6
Context-Free Grammars

6.1 INTRODUCTION

In this chapter, we present a method of defining sets of sequences based on the concept of a *context-free grammar.* Unlike the previous methods given for describing such sets, the context-free methods are powerful enough to describe almost all of the so-called syntactic features of programming languages. Indeed context-free grammars are often used in language manuals to describe a programming language.

Using a particular set-description method to describe a programming language can be a valuable step in the design of a compiler for the language if there are systematic methods of converting the set description into a program that processes the set. In the case of context-free grammars, we will develop in later chapters methods of converting certain context-free grammars into pushdown machines that recognize and even translate the sets defined by the grammars. Before these methods can be explained, we must understand how sets are defined by context-free grammars (Chapter 6) and how translations can be defined in terms of these grammars (Chapter 7).

6.2 FORMAL LANGUAGES AND FORMAL GRAMMARS

Many of the concepts in this chapter are analogous to concepts used in discussing natural languages such as English. Some of the definitions introduced in this chapter were in fact originally developed with natural languages in mind.

The most basic concept is that of a *language*. For theoretical purposes the word "language" is synonymous with the term "set of sequences". Thus we might think of the FORTRAN IV language as being the set of sequences specified by some official set of specifications. Most interesting languages, like FORTRAN IV, consist of an infinite set of sequences.

To distinguish between the use of the word "language" as meaning a precisely specified set of sequences and the use of the word "language" as used in every day nontechnical conversation, one sometimes refers to a set of sequences as a *formal language*.

In order to attack problems of languages and language processing in a mathematical way, we must confine ourselves to sets of sequences that can be specified in some precise way. There are many ways of precisely specifying such sets. One way, for example, is to define a language as the set accepted by some kind of a sequence recognizer such as a finite-state or pushdown machine. Another approach is the use of methods that can be thought of as grammatical.

The term "formal grammar" is applied to any formal-language specification based on "grammatical rules" whereby sequences can be generated or analyzed in a way analogous to the use of grammars in the study of natural languages. In this chapter, we are concerned with a specific kind of formal grammar, called a context-free grammar.

6.3 FORMAL GRAMMARS—AN EXAMPLE

In this section, we give a specific formal grammar that looks something like an English grammar and which defines a formal language consisting of four English sentences.

The formal grammar uses certain entities that are like parts of speech.

⟨sentence⟩
⟨subject⟩
⟨verb⟩
⟨object⟩
⟨noun⟩
⟨article⟩

We enclose these in pointed brackets to distinguish them from the actual dictionary words that make up the sequences in the language. In our example, the dictionary contains the following five words or symbols.

THE
BOY
GIRL
SEES
. (period)

The grammar has certain rules that tell how sequences in the language can be made up from these symbols. One such rule is:

1. ⟨sentence⟩ → ⟨subject⟩ ⟨verb⟩ ⟨object⟩.

The interpretation of this rule is: "An example of a sentence is a subject followed by a verb followed by an object followed by a period." The grammar could well have other rules giving other examples of sentences. This particular grammar does not have such rules.

The remaining rules are:

2. ⟨subject⟩ → ⟨article⟩ ⟨noun⟩
3. ⟨object⟩ → ⟨article⟩ ⟨noun⟩
4. ⟨verb⟩ → SEES
5. ⟨article⟩ → THE
6. ⟨noun⟩ → BOY
7. ⟨noun⟩ → GIRL

We now use this grammar to generate or derive a sentence in the language. By rule 1, an example of a sentence in the language is

⟨subject⟩ ⟨verb⟩ ⟨object⟩ .

Since, by rule 2, an example of a subject is

⟨article⟩ ⟨noun⟩

these can be substituted for ⟨subject⟩ to obtain the following example of a sentence

⟨article⟩ ⟨noun⟩ ⟨verb⟩ ⟨object⟩ .

Similarly we can use rule 3 to substitute for ⟨object⟩ to obtain

⟨article⟩ ⟨noun⟩ ⟨verb⟩ ⟨article⟩ ⟨noun⟩ .

Now we can use rule 5 to substitute for ⟨article⟩ in both places to obtain

THE ⟨noun⟩ ⟨verb⟩ THE ⟨noun⟩.

Finally we can use rule 4 to substitute for ⟨verb⟩ and rules 6 and 7 to substitute for the two occurrences of ⟨noun⟩ to obtain a complete sentence:

THE BOY SEES THE GIRL.

The derivation can be shown pictorially as a tree (see Fig. 6.1). The tree shows which rules were applied to the various intermediate entities but conceals the order in which the rules were applied. Thus the tree demonstrates that the resulting sequence is independent of the order in which substitutions are made for intermediate entities. The tree is sometimes said to display the "syntactic structure" of the sentence.

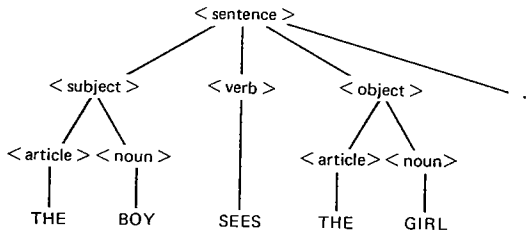


Figure 6.1

The idea of a derivation suggests other interpretations of a rule such as

⟨subject⟩ → ⟨article⟩ ⟨noun⟩

Instead of saying "an example of a ⟨subject⟩ is an ⟨article⟩ followed by a ⟨noun⟩" we might say ⟨subject⟩ "generates" (or "derives" or "can be replaced by") ⟨article⟩ ⟨noun⟩.

This grammar can also be used to derive three other sentences, namely:

THE GIRL SEES THE BOY.
THE GIRL SEES THE GIRL.
THE BOY SEES THE BOY.

These three sentences together with the one sentence derived previously are the only sentences that can be derived from the grammar. We say that the set consisting of these four sequences is the language specified by (or generated by, or derived from) the grammar.

6.4 CONTEXT-FREE GRAMMARS

The grammar given in the last section is a simple example of the class of grammars that is our main concern: the context-free grammars. In this section we define this type of grammar and introduce the conventional notation for its description.

The entities such as ⟨sentence⟩, ⟨verb⟩, etc., that are analogous to parts of speech are called *nonterminal symbols* or *nonterminals.* A context-free grammar can be specified to have any finite number of nonterminals. For programming languages, the nonterminals may be such entities as ⟨statement⟩, ⟨arithmetic expression⟩, etc.

The entities such as THE, BOY, etc., which are analogous to dictionary words are called *terminal symbols* or *terminals.* A context-free grammar can be specified to have any finite number of terminals. For programming languages, the terminals are the actual words and symbols in the language, such as DO, +, etc.

The rules of the grammar are sometimes called *productions* and are of the general form:

Any single nonterminal → *any finite sequence of terminals and nonterminals*

The sequence on the right of the arrow can be the null sequence. An example of such a production is

$$\langle A \rangle \rightarrow \epsilon$$

We sometimes refer to a production with a null righthand side as an epsilon production. A context-free grammar can have any finite set of productions. An example of a production in a programming language is

⟨statement⟩ → IF ⟨Boolean expression⟩ THEN ⟨statement⟩

One of the nonterminals is specified to be the *starting nonterminal* or *starting symbol* from which derivations of sequences in the language must start. For a natural language this nonterminal might be ⟨sentence⟩; for a programming language it might be ⟨program⟩. We frequently use $\langle S \rangle$ for the starting symbol.

Summarizing the above, a context-free grammar is specified by

a) a finite set of nonterminals;

b) a finite set of terminals which is disjoint from the set of nonterminals;

c) a finite set of productions of the form

$$\langle A \rangle \rightarrow \alpha$$

where $\langle A \rangle$ is a nonterminal and α is a sequence (possibly the null sequence) of terminals and nonterminals—nonterminal $\langle A \rangle$ is

and we wish to apply production 5 to the nonterminal $\langle B \rangle$ pointed to by the arrow the result of the corresponding substitution is

$$a \langle A \rangle b \langle B \rangle c$$

We express this substitution by writing

$$\begin{array}{c} a \langle A \rangle \langle B \rangle c \Rightarrow a \langle A \rangle b \langle B \rangle c \\ \uparrow \\ 5 \end{array}$$

In this notation, the vertical arrow points to the nonterminal to be replaced, the number under the arrow indicates the production to be applied, and the symbol $\Rightarrow$ is used to separate the *before* string from the *after* string.

Sometimes this notation is used without the vertical arrow to indicate just the before and after strings. For instance, one might write

$$a \langle A \rangle \langle B \rangle c \Rightarrow a \langle A \rangle b \langle B \rangle c$$

as a shorthand for "string $a \langle A \rangle b \langle B \rangle c$ can be obtained from string $a \langle A \rangle \langle B \rangle c$ as the result of one substitution."

However this abbreviated notation is not always sufficient to describe the substitution involved. For instance the same after string can also be obtained by the substitution

$$\begin{array}{c} a \langle A \rangle \langle B \rangle c \Rightarrow a \langle A \rangle b \langle B \rangle c \\ \uparrow \\ 4 \end{array}$$

A series of substitutions is called a *derivation.* For instance the string *acabac* can be obtained from the starting nonterminal by the following derivation.

$$\begin{array}{c} \langle S \rangle \Rightarrow a \langle A \rangle \langle B \rangle c \Rightarrow a \langle A \rangle b \langle B \rangle c \Rightarrow a c \langle S \rangle \langle B \rangle b \langle B \rangle c \\ \uparrow \quad\quad \uparrow \quad\quad\quad\quad \uparrow \quad\quad\quad\quad\quad\quad \uparrow \\ 1 \quad\quad 4 \quad\quad\quad\quad 3 \quad\quad\quad\quad\quad\quad 6 \\ \Rightarrow a c \langle S \rangle a b \langle B \rangle c \Rightarrow a c a b \langle B \rangle c \Rightarrow a c a b a c \\ \uparrow \quad\quad\quad\quad\quad\quad \uparrow \\ 2 \quad\quad\quad\quad\quad\quad 6 \end{array}$$

Each of the strings of terminals and nonterminals that appears in a derivation is called an *intermediate string* of the derivation. Thus, in the above derivation, there are seven intermediate strings, counting the initial and final strings. (In the literature an intermediate string derived from the starting symbol is sometimes called a "sentential form.")

We frequently use the word "derivation" without specifying the initial string of the derivation. In such cases, the initial string is assumed to be the starting nonterminal. If a different initial string is meant, it is explicitly indicated.

To assert the existence of a derivation from one string to another string we use the symbol

$$\stackrel{*}{\Rightarrow}$$

For instance we write

$$\langle S \rangle \stackrel{*}{\Rightarrow} a c a b a c$$

to mean that "string *acabac* can be obtained from string $\langle S \rangle$ by a derivation," or equivalently, that "from string $\langle S \rangle$, the string *acabac* can be derived."

For purposes of uniformity, we allow the number of substitutions in a derivation to be zero. For instance

$$b \langle A \rangle c$$

can be regarded as a derivation of length zero where $b \langle A \rangle c$ is both the initial and final string. Therefore, for any string α, we can write

$$\alpha \stackrel{*}{\Rightarrow} \alpha$$

since α can be obtained from itself by a length zero sequence of substitutions.

The "$*$" is analogous to the Kleene star in that it suggests zero or more uses of the relation $\Rightarrow$. If one wishes to exclude the rather trivial zero length derivation, one replaces the $*$ by a $+$ and writes

$$\langle S \rangle \stackrel{+}{\Rightarrow} a c a b a c$$

to mean that "string *acabac* can be obtained from string $\langle S \rangle$ by a derivation of length greater than zero."

We define the language specified by a grammar to be the set of terminal strings that can be derived from the starting symbol of the grammar. Sometimes, one says that the language is "defined by," "generated from," or "derived from" the grammar. Any language that can be specified by a context-free grammar is called a *context-free language.*

For the grammar given above, *acabac* can be derived from the starting symbol of the grammar, and therefore *acabac* is in the language specified by the grammar. On the other hand, inspection of the grammar reveals that the string *bb* cannot possibly be derived from $\langle S \rangle$; therefore, *bb* is not in the language specified by the grammar.

6.6 TREES

We have defined the context-free language specified by a grammar to be the set of terminal strings that can be derived from the starting symbol. Given a derivation of a string in a context-free language, it is possible to construct an

associated *derivation tree.* This construction is carried out by interpreting the substitutions in the derivation as tree-building operations. This tree construction is easier to demonstrate than to describe, so we illustrate with an example.

The previous section contains a grammar and a derivation of the terminal string *acabac* according to the grammar. This derivation can be used to construct an associated derivation tree as shown in Fig. 6.3 where each step in the derivation is shown together with an associated tree. In Fig. 6.3(a), the starting symbol $\langle S\rangle$ is associated with a one node tree $\langle S\rangle$. When production 1 is applied to symbol $\langle S\rangle$, that symbol is replaced by the righthand side of production 1, namely $a\,\langle A\rangle\,\langle B\rangle\,c$. The corresponding tree-building action is to add the symbols a, $\langle A\rangle$, $\langle B\rangle$, and c as tree nodes attached below the symbol being replaced. The result is Fig. 6.3(b). The derivation and associated tree construction continue until Fig. 6.3(g) is achieved. The final tree is then said to be a derivation tree associated with the terminal string *acabac.*

The derivation tree shows which productions were applied during the derivation and to which occurrence of a nonterminal they were applied. However, the tree does not provide information about the order in which the productions were applied except for the obvious implication that the productions for each node must have been used before the productions for the nonterminal nodes below it. Thus in the tree of Fig. 6.3(g), one can see that production 4 was used before production 3 but one cannot tell the order in which production 2 and the two applications of production 6 were used.

Because the associated tree may conceal the order of substitution, there may be many derivations associated with the same derivation tree. For example, the tree in Fig. 6.3 (g) is also associated with the derivation

$$\underset{1}{\underset{\uparrow}{\langle S\rangle}} \Rightarrow a\,\underset{4}{\underset{\uparrow}{\langle A\rangle}}\,\langle B\rangle\,c \Rightarrow a\,\underset{3}{\underset{\uparrow}{\langle A\rangle}}\,b\,\langle B\rangle\,c \Rightarrow a\,c\,\underset{2}{\underset{\uparrow}{\langle S\rangle}}\,\langle B\rangle\,b\,\langle B\rangle\,c$$

$$\Rightarrow a\,c\,\underset{6}{\underset{\uparrow}{\langle B\rangle}}\,b\,\langle B\rangle\,c \Rightarrow a\,c\,a\,b\,\underset{6}{\underset{\uparrow}{\langle B\rangle}}\,c \Rightarrow a\,c\,a\,b\,a\,c$$

This particular derivation is called a *leftmost derivation* because at each step the substitution was performed on the leftmost nonterminal symbol. For each tree, there is a unique leftmost derivation since the convention of selecting the leftmost nonterminal determines a unique place to perform a substitution and the tree determines a unique production to use for the substitution.

For each tree there is also a unique *rightmost derivation* obtained by the convention that a substitution is always made at the rightmost nonterminal.

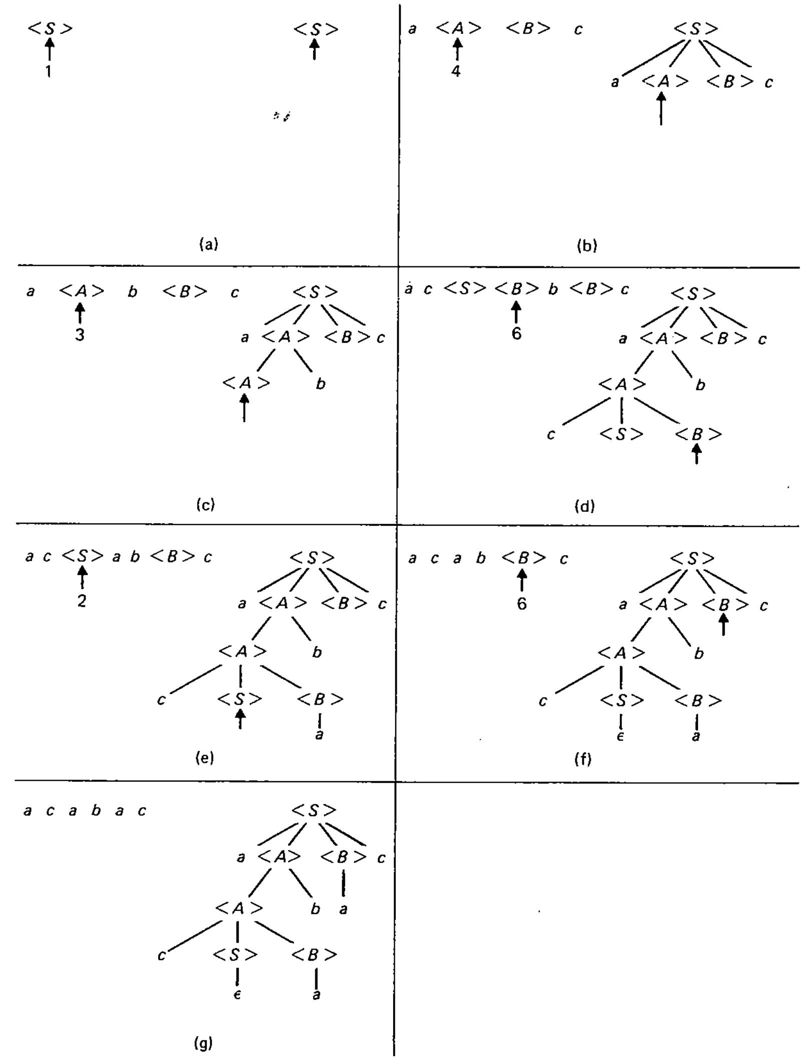


Figure 6.3

The rightmost derivation for the tree of Fig. 6.3(g) is

$$\underset{\uparrow}{\langle S\rangle} \Rightarrow a \underset{\uparrow}{\langle A\rangle} \underset{\uparrow}{\langle B\rangle} c \Rightarrow a \underset{\uparrow}{\langle A\rangle} a c \Rightarrow a \underset{\uparrow}{\langle A\rangle} b a c$$
$$1 \qquad 6 \qquad 4 \qquad 3$$
$$\Rightarrow a c \underset{\uparrow}{\langle S\rangle} \langle B\rangle b a c \Rightarrow a c \underset{\uparrow}{\langle S\rangle} a b a c \Rightarrow a c a b a c$$
$$6 \qquad 2$$

Many language-processing methods deal exclusively with leftmost or rightmost derivations since these derivations are very suitable for systematic treatment. In discussing these cases, we write

$$\alpha \Rightarrow_L \beta$$

to mean that "string β can be obtained from string α as the result of one leftmost substitution," and we write

$$\alpha \Rightarrow_R \beta$$

to mean that "string β can be obtained from string α as the result of one rightmost substitution." In these cases, the substitution can be deduced from the two intermediate strings. It is clear for example that

$$a \langle A\rangle \langle B\rangle c \Rightarrow_L a \langle A\rangle b \langle B\rangle c$$

can only result from substitution

$$\underset{\uparrow}{a \langle A\rangle} \langle B\rangle c \Rightarrow a \langle A\rangle b \langle B\rangle c$$
$$4$$

and that

$$a \langle A\rangle \langle B\rangle c \Rightarrow_R a \langle A\rangle b \langle B\rangle c$$

can only result from substitution

$$a \underset{\uparrow}{\langle A\rangle} \langle B\rangle c \Rightarrow a \langle A\rangle b \langle B\rangle c$$
$$5$$

We also write

$$\alpha \overset{*}{\Rightarrow}_L \beta$$

if there is a leftmost derivation of β from α and write

$$\alpha \overset{*}{\Rightarrow}_R \beta$$

if there is a rightmost derivation.

A string in a language may have more than one tree because the string may have different derivations which yield distinct trees. This is in fact the case for our example *acabac*. In addition to the tree of Fig. 6.3(g) which has been redrawn in Fig. 6.4(a), it also has the tree of Fig. 6.4(b) obtained from the following derivation:

$$\underset{\uparrow}{\langle S\rangle} \Rightarrow a \underset{\uparrow}{\langle A\rangle} \underset{\uparrow}{\langle B\rangle} c \Rightarrow a \underset{\uparrow}{\langle A\rangle} b \underset{\uparrow}{\langle B\rangle} c \Rightarrow a c \underset{\uparrow}{\langle S\rangle} \underset{\uparrow}{\langle B\rangle} b \underset{\uparrow}{\langle B\rangle} c$$
$$1 \qquad 5 \qquad 3 \qquad 6$$
$$\Rightarrow a c \underset{\uparrow}{\langle S\rangle} a b \underset{\uparrow}{\langle B\rangle} c \Rightarrow a c a b \underset{\uparrow}{\langle B\rangle} c \Rightarrow a c a b a c$$
$$2 \qquad 6$$

In this derivation, even the intermediate strings are the same as in our original derivation. However, this derivation uses production 5 when the original used production 6 and the associated trees are clearly different.

In cases like this where one string can have several trees, the grammar is said to be *ambiguous*.

The discussion up to this point can be summarized as follows:

1. Corresponding to each string derived from a given context-free grammar, there are one or more derivation trees.
2. Corresponding to each derivation tree, there are one or more derivations.

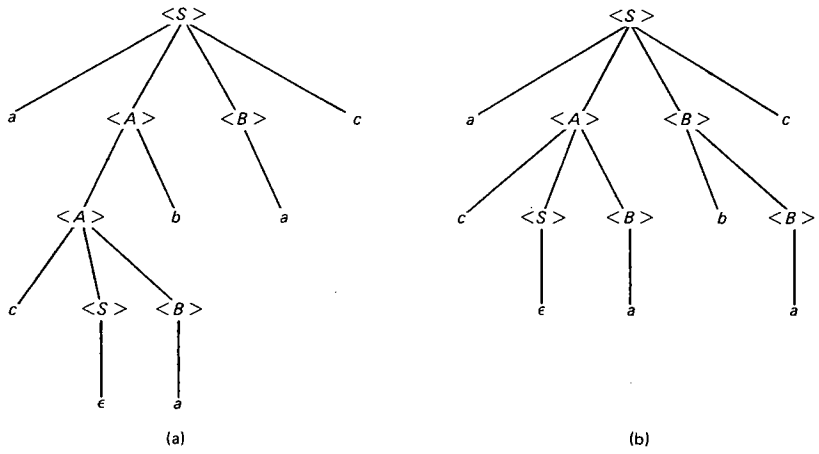


Figure 6.4

3. Corresponding to each derivation tree, there is a unique rightmost derivation and a unique leftmost derivation.
4. If each string derived from a given context-free grammar has only one derivation tree, the grammar is said to be unambiguous; otherwise it is said to be ambiguous.

6.7 A GRAMMAR FOR MINI-BASIC CONSTANTS

It is frequently necessary to find a context-free grammar for some language that is only specified in some informal way. We demonstrate how this might be done by finding a grammar for MINI-BASIC constants.

We give the nonterminals names corresponding to the strings that can be derived from them. The starting nonterminal is ⟨constant⟩.

The MINI-BASIC manual describes two forms of constants: those with and without the E, representing power of 10. Those without the E will be derived from the production

1. ⟨constant⟩ → ⟨decimal number⟩

where ⟨decimal number⟩ generates a sequence of digits with an optional decimal point. Constants with the E will be derived from the production

2. ⟨constant⟩ → ⟨decimal number⟩ E ⟨integer⟩

where ⟨decimal number⟩ is the same as before and ⟨integer⟩ generates a sequence of digits preceded by an optional sign.

To complete the grammar we need productions for ⟨decimal number⟩ and ⟨integer⟩. We begin with ⟨integer⟩.

3. ⟨integer⟩ → + ⟨unsigned integer⟩
4. ⟨integer⟩ → − ⟨unsigned integer⟩
5. ⟨integer⟩ → ⟨unsigned integer⟩

where ⟨unsigned integer⟩ generates a sequence of digits.

The productions for ⟨unsigned integer⟩ are

6. ⟨unsigned integer⟩ → *d* ⟨unsigned integer⟩
7. ⟨unsigned integer⟩ → *d*

where *d* represents an arbitrary digit. It is easy to see that these two productions generate the required sequence of digits.

The productions for ⟨decimal number⟩ can be expressed in terms of the nonterminal ⟨unsigned integer⟩.

8. ⟨decimal number⟩ → ⟨unsigned integer⟩
9. ⟨decimal number⟩ → ⟨unsigned integer⟩ .

10. ⟨decimal number⟩ → . ⟨unsigned integer⟩
11. ⟨decimal number⟩ → ⟨unsigned integer⟩ . ⟨unsigned integer⟩

Production 8 is for numbers without a decimal point; production 9 is for numbers with the decimal point after the last digit; production 10 is for numbers with the decimal point at the beginning; and production 11 is for numbers with digits on each side of the decimal point. The righthand sides of these productions only use the nonterminal ⟨unsigned integer⟩ whose productions have already been given. All nonterminals are thus accounted for and the grammar is complete.

As an example, the constant

3.1 E−21

can be generated from the following leftmost derivation

⟨constant⟩ ⇒ ⟨decimal number⟩ E ⟨integer⟩ ⇒

⟨unsigned integer⟩ . ⟨unsigned integer⟩ E ⟨integer⟩ ⇒

3. ⟨unsigned integer⟩ E ⟨integer⟩ ⇒

3.1 E ⟨integer⟩ ⇒

3.1 E − ⟨unsigned integer⟩ ⇒

3.1 E − 2 ⟨unsigned integer⟩ ⇒

3.1 E − 21

The corresponding derivation tree is shown in Fig. 6.5.

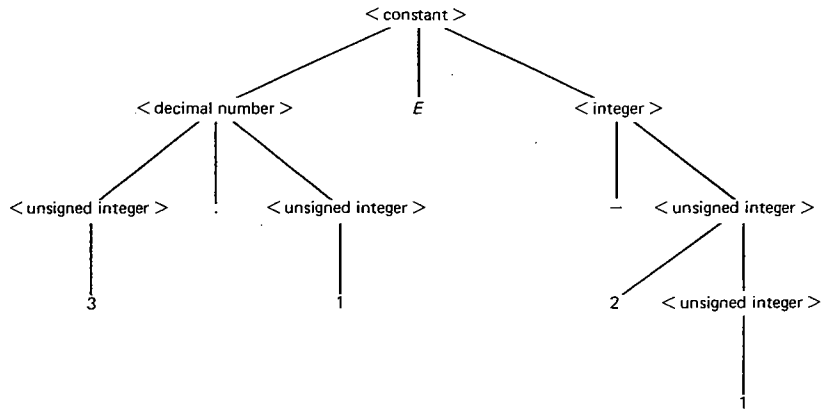


Figure 6.5

6.8 A GRAMMAR FOR S-EXPRESSIONS IN LISP

As another example we find a grammar for S-expressions in LISP. In the LISP manual, S-expressions are defined recursively in terms of other S-expressions and atoms which are analogous to identifiers:

> "An S-expression is either an atom or a left parenthesis followed by an S-expression followed by a dot followed by an S-expression followed by a right parenthesis."

Using ATOM as a terminal symbol, a grammar for S-expressions is:

$$\langle S\rangle \rightarrow \text{ATOM}$$
$$\langle S\rangle \rightarrow (\langle S\rangle \cdot \langle S\rangle)$$

This grammar demonstrates two of the nice features of context-free languages–that they can handle nested parentheses, and that they can define sets of strings in a recursive fashion.

A sample derivation in this grammar is

$$\langle S\rangle \Rightarrow (\langle S\rangle \cdot \langle S\rangle) \Rightarrow ((\langle S\rangle \cdot \langle S\rangle) \cdot \langle S\rangle) \Rightarrow ((\text{ATOM} \cdot \langle S\rangle) \cdot \langle S\rangle) \Rightarrow$$
$$((\text{ATOM} \cdot (\langle S\rangle \cdot \langle S\rangle)) \cdot \langle S\rangle) \Rightarrow$$
$$((\text{ATOM} \cdot (\text{ATOM} \cdot \langle S\rangle)) \cdot \langle S\rangle) \Rightarrow$$
$$((\text{ATOM} \cdot (\text{ATOM} \cdot \text{ATOM})) \cdot \langle S\rangle) \Rightarrow$$
$$((\text{ATOM} \cdot (\text{ATOM} \cdot \text{ATOM})) \cdot \text{ATOM})$$

A corresponding derivation tree is shown in Fig. 6.6

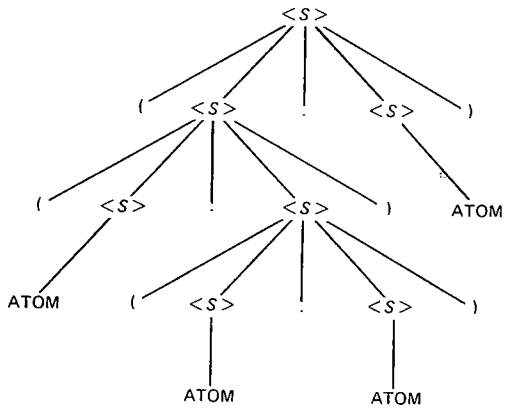


Figure 6.6

6.9 A GRAMMAR FOR ARITHMETIC EXPRESSIONS

In addition to specifying which sequences are in a language, a grammar can sometimes be used to display a certain "structure" of the language. For example, consider the grammar

1. $\langle E\rangle \rightarrow \langle E\rangle + \langle T\rangle$
2. $\langle E\rangle \rightarrow \langle T\rangle$
3. $\langle T\rangle \rightarrow \langle T\rangle * \langle F\rangle$
4. $\langle T\rangle \rightarrow \langle F\rangle$
5. $\langle F\rangle \rightarrow \langle F\rangle \uparrow \langle P\rangle$
6. $\langle F\rangle \rightarrow \langle P\rangle$
7. $\langle P\rangle \rightarrow (\langle E\rangle)$
8. $\langle P\rangle \rightarrow I$

where $\langle E\rangle$ is the starting symbol and I represents any integer. This grammar, which is based on the grammar for arithmetic expressions given in the ALGOL 60 report, generates all ALGOL 60 arithmetic expressions using integers and the binary operators $+$, $*$, and $\uparrow$. Figure 6.7 shows a derivation tree based on this grammar for the expression

$$1 + 2 * 3 + (5 + 6) * 7$$

where the integers are considered instances of the terminal symbol I.

A specification of a programming language must indicate not only which sequences are in the language, but also how they are to be executed. Therefore, the specification of arithmetic expressions includes rules for determining the operands of each operator in the expression. For instance, the operands of the first $+$ in the above expression are the integer 1 and the result of evaluating the subexpression $2 * 3$. Observe that in Fig. 6.7, the subexpression $1 + 2 * 3$ is generated from a single node labeled with an $\langle E\rangle$ and that this node has three descendants: an $\langle E\rangle$ that generates the operand 1, the $+$, and a $\langle T\rangle$ that generates the subexpression $2 * 3$. Similarly each of the other operators in the tree descends from a node whose other two nodes generate the operands of the operator. Thus the tree reflects which parts of the input sequence form subexpressions that are to be evaluated and what the operands of each operator are. In this sense, the tree displays the "structure" of the expression.

6.10 DIFFERENT GRAMMARS FOR THE SAME LANGUAGE

For any context-free language, there are an infinite number of grammars that generate that language. Although most of these grammars are need-

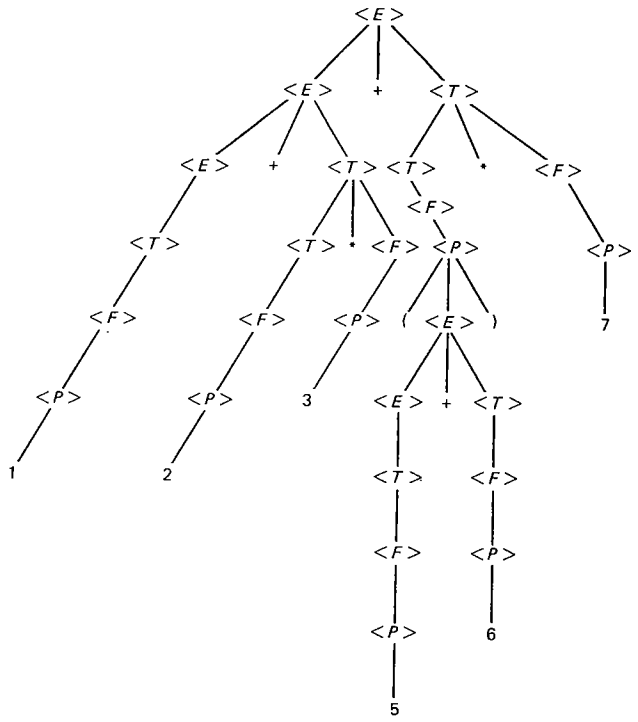


Figure 6.7

lessly complex, it frequently happens that there are several different grammars which might be considered useful for describing some particular language.

For example, the following grammar, with starting symbol $\langle E \rangle$, generates the same arithmetic expression language as the grammar of Section 6.9.

1. $\langle E \rangle \rightarrow \langle T \rangle \langle E\text{-list} \rangle$
2. $\langle E\text{-list} \rangle \rightarrow + \langle T \rangle \langle E\text{-list} \rangle$
3. $\langle E\text{-list} \rangle \rightarrow \epsilon$
4. $\langle T \rangle \rightarrow \langle F \rangle \langle T\text{-list} \rangle$
5. $\langle T\text{-list} \rangle \rightarrow * \langle F \rangle \langle T\text{-list} \rangle$
6. $\langle T\text{-list} \rangle \rightarrow \epsilon$
7. $\langle F \rangle \rightarrow \langle P \rangle \langle F\text{-list} \rangle$

8. $\langle F\text{-list} \rangle \rightarrow \uparrow \langle P \rangle \langle F\text{-list} \rangle$
9. $\langle F\text{-list} \rangle \rightarrow \epsilon$
10. $\langle P \rangle \rightarrow (\langle E \rangle)$
11. $\langle P \rangle \rightarrow \mathrm{I}$

In fact, nonterminals $\langle E \rangle$, $\langle T \rangle$, $\langle F \rangle$, and $\langle P \rangle$ generate the same sequences in both grammars.

First observe that both grammars have the same productions for nonterminal $\langle P \rangle$. Now consider nonterminal $\langle F \rangle$. Both grammars embody the concept that an $\langle F \rangle$ is a $\langle P \rangle$ with zero or more $\uparrow\langle P \rangle$'s appended to it. In the grammar of Section 6.9, n applications of production 5 followed by an application of production 6 generate a list of $(n + 1)$ $\langle P \rangle$'s separated by $\uparrow$'s. For example

$$\langle F \rangle \Rightarrow \langle F \rangle \uparrow \langle P \rangle \Rightarrow \langle F \rangle \uparrow \langle P \rangle \uparrow \langle P \rangle \Rightarrow \langle P \rangle \uparrow \langle P \rangle \uparrow \langle P \rangle$$

In the new grammar the same string is generated by an application of production 7 followed by n applications of production 8 followed by an application of production 9. For example

$$\langle F \rangle \Rightarrow \langle P \rangle \langle F\text{-list} \rangle \Rightarrow \langle P \rangle \uparrow \langle P \rangle \langle F\text{-list} \rangle \Rightarrow \langle P \rangle \uparrow \langle P \rangle \uparrow \langle P \rangle \langle F\text{-list} \rangle \Rightarrow \langle P \rangle \uparrow \langle P \rangle \uparrow \langle P \rangle$$

The nonterminal $\langle F\text{-list} \rangle$ is introduced to generate the list of zero or more $\uparrow \langle P \rangle$'s. Production 8 generates one $\uparrow \langle P \rangle$ followed by $\langle F\text{-list} \rangle$ to generate more. Production 9 generates a list of zero items.

The "structure" of an expression is reflected in the grammar. The right operand of each $\uparrow$ is generated from the $\langle P \rangle$ immediately following the $\uparrow$ in production 8. The left operand of each $\uparrow$ is generated from the string of $\langle P \rangle$'s and $\uparrow$'s preceding the occurrence of $\langle F\text{-list} \rangle$ to which production 8 is applied.

Next consider the nonterminal $\langle T \rangle$. Both sets of productions embody the concept that a $\langle T \rangle$ is an $\langle F \rangle$ followed by a list of zero or more $* \langle F \rangle$'s. The nonterminal $\langle T\text{-list} \rangle$ is introduced to generate the list of zero or more items. Productions 4, 5, and 6 provide for the first item, the list continuation, and the list termination.

Nonterminal $\langle E \rangle$ has a similar interpretation arising from productions 1, 2, and 3.

6.11 REGULAR SETS AS CONTEXT-FREE LANGUAGES

Two classes of languages have been introduced in this book: regular sets and context-free languages. In this section we show that context-free languages

are the more powerful concept. In particular we demonstrate the following fact:

Any regular set can be described by a context-free grammar.

This is, of course, another way of saying that regular sets are also context-free languages.

Suppose that one is given a finite-state recognizer for some regular set. A context-free grammar for that set can be obtained from the machine description as follows:

1. The input set of the machine is used as the terminal set of the grammar.
2. The state set of the machine is used as the nonterminal set and the starting state is used as the starting nonterminal.
3. If the machine has a transition from state A to state B under input x, then the grammar is given the rule

$$A \rightarrow x\,B$$

4. If A is an accepting state of the machine, then the grammar is given the rule

$$A \rightarrow \epsilon$$

To see that this construction results in a grammar for the set of sequences generated by the machine, interpret the nonterminal corresponding to state Z as

$\langle$string accepted by the machine started in state $Z\rangle$.

The production $A \rightarrow x\,B$ constructed in Step 3 is then interpreted as saying

> an example of a string accepted by the machine starting in state A is an x followed by an example of a string accepted by the machine started in state B.

The production $A \rightarrow \epsilon$ constructed in step 4 is interpreted as saying

> an example of a string accepted by the machine started in (accepting) state A is the null string.

Thus the productions express simple truths about how the finite-state machine works.

Under the interpretations just given, derivations correspond in a one-to-one fashion with the actions of the finite-state machine. As an example, we show a machine in Fig. 6.8(a) and the grammar constructed from the machine in Fig. 6.8(b). The sequence *aba* is accepted by the machine because it causes the following transition sequence

$$S \xrightarrow{a} A \xrightarrow{b} A \xrightarrow{a} B$$

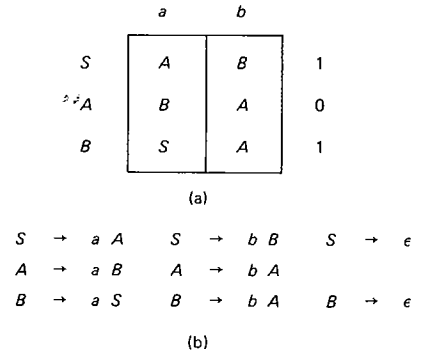


Figure 6.8

where S is the starting state and B is an accepting state. A corresponding derivation is obtained by applying the corresponding production for each transition and then removing the final nonterminal by applying an epsilon production. Thus the corresponding derivative of *aba* is

$$S \Rightarrow aA \Rightarrow abA \Rightarrow abaB \Rightarrow aba$$

Conversely, if one wants to find the state transitions corresponding to a given derivation, the nonterminal and rightmost terminal at each step give the new state and the input used to reach that state. Thus derivation

$$S \Rightarrow bB \Rightarrow baS \Rightarrow ba$$

implies the state transitions

$$S \xrightarrow{b} B \xrightarrow{a} S$$

and the application of the final epsilon production implies that final state S is an accepting state.

6.12 RIGHT-LINEAR GRAMMARS

Consider a special-form grammar in which the productions have one of the two forms

$$\langle A\rangle \rightarrow x\,\langle B\rangle \qquad \text{or} \qquad \langle A\rangle \rightarrow \epsilon$$

where x is a single terminal symbol. The procedure of the previous section can be reversed to obtain a finite-state recognizer to recognize the language generated by any grammar of the special form. Specifically, the inverse procedure is as follows:

1. the terminal set of the grammar is used as the input set of the machine;

2. the nonterminal set of the grammar is used as the state set of the machine and the starting nonterminal is used as the starting state;
3. if the grammar has a rule

$$\langle A \rangle \rightarrow x \langle B \rangle$$

then the machine is given a transition from state $\langle A \rangle$ to state $\langle B \rangle$ under input x;

4. if the grammar has a rule

$$\langle A \rangle \rightarrow \epsilon$$

then $\langle A \rangle$ is made an accepting state.

The result of this construction is a nondeterministic machine with one starting state. The construction of the previous section is valid for this kind of nondeterministic machine although it was presented as a procedure for deterministic machines. Applying the procedure to the grammar of Fig. 6.8(b), one gets Fig. 6.8(a). Derivation steps and machine transitions correspond as before.

Knowing that grammars of the above special form generate regular sets can be useful for spotting problems that can be done on a finite-state machine and the procedure can be used to obtain the machine. For example, the identifiers in ALGOL 60 can be described by the following grammar with starting nonterminal ⟨identifier⟩:

⟨identifier⟩ → *l* ⟨letters and digits⟩
⟨letters and digits⟩ → *l* ⟨letters and digits⟩
⟨letters and digits⟩ → *d* ⟨letters and digits⟩
⟨letters and digits⟩ → ϵ

where *l* and *d* represent letter and digit respectively. As the grammar has the special form, it is immediately apparent that it can be recognized by a finite-state machine. The procedure gives the nondeterministic machine of Fig. 6.9.

Grammars of the above special form are by no means the only grammars that generate regular sets. Another easily recognized class of grammars

	l	*d*	
< identifier >	< letters and digits >		0
< letters and digits >	< letters and digits >	< letters and digits >	1

Figure 6.9

known to generate regular sets is the class known as "right-linear" grammars. A grammar is called *right linear* if the righthand side of each production has at most one instance of a nonterminal and this nonterminal is the rightmost symbol of the righthand side. Symbolically, the productions of a right linear grammar have one of the two forms

$$\langle A \rangle \rightarrow w \langle B \rangle \qquad \text{or} \qquad \langle A \rangle \rightarrow w$$

where $\langle A \rangle$ and $\langle B \rangle$ are nonterminals and w is a string of terminal symbols.

An example of a right-linear grammar is the following grammar with starting symbol $\langle S \rangle$:

1. $\langle S \rangle \rightarrow a \langle A \rangle$
2. $\langle S \rangle \rightarrow b\ c$
3. $\langle S \rangle \rightarrow \langle A \rangle$
4. $\langle A \rangle \rightarrow a\ b\ b \langle S \rangle$
5. $\langle A \rangle \rightarrow c \langle A \rangle$
6. $\langle A \rangle \rightarrow \epsilon$

Productions 1, 3, 4, and 5 have the form $\langle A \rangle \rightarrow w \langle B \rangle$ and productions 2 and 6 have the form $\langle A \rangle \rightarrow w$. In productions 3 and 6, w is the null string.

Right-linear grammars are easily transformed into grammars of the special form previously discussed. We show how this is done on the current example. Production 4 is of the wrong form because the nonterminal is preceded by three terminals instead of one. To cure this, replace the production by the three productions:

$$\begin{aligned} \langle A \rangle &\rightarrow a \langle bbS \rangle \\ \langle bbS \rangle &\rightarrow b \langle bS \rangle \\ \langle bS \rangle &\rightarrow b \langle S \rangle \end{aligned}$$

each of the correct form. The result is that we now have

$$\langle A \rangle \Rightarrow a \langle bbS \rangle \Rightarrow a\ b \langle bS \rangle \Rightarrow a\ b\ b \langle S \rangle$$

instead of the single production:

$$\langle A \rangle \rightarrow a\ b\ b \langle S \rangle$$

One reason that production 2 has the wrong form is that its terminal symbols are not followed by a nonterminal. To cure this, we replace it by the two productions

⟨S⟩ → b c ⟨epsilon⟩
⟨epsilon⟩ → ϵ

The new production for $\langle S \rangle$ must now be transformed by the technique used previously for production 4.

Finally, we have production 3 which is the wrong form because the righthand side is a single nonterminal. To cure this, we replace production 3 by the productions of the form

$$\langle S \rangle \rightarrow \text{righthand side of a production for } \langle A \rangle$$

for all productions for $\langle A \rangle$.

The resulting new grammar is as follows:

1. $\langle S \rangle \rightarrow a \langle A \rangle$
2a. $\langle S \rangle \rightarrow b \langle c \text{ epsilon} \rangle$
2b. $\langle c \text{ epsilon} \rangle \rightarrow c \langle \text{epsilon} \rangle$
2c. $\langle \text{epsilon} \rangle \rightarrow \epsilon$
3a. $\langle S \rangle \rightarrow a \langle bbS \rangle$
3b. $\langle S \rangle \rightarrow c \langle A \rangle$
3c. $\langle S \rangle \rightarrow \epsilon$
4a. $\langle A \rangle \rightarrow a \langle bbS \rangle$
4b. $\langle bbS \rangle \rightarrow b \langle bS \rangle$
4c. $\langle bS \rangle \rightarrow b \langle S \rangle$
5. $\langle A \rangle \rightarrow c \langle A \rangle$
6. $\langle A \rangle \rightarrow \epsilon$

The transformed grammar can now be used to build a finite-state recognizer as shown in Fig. 6.10.

Although the transformed right-linear grammar is ideally suited for building a finite-state recognizer, the original untransformed grammar is probably preferable for describing the set to humans because it has fewer productions and nonterminals.

		a	*b*	*c*	
→	<*S*>	<*A*>, <*bbS*>	<*c epsilon*>	<*A*>	1
	<*c* epsilon>			<*epsilon*>	0
	<epsilon>				1
	<*A*>	<*bbS*>		<*A*>	1
	<*bbS*>		<*bS*>		0
	<*bS*>		<*S*>		0

Figure 6.10

Sometimes, particularly after the application of a grammatical transformation technique, a grammar contains a production of the form:

$$\langle A \rangle \rightarrow \langle A \rangle$$

Such productions say that a nonterminal is an example of itself. These productions serve no useful purpose and can be deleted from the grammar without affecting the language generated.

The transformation techniques of this section can be combined into a single procedure to convert a right-linear grammar into the special form described at the beginning of the section.

1. If there are productions of the form

$$\langle A \rangle \rightarrow w$$

where w is a nonnull string of terminals, invent a new nonterminal, say $\langle \text{epsilon} \rangle$, and add the production:

$$\langle \text{epsilon} \rangle \rightarrow \epsilon$$

Then replace each of the $\langle A \rangle \rightarrow w$ productions with production:

$$\langle A \rangle \rightarrow w \langle \text{epsilon} \rangle$$

2. Replace each production of the form

$$\langle A \rangle \rightarrow a_1 \ldots a_n \langle B \rangle$$

for $n > 1$, with productions

$$\langle A \rangle \rightarrow a_1 \langle a_2 \ldots a_n B \rangle$$
$$\langle a_i \ldots a_n B \rangle \rightarrow a_i \langle a_{i+1} \ldots a_n B \rangle \text{ for } 1 < i < n$$
$$\langle a_n B \rangle \rightarrow a_n \langle B \rangle$$

where $\langle a_i \ldots a_n B \rangle$ for $1 < i \leq n$ are new nonterminals.

3. If there is a nonterminal $\langle B \rangle$ such that there are productions of the form

$$\langle A \rangle \rightarrow \langle B \rangle$$

first delete the production

$$\langle B \rangle \rightarrow \langle B \rangle$$

if it is present and then if there are any remaining productions having $\langle B \rangle$ as the righthand side, replace them with productions of the form

$$\langle A \rangle \rightarrow y$$

for all combinations of $\langle A \rangle$ and righthand side y such that there are productions

$$\langle A \rangle \rightarrow \langle B \rangle \text{ and } \langle B \rangle \rightarrow y$$

Repeat step 3 if there are any remaining productions whose righthand side consists of a single nonterminal.

The effect of step 3 is to completely eliminate nonterminal $\langle B \rangle$ as an isolated righthand side. Once step 3 has eliminated a particular nonterminal as a righthand side then subsequent applications of step 3 to other nonterminals cannot reintroduce that nonterminal as a righthand side. Consequently, step 3 need only be applied once for each nonterminal appearing as a righthand side.

It sometimes happens that steps in the procedure produce duplicate productions. For example, consider the following grammar with starting symbol $\langle S \rangle$

1. $\langle S \rangle \rightarrow a\ b\ \langle A \rangle$
2. $\langle A \rangle \rightarrow b\ b\ \langle A \rangle$
3. $\langle A \rangle \rightarrow b\ \langle S \rangle$
4. $\langle S \rangle \rightarrow \epsilon$

Step 1 of the procedure need not be applied since there are no productions of the appropriate form. Step 2 applies to both productions 1 and 2 and the result is

1a. $\langle S \rangle \rightarrow a\ \langle bA \rangle$
1b. $\langle bA \rangle \rightarrow b\ \langle A \rangle$
2a. $\langle A \rangle \rightarrow b\ \langle bA \rangle$
2b. $\langle bA \rangle \rightarrow b\ \langle A \rangle$
3. $\langle A \rangle \rightarrow b\ \langle S \rangle$
4. $\langle S \rangle \rightarrow \epsilon$

However, productions 1b and 2b are identical and one copy should be discarded. Step 3 does not apply to this grammar as it already has the desired form.

As a further illustration, we now apply the procedure to the following grammar with starting symbol $\langle S \rangle$:

1. $\langle S \rangle \rightarrow a\ \langle A \rangle$
2. $\langle S \rangle \rightarrow \langle A \rangle$
3. $\langle A \rangle \rightarrow \langle S \rangle$
4. $\langle A \rangle \rightarrow \epsilon$

Steps 1 and 2 of the procedure do not apply. Applying step 3 to nonterminal $\langle A \rangle$ gives the productions

1. $\langle S \rangle \rightarrow a\ \langle A \rangle$

2a. $\langle S \rangle \rightarrow \langle S \rangle$
2b. $\langle S \rangle \rightarrow \epsilon$
3. $\langle A \rangle \rightarrow \langle S \rangle$
4. $\langle A \rangle \rightarrow \epsilon$

Applying step 3 to nonterminal $\langle S \rangle$, we first discard production 2a and then replace production 3 by two new productions. The new grammar is

1. $\langle S \rangle \rightarrow a\ \langle A \rangle$
2b. $\langle S \rangle \rightarrow \epsilon$
3a. $\langle A \rangle \rightarrow a\ \langle A \rangle$
3b. $\langle A \rangle \rightarrow \epsilon$
4. $\langle A \rangle \rightarrow \epsilon$

Productions 3b and 4 are duplicates so one of them should be dropped resulting in a four-production grammar of the desired form.

Because of the procedure for converting right-linear grammars to the special form, we can assert the following fact:

The language generated by a right-linear grammar is regular.

To see how right-linear grammars might arise in practice, consider ALGOL 60 variable declarations, which are of the form

TYPE IDENT, IDENT, ..., IDENT

where TYPE and IDENT are lexical tokens. These declarations are described rather compactly by the right-linear grammar:

⟨declaration⟩ → TYPE IDENT ⟨declared variable list⟩
⟨declared variable list⟩ → , IDENT ⟨declared variable list⟩
⟨declared variable list⟩ → ϵ

6.13 ANOTHER GRAMMAR FOR MINI-BASIC CONSTANTS

A grammar for MINI-BASIC constants was given in Section 6.7. Now we derive a second grammar by applying the procedure of Section 6.11 to a finite-state recognizer for MINI-BASIC constants.

A recognizer for unsigned MINI-BASIC constants was developed in Section 2.14 and was shown in Fig. 2.24(b). For this example, we explicitly add an error state, ER, to obtain the recognizer of Fig. 6.11. The names of the input symbols have been changed to conform to Section 6.7.

Applying the procedure of Section 6.11 to this transition table gives a grammar with 40 productions of the form $\langle A \rangle \rightarrow x\ \langle B \rangle$ and three of the

	d	*E*	.	+	−	
0	1	ER	7	ER	ER	0
1	1	4	$\overline{23}$	ER	ER	1
$\overline{23}$	$\overline{23}$	4	ER	ER	ER	1
4	6	ER	ER	5	5	0
5	6	ER	ER	ER	ER	0
6	6	ER	ER	ER	ER	1
7	$\overline{23}$	ER	ER	ER	ER	0
ER	ER	ER	ER	ER	ER	0

Figure 6.11

form $\langle A \rangle \rightarrow \epsilon$. However, closer inspection reveals that this size is somewhat misleading. Consider the productions that apply to the nonterminal ⟨ER⟩. These may be found by inspecting the row for state ⟨ER⟩ in the transition table and are specifically:

$$\langle \text{ER} \rangle \rightarrow d \langle \text{ER} \rangle$$
$$\langle \text{ER} \rangle \rightarrow \text{E} \langle \text{ER} \rangle$$
$$\langle \text{ER} \rangle \rightarrow . \langle \text{ER} \rangle$$
$$\langle \text{ER} \rangle \rightarrow + \langle \text{ER} \rangle$$
$$\langle \text{ER} \rangle \rightarrow - \langle \text{ER} \rangle$$

It is evident from these productions that the nonterminal ⟨ER⟩ cannot be used in the derivation of any string of terminals since any substitution for nonterminal ⟨ER⟩ simply results in a new string containing nonterminal ⟨ER⟩. In fact, the interpretation of nonterminal ⟨ER⟩ as

⟨string accepted by the machine started in state ER⟩

indicates that it does not generate any terminal strings at all since ER is an error state with the property that once the machine enters state ER, it will never subsequently accept any sequence. Thus, if nonterminal ⟨ER⟩ is ever introduced in a derivation all subsequent intermediate strings must contain ⟨ER⟩ and cannot be terminal strings.

Because nonterminal ⟨ER⟩ does not generate any terminal strings we can eliminate from the grammar all productions containing ⟨ER⟩ on either the left or right side. Since ⟨ER⟩ cannot be involved in the generation of any string in the language, elimination of ⟨ER⟩ and all its productions from the grammar cannot possibly have any effect on the language generated by the grammar.

After eliminating all the productions containing ⟨ER⟩ the 43 original productions reduce to the following 16.

$$\langle 0 \rangle \rightarrow d \langle 1 \rangle$$
$$\langle 0 \rangle \rightarrow . \langle 7 \rangle$$
$$\langle 1 \rangle \rightarrow d \langle 1 \rangle$$
$$\langle 1 \rangle \rightarrow \text{E} \langle 4 \rangle$$
$$\langle 1 \rangle \rightarrow . \langle \overline{23} \rangle$$
$$\langle 1 \rangle \rightarrow \epsilon$$
$$\langle \overline{23} \rangle \rightarrow d \langle \overline{23} \rangle$$
$$\langle \overline{23} \rangle \rightarrow \text{E} \langle 4 \rangle$$
$$\langle \overline{23} \rangle \rightarrow \epsilon$$
$$\langle 4 \rangle \rightarrow d \langle 6 \rangle$$
$$\langle 4 \rangle \rightarrow + \langle 5 \rangle$$
$$\langle 4 \rangle \rightarrow - \langle 5 \rangle$$
$$\langle 5 \rangle \rightarrow d \langle 6 \rangle$$
$$\langle 6 \rangle \rightarrow d \langle 6 \rangle$$
$$\langle 6 \rangle \rightarrow \epsilon$$
$$\langle 7 \rangle \rightarrow d \langle \overline{23} \rangle$$

In order to use this grammar as a means of communicating the form of MINI-BASIC constants, the nonterminals should be given more suggestive names. One possibility is as follows:

⟨0⟩ = ⟨constant⟩
⟨1⟩ = ⟨digits with optional decimal and exponent⟩
$\langle \overline{23} \rangle$ = ⟨decimal digits and optional exponent⟩
⟨4⟩ = ⟨integer⟩
⟨5⟩ = ⟨unsigned integer⟩
⟨6⟩ = ⟨digits⟩
⟨7⟩ = ⟨decimal integer and optional exponent⟩

6.14 EXTRANEOUS NONTERMINALS

In the last section, we came upon an example of a nonterminal which could not be used in the derivation of a terminal string and which could therefore be discarded from the grammar along with all productions involving that nonterminal. Nonterminals which do not generate any terminal strings are called *dead nonterminals.*

In the previous section, the nonterminal ⟨ER⟩ was dead because each production with ⟨ER⟩ on the lefthand side also had ⟨ER⟩ on the righthand side. Nonterminals can also be dead for more subtle reasons. Consider for example the following grammar with starting symbol ⟨*S*⟩:

1. ⟨*S*⟩ → *a* ⟨*S*⟩ *a*
2. ⟨*S*⟩ → *b* ⟨*A*⟩ *d*
3. ⟨*S*⟩ → *c*
4. ⟨*A*⟩ → *c* ⟨*B*⟩ *d*
5. ⟨*A*⟩ → *a* ⟨*A*⟩ *d*
6. ⟨*B*⟩ → *d* ⟨*A*⟩ *f*

Here, both ⟨*A*⟩ and ⟨*B*⟩ are dead nonterminals. A string with symbol ⟨*A*⟩ can be changed to a string with symbol ⟨*B*⟩ by applying production 4 and a ⟨*B*⟩ can be changed back to ⟨*A*⟩ using production 6. But no matter how one substitutes, a string with an ⟨*A*⟩ or ⟨*B*⟩ is always transformed into a string with an ⟨*A*⟩ or ⟨*B*⟩. The productions involving ⟨*A*⟩ and ⟨*B*⟩ can therefore be eliminated from the grammar leaving only productions 1 and 3. Productions 1 and 3 generate the same terminal strings as productions 1 through 6.

A similar situation occurs when there are nonterminals that cannot be reached from the starting nonterminal. Consider for example the following grammar with starting nonterminal ⟨*S*⟩:

1. ⟨*S*⟩ → *a* ⟨*S*⟩ *b*
2. ⟨*S*⟩ → *c*
3. ⟨*A*⟩ → *b* ⟨*S*⟩
4. ⟨*A*⟩ → *a*

Neither of the two productions with starting symbol ⟨*S*⟩ as a lefthand side (i.e., productions 1 and 2) has the symbol ⟨*A*⟩ on the righthand side. No string derived from symbol ⟨*S*⟩ can contain an ⟨*A*⟩. Therefore ⟨*A*⟩ cannot be used to derive a string from an ⟨*S*⟩, even though ⟨*A*⟩ itself is not dead. Nonterminal ⟨*A*⟩ and its productions can be eliminated from the grammar leaving only productions 1 and 2. Nonterminals which do not appear in any string derived from the starting symbol are called *unreachable nonterminals.*

Nonterminals which are either dead or unreachable are called *extraneous.* When a grammar is obtained by mechanical means such as by using the procedure of Section 6.11, there is often a possibility that extraneous nonterminals have been created. Such grammars should generally be checked to see if they can be simplified by eliminating extraneous nonterminals. Even with grammars that are made by hand, extraneous nonterminals frequently arise when the designer makes an error. Because of this, a search for extraneous nonterminals can often be useful in debugging a handmade grammar. For these reasons, we now describe a procedure for detecting extraneous nonterminals.

The procedure consists of two parts, one to detect dead nonterminals and one to detect unreachable nonterminals. The procedure for dead nonterminals should be done first since the elimination of dead nonterminals from the grammar may cause other nonterminals to become unreachable.

We call a symbol alive if it is either a terminal symbol or a nonterminal symbol from which a terminal string can be derived (i.e., if it is not a dead nonterminal). The procedure for detecting dead nonterminals is based on the following property of alive symbols

> **Property A:** If all symbols on the righthand side of a production are alive, then so is the symbol on the lefthand side.

To see that this property must hold, observe that given such a production, a terminal string for the symbol on the lefthand side can be obtained by first applying the given production and then replacing each nonterminal in the righthand side with one of the strings that makes it alive.

The basic idea of the procedure is to start a list of nonterminals "known to be alive" and then use Property A to detect other alive nonterminals and expand the list. The steps of the procedure are as follows:

1. Make a list of nonterminals that have at least one production with no nonterminals on the righthand side.
2. If a production is found such that all the nonterminals on the righthand side are on the list, then the nonterminal on the lefthand side of the production is added to the list.
3. When no further nonterminals can be added to the list using rule 2, then the list is the list of all alive nonterminals and all nonterminals not on the list are dead.

The fact that the list obtained from step 3 contains only alive nonterminals is due to the fact that nonterminals are only added when they are known to be alive by Property A. To see that all alive nonterminals must appear on the list, observe that given a sequence of productions used to derive a terminal string from a given nonterminal, the production sequence can be used in reverse to show that all nonterminals involved are alive by Property A.

To see the procedure in action, consider the grammar of Fig. 6.12(a) with starting symbol ⟨*S*⟩. By step 1, ⟨*C*⟩ can be placed on the list because of production 9. Then ⟨*A*⟩ can be added by step 2 because of production 5. Finally, ⟨*S*⟩ can be added because of production 2. Further attempts to

(a)	(b)
1. $\langle S\rangle \to a\,\langle A\rangle\,\langle B\rangle\,\langle S\rangle$	
2. $\langle S\rangle \to b\,\langle C\rangle\,\langle A\rangle\,\langle C\rangle\,d$	2. $\langle S\rangle \to b\,\langle C\rangle\,\langle A\rangle\,\langle C\rangle\,d$
3. $\langle A\rangle \to b\,\langle A\rangle\,\langle B\rangle$	
4. $\langle A\rangle \to c\,\langle S\rangle\,\langle A\rangle$	4. $\langle A\rangle \to c\,\langle S\rangle\,\langle A\rangle$
5. $\langle A\rangle \to c\,\langle C\rangle\,\langle C\rangle$	5. $\langle A\rangle \to c\,\langle C\rangle\,\langle C\rangle$
6. $\langle B\rangle \to b\,\langle A\rangle\,\langle B\rangle$	
7. $\langle B\rangle \to c\,\langle S\rangle\,\langle B\rangle$	
8. $\langle C\rangle \to c\,\langle S\rangle$	8. $\langle C\rangle \to c\,\langle S\rangle$
9. $\langle C\rangle \to c$	9. $\langle C\rangle \to c$

Figure 6.12

apply step 2 fail and so we know that the alive nonterminals are $\langle C\rangle$, $\langle A\rangle$, and $\langle S\rangle$ and the remaining nonterminal $\langle B\rangle$ is dead. Eliminating all productions involving the dead nonterminal $\langle B\rangle$ gives the grammar of Fig. 6.12(b).

A symbol is called *reachable* in a grammar if it appears in a string derived from the starting nonterminal (i.e., if it is not unreachable). The procedure for detecting unreachable nonterminals is based on the following property of reachable symbols:

> **Property B.** If the nonterminal on the lefthand side of a production is reachable, then so are all the symbols on the righthand side.

This property holds because the righthand-side symbols can be reached by first deriving a string containing the lefthand side and then applying the given production. The basic idea of the procedure is to start a list of nonterminals "known to be reachable" and then use Property B to detect other reachable nonterminals and expand the list. The steps of the procedure are as follows.

1. Make a one-element list of nonterminals consisting of the starting nonterminal.

2. If a production is found such that the lefthand side is on the list, then the nonterminals appearing on the righthand side are added to the list.

3. When no further nonterminals can be added to the list using rule 2, then the list is the list of all reachable nonterminals and all nonterminals not on the list are unreachable.

The list obtained from step 3 contains only reachable nonterminals since nonterminals are only added when known to be reachable by Property B. All reachable nonterminals must be on the final list as the sequence of productions used to reach a given nonterminal can be used with Property B to show that all nonterminals in the sequence of productions are placed on the list.

(a)	(b)
1. $\langle S\rangle \to a\,\langle A\rangle\,\langle B\rangle$	1. $\langle S\rangle \to a\,\langle A\rangle\,\langle B\rangle$
2. $\langle S\rangle \to \langle E\rangle$	2. $\langle S\rangle \to \langle E\rangle$
3. $\langle A\rangle \to d\,\langle D\rangle\,\langle A\rangle$	3. $\langle A\rangle \to d\,\langle D\rangle\,\langle A\rangle$
4. $\langle A\rangle \to e$	4. $\langle A\rangle \to e$
5. $\langle B\rangle \to b\,\langle E\rangle$	5. $\langle B\rangle \to b\,\langle E\rangle$
6. $\langle B\rangle \to f$	6. $\langle B\rangle \to f$
7. $\langle C\rangle \to c\,\langle A\rangle\,\langle B\rangle$	
8. $\langle C\rangle \to d\,\langle S\rangle\,\langle D\rangle$	
9. $\langle C\rangle \to a$	
10. $\langle D\rangle \to e\,\langle A\rangle$	10. $\langle D\rangle \to e\,\langle A\rangle$
11. $\langle E\rangle \to f\,\langle A\rangle$	11. $\langle E\rangle \to f\,\langle A\rangle$
12. $\langle E\rangle \to g$	12. $\langle E\rangle \to g$

Figure 6.13

As an example of the procedure, consider the grammar of Fig. 6.13(a) with starting symbol $\langle S\rangle$. By rule 1, we begin with starting symbol $\langle S\rangle$ on the list. Applying rule 2 to the productions with $\langle S\rangle$ on the lefthand side, namely productions 1 and 2, we find that nonterminals $\langle A\rangle$, $\langle B\rangle$, and $\langle E\rangle$ must be added to the list. Applying rule 2 to production 3, we find that $\langle D\rangle$ must also be added. Checking the other productions does not give any additional nonterminals and we conclude that $\langle S\rangle$, $\langle A\rangle$, $\langle B\rangle$, $\langle D\rangle$, and $\langle E\rangle$ are reachable and the remaining nonterminal $\langle C\rangle$ is unreachable. Dropping the productions involving $\langle C\rangle$, we get the simplified grammar in Fig. 6.13(b).

Finally, we illustrate the two procedures together using the grammar of Fig. 14(a), with starting symbol $\langle S\rangle$. Applying the procedure for dead nonterminals, we discover that $\langle A\rangle$ and $\langle B\rangle$ are dead. Eliminating the productions involving these dead nonterminals gives the grammar of Fig. 6.14(b).

(a)	(b)	(c)
1. $\langle S\rangle \to a\,c$	1. $\langle S\rangle \to a\,c$	1. $\langle S\rangle \to a\,c$
2. $\langle S\rangle \to b\,\langle A\rangle$	5. $\langle C\rangle \to b\,\langle C\rangle$	
3. $\langle A\rangle \to c\,\langle B\rangle\,\langle C\rangle$	6. $\langle C\rangle \to d$	
4. $\langle B\rangle \to a\,\langle S\rangle\,\langle A\rangle$		
5. $\langle C\rangle \to b\,\langle C\rangle$		
6. $\langle C\rangle \to d$		

Figure 6.14

Now testing for unreachable nonterminals, we find that ⟨C⟩ is unreachable. Eliminating the productions involving ⟨C⟩ gives us the grammar of Fig. 6.14(c). It is now evident that the language generated from the grammar of Fig. 6.14(a) consists of the one string *ac*. Observe that nonterminal ⟨C⟩ becomes unreachable only after production 3 is eliminated. Thus if the test for unreachable nonterminals were performed before the test for dead nonterminals, the full simplification would not be achieved.

6.15 A MINI-BASIC GRAMMAR FOR THE MINI-BASIC LANGUAGE MANUAL

The MINI-BASIC Language Manual of Appendix A is designed to explain the MINI-BASIC language to people with no formal language skills. The approach used is to explain the language structures by means of example. It is hoped that by looking at a few examples, the reader can write arbitrary constructions of a given form. The weakness of such a scheme is that the reader may nevertheless be confused as to exactly what is allowed. To assist the more sophisticated user, it is good practice to supplement the elementary description with a context-free grammar for the language so that the reader familiar with formal languages can determine more exactly what language constructs are permitted. In this section, we give a MINI-BASIC grammar which might be suitable as an appendix to the MINI-BASIC manual.

The terminals of the grammar are the characters that normally go into a MINI-BASIC program. The blank character is not part of the terminal set as the rule "spaces are ignored" is best understood without being built into the grammar.

Starting Nonterminal

The starting nonterminal is called ⟨statement list⟩ and has two productions:

1. ⟨statement list⟩ → ⟨number⟩ ⟨statement⟩ (CR) ⟨statement list⟩
2. ⟨statement list⟩ → ⟨number⟩ END (CR)

The symbol (CR) stands for the character "carriage return." These productions ensure that the program is a list of statements preceded by line numbers and that the last statement is the END statement.

Now we consider each of the statements.

Null Statement

A line can contain no statement at all.

3. ⟨statement⟩ → ϵ

Assignment Statement

4. ⟨statement⟩ → LET ⟨variable⟩ = ⟨expression⟩

GOTO Statement

5. ⟨statement⟩ → GOTO ⟨number⟩

IF Statement

6. ⟨statement⟩ → IF ⟨expression⟩ ⟨relational operator⟩ ⟨expression⟩ GOTO ⟨number⟩

GOSUB Statement

7. ⟨statement⟩ → GOSUB ⟨number⟩

RETURN Statement

8. ⟨statement⟩ → RETURN

FOR Statement

9. ⟨statement⟩ → FOR ⟨variable⟩ = ⟨expression⟩ TO ⟨expression⟩
10. ⟨statement⟩ → FOR ⟨variable⟩ = ⟨expression⟩ TO ⟨expression⟩ STEP ⟨expression⟩

NEXT Statement

11. ⟨statement⟩ → NEXT ⟨variable⟩

REM Statement

12. ⟨statement⟩ → REM ⟨characters⟩

Because there are so many characters we do not bother to give productions for the nonterminal ⟨characters⟩, but merely observe that it generates an arbitrary sequence composed of any characters except (CR).

Expressions

The grammar for expressions is a larger version of that given in Section 6.9. Productions 13 and 14 generate the unary + and − operators.

13. ⟨expression⟩ → + ⟨term⟩
14. ⟨expression⟩ → − ⟨term⟩
15. ⟨expression⟩ → ⟨expression⟩ + ⟨term⟩
16. ⟨expression⟩ → ⟨expression⟩ − ⟨term⟩
17. ⟨expression⟩ → ⟨term⟩
18. ⟨term⟩ → ⟨term⟩ * ⟨factor⟩
19. ⟨term⟩ → ⟨term⟩ / ⟨factor⟩
20. ⟨term⟩ → ⟨factor⟩
21. ⟨factor⟩ → ⟨factor⟩ ↑ ⟨primary⟩

22. ⟨factor⟩ → ⟨primary⟩
23. ⟨primary⟩ → (⟨expression⟩)
24. ⟨primary⟩ → ⟨variable⟩
25. ⟨primary⟩ → ⟨unsigned constant⟩

Numbers and Constants

26. ⟨number⟩ → ⟨digit⟩ ⟨digits⟩
27. ⟨digits⟩ → ⟨digit⟩ ⟨digits⟩
28. ⟨digits⟩ → ϵ

Productions for ⟨digit⟩ will be given later.

29. ⟨unsigned constant⟩ → ⟨number⟩ ⟨exponent⟩
30. ⟨unsigned constant⟩ → ⟨number⟩ . ⟨digits⟩ ⟨exponent⟩
31. ⟨unsigned constant⟩ → . ⟨number⟩ ⟨exponent⟩
32. ⟨exponent⟩ → E + ⟨number⟩
33. ⟨exponent⟩ → E − ⟨number⟩
34. ⟨exponent⟩ → E ⟨number⟩
35. ⟨exponent⟩ → ϵ

Variables

36. ⟨variable⟩ → ⟨letter⟩
37. ⟨variable⟩ → ⟨letter⟩ ⟨digit⟩

Other Productions

To conserve space, we write several productions on one line

38–43. ⟨relational operator⟩ → = | <> | < | <= | > | >=

44–53. ⟨digit⟩ → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

54–79. ⟨letter⟩ → A | B | C | D | E | F | G | H | I | J | K | L | M | N | O | P | Q | R | S | T | U | V | W | X | Y | Z

Some aspects of the MINI-BASIC language are not specified by this grammar; for example,

a) no two lines of a MINI-BASIC program can begin with the same line number;

b) the line number used in a GOTO, IF, or GOSUB statement must actually occur at the beginning of some line in the program;

c) each FOR statement must have a corresponding NEXT statement with the same variable, and these statements must be properly nested.

Published context-free grammars for other common programming languages such as ALGOL and PL/I also give an incomplete specification. Usually, a programming language is specified by a context-free grammar plus additional informal descriptive material. The programming language is then defined to consist of those sequences that can be generated by the grammar *and* satisfy the additional restrictions specified in the descriptive material. Hence the grammar alone generates all the programs in the language plus some additional programs not in the language; for example, the MINI-BASIC grammar generates some programs where two lines have the same line number.

Customarily those features of the language that are described by the grammar in its manual are called *syntactic* or *grammatical* features and those features not described by its grammar are called *semantic* features. In the language manual the semantic features are usually described informally in a natural language such as English.

The words "syntactic" and "semantic" should be not taken too seriously. When different grammars are used to describe the same programming language, some features of the language that are not described by one grammar (and hence might be called semantic) may be described by another grammar (and hence would be syntactic for that grammar). As an example of this situation, the grammar actually used in the syntax box of our MINI-BASIC compiler (as given in Sections 10.1 and 12.8) does describe grammatically the requirement that each FOR statement have a matching NEXT statement.

However there are some aspects of programming languages that cannot possibly be described by a context-free grammar. For instance if line numbers are allowed to be arbitrarily long, it can be proved that no context-free grammar generates exactly those MINI-BASIC programs satisfying the restriction that no two lines begin with the same line number. As another example, a context-free grammar cannot generate exactly those ALGOL programs satisfying the restriction that identifiers are not declared more than once in the same block.

There are other aspects of programming languages that *can* be described by a context-free grammar, but only by a grammar that is excessively large. For instance it *is* possible to find a grammar that generates the MINI-BASIC programs satisfying the restriction that each FOR statement and its matching NEXT statement actually use the same variable. However, this grammar would be quite large; hence we choose to describe this restriction by non-grammatical means.

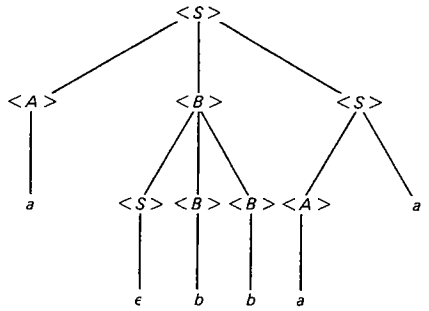
In practice, context-free grammars are only used to specify as much of a programming language as can be done compactly. For the remaining aspects, informal descriptive methods are used.

6.16 REFERENCES

Context-free grammars were first formalized by Chomsky [1956, 1957, 1959]. Many of the properties of context-free languages first appeared in Bar-Hillel, Perles, and Shamir [1961]. The relationship between regular sets and right-linear grammars is developed in Chomsky and Miller [1958]. A survey of context-free grammars is given in Chomsky [1963]. The first use of Backus-Naur Form to describe a programming language is in Naur, et al. [1960, 1962]. Ginsburg [1966] is a mathematical text that is completely devoted to context-free languages. Methods for obtaining a recognizer for any context-free grammar are discussed in Aho and Ullman [1972a].

PROBLEMS

1. Find a context-free grammar that will generate each of the following languages:
   a) $\{1^n 0^m\}$ for $n > m > 0$;
   b) $\{1^n 0^n 1^m 0^m\}$ for $n, m \geq 0$;
   c) $\{1^n 0^m 1^m 0^n\}$ for $n, m \geq 0$;
   d) $\{1^n 0^n\} \cup \{0^m 1^m\}$ for $n, m \geq 0$;
   e) $\{1^{3n+2} 0^n\}$ for $n \geq 0$;
   f) $\{w\, a\, w^r\}$ where w is an arbitrary sequence of 1's and 0's;
   g) All sequences of 1's and 0's containing an equal number of each;
   h) All sequences of 1's and 0's containing an equal number of each and such that each subsequence beginning at the left end contains at least as many 1's as 0's;
   i) $\{1^n 0^m 1^p\}$ for $n + p > m \geq 0$.
2. A context-free grammar generates the following derivation tree.



   a) Give the leftmost derivation corresponding to this derivation tree.
   b) How many different derivations correspond to this derivation tree?
   c) Draw a derivation tree for the terminal string *ab*.
3. Describe the language generated by each of the following grammars with starting nonterminal $\langle S \rangle$.
   a) $\langle S \rangle \rightarrow 1\ 0\ \langle S \rangle\ 0$
      $\langle S \rangle \rightarrow a\ \langle A \rangle$
      $\langle A \rangle \rightarrow b\ \langle A \rangle$
      $\langle A \rangle \rightarrow a$
   b) $\langle S \rangle \rightarrow \langle S \rangle\ \langle S \rangle$
      $\langle S \rangle \rightarrow 1\ \langle A \rangle\ 0$
      $\langle A \rangle \rightarrow 1\ \langle A \rangle\ 0$
      $\langle A \rangle \rightarrow \epsilon$
   c) $\langle S \rangle \rightarrow 1\ \langle A \rangle$
      $\langle S \rangle \rightarrow \langle B \rangle\ 0$
      $\langle A \rangle \rightarrow 1\ \langle A \rangle$
      $\langle A \rangle \rightarrow \langle C \rangle$
      $\langle B \rangle \rightarrow \langle B \rangle\ 0$
      $\langle B \rangle \rightarrow \langle C \rangle$
      $\langle C \rangle \rightarrow 1\ \langle C \rangle\ 0$
      $\langle C \rangle \rightarrow \epsilon$
   d) $\langle S \rangle \rightarrow B\ \langle A \rangle\ \langle D \rangle\ C$
      $\langle D \rangle \rightarrow \langle G \rangle\ I$
      $\langle A \rangle \rightarrow A\ \langle G \rangle\ S$
      $\langle G \rangle \rightarrow \epsilon$
   e) $\langle S \rangle \rightarrow a\ \langle S \rangle\ \langle S \rangle$
      $\langle S \rangle \rightarrow a$
4. Which of the following sequences can be derived from the given grammar with starting nonterminal $\langle S \rangle$? In each case give a leftmost derivation, a rightmost derivation, and a derivation tree.

$\langle S \rangle \rightarrow a\ \langle A \rangle\ c\ \langle B \rangle$
$\langle S \rangle \rightarrow \langle B \rangle\ d\ \langle S \rangle$
$\langle B \rangle \rightarrow a\ \langle S \rangle\ c\ \langle A \rangle$
$\langle B \rangle \rightarrow c\ \langle A \rangle\ \langle B \rangle$
$\langle A \rangle \rightarrow \langle B \rangle\ a\ \langle B \rangle$
$\langle A \rangle \rightarrow a\ \langle B \rangle\ c$
$\langle A \rangle \rightarrow a$
$\langle B \rangle \rightarrow b$

a) *aacb*

b) *aababcbadcd*

c) *aacbccb*

d) *aacabcbcccaacdca*

e) *aacabcbcccaacbca*

5. Find a context-free grammar for expressions in the Lambda Calculus as described in the following paragraph:

   An expression in Lambda Calculus is a variable; or the symbol λ followed by a variable followed by an expression; or a left parenthesis followed by an expression followed by an expression followed by a right parenthesis.

6. Is it possible for a sequence to have two leftmost derivations, but only one rightmost derivation?

7. Show that if a language is context free, then the language obtained by regarding the endmarker as a terminal symbol and inserting it at the end of every sequence in the original language is also context free.

8. Find a context-free grammar for the two IF statements in FORTRAN.

9. Find a context-free grammar for the IF statement in COBOL.

10. Draw the derivation tree for the following ALGOL program using the standard grammar in the ALGOL report (Naur [1963]):

    begin integer *A*1;
    *A*1 := 12; **end**

11. Find a context-free grammar for a statement in SNOBOL.

12. Find a context-free grammar for regular expressions (Hennie [1968]) using the + and * operators and indicating concatenation with juxtaposition.

13. Describe the language generated by the following grammar and draw the derivation tree for three of the sequences in its language.

    $\langle E\rangle \rightarrow \langle P\rangle\ \langle R\rangle$
    $\langle P\rangle \rightarrow (\ \langle E\rangle\ )$
    $\langle P\rangle \rightarrow I$
    $\langle R\rangle \rightarrow +\ \langle P\rangle\ \langle R\rangle$
    $\langle R\rangle \rightarrow *\ \langle P\rangle\ \langle R\rangle$
    $\langle R\rangle \rightarrow \uparrow\ \langle P\rangle\ \langle R\rangle$
    $\langle R\rangle \rightarrow \epsilon$

14. Find a grammar which generates the same set of arithmetic expressions as the grammar of Section 6.9, but which has only one nonterminal.

15. a) Find a context-free grammar for Boolean expressions made up of Boolean variables, constants, parentheses, and the NOT, OR, and AND operators. Assume the usual precedence of NOT before AND and AND before OR.

b) Add to your grammar Boolean primaries consisting of an arithmetic expression followed by a relational operator ($>, \geq, =, \neq, <, \leq$) followed by another arithmetic expression.

16. Find a right-linear grammar for the language recognized by the lexical box of the MINI-BASIC compiler.

17. The language recognized by this machine is denoted by L

	0	1	
A	*B*	*A*	0
B	*C*	*B*	1
C	*B*	*A*	0

    Find right-linear grammars for the languages

    a) $L + \epsilon$

    b) $L \cdot L$

    c) L^*

    d) L^+

18. Consider the grammar

    $S \rightarrow S\,a$
    $S \rightarrow S\,b$
    $S \rightarrow a$

    a) Find a right-linear grammar that generates the same language.

    b) Draw the derivation tree for the sequence *ababb* according to both grammars.

19. Find right-linear grammars for the languages and/or machines described in the following problems from Chapter 2.

a) 1	e) 13	i) 17
b) 3	f) 14	j) 18
c) 4	g) 15	k) 22
d) 6	h) 16	l) 23

20. Show that the following two right-linear grammars generate the same language.

    a) (starting nonterminal $\langle X\rangle$)

    $\langle X\rangle \rightarrow 0$
    $\langle X\rangle \rightarrow 0\ \langle Y\rangle$
    $\langle X\rangle \rightarrow 1\ \langle Z\rangle$
    $\langle Y\rangle \rightarrow 0\ \langle X\rangle$
    $\langle Y\rangle \rightarrow 1\langle Y\rangle$
    $\langle Y\rangle \rightarrow 1$
    $\langle Z\rangle \rightarrow 0\ \langle Z\rangle$
    $\langle Z\rangle \rightarrow 1\ \langle X\rangle$

b) (starting nonterminal $\langle A \rangle$)

$\langle A \rangle \rightarrow 0\ \langle B \rangle$
$\langle A \rangle \rightarrow 1\ \langle E \rangle$
$\langle B \rangle \rightarrow 0\ \langle A \rangle$
$\langle B \rangle \rightarrow 1\ \langle F \rangle$
$\langle B \rangle \rightarrow \epsilon$
$\langle C \rangle \rightarrow 0\ \langle C \rangle$
$\langle C \rangle \rightarrow 1\ \langle A \rangle$
$\langle D \rangle \rightarrow 0\ \langle A \rangle$
$\langle D \rangle \rightarrow 1\ \langle D \rangle$
$\langle D \rangle \rightarrow \epsilon$
$\langle E \rangle \rightarrow 0\ \langle C \rangle$
$\langle E \rangle \rightarrow 1\ \langle A \rangle$
$\langle F \rangle \rightarrow 0\ \langle A \rangle$
$\langle F \rangle \rightarrow 1\ \langle B \rangle$
$\langle F \rangle \rightarrow \epsilon$

21. Find the extraneous nonterminals in the following grammar with starting nonterminal $\langle S \rangle$

$\langle S \rangle \rightarrow a\ \langle A \rangle\ \langle B \rangle\ \langle C \rangle$
$\langle S \rangle \rightarrow b\ \langle C \rangle\ \langle E \rangle\ \langle S \rangle$
$\langle S \rangle \rightarrow a\ \langle E \rangle$
$\langle A \rangle \rightarrow b\ \langle E \rangle$
$\langle A \rangle \rightarrow \langle S \rangle\ \langle C \rangle\ \langle D \rangle$
$\langle A \rangle \rightarrow d$
$\langle B \rangle \rightarrow d\ \langle F \rangle\ \langle S \rangle$
$\langle B \rangle \rightarrow a\ \langle B \rangle\ \langle C \rangle$
$\langle C \rangle \rightarrow a\ \langle E \rangle\ \langle S \rangle$
$\langle C \rangle \rightarrow b\ \langle E \rangle$
$\langle D \rangle \rightarrow a\ \langle A \rangle\ \langle C \rangle$
$\langle D \rangle \rightarrow d$
$\langle E \rangle \rightarrow a\ \langle C \rangle\ \langle E \rangle$
$\langle E \rangle \rightarrow \epsilon$
$\langle F \rangle \rightarrow \langle A \rangle\ \langle B \rangle$
$\langle F \rangle \rightarrow a\ \langle F \rangle$

22. Find a right-linear grammar with no extraneous nonterminals corresponding to each of the machines in Fig. 6.15.

23. What is the value of each of the following expressions in MINI-BASIC? What is their value in FORTRAN (making the lexical change of replacing ↑ with **)?

a) $-2 \uparrow 2$
b) $2 + 2 / 2 + 2$
c) $-2 * 2$

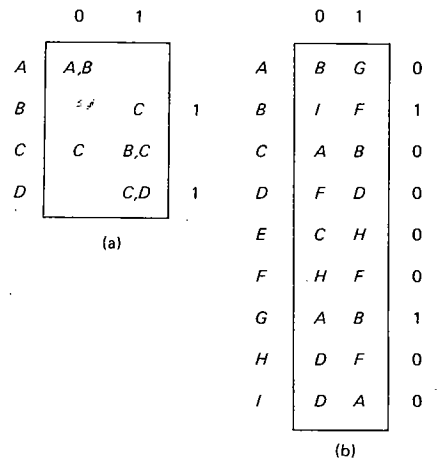


Figure 6.15

d) $2 \uparrow 2 / 2 \uparrow 2$
e) $2 \uparrow 2 \uparrow 2 \uparrow 2$

24. Describe in words the language generated by this grammar with $\langle E \rangle$ as the starting symbol:

$\langle E \rangle \rightarrow \langle E \rangle\ \langle T \rangle\ +$
$\langle E \rangle \rightarrow \langle T \rangle$
$\langle T \rangle \rightarrow \langle T \rangle\ \langle F \rangle\ *$
$\langle T \rangle \rightarrow \langle F \rangle$
$\langle F \rangle \rightarrow \langle F \rangle\ \langle P \rangle \uparrow$
$\langle F \rangle \rightarrow \langle P \rangle$
$\langle P \rangle \rightarrow \langle E \rangle$
$\langle P \rangle \rightarrow I$

25. Find a grammar for arithmetic expressions similar to that of Section 6.9 except that unary plus and minus operators are allowed before every number or variable (for example $3 * -4$ is allowed) with each of the following interpretations:

a) the same precedence as binary addition and subtraction (for example $-2 \uparrow 2 = -(2 \uparrow 2) = -4$);

b) the unary operators are always carried out first (for example $-2 \uparrow 2 = (-2) \uparrow 2 = 4$).

26. Draw the derivation tree corresponding to the grammar of Section 6.15 for the following MINI-BASIC programs.

a)
```
05 LET X1 = -4 + 3 * (+1 - 7)
10 END
```

b)
```
12 FOR X1 = 1 TO 12
22 FOR X2 = 1 TO 10
24 LET X1 = X2 + 1
25 NEXT X2
26 NEXT X1
27 END
```

27. Using the terminal alphabet of the grammar of Section 6.9, write a grammar that generates all strings composed from these terminals that are not arithmetic expressions (according to the grammar of Section 6.9).

28. In ordinary algebra, variables are single letters and hence the multiplication operator can often be omitted (for example $2z$ denotes two times z). Find a context-free grammar for polynomials made up of numbers, single-letter variables, and $+$, $-$, and $\uparrow$ operators with multiplication implied between adjacent variables or between an adjacent integer and variable.

29. Given two context-free grammars, G_1 and G_2 that generate languages L_1 and L_2, describe general procedures that will give a context-free grammar for:

a) $L_1 \cup L_2$

b) $L_1 \cdot L_2$

c) $L_1{}^*$

Use the grammars directly in your procedure.

30. Find a context-free grammar that will generate the FORMAT statement in FORTRAN (excluding the Hollerith field).

31. Find a context-free grammar for the BASIC language as described in some convenient manual. Which features of the language are not described by your grammar?

32. Find a context-free grammar that will generate S-expressions in LISP using

a) dot notation,

b) list notation,

c) LISP metalanguage.

33. Suppose that a MINI-BASIC program could only use the three line numbers 1, 2, and 3. Write a context-free grammar that generates all MINI-BASIC programs satisfying the requirements that each line begins with a distinct line number and that the line number used in a GOTO, IF, or GOSUB statement actually occurs at the beginning of some line in the program.

34. How would you determine whether the language generated by a grammar contains an infinite number of sequences?

35. How would you determine if a grammar generates any terminal sequences at all?

36. Show that if a context-free language does not contain the null string, there exists a context-free grammar for that language with no epsilon productions.

37. Give an example of a grammar with no terminal symbols, but whose language contains at least one sequence.

38. Give an example of a grammar for which each sequence in the language has an infinite number of derivation trees.

39. What property must a grammar have if every derivation tree has only one corresponding derivation?

40. a) Suppose that we write $x \Longleftrightarrow y$ if either $x \Rightarrow y$ or $y \Rightarrow x$. Also suppose that $\overset{*}{\Longleftrightarrow}$ represents zero or more uses of the relation $\Longleftrightarrow$. Give an example of a context-free grammar with starting symbol $\langle S \rangle$ for which the set of terminal strings w such that $\langle S \rangle \overset{*}{\Rightarrow} w$ is different from the set of terminal strings w such that $\langle S \rangle \overset{*}{\Longleftrightarrow} w$.

b) What would the "derivation tree" for a "derivation" using the relation $\Longleftrightarrow$ look like?

41. Show that the language $a^* + b^*$ cannot be generated by any context-free grammar with a single nonterminal.

42. Show that for each grammar there exists a constant such that every sequence in the language has a derivation with a number of steps less than the constant times the length of the sequence.

43. Find a context-free grammar that will generate all the well-formed formulas in the propositional calculus.

44. Give a procedure that will test two right-linear grammars to determine whether or not they generate the same language.

45. Draw the derivation trees for each of the following expressions according to the grammars of Section 6.9 and 6.10.

a) $I * I + I$

b) $I * (I + I)$

c) $I + (I * I) \uparrow I + I$