq 0\} \subseteq \{a, b\}^*;$$

$$P = \{ww^T \mid w \in \{a, b\}^*\}.$$ 

$P$ is the set of even-length "palindromes." A palindrome is a string that is the same whether written forwards or backward. We conclude this section by giving some natural-language examples.†

RADAR
ABLE WAS I SAW ELBA
MADAM IM ADAM
A MAN A PLAN A CANAL PANAMA

### 1.3* $\Sigma^*$ REVISITED — A COMBINATORIAL VIEW

In this section, we shall take a brief, more algebraic look at the monoid $\Sigma^*$. We begin by recalling some concepts from semigroup theory.

If we have some semigroup $S$, let $T$ be a subset of $S$ (not necessarily a subsemigroup). $T$ is contained in at least one subsemigroup, namely $S$. Let $T^*$ be the least subsemigroup containing $T$ (i.e., the intersection of all subsemigroups of $S$ containing $T$). We say that $T^*$ is the semigroup generated by $T$. If $T^* = S$, we say that $T$ is a set of generators for $S$. Similar remarks hold for monoids, except that we would write $T^*$ instead of $T^*$. There is another important type of generation.

**Definition** Let $T$ be a set of generators of a semigroup (monoid) $S$. $S$ is free over $T$ if each element of $S$ has a unique representation as a product of elements from $T$.

**Fact** $\Sigma^*$ is free over $\Sigma$. Thus if we have $x, y \in \Sigma^*$, where $x = a_1 \cdots a_m, a_i \in \Sigma, m \geq 0,$ and $y = b_1 \cdots b_n, b_i \in \Sigma, n \geq 0$, then we may have $x = y$ if and only if $m = n$ and $a_i = b_i$ for all $i, 1 \leq i \leq m = n$.

There is an important but simple theorem that lies at the root of the study of equations in $\Sigma^*$.

**Theorem 1.3.1 (Levi)** Let $\nu, \omega, x,$ and $y$ be words in $\Sigma^*$. If $\nu \omega = x y$, then:

i) if $\lg(\nu) \geq \lg(x)$, then there exists a unique word $z \in \Sigma^*$ such that $\nu = xz$ and $y = \omega$;

ii) if $\lg(\nu) = \lg(x)$, then $\nu = x$ and $\omega = y$;

† The spaces between words are included only for clarity in the last two examples.
iii) If \( \lg(y) \leq \lg(x) \), then there exists a unique word \( z \in \Sigma^* \) such that \( x = yz \) and \( w = yz \).

**Proof** The most intuitive way to deal with general problems of this type is to draw a picture of the strings, as follows: 

```
  \[
  \begin{array}{c|c|c|c}
    \ & x & \ & z & \ & \ \\
    \ & \ & \ & \ & \ & \ \\
  \end{array}
  \]
```

From the picture, it is clear that the desired relationships hold.

The proof follows by considering whether the line between \( x \) and \( y \) in the second row of the picture goes to the left or right of the line between \( z \) and \( w \).

A formal proof of the result would proceed by induction on \( \lg(\omega w) \), and is omitted.

**Definition** Let \( w \) and \( y \) be in \( \Sigma^* \). Write \( w \leq y \), \( w \) is a *prefix* of \( y \), if there exists \( y' \in \Sigma^* \) so that 

\[
y' = w y'.
\]

Also, \( w \) is a *suffix* of \( y \) if \( y = y' w \) for some \( y' \in \Sigma^* \).

This allows us to state a corollary to Theorem 1.3.1.

**Corollary** Let \( u, w, x, y \in \Sigma^* \). If \( \omega w = xy \), then 

\[
x \leq u \quad \text{or} \quad u \leq x
\]

As an example of the usefulness of these little results, we prove the following easy proposition.

**Proposition 1.3.1** Let \( w, x, y \), and \( z \) be in \( \Sigma^* \). If \( w \leq y \) and \( x \leq yz \) then either 

\[
w \leq x \quad \text{or} \quad x \leq w.
\]

**Proof** \( w \leq y \) and \( x \leq yz \) imply that there exist \( y' \) and \( x' \) so that 

\[
y = wy' \quad (1.3.1)
\]

\[
yz = xx' \quad (1.3.2)
\]

Thus (1.3.1) and (1.3.2) imply that 

\[
yz = wy'z = xx'
\]

but \( w(y'z) = xx' \) implies that 

\[
x \leq w \quad \text{or} \quad w \leq x.
\]

A more interesting and useful result is the following.

**Theorem 1.3.2** Let \( y \in \Sigma^* \) and \( x, z \in \Sigma^* \) such that \( xy = yz \). Then there exist \( u, v \in \Sigma^* \) and \( p, q \geq 0 \) such that \( x = uv, z = vu \), and \( y = (uv)^p u = u(vu)^q \).

**Proof** We induct on the length of \( y \).

**Basis.** If \( \lg(y) = 0 \), we have \( u = z = x = 0 \), and \( p = 0 \).

**Induction step.** There are two cases depending on the comparative lengths of \( x \) and \( v \).

**CASE 1.** \( \lg(y) > \lg(v) \).

By Theorem 1.3.1 applied to \( xy = yz \), we know that there is \( v \in \Sigma^* \) so that 

\[
x = yv \quad \text{and} \quad z = vy
\]

We may choose \( p = 0 \) and \( u = y \) and the result is proven. Note that if \( \lg(x) = \lg(y) \), then \( v = 0 \), but the proof is still valid.

**CASE 2.** \( \lg(x) < \lg(v) \).

Again by Theorem 1.3.1, there exists \( w \in \Sigma^* \) so that 

\[
y = xw \quad \text{and} \quad y = wz
\]

and \( \lg(w) < \lg(y) \). Since we have 

\[
xw = wz
\]

with \( \lg(w) < \lg(y) \), we can assert, by the induction hypothesis, that there exist \( u, v \in \Sigma^* \), and \( p, q \geq 0 \) so that \( x = uv, z = vu \), and \( w = (uv)^p u = (uv)^q u \). From the above, we have 

\[
y = xw = u(vw) = u(vw) = (uv)^p u = (uv)^q u = (uv)^q u.
\]

The induction has been extended. 

Now we approach a very useful theorem which has many applications. The sufficient condition is particularly valuable.

**Theorem 1.3.3** Let \( u, v \in \Sigma^* \), \( u = w^m \) and \( v = w^n \) for some \( w \in \Sigma^* \), \( m, n \geq 0 \), if and only if there exist \( p, q \geq 0 \) so that \( u^p \) and \( v^q \) contain a common prefix (suffix) of length \( \lg(u) + \lg(v) - (\lg(u), \lg(v)) \), where \( (i, j) \) denotes the greatest common divisor of \( i \) and \( j \).

**Proof** Let \( \lg(u) = s \) and \( \lg(v) = t \). Let 

\[
u = a_0 \cdots a_s, \quad v = b_0 \cdots b_t
\]

with \( a_i, b_j \in \Sigma \), and suppose that \( s < t \).

Let us first suppose that the second condition is satisfied, that is, we deal with sufficiency first. Suppose that \( s < t \).
Claim: \( a_i = b_0 \) for each \( 0 \leq i < s \). That is, all the \( a_i \) are equal and thus \( \log(w) = 1 \).

**Proof.** We simply begin to pair off the first \( s + t - 1 \) letters of \( u^p \) and \( v^q \), and we get:

<table>
<thead>
<tr>
<th>( i )</th>
<th>( u^p )</th>
<th>( v^q )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>( a_0 = b_0 )</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>( a_1 = b_1 )</td>
<td></td>
</tr>
<tr>
<td>( s + t - 2 )</td>
<td>( a_k = b_k )</td>
<td></td>
</tr>
</tbody>
</table>

It is clear that \( a_i = b_i \) for each \( i \), \( 0 \leq i < s \).

Moreover, the following is true.

**Fact.** For each \( i \), \( 0 \leq i < s - 1 \),

\[
a_{(s+t) \mod s} = b_i.
\]

This fact is easily seen to be true for \( i = 0 \). From that, the case of \( i > 0 \) follows easily from the form of \( u^p \) and \( v^q \). Now let us consider the set of letters

\[
A = \{ a_{(s+t) \mod s} : 0 \leq k < s \}.
\]

Since \((s,t) = 1\), we have \( \{ s \mod s \} = \{ 0, 1, \ldots, s - 1 \} \). By this fact, every letter in \( u \) is in \( A \) and equals \( b_0 \). Thus all the letters in \( u \) are identical.

Therefore \( u = b_0^p \) and \( v = b_0^q \) since every letter in \( v \) is equal to some letter in \( u \).

To extend the result to the case where \( d = (s,t) > 1 \), simply take words of length \( d \) and work over \( \Sigma^d \), using the previous case.

**Necessity.** Let \( u = w^m \) and \( v = w^n \) for some \( w \in \Sigma^* \); \( m, n \geq 0 \). Let \( p = \left\lfloor \frac{m + n - (m,n)}{m} \right\rfloor \) and \( q = \left\lfloor \frac{m + n - (m,n)}{n} \right\rfloor \).

Then we have

\[
\begin{align*}
    u^p &= w^{mp} \\
v^q &= w^{nq}
\end{align*}
\]

but \( mp \geq m + n - (m,n) \) and \( nq \geq m + n - (m,n) \) so that \( u^p \) and \( v^q \) have a common prefix of length

\[
\geq \log(w)(m + n - (m,n)) = \log(u) + \log(v) - \log(u) \log(v).
\]

**Corollary.** If \( uv = vu \), where \( u, v \in \Sigma^* \), then there is \( w \in \Sigma^* \) so that \( u = w^m \) and \( v = w^n \) for some \( m, n \geq 0 \).

Now, an application of the previous theorem will be presented.

**Theorem 13.4.** Let \( X = \{ x, y \} \subseteq \Sigma^* \) with \( x \neq y \). \( X^* \) is free if and only if \( xy \neq yx \).

**Proof.** If \( X^* \) is free, then \( xy \neq yx \).

Conversely, suppose \( xy \neq yx \) with \( \Lambda \neq x \neq y \neq \Lambda \). Assume, for the sake of contradiction, that there is a nontrivial relation involving \( x \) and \( y \), say,

\[
x^{n_1}y^{m_1} \cdots x^{n_k}y^{m_k} = x^{m_1}x^{m_2} \cdots x^{m_k}y^{n_k} \quad (1.3.3)
\]

with \( n_i, m_i \geq 0 \) for all \( i \). [Note that (1.3.3) can be assumed to begin (end) with different words by left (right) cancellation.] We may assume that \( \log(y) \leq \log(x) \). This and (1.3.3) imply that \( y \) is a prefix of \( x \). Thus,

\[
x = y^q t_0 \quad (1.3.4)
\]

for some \( t_0 \in \Sigma^* \), where \( q \geq 1 \) is the largest power of \( y \) that is a prefix of \( x \). Substituting (1.3.4) in (1.3.3) yields:

\[
y^q t_0 y \cdots y^{n_k} = y^{m_1}y^q t_0 \cdots x^{m_k}.
\]

By left cancellation,

\[
t_0 y \cdots y^{n_k} = y^{m_1} t_0 \cdots x^{m_k}. \quad (1.3.5)
\]

Note that \( y \) cannot be a prefix of \( t_0 \) because, if it were, the definition of \( q \) would be contradicted. By (1.3.5), we know that \( t_0 \) is a prefix of \( y \). Hence,

\[
y = t_0 t_1
\]

for some \( t_1 \in \Sigma^* \). From (1.3.3) we must have that \( y \) is a suffix of \( x \). [Recall that \( \log(y) \leq \log(x) \)]. Since \( y = t_0 t_1 \) is a suffix of \( x \) and \( \log(y) = \log(t_0 t_1) \), we must have, from Eq. (1.3.4), that:

\[
y = t_1 t_0 = t_0 t_1 \quad (1.3.6)
\]
From the Corollary to Theorem 1.3.3 and Eq. (1.6), we know that there exists \( z \in \Sigma^* \) so that

\[
t_0 = z^{p_0}, \quad t_1 = z^{p_1}, \quad y = z^{p_2}
\]

for some \( p_0, p_1, p_2 \geq 0, p_0 + p_1 = p_2 > 0 \). But now we know from (1.3.4) that

\[
x = y^z t_0 = (t_0 t_1)^y t_0 = z^{p_2 y^z p_0}.
\]

Since both \( x \) and \( y \) are powers of \( z \), we know that:

\[
xy = yx = z^{(a \cdot y^z) p_2 - p_0}
\]

which is a contradiction. \( \square \)

In certain applications, it is useful to order strings.

**Definition** Let \( S \) be any set. A binary relation, to be written \( \leq \), is a partial order on \( S \) if it is reflexive, antisymmetric, and transitive, that is, if for each \( a, b, c \) in \( S \),

1) \( a \leq a \);
2) if \( a \leq b \) and \( b \leq a \), then \( a = b \);
3) if \( a \leq b \) and \( b \leq c \), then \( a \leq c \).

Moreover, \( \leq \) is said to be a total order if it is a partial order which satisfies the following additional property: For each \( a, b \in S \), either

\[
a < b \quad \text{or} \quad b < a.
\]

where the relation \( a < b \) is an abbreviation for \( a \leq b \) and \( a \neq b \).

For example, the "prefix relation" is a partial ordering of \( \Sigma^* \). The ordinary relation \( < \) on natural numbers is an example of a total order. Now we introduce a mathematical counterpart to a dictionary ordering.

**Definition** Let \( \Sigma \) be any alphabet and suppose that \( \Sigma \) is totally ordered by \( < \). Let \( x = x_1 \cdots x_r, y = y_1 \cdots y_s \), where \( r, s \geq 0 \), \( x_i, y_j \in \Sigma \). We say that \( x < y \) if

i) \( x_i = y_i \) for \( 1 \leq i < r \) and \( r < s \); or
ii) there exists \( k \geq 1 \) so that \( x_i = y_i \) for \( 1 \leq i < k \) and \( x_k < y_k \). \( x < y \) is referred to as lexicographic order.

This definition is not hard to understand:

i) says that if \( x \) is a proper prefix of \( y \) (that is, \( y \in \{ x \} \Sigma^+) \), then \( x < y \);
ii) says that if there exist \( u, v, w \in \Sigma^+ ; a, b, c \in \Sigma \) so that \( x = uav, y = ubv \)

and \( a \neq b \), then \( x < y \).

Note that if \( \Sigma = \{ a, b, \ldots, z \} \) and the total order is alphabetical, then the lexicographic order on \( \Sigma^* \) is the usual dictionary order.

**Example** Consider \( \Sigma = \{ 0, 1 \} \). We begin to enumerate the elements of \( \Sigma^* \) in lexicographic order.

\[
\begin{array}{c}
\Lambda \\
0 \\
00 \\
000 \\
\vdots \\
01 \\
\vdots \\
1 \\
\vdots \\
11 \\
\vdots
\end{array}
\]

First we justify that \( \cdot < \cdot \) is a total order on \( \Sigma^* \).

**Proposition 1.3.2** Let \( x, y, z \in \Sigma^* \).

1) If \( x < y, y < z \), then \( x < z \).
2) If \( x \preceq y \) and \( y \preceq x \), then \( x = y \).
3) For every \( x, y \in \Sigma^* \), either \( x = y \) or \( x < y \) or \( y < x \).

This proposition establishes \( \cdot < \cdot \) as a total order on \( \Sigma^* \). Next we establish the connection between concatenation and lexicographic order.

**Proposition 1.3.3** Let \( x, y, z \in \Sigma^* \). \( x \cdot y \) if and only if \( xz < yz \).

Next we show that things are different on the right.

**Proposition 1.3.4** Concatenation is not "monotone" on the right. That is, there exist \( x, y, z \in \Sigma^* \), such that \( x < y \), but \( xz < yz \).

**Proof** Let \( \Sigma = \{ 0, 1, 2 \} \). Choose \( x = 0 \), \( y = 01 \), and \( z = 2 \). Clearly \( x < y \), but \( xz = 02 \preceq 012 = yz \). \( \square \)

Now we will state and prove one of the main properties of lexicographic order.

**Theorem 1.3.5** Let \( x, y \in \Sigma^* \) and \( x < y \). Either

i) \( x \) is a prefix of \( y \), or

ii) for any \( z, z' \in \Sigma^* \), \( xz < yz' \).
Proof. Suppose \( x \) is not a proper prefix of \( y \). Then, since \( x < y \), there exists \( u, v, v' \in \Sigma^*; a, b \in \Sigma \), so that
\[
\begin{align*}
x &= uvv' \\
y &= ubv'
\end{align*}
\]
with \( a < b \). But \( xz = uvzv \) and \( yz = ubv'z \), so that \( xz < yz \).

Some examples will help to motivate and clarify these concepts.

Example of natural language The alphabet \( \Sigma \) would consist of all words in the language. Although large, \( \Sigma \) is finite. \( N \) would contain variables which stand for concepts that need to be added to a grammar to understand its structure, for example

\( \alpha \rightarrow \beta \) where \( \alpha \in \Sigma^* \Sigma^* \) and \( \beta \in \Sigma^* \).

Example from programming languages Here \( \Sigma \) would be all the symbols in the language. For some reasonable language, it might contain

\( A, B, C, \ldots, Z, \ :=, \ begin, \ end, \ if, \ then, \ etc. \)

The set of variables would contain the start symbol (program), as well as others like (compound-statement), (block), etc. A typical rule might be as follows:

\( \text{(for statement)} \rightarrow \text{(for control variable)} := \text{(for list)} \text{do (statement)} \)

Example 1.4.1 (of a more abstract grammar) Let \( \Sigma = \{a, b\}, N = \Sigma = \{A, S\} \) with the productions:

\[
\begin{align*}
S &\rightarrow ab \\
SA &\rightarrow bSb \\
S &\rightarrow aASb \\
A &\rightarrow \Lambda \\
A &\rightarrow bSb \\
A &\rightarrow aAAb \rightarrow aa
\end{align*}
\]

Note that a phrase-structure grammar has very general rules. The only requirement is that the left-hand side has at least one variable.

Grammars are made for “rewriting” or “generation,” as given by the following definition.

1 1.4 PHRASE-STRUCTURE GRAMMARS

There is an underlying framework for describing “grammars.” Our intuitive concept of a grammar is as a (finite) mechanism for producing sets of strings. The following concept has proved to be very useful in linguistics, in programming languages and even in biology.

Definition A phrase-structure grammar is a 4-tuple: \( G = (V, \Sigma, P, S) \), where:

\( V \) is a finite nonempty set called the total vocabulary;
\( \Sigma \subseteq V \) is a finite nonempty set called the terminal alphabet;
\( S \subseteq V - \Sigma = N \) is called the start symbol.

\( P \) is a finite set of rules (or productions) of the form

\( \alpha \rightarrow \beta \) where \( \alpha \in V^* \) and \( \beta \in V^* \).

Some examples will help to motivate and clarify these concepts.

Example of natural language The alphabet \( \Sigma \) would consist of all words in the language. Although large, \( \Sigma \) is finite. \( N \) would contain variables which stand for concepts that need to be added to a grammar to understand its structure, for example

\( \alpha \rightarrow \beta \) where \( \alpha \in \Sigma^* \Sigma^* \) and \( \beta \in \Sigma^* \).

Example from programming languages Here \( \Sigma \) would be all the symbols in the language. For some reasonable language, it might contain

\( A, B, C, \ldots, Z, \ :=, \ begin, \ end, \ if, \ then, \ etc. \)

The set of variables would contain the start symbol (program), as well as others like (compound-statement), (block), etc. A typical rule might be as follows:

\( \text{(for statement)} \rightarrow \text{(for control variable)} := \text{(for list)} \text{do (statement)} \)

Example 1.4.1 (of a more abstract grammar) Let \( \Sigma = \{a, b\}, N = \Sigma = \{A, S\} \) with the productions:

\[
\begin{align*}
S &\rightarrow ab \\
SA &\rightarrow bSb \\
S &\rightarrow aASb \\
A &\rightarrow \Lambda \\
A &\rightarrow bSb \\
A &\rightarrow aAAb \rightarrow aa
\end{align*}
\]

Note that a phrase-structure grammar has very general rules. The only requirement is that the left-hand side has at least one variable.

Grammars are made for “rewriting” or “generation,” as given by the following definition.

\( N \) is called the set of nonterminals, or the set of variables.

For the moment only, variables are being written in angeld brackets to distinguish them.
Definition Let $G = (V, \Sigma, P, S)$ be a phrase-structure grammar and let $\alpha', \beta' \in V^*$. $\alpha'$ is said to 
 directly generate $\beta'$, written $\alpha' \Rightarrow \beta'$ if there exist $\alpha_1, \alpha_2, \beta_1, \beta_2 \in V^*$, such that $\alpha' = \alpha_1 \alpha_2, \beta' = \alpha_1 \beta_2$ and $\alpha \Rightarrow \beta$ is in $P$. We write $\Rightarrow^*$ for the reflexive-transitive closure of $\Rightarrow$.

Example We use the grammar of Example 1.4.1:

\[
S \Rightarrow ab \\
S \Rightarrow aSb \Rightarrow abbb \Rightarrow ababb \\
S \Rightarrow^* (ab)^2 b^2
\]

A sequence like (1.4.1) is called a generation or derivation. It is convenient in displaying derivations, to underline the subword being rewritten, as is done in (1.4.1).

Certain conventions are adopted in usage of symbols. Capital letters near the beginning of the alphabet are used for elements of $V$ or $N$. Lower-case elements like $a, b, c$ designate elements of $\Sigma$ or $\Sigma_\Lambda = \Sigma \cup \{\Lambda\}$. One uses $\alpha, \beta, \gamma, \ldots$ for elements of $V^*$ and $a, b, c, \ldots$ for elements of $\Sigma^*$.

Definition Let $G = (V, \Sigma, P, S)$ be a phrase-structure grammar. The set of

sentential forms of $G$, written $S(G)$, is the set:

\[S(G) = \{\alpha \in V^* | S \Rightarrow^* \alpha\} \]

Intuitively, a sentential form is something derivable from $S$. Historically, $S$ stood for "sentencehood" and so, potentially, a sentential form could be expanded to a sentence. The set of sentences can be defined as the language generated by the grammar, as is done in the following definition.

Definition Let $G = (V, \Sigma, P, S)$ be a phrase-structure grammar. The language

generated by $G$, written $L(G)$, is the set:

\[L(G) = S(G) \cap \Sigma^* = \{w \in \Sigma^* | S \Rightarrow^* w\} \]

An important concept is expressed when we say that two grammars are

"equivalent." Since there are a number of concepts of equivalence which make sense, the term weak equivalence is sometimes used for this concept.

Definition If $G$ and $G'$ are phrase-structure grammars, then $G$ is said to be

(weakly) equivalent to $G'$ if $L(G) = L(G')$.

Let us work more closely with a grammar in order to get a feeling for what is involved. The example will show that these constructs do possess a certain complexity. Let $G = (V, \Sigma, P, S)$, where

\[
\Sigma = \{0, 1\}, \quad V = \Sigma \cup \{A, B, C, D, S\}.
\]

The rules of $P$ are listed in Table 1.1, with numbers and explanatory comments.

**TABLE 1.1**

<table>
<thead>
<tr>
<th>Number</th>
<th>Rule</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$S \rightarrow ABC$</td>
<td>start</td>
</tr>
<tr>
<td>2</td>
<td>$AB \rightarrow 0AD$</td>
<td>add a 0</td>
</tr>
<tr>
<td>3</td>
<td>$AB \rightarrow 1AE$</td>
<td>add a 1</td>
</tr>
<tr>
<td>4</td>
<td>$DC \rightarrow BOC$</td>
<td>drop a 0</td>
</tr>
<tr>
<td>5</td>
<td>$EC \rightarrow B1C$</td>
<td>drop a 1</td>
</tr>
<tr>
<td>6</td>
<td>$DO \rightarrow 0D$</td>
<td>skip right</td>
</tr>
<tr>
<td>7</td>
<td>$DI \rightarrow 1D$</td>
<td>remembering a 0</td>
</tr>
<tr>
<td>8</td>
<td>$EO \rightarrow 0E$</td>
<td>skip right</td>
</tr>
<tr>
<td>9</td>
<td>$F1 \rightarrow 1E$</td>
<td>remembering a 1</td>
</tr>
<tr>
<td>10</td>
<td>$AB \rightarrow \Lambda$</td>
<td>terminate</td>
</tr>
<tr>
<td>11</td>
<td>$C \rightarrow \Lambda$</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>$0B \rightarrow B0$</td>
<td>move B left to</td>
</tr>
<tr>
<td>13</td>
<td>$1B \rightarrow B1$</td>
<td>continue</td>
</tr>
</tbody>
</table>

What is $L(G)$ for this grammar? At this point, the answer is not clear, and even the intuitive comments given in the grammatical description do not help very much. It turns out that

\[L(G) = \{xx | x \in \{0, 1\}^*\}, \quad (1.4.2)\]

Even knowing the answer is of little help, since we must prove it. Proofs of this kind are important because we must verify that:

i) $xx$ is in $L(G)$ for each $x \in \{0, 1\}^*$;

ii) every string in $L(G)$ is of the form $xx$, where $x \in \{0, 1\}^*$.

Step (i) is not too hard in general. Step (ii) is necessary for making sure that a construction is correct.

Let us now work out some of the propositions needed to prove (1.4.2).

Consider derivations that start from the string $xABxC$, where $x \in \{0, 1\}^*$. We will determine all the ways in which the derivation can continue and ultimately be able to generate a sentence.

First of all, these considerations suffice to deal with (1.4.2), since

\[S \Rightarrow ABC = \Lambda AB\]

Now consider

\[xABxC \]

How shall we continue?

**CASE 1.** Apply production 10 to (1.4.3). We get:

\[xABxC \Rightarrow xxC \] (1.4.4)
Our only choice is to use production 11, to get:

\[ xx \in L(G). \]

This is fine, since we need this string in the language.

CASE 2. Apply production 11 to (1.4.3). We get:

\[ xABx \Rightarrow xABx \]

We could again apply production 10 but that would reproduce Case 1. So we'll use one of productions 2 or 3, say 2. Then we get:

\[ xABx \Rightarrow x0ADx \]

Now rules 6 and 7 are applicable (depending on the structure of \( x \)). There is no way that the \( D \) can ever be made to go away, because it will never find a \( C \) with which to “collaborate” (as in rule 4). Thus the initial choice of rule 11 leads us to block. (The argument would be the same if production 3 instead of production 2 had been employed.)

CASE 3. Apply production 2 to (1.4.3). (There is an analogous argument for production 3 by the symmetry of the rules.)

\[
\begin{align*}
 xABx & \Rightarrow x0ADx \quad \text{rules 6 and 7} \\
 & \Rightarrow x0AxDC \\
 & \Rightarrow x0AxBOC \quad \text{drop a 0} \\
 & \Rightarrow x0ABxC \quad \text{continue}
\end{align*}
\]

In this case, a string was found which is one longer than \( x \) and of the same form.

The cases we have studied exhaust all possible rewritings of \( xABx \), and it now follows that (1.4.2) is true.

Doing these proofs is important because it helps to prevent errors. In the example, note that rules 12 and 13 occur after the termination rules. The reason is that the grammar was designed first. When the proof was carried out, it became apparent that a possible set of rules had been made.

Writing a grammar for a problem is very much like composing a program (in a primitive programming language). The techniques for writing an understandable or structured program are even more useful here and should be employed.

**PROBLEM**

1. Give a phrase-structure grammar that generates the set

\[ L = \{ a^i b^j a^i b^j : i, j \geq 1 \}. \]

Explain clearly how the grammar is supposed to work, and also prove that it generates \( L \); i.e., every string in \( L(G) \) is in \( L \) and every string in \( L \) is in \( L(G) \).

## 1.5 OTHER FAMILIES OF GRAMMARS – THE CHOMSKY HIERARCHY

We shall now survey some of the other families of grammars which are known, and characterize some of them by automata. This will be a somewhat informal survey. In later chapters, these devices will be studied in great detail and the actual characterizations will be proved.

First, we consider some variations on phrase-structure grammars.

**Definition** A phrase-structure grammar \( G = (V, \Sigma, P, S) \) is said to be of type 0 if each rule is of the form

\[ \alpha \rightarrow \beta \]

where \( \alpha \in N^* \) and \( \beta \in V^* \).

We shall see that type 0 and phrase-structure grammars are equivalent. This is, in fact, a general theme that will run throughout the book. For a given family of languages, \( \mathcal{L} \), we will be interested in very powerful types of grammars that generate \( \mathcal{L} \). Moreover, we are very interested in the weakest family of grammars that also generates \( \mathcal{L} \). When we have some particular set \( L \) and wish to show it is in \( \mathcal{L} \), we use a powerful type of grammar. On the other, if we wish to show that some given set \( L' \) is not in \( \mathcal{L} \), then it is convenient to assume it is in \( \mathcal{L} \) and is generated by a very constrained type of grammar. It then becomes much easier to find a contradiction.

**Definition** A phrase-structure grammar \( G = (V, \Sigma, P, S) \) is said to be context-sensitive with erasing if each rule is of the form

\[ AA' \gamma \rightarrow a\beta \gamma \]

where \( A \in N \) and \( \alpha, \beta, \gamma \in V^* \).

**Examples** Consider the grammar \( G_1 \) given by the following rules:

\[
\begin{align*}
S & \rightarrow ABC \\
A & \rightarrow a \\
bA & \rightarrow b \\
C & \rightarrow c
\end{align*}
\]

This grammar is neither type 0 nor context-sensitive with erasing, but is phrase-structure. Of course, every type 0 grammar and context-sensitive with erasing grammar must be phrase-structure.

Now consider grammar \( G_2 \) shown below:

\[
\begin{align*}
S & \rightarrow AB \\
A & \rightarrow a \\
AB & \rightarrow BA \\
B & \rightarrow b
\end{align*}
\]

\( G_2 \) is type 0 but not context-sensitive with erasing. It is easy to modify \( G_1 \) to obtain a grammar \( G_3 \) which is context-sensitive with erasing but not type 0. This is left to the reader.

The next definition will lead to a different family of languages.
Definition A phrase-structure grammar \( G = (V, \Sigma, P, S) \) is context-sensitive if each rule is of the form

\[
\alpha A \gamma \rightarrow \alpha \beta \gamma
\]

where \( A \in N; \alpha, \gamma \in V^*; \beta \in V^* \); or

\[
S \rightarrow \Lambda
\]

If this rule occurs, then \( S \) does not appear in the right-hand side of any rule.

The motivation for the term context-sensitive comes from rules like

\[
\alpha A \gamma \rightarrow \alpha \beta \gamma
\]

The idea is that \( A \) is rewritten by \( \beta \) only if it occurs in the context of \( \alpha \) on the left and \( \gamma \) on the right.

The purpose of the restriction \( \beta \in V^* \) is to ensure that when \( A \) is rewritten, it has an effect. But this condition alone would rule out \( \Lambda \) from being a sentence. For certain technical reasons and for certain applications, we would like to include \( \Lambda \). This leads to condition (ii) in the definition. We must constrain the appearance of \( S \) in the rules; otherwise erasing could be simulated by a construction using rules \( \alpha A \gamma \rightarrow \alpha S \gamma \) and \( S \rightarrow \Lambda \), to yield

\[
\alpha A \gamma \rightarrow \alpha S \gamma \rightarrow \alpha \gamma
\]

Now we give a definition of context-free grammars, which will be one of the main topics of this book.

Definition A phrase-structure grammar \( G = (V, \Sigma, P, S) \) is a context-free grammar if each rule is of the form

\[
A \rightarrow \alpha
\]

where \( A \in N, \alpha \in V^* \).

The term “context free” means that \( A \) can be replaced by \( \alpha \) wherever it appears, no matter the context.

Some authors give an alternative definition in which \( \alpha \in V^* \), which prohibits \( \Lambda \) rules. The presence of \( \Lambda \) rules in our definition must be accounted for eventually; and indeed it will be shown that, for each language \( L \) generated by a context-free grammar with \( \Lambda \) rules, there exists a \( \Lambda \)-free, context-free grammar that generates \( L \rightarrow \{A\} \). In the meantime \( \Lambda \) rules will play an important role in proofs, since they frequently give rise to special cases that must be carefully checked. Because \( \Lambda \) rules do cause technical problems, their omission is a common suggestion of novices and a temptation to specialists. Systematic omission of \( \Lambda \) rules changes the class of theorems one can prove. We shall include them.

There is an alternative form of context-free grammar which is used in the specification of programming languages. This system is called the Backus normal form or Backus-Naur form; in either case it is commonly abbreviated BNF. The form uses four meta characters which are not in the working vocabulary. These are

\[
( ) \quad ::= \quad 1
\]

The idea is that strings (which do not contain the meta characters) are enclosed by (and) and \( ::= \) denote variables. The symbol \( ::= \) serves as a replacement operator just like \( \rightarrow \) and \( 1 \) is read “or”.

Example An ordinary context-free grammar for unsigned digits in a programming language might be as follows, where \( D \) stands for the class of digits and \( U \) stands for the class of unsigned integers:

\[
\begin{align*}
D & \rightarrow 0 \\
D & \rightarrow 1 \\
D & \rightarrow 2 \\
D & \rightarrow 3 \\
D & \rightarrow 4 \\
U & \rightarrow D
\end{align*}
\]

\[
\begin{align*}
D & \rightarrow 5 \\
D & \rightarrow 6 \\
D & \rightarrow 7 \\
D & \rightarrow 8 \\
D & \rightarrow 9 \\
U & \rightarrow UD
\end{align*}
\]

The example, when rewritten in BNF, becomes

\[
\text{(digit)} ::= 0\,|\,1\,|\,2\,|\,3\,|\,4\,|\,5\,|\,6\,|\,7\,|\,8\,|\,9
\]

\[
\text{(unsigned int)} ::= (\text{(digit})(\text{(unsigned int)})\,(\text{(digit)})
\]

In this book, we will adopt a blend of the two notations. We shall use \( \mid \) but not the other meta characters. This allows for the compact representation of context-free grammars.

There are a few special classes of context-free grammars which will now be singled out.

Definition A context-free grammar \( G = (V, \Sigma, P, S) \) is linear if each rule is of the form

\[
A \rightarrow uBv \quad \text{or} \quad A \rightarrow u
\]

where \( A, B \in N \) and \( u, v \in \Sigma^* \).

In a linear grammar, there is at most one variable on the right side of each rule.

In a linear grammar, if the one variable that appears is on the right end of the word, the grammar is called “right linear.” A left linear grammar is defined in analogous fashion.

Definition A context-free grammar \( G = (V, \Sigma, P, S) \) is right linear if each rule is of the form

\[
A \rightarrow uB \quad A, B \in N
\]

or

\[
A \rightarrow u \quad u \in \Sigma^*
\]

Examples \( S \rightarrow aSa|bSb|\Lambda \) is a linear context-free grammar for the even-length palindromes; that is,

\[
L(G) = \{ww^T \mid w \in \{a,b\}^*\}
\]

The grammar \( G' \) given by

\[
S \rightarrow aSa \mid a
\]
is a right linear grammar which generates

\[ L(G') = a^+ \]

Any of these families of grammars gives rise to a family of languages. The following definition makes the connection more precise.

**Definition** A language \( L \) is said to be of type \( X \) (e.g., phrase-structure, context-sensitive, etc.) if there exists a type-\( X \) grammar \( G \) such that \( L(G) = L \).

For any language, there will be infinitely many grammars generating it.

**Example** Let \( G_1 \) be the following grammar:

\[
\begin{align*}
S & \rightarrow aSa \\
S & \rightarrow aa \\
S & \rightarrow a
\end{align*}
\]

Clearly, \( L(G_1) = a^+ \).

This example shows that \( a^+ \) is a linear language, but we know still more. The previous example establishes that \( a^+ \) is a right linear language.

A great deal is known about these classes of languages and grammars. In the rest of this section, we shall give an outline. To summarize the material, we use Table 1.2. The undefined terms in the table will be explained subsequently.

**TABLE 1.2** The Chomsky Hierarchy

<table>
<thead>
<tr>
<th>Grammars</th>
<th>Languages</th>
<th>Automata</th>
</tr>
</thead>
<tbody>
<tr>
<td>Phrase-structure</td>
<td>Recursively enumerable sets</td>
<td>Nondeterministic or deterministic Turing machines</td>
</tr>
<tr>
<td>Type 0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context-sensitive</td>
<td>Context-sensitive</td>
<td>Nondeterministic linearly space-bounded Turing machines, or lba's, for short</td>
</tr>
<tr>
<td>with erasing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context-sensitive</td>
<td>Context-sensitive</td>
<td>Deterministic pushdown automata</td>
</tr>
<tr>
<td>Monotonic</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context-free</td>
<td>Context-free</td>
<td></td>
</tr>
<tr>
<td>( L(k) )</td>
<td>Deterministic context-free</td>
<td>Deterministic pushdown automata</td>
</tr>
<tr>
<td>Linear</td>
<td>Linear context-free</td>
<td>Two-tape nondeterministic finite automata of a special type or 1-turn pda's</td>
</tr>
<tr>
<td>Right linear</td>
<td>Regular sets</td>
<td>Nondeterministic or deterministic, one-way or two-way finite automata</td>
</tr>
<tr>
<td>Left linear</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.5 OTHER FAMILIES OF GRAMMARS – THE CHOMSKY HIERARCHY

We shall now try to give some intuitive meaning to the technical terms in this table. (This organization is sometimes called the Chomsky hierarchy.) The reader who has trouble understanding the table should not be distressed. The definitions will be given later in full detail, and the results will be carefully proven.

A deterministic automaton which is in a given state reading a given input always does the same thing (i.e., transfers to the same new state, writes the same output, etc.). On the other hand, a nondeterministic automaton may "choose" any one of a finite set of actions. A nondeterministic automaton is said to accept a string \( x \) if there exists some sequence of transitions which start from an initial configuration, end in a final configuration, and are "controlled by \( x \)."

A recursive set is a set for which there exists an algorithm for deciding set membership. That is, for any string \( x \), the algorithm will halt in a finite amount of time, either accepting or rejecting \( x \). A recursively enumerable set is a set for which there exists a procedure for recognizing the set. That is, given a string \( x \) in the set, the procedure will halt after a finite length of time, accepting \( x \). On the other hand, if \( x \) is not in the set, the procedure may either halt, rejecting \( x \), or never halt.

Recursive sets are a proper subset of the recursively enumerable sets.

A Turing machine is simply a finite-state control device with a finite but potentially unbounded read–write tape. The sets accepted by deterministic and nondeterministic Turing machines are the same.

A linear bounded automaton (lba) is a Turing machine with a bounded tape. It should be noted that if the automaton is limited to \( n \) squares of tape, by suitably enlarging its alphabet to \( k \)-tuples, it can increase its effective tape size to \( kn \) for any fixed \( k \). Hence the word linear.

While many properties of context-sensitive languages and deterministic context-sensitive languages (languages accepted by deterministic lba's) are known, it is an open question as to whether the two families of languages are equivalent.

A pushdown automaton (pda) has a finite-state control with an input tape and an auxiliary pushdown store (LIFO, or last-in, first-out, store). In a given state, reading a given input and the symbol from the top of its pushdown store, it will make a transition to a new state, writing \( n \geq 0 \) symbols on the top of its pushdown store \((n = 0\), corresponding to erasing the top symbol of the pushdown store).

\( L((k) \) grammars are grammars that may be parsed from left to right, with \( k \) symbols looked ahead, by working from the bottom to the top. This class of grammar is rich enough to contain the syntax of many programming languages but restrictive enough to be parseable in "linear" time.

The deterministic context-free languages are characterized by the automata which accept them. It will be shown that they are a proper subset of the family of context-free languages.

**PROBLEMS**

**Definition** A phrase-structure grammar \( G = (V, \Sigma, P, S) \) is called monotonic if each rule is of the form:

i) \( \alpha \rightarrow \beta \) where \( \alpha \in V^* \), \( \beta \in V^* \), and \( \text{lg}(\alpha) \leq \text{lg}(\beta) \);

ii) \( S \rightarrow \lambda \) and if this rule occurs, then \( S \) does not appear on the righthand side of any rule in \( P \).
1. Given a phrase-structure (context-sensitive with erasing) \{context-sensitive\} \{monotonic\} grammar, show that there is a phrase-structure (context-sensitive with erasing) \{context-sensitive\} \{monotonic\} grammar \(G' = (V', \Sigma, P', S')\) with \(L(G') = L(G)\) and for each rule \(\alpha \to \beta\) in \(P'\), we have \(\alpha \in N^*\).

2. Prove the following statement (if it is true; if not, give a counter example).

- **Fact** For each phrase-structure (context-sensitive with erasing) \{context-sensitive\} \{monotonic\} grammar \(G = (V, \Sigma, P, S)\), there is a phrase-structure (context-sensitive with erasing) \{context-sensitive\} \{monotonic\} grammar \(G' = (V', \Sigma, P', S')\) such that \(L(G') = L(G)\) and each rule in \(P'\) is of the form

\[
\alpha \to \beta
\]

with \(\alpha \in N^*, \beta \in N^*, \) or

\[
A \to a
\]

with \(A \in N\) and \(a \in \Sigma \cup \{\Lambda\}\).

In the next few problems, we shall show that the family of monotonic languages is coextensive with the family of context-sensitive languages.

**Definition** For any phrase-structure grammar \(G = (V, \Sigma, P, S)\), let the weight of \(G\) be \(\max \{\log |\beta| : \alpha \to \beta \in P\}\).

3. Prove the following fact.

- **Fact** For each monotonic grammar \(G = (V, \Sigma, P, S)\), there is a monotonic grammar \(G' = (V', \Sigma, P', S)\) of weight at most 2, so that \(L(G') = L(G)\).

4. Prove the following statement.

- **Fact** For each monotonic grammar \(G = (V, \Sigma, P, S)\), there is a context-sensitive grammar \(G' = (V', \Sigma, P', S)\) such that \(L(G') = L(G)\).

5. Consider the following "proof" of Problem 4.

Assume without loss of generality that the weight of \(G\) is 2.

- i) If a production \(\pi \in P\) has weight < 2, then \(\pi \in P'\).
- ii) If \(\pi = AB \to CD\) and \(C = A\) or \(B = D\), then \(\pi \in P'\).
- iii) If \(\pi = AB \to CD\) with \(C \neq A\) and \(D \neq B\), then the following productions are in \(P'\):

\[
AB \to (\pi, A)B \\
(\pi, A)B \to (\pi, A)D \\
(\pi, A)D \to CD
\]

where \((\pi, A)\) is a new variable.

Surely \(P'\) is a finite set of context-sensitive rules, and since

\[
AB \xrightarrow{\alpha} (\pi, A)B \xrightarrow{\alpha} (\pi, A)D \xrightarrow{\alpha} CD
\]

we have that \(L(G') = L(G)\).

**c)** If your answer to (a) was yes and to (b) was no, then what do you think is the point of this problem?

6. Give a context-sensitive or monotonic grammar that generates

\[
\{xx | x \in \{a, b\}^*\}
\]

Be sure to explain your grammar and prove that it works.

7. Show that the family of languages generated by context-sensitive grammars with erasing is identical to the phrase-structure languages.

### 1.6 CONTEXT-FREE GRAMMARS AND TREES

The systematic use of trees to illustrate context-free derivations is an important device that greatly sharpens our intuition. Since we use these concepts in formal proofs, it is necessary to have precise formal definitions of the concepts. It turns out that the usual graph-theoretic terminology is insufficient for our purposes. On the other hand, a completely formal treatment becomes tedious and obscure the very intuition we wish to develop. For these reasons, a semiformal approach is developed here, which can be completely formalized. We leave a number of alternative approaches for the exercises.

An example of a tree \(T\) is given in Fig. 1.1. It has one or more nodes \((x_0, x_1, \ldots, x_{10})\), one of which is the root \((x_0)\). We denote the relation of immediate descendency by \(\triangleright\) (thus, on Fig. 1.1(a), \(x_2 \triangleright x_3\) but not \(x_2 \triangleright x_{10}\)). The descendency relation is the reflexive and transitive closure \(\triangleright^*\) of \(\triangleright\) (now \(x_3 \triangleright^* x_{10}\)). If \(x \triangleright^* y\), then the path from \(x\) to \(y\) (or path to \(y\) if \(x\) is the root) is the sequence of all nodes between and including \(x\) and \(y\) (for example, \((x_0, x_2, x_4, x_8)\) is the path from \(x_0\) to \(x_8\)).

**FIG. 1.1**
in Fig. 1.1(a). If \( n \) is the number of nodes in one of the longest paths in \( T \), then the height of \( T \) is, by definition, \( m - 1 \) (the tree in Fig. 1.1(a) has height 4). A node \( x \) is a leaf in \( T \) if for no \( y \) in \( T \), \( x \not\rightarrow y \). Nodes that are not leaves are called internal.

We must pay attention to the left-to-right order of nodes. Let

\[
y_1, y_2, \ldots, y_m, \quad m \geq 1
\]

be a sequence of all the leaves in \( T \) without repetition. (There is no other restriction on this sequence except the informal assumption that sequence (1.6.1) represents the intuitive left-to-right orders of leaves.) Now we define a special relation \( \preceq \) between certain pairs of nodes in \( T \).

For any \( x, y \) in \( T \),

\[
x \preceq y
\]

if and only if

i) \( x \) and \( y \) are not on the same path, and

ii) for some leaves \( y_i, y_{i+1} \) (\( 1 \leq i < m \)) in (1.6.1) we have \( x \preceq y_i \) and \( y \preceq y_{i+1} \).

Thus, in particular, there is no leaf "between" \( x \) and \( y \). (Thus, for instance, we have \( x_5 \preceq x_8 \) but not \( x_5 \preceq x_6 \) in Fig. 1.1(a).) Note that relation \( \preceq \) is determined uniquely by (1.6.1). The left-to-right order is then the reflexive and transitive closure \( \preceq \) of \( \preceq \).

Note that we have \( \preceq \preceq \preceq = \emptyset \). On the other hand, one of the two relations \( \preceq \) or \( \preceq \) (or their inverses) holds between any two nodes of a tree.

Every node \( x \in T \) has a label \( \lambda(x) \) from a given finite set of labels — in our cases it is always the set \( V_\Lambda = V \cup \{ \Lambda \} \), where \( V \) is the alphabet of a given grammar. The corresponding function \( \lambda : T \rightarrow V_\Lambda ; \ x \mapsto \lambda(x) \) is the labeling (thus Fig. 1.1(a), 1.1(b)). Of special importance are the root label of \( T \), \( \lambda(T) \), and the frontier of \( T \), \( \lambda(T) \). The latter is defined as a concatenation of labels of leaves in (1.6.1):

\[
\lambda(T) = \lambda(y_1) \lambda(y_2) \cdots \lambda(y_m) \in V^*.
\]

Note that some leaves may be labeled by \( \Lambda \) and thus \( \lambda(T) \) may be shorter than sequence (1.6.1) (for instance, in Fig. 1.1, \( \lambda(T) = abaA \)).

For any internal node \( x \) in \( T \) the set \( \{ y \mid y \equiv x \} \) or \( x \preceq y \) defines an elementary subtree of \( T \) in an obvious way. A cross section of \( T \) is any maximal sequence of nodes in \( T \) (with respect to the partial ordering obtained from the prefix relation)

\[
\xi = (x_1, x_2, \ldots, x_n)
\]

such that \( x_1 \preceq x_2 \preceq \cdots \preceq x_n \) (\( n \geq 1 \)). Thus, in particular, sequence (1.6.1) is a cross section of \( T \).

Two trees \( T, T' \) are structurally isomorphic, \( T \cong T' \), if and only if there is a bijection \( T \rightarrow T' \), \( x \rightarrow x' \), called a structural isomorphism from \( T \) to \( T' \), such that \( x \preceq y \) if and only if \( x' \preceq y' \) (intuitively, \( T \) and \( T' \) are "identical except for the labeling"). Where there is no danger of confusion, we write \( T = T' \) when \( T \cong T' \), and the labeling is preserved (this is in agreement with the customary understanding that \( T \) and \( T' \) are "identical," but not with the algebraic definition of trees). Also we use the same symbols \( \tau, \preceq, \lambda \), even for different trees.

We now define certain trees and sets of trees related to grammars. Let \( G = (V, \Sigma, P, S) \) be a context-free grammar. We interpret the productions of \( G \) as trees (of height 1) in a natural way: The production \( A_i \rightarrow A_1A_2 \cdots A_n \) corresponds to the tree shown in Fig. 1.2 (or, formally, \( x_0 \preceq x_1 \preceq x_2 \preceq \cdots \preceq x_n \preceq x_m \), \( x_1 \preceq x_2 \preceq \cdots \preceq x_n \), and \( \lambda(x_i) = A_i, 0 \leq i \leq n \)). We make a convention that, in this correspondence, \( \lambda(x_i) = \Lambda \) if and only if \( i = n + 1 \) and \( A_0 \rightarrow \Lambda \) is in \( P \).

**Definition** Let \( G = (V, \Sigma, P, S) \) be a context-free grammar and let \( T \) be a tree with labeling \( \lambda : T \rightarrow V_\Lambda \). \( T \) is called a grammatical tree of \( G \) if and only if, for every elementary subtree \( T' \) of \( T \), there exists a production \( p \in P \) corresponding to \( T' \). Moreover, a derivation tree \( T \) is a grammatical tree such that \( \lambda(T) \subseteq \Sigma^* \).

Consequently, the leaves in a derivation tree are precisely the nodes labeled by letters in \( \Sigma \) (called the terminal nodes) or by \( \Lambda \) (the \( \Lambda \)-nodes). All other nodes are labeled by letters in \( N = V - \Sigma \) (and are accordingly called the nonterminal nodes). Note, that in particular, a trivial tree consisting of a single node, labeled by \( a \in \Sigma \) or by \( \Lambda \), is a derivation tree of \( G \).

![Fig. 1.2](image_url)

The tree in Fig. 1.1 may serve as an example of a grammatical tree for any grammar with productions

\[
\begin{align*}
S &\rightarrow aAB \mid bBa \\
A &\rightarrow SA \\
B &\rightarrow \Lambda
\end{align*}
\]

then the derivation

\[
S \rightarrow AB \rightarrow Ab \rightarrow ab
\]

is associated with the tree
Note that the derivation
\[ S \Rightarrow AB \Rightarrow aB \Rightarrow ab \]  
(1.6.3)
is also a derivation of \( ab \), and the same tree is associated with it. Thus there are many
different derivations associated with the same tree.

**Example** If \( G \) is the grammar
\[
\begin{align*}
S &\rightarrow aSa \\
S &\rightarrow bSb \\
S &\rightarrow \lambda
\end{align*}
\]
which may be compactly written
\[ S \rightarrow aSa | bSb | \lambda \]
then the derivation
\[ S \Rightarrow aSa \Rightarrow abSba \Rightarrow abaSaba \Rightarrow ababa \]
is represented by the tree in Fig. 1.3, where the derived string may be read off by
concatenating in left-to-right order the symbols on the frontier of the tree; i.e., “reading the tree leaves.”

Since many derivations may correspond to the same tree, we would like to
identify one of these as *canonical* so as to set up a one-to-one correspondence between
trees and derivations.

![Tree](image)

**FIG. 1.3**

**Definition** Let \( G = (V, \Sigma, P, S) \) be a context-free grammar and let \( \alpha, \beta \in V^* \).
If
\[ \alpha \Rightarrow^\ast \beta \]  
(1.6.4)
then there exist \( \alpha_1, \alpha_2, A, \theta \), so that \( \alpha = \alpha_1 A \alpha_2 \), \( \beta = \alpha_1 \theta \alpha_2 \), and \( A \theta \) is in \( P \). If
\( \alpha_2 \in \Sigma^* \), then (1.6.4) is called a *rightmost* (or *canonical*) generation. If \( \alpha_1 \in \Sigma^* \), then
(1.6.4) is a *leftmost* generation. The notation
\[ \alpha \Rightarrow^r \beta \quad \text{or} \quad \alpha \Rightarrow^L \beta \]
is used in the respective cases.

Note that (1.6.2) and (1.6.3) provide examples of rightmost and leftmost
derivations.

Now we can state the correspondence between trees and rightmost derivations.

**Theorem 1.6.1** Let \( G = (V, \Sigma, P, S) \) be a context-free grammar. There is a
one-to-one correspondence between rightmost (leftmost) derivations of a string
\( w \in \Sigma^* \) and the derivation trees of \( w \) with root labeled \( S \).

**Proof** The proof is obvious intuitively, the correspondence being the natural
one. A formal proof is omitted, since trees have been introduced only informally. \( \square \)

The condition in Theorem 1.6.1 that \( w \in \Sigma^* \) is necessary. To illustrate this
point, consider the grammar:
\[
\begin{align*}
S &\rightarrow ABC & B &\rightarrow b \\
A &\rightarrow a & C &\rightarrow c
\end{align*}
\]
The tree

![Tree](image)

has frontier \( AbC \). There is no rightmost or leftmost derivation that contains that
string.

**Definition** Let \( G = (V, \Sigma, P, S) \) be a context-free grammar; \( \alpha \in V^* \) is a
*canonical sentential form* if \( S \Rightarrow^* \alpha \).

Not every sentential form is canonical, as the following trivial example shows.
Let \( G \) be
\[
\begin{align*}
S &\rightarrow AA \\
A &\rightarrow a
\end{align*}
\]
then \( aA \in S(G) \) but is not canonical.

**Definition** A context-free grammar \( G = (V, \Sigma, P, S) \) is *ambiguous* if there
exists \( x \in L(G) \) such that \( x \) has at least two rightmost generations from \( S \). A grammar
that is not ambiguous is said to be *unambiguous*. Alternatively, \( G \) is unambiguous if,
for any two derivation trees \( T \) and \( T' \) (from \( S \)), \( \text{fr}(T) = \text{fr}(T') \) implies \( T = T' \).
Example. Let $G$ be $S \rightarrow SBSd$. Then $G$ is ambiguous, since the string $(ab)^2a$ has the two rightmost generations shown in Fig. 1.4.

![Fig. 1.4](image)

**Definition.** A context-free language $L$ is *unambiguous* if there exists an unambiguous context-free grammar $G$ such that $L = L(G)$.

It can easily be shown that the language generated by the grammar in the preceding example is $L(G) = \{(ab)^n a | n \geq 0\}$. Consider the grammar $G'$:

$$S \rightarrow Ta$$
$$T \rightarrow abT \mid \Lambda$$

Then we have

$$S \Rightarrow_R Ta$$
$$T \Rightarrow_R abT \Rightarrow_R (ab)^n$$
$$S \Rightarrow_R (ab)^n a, \quad n \geq 0$$

We have that $L(G) = L(G')$ and hence the language $L = \{(ab)^n a | n \geq 0\}$ is unambiguous.

This discussion raises a deep question. Can ambiguity always be disposed of by changing grammars? Assuming the worst, the following definition would be useful.

**Definition.** A context-free language is *inherently ambiguous* if $L$ is not unambiguous (i.e., there does not exist any unambiguous grammar $G$ such that $L = L(G)$).

In a later chapter, it will be shown that there exist inherently ambiguous context-free languages. The following is an example.

$$L = \{ab^{i+1}c^k | i = j \text{ or } j = k, \text{ where } i, j, k \geq 1\}$$

Returning to the relationship between trees and grammars we note that if $G = (V, \Sigma, P, S)$ is linear, then each rule in $G$ is of the form

$$A \rightarrow a_1 a_2 \cdots a_k B a_{k+1} \cdots a_l, \quad k \in N, \quad a_i \in \Sigma,$$

or

$$A \rightarrow a_1 a_2 \cdots a_k, \quad k \geq 0.$$  

Thus trees associated with a linear grammar have a special form. At each level, at most one node may have descendants. An example of such a tree is shown in Fig. 1.5.

![Fig. 1.5](image)

**Example.** Let $G$ be $S \rightarrow aSBa$, which is a "minimal linear" grammar in that it has only one variable. Then $L(G) = \{a^i b^j | i \geq 0\}$.

![Fig. 1.6](image)

Then a derivation tree is of the form shown in Fig. 1.6.

If the grammar is right linear, then the generation trees have an even more restricted form. At each level at most one node may have descendants, and this must be the rightmost node. Thus, Fig. 1.7 would be an example of a derivation tree for a right linear grammar.

![Fig. 1.7](image)