CHAPTER 3

SYNTACTIC ANALYSIS I

The syntactic analyzer, or *parser*, is the heart of the front end of the compiler. The parser's main task is analyzing the structure of the program and its component statements and checking these for errors. It frequently controls the lexical analyzer, which provides tokens in response to the parser's requests, and it may also supervise the operation of the intermediate code generator.

Our principal resource in parser design is the theory of formal languages. This can be defined roughly as the mathematical theory of grammars and languages. This theory was developed originally with natural languages in mind (i.e., languages like English, French, Czech, Navaho, and Chinese); it was only later that computer scientists discovered its usefulness in compiler design. We use this theory by providing a grammar for the programming language and then using the grammar to construct the parser.

To be quite accurate, I should point out that most practical programming languages cannot be completely described by the kind of grammar we will use. In a programming language there are frequently restrictions that cannot be enforced by these grammars. For example, strongly typed languages require that every variable be declared before it is used. The *context-free grammars*, which are the type we will study, are not capable of enforcing this requirement. The grammars that can enforce requirements like this are too complex and unwieldy for use in compiler construction. (We'll come back to this later.) In any case, exceptions like these are rare and can be handled simply by other means, and context-free grammars are so convenient to use and lend themselves so readily to efficient methods of parser construction that they are universally used in compiler construction.

3.1 GRAMMARS

I will provide a formal definition of a grammar presently, but for the time being, we will say that a grammar is a finite set of rules for generating an infinite set

of sentences. (We can also write grammars that generate finite languages, and we will use many such examples, but they are of little practical interest.) In natural languages, the sentences are made up of words; in programming languages, they are made up of tokens. The rules impose some desired structure on the sentences. Any sentence that can be generated by a given grammar is by definition grammatical (in the terms of that grammar, anyway); any sentence that cannot be so generated is ungrammatical.

The grammars one traditionally learns in studying foreign languages are different from the ones we will consider. (I have to say "traditionally" here, because the theory of formal languages has had some influence on modern foreign language teaching.) We use a type of grammar called a *generative grammar*; this type of grammar builds a sentence in a series of steps, starting with the abstract concept of a sentence and refining that concept until an actual sentence emerges as the result. Analyzing, or parsing, the sentence consists of reconstructing the way in which the sentence was formed. Much of the theory of generative grammars is based on the work of Noam Chomsky [1956, 1957, 1959], and we will draw on that work here.

Some examples will be useful in laying the groundwork for the formal definition. Consider the sentence, "The dog gnawed the bone." This sentence consists of a noun phrase, "the dog," and a verb phrase, "gnawed the bone." The verb phrase, in turn, can be analyzed into a verb, "gnawed," followed by another noun phrase, "the bone." Finally, each noun phrase consists of an article, "the," followed by a noun, in one case "dog" and in the other "bone."

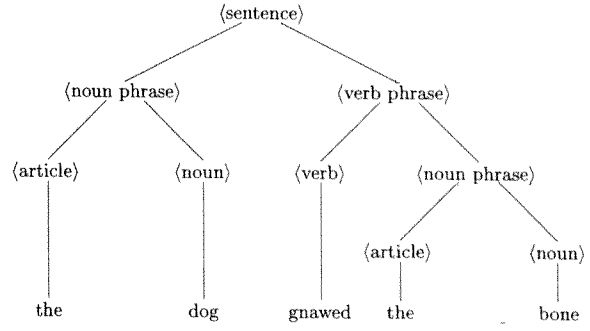

Figure 3.1

This analysis can be represented by the tree shown here. Such a structure is called a *parse tree*. The root of the tree is our starting point, a *sentence*. A sentence can take many forms; in this case, it consists of a noun phrase and a verb phrase. These are the children of the node *sentence*, as shown in this partial tree:

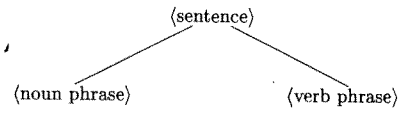

Figure 3.2(a)

When a node has two children, as here, they are read off from left to right: noun phrase, verb phrase. In similar fashion, the specific form of the verb phrase is indicated by its children, the nodes *verb* and *noun phrase*, as here:

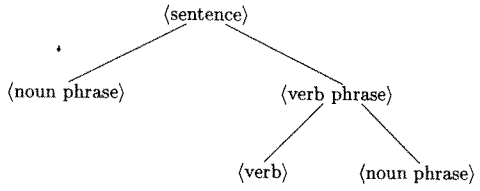

Figure 3.2(b)

The rest of the tree is constructed in the same way, and the leaves of the tree are the final sentence.

The tree lays bare the structure or organization of the sentence. Sentences are not just random strings of words like "Bone dog gnawed the the." A grammatical sentence is made up of parts, and the parts themselves are made up of smaller parts, and so on. The tree shows these parts and how they are related so we can examine them at leisure.

If you start at the root of the tree and work downward, you are retracing the steps by which the sentence was generated. It is almost as if someone were thinking, "I guess I'll speak a sentence. Let's see—how will I do it this time? I guess I'll fall back on the old noun-phrase–verb-phrase structure. Now, what should I use for the noun phrase?..." I don't really believe people go through such a process when they talk, even unconsciously. The point is that you can reconstruct the derivation of the sentence and lay bare its structure *as if* the speaker had thought of it that way.

If you start at the leaves of the tree and work upward, you are retracing the process by which a listener might understand what was being said or what was written. "Hmmm...it says, 'the.' That's usually the beginning of a noun phrase. Now we have, 'dog'; yep, that's a noun phrase, all right. Will they give me a verb next?..." and so forth. Again, whether people really analyze sentences that way is doubtful. (Part of the problem is that we learned to speak and understand as infants and no longer recall how we did it.) This is a usable model of comprehension, however, or at least of analysis, and, more to the point, we can write parsers for programming languages that proceed almost exactly along these lines.

A generative grammar is a formalization of this process. Returning to our model of speaking the sentence, we note that the starting point was the idea of a sentence. We can paraphrase our hypothetical talker's thoughts as follows:

A sentence can consist of a noun phrase and a verb phrase.

A noun phrase can consist of an article and a noun.

A verb phrase can consist of a verb and a noun phrase.

Possible nouns are "dog," "cat," "bone,"..., etc.

Possible articles are "the," "a,"..., etc.

Possible verbs are "gnawed," "saw," "walks,"..., etc.

These are, in fact, rules of English grammar. They are not a complete set, but they are enough to analyze our sample sentence.

For clarity, it is convenient to identify structural elements like *sentence*, *noun phrase*, and *verb phrase* by enclosing them in angle brackets, the way we did on the tree. Thus we write ⟨sentence⟩, ⟨noun phrase⟩, ⟨verb phrase⟩, and so on. This lets us omit the quotation marks around the actual words in the sentence; it also avoids confusion when analyzing sentences like "The sentence contains a verb." Later, for brevity, we will abbreviate these structural elements with capital letters. We will also abbreviate "can consist of" (or "can be replaced by") with the symbol →.

Using these conventions, we can say

⟨sentence⟩ → ⟨noun phrase⟩ ⟨verb phrase⟩
⟨noun phrase⟩ → ⟨article⟩ ⟨noun⟩
⟨verb phrase⟩ → ⟨verb⟩ ⟨noun phrase⟩
⟨noun⟩ → dog, cat, bone, sentence, verb, ...
⟨article⟩ → the, a, an
⟨verb⟩ → contains, gnawed, saw, walks, ...

We can parallel this entire discussion in the domain of programming languages. Consider the expression $a * b + c$. This expression consists of a smaller expression $a * b$ and a second expression c, joined by a $+$. The expression $a * b$, in turn, consists of the expression a and the expression b, joined by a $*$. We can construct a parse tree for this expression in the same way that we did for the English sentence:

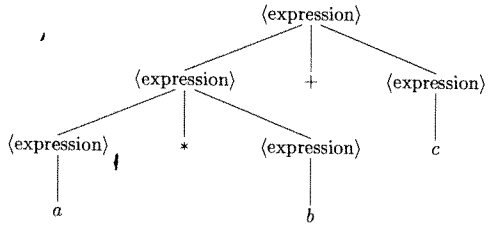

Figure 3.3

We can also prbduce a partial set of rules for forming expressions, as follows:

⟨expression⟩ → ⟨expression⟩ * ⟨expression⟩

⟨expression⟩ → ⟨expression⟩ + ⟨expression⟩

⟨expression⟩ → a, b, c, ...

These rules are called *productions* or *rewrite rules* (because ⟨sentence⟩ can be rewritten as ⟨noun phrase⟩ ⟨verb phrase⟩ and so on), and they are the heart of a generative grammar. The actual words in the sentence are called *terminal symbols*, or just *terminals*, because they are what we end up with, and the things in angle brackets are called *nonterminal symbols*, or just *nonterminals*.

3.1.1 Syntax and Semantics

Before we go on, we must draw a couple of distinctions. We must differentiate between syntax and semantics, and it is well to contrast the theoretical powers of grammars with the practical limitations found in languages.

Syntax deals with the way a sentence is put together; semantics deals with what the sentence means. Grammars define proper sentence structure, but they have no bearing on meaning. It is possible to write grammatically correct sentences that are nonsense; for example,

> The blotter subtracted the sympathy.

It is also possible to write ungrammatical sentences that make sense, for example, the famous reason for climbing Mount Everest:

> Because it was there.

This sentence lacks a main verb. In fact, it isn't really a sentence at all, only a dependent clause; but it makes sense to the listener in context because the listener mentally supplies the missing part:

> [We climbed it] because it was there.

We can supply the missing part, but the grammar won't; hence by our definition this utterance is ungrammatical even though sensible.

It is much easier to determine whether a sentence, or a program, is grammatical than whether it makes sense. The compiler, during syntactic analysis, will automatically check for grammatical errors; it is generally left to the programmer to see, during debugging, whether the program makes sense. (Later on, we will consider *semantic analysis* in the compiler, but that will turn out to cover much less ground than debugging does.)

The practical limitations come about as follows: there are grammatical sentences that nobody can utter and grammatical sentences that are almost impossible to understand. There is a story about a child who was told to write a composition 100 words long and who submitted the following:

> Perspective is what you see when you look down the railroad tracks and they get smaller and smaller and smaller and smaller and smaller and smaller and smaller and ...

and so on, out to 100 words. This is not a practical sentence, but it is unquestionably grammatical, and it could have been made arbitrarily long. A sentence that takes 1,000 years to speak cannot be spoken.

That grammatical sentences can be nearly impossible to understand must be clear to anyone who has read many textbooks, but I have something different in mind. Robinson [1975] provides this example:

> The house the cheese the rat the cat the dog chased caught ate lay in was built by Jack.

This grammatical sentence is virtually incomprehensible until you recognize the nursery jingle, "This is the house that Jack built..." and recall how it goes. The problems here are the large number of nested clauses and the fact that the subjects and verbs in all but the lowest level of nesting are separated by further levels of nesting. We could easily follow

> The house the cheese lay in was built by Jack.

and we might even be able to handle

> The house the cheese the rat ate lay in was built by Jack.

if we read it carefully.

The problem is that the mental stack we use for handling nested clauses doesn't work very well when the depth of nesting exceeds approximately two. Practical limitations inside the computer, or in the compiler itself, impose similar limits. There may be limits on the number of dimensions an array may have, for example,

or limits on the number of levels of nesting in a record or structure. We saw in Chapter 2 that regular expressions could not in themselves enforce a limit on the length of an identifier. Here, too, just as English grammatical rules permit the monstrosities shown above, programming-language grammars do not cover the consequences of practical limitations in their languages.

There are also grammatical sentences in natural languages that do not lend themselves well to analysis with the simple grammars we will consider here; I will have more to say about this later.

3.1.2 Grammars: Formal Definition

We have now seen enough background to permit a formal definition of a grammar. A grammar G is a quadruple $(T,\ N,\ S,\ R)$, where

> T is a finite set of terminal symbols,
>
> N is a finite set of nonterminal symbols (note that N and T are disjoint sets),
>
> S is a unique starting symbol ($S \in N$),
>
> R is a finite set of productions of the form $\alpha \rightarrow \beta$, where α and β are strings of nonterminals and terminals.

Nonterminals are things like ⟨sentence⟩, ⟨noun phrase⟩, ⟨expression⟩, and the like; terminals are things like "dog," "gnawed," "+," and so on. In our dog-bone example, the starting symbol was ⟨sentence⟩; in the programming example, it was ⟨expression⟩.

A production means that any occurrence of the string on the left-hand side can be replaced by the string on the right-hand side. At the moment, α will be a single nonterminal, although it does not have to be; we will come back to this later. When α is a single nonterminal, the grammar is *context-free*. These are the grammars of particular interest to us. Nonterminals are also called *syntactic categories* or *syntactic variables*. S is sometimes called the starting symbol and sometimes the *sentence symbol*. Some writers require that S be distinct from N; this requirement makes it slightly easier to construct one kind of parser, but otherwise it makes no practical difference, and we will not impose this requirement.

Sentences in the language are generated by starting with S and applying productions until we are left with nothing but terminals. (You aren't done as long as there are any nonterminals left.) The set of all sentences that can be generated in this way from a given grammar G are the *language* generated by G, written $L(G)$.

Before we go on, there are some notational details to cover. First, in talking about grammars generally, I will follow Chomsky's convention whereby nonterminals are represented by capital letters, terminals by lowercase letters early in the alphabet, strings of terminals by lowercase letters late in the alphabet, and mixtures of nonterminals and terminals by lowercase Greek letters. Thus for example we will

speak of the nonterminals A, B, and C, the terminals a, b, and c, the strings w and x, and the sentential forms α, β, and γ. (I will define sentential forms shortly.) When dealing with specific programming-language examples, however, I will write the terminals in **boldface** characters. In particular, I will use a boldface **i** for an identifier.

Second, the list of productions can be used to define the entire grammar, in much the same way that the state table was used to define an entire finite-state machine, if we agree to list the productions from the sentence symbol first. Then the nonterminals can be identified by scanning the left-hand sides of the productions and the terminals can be identified by a careful examination of the right-hand sides.

Finally, we must mention BNF notation. BNF stands for Backus-Naur Form, named for J. W. Backus (who played a leading rôle in the development of the first FORTRAN compiler) and P. Naur (who played a leading rôle in the design of Algol-60). In this notation, nonterminals are placed in angle brackets, as mentioned previously; the symbol ::= is used in place of the arrow; and when several productions share the same left-hand side, as for example,

⟨expression⟩ ::= ⟨expression⟩ + ⟨expression⟩

⟨expression⟩ ::= (⟨expression⟩)

⟨expression⟩ ::= **i**

we write

⟨expression⟩ ::= ⟨expression⟩ + ⟨expression⟩ | (⟨expression⟩) | **i**

where the vertical line has its familiar meaning of "or." We will make free use of the vertical line but will otherwise follow Chomsky's notation and use the arrow. You should be aware that some writers use the term "BNF grammars" as a synonym for "context-free grammars."

Extended BNF includes two constructs useful in defining practical programming-language grammars concisely. An indefinite number of repetitions is indicated by enclosing a sentential form in curly braces. For example, in the syntax for Turbo Pascal [Borland, 1985], the syntax for the `var` statement goes, in part (and slightly paraphrased),

⟨variable-declaration-part⟩ ::= `var` ⟨variable-declaration⟩
{; ⟨variable-declaration⟩} ;

The curly braces function like the Kleene closure, in that they mean that their contents can be repeated zero or more times.

The second construct uses square brackets to indicate an optional element. For example, an integer constant might be defined as

⟨integer-constant⟩ ::= [+ | −] ⟨digit⟩ {⟨digit⟩}

Here the brackets indicate that the sign before the number is optional.

We will not use BNF in our discussion of parsing, but it is well to be familiar with the notation, because it is widely used in descriptions of programming languages.

3.1.3 Parse Trees and Derivations

Now let us return to expressions in a programming language. One simple grammar for expressions is

$$
\begin{aligned}
& G_E = (T,\ N,\ S,\ R), \\
\text{where } & T = \{\mathbf{i},\ +,\ -,\ *,\ /,\ (,\ )\}, \\
& N = \{E\}, \\
& S = E, \\
& R = \{E \to E + E, \\
& \quad E \to E - E, \\
& \quad E \to E * E, \\
& \quad E \to E/E, \\
& \quad E \to (\ E\ ), \\
& \quad E \to \mathbf{i}\},
\end{aligned}
$$

and where E stands for ⟨expression⟩. (This is the first and last time I will use the full $(T,\ N,\ S,\ R)$ formalism in specifying a grammar instead of just listing the productions; but you should see the formal way at least once.)

Using this grammar, let us analyze the expression $(a + b)/(a - b)$. The lexical analyzer will pass this expression to us in tokenized form, representing the lexemes a and b with the token i. Hence our terminal string is $(\mathbf{i} + \mathbf{i})/(\mathbf{i} - \mathbf{i})$. We recognize this expression as a fraction whose numerator is $(\mathbf{i} + \mathbf{i})$ and whose denominator is $(\mathbf{i} - \mathbf{i})$. Since the expression is basically a fraction, the first production we use is $E \to E/E$:

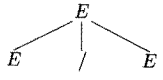

Figure 3.4(a)

Note that every production is represented by making the left-hand side the parent node and the symbols on the right-hand side the children. The numerator and denominator are both parenthesized expressions, so we replace both of these new E's using the production $E \to (\ E\ )$:

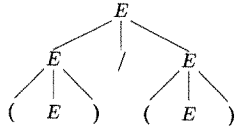

Figure 3.4(b)

The contents of the numerator parentheses is a sum; hence we replace the E inside its parentheses using the production $E \to E + E$. The denominator is a difference,

so we replace the E inside its parentheses using the production $E \to E - E$. The remaining E's give us our identifiers via the production $E \to \mathbf{i}$:

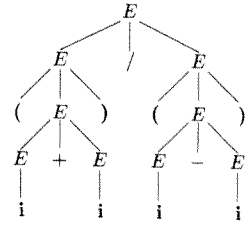

Figure 3.4(c)

There is another way in which we can represent this process. We can simply write the results of applying each production as a series of steps. We use the notation

$$\alpha_1 \Rightarrow \alpha_2$$

to show the result of applying a single production to α_1. For example, in the second level of the tree above, we have E/E. When we apply the production $E \to (\ E\ )$ to the right-hand E, we obtain $E/(E)$, and we write

$$E/E \underset{G_E}{\Rightarrow} E/(E)$$

The subscript G_E specifies the grammar we are using and is omitted whenever there is no question of which grammar is being used.

With this notation, we may write

$$
\begin{aligned}
E &\Rightarrow E/E && \text{[using production } E \to E/E\text{]} \\
&\Rightarrow E/(E) && \text{[using production } E \to (\ E\ )\text{]} \\
&\Rightarrow E/(E - E) && \text{[using production } E \to E - E\text{]} \\
&\Rightarrow E/(E - \mathbf{i}) && \text{[using production } E \to \mathbf{i}\text{]} \\
&\Rightarrow E/(\mathbf{i} - \mathbf{i}) && \text{[using production } E \to \mathbf{i}\text{]} \\
&\Rightarrow (E)/(\mathbf{i} - \mathbf{i}) && \text{[using production } E \to (\ E\ )\text{]} \\
&\Rightarrow (E + E)/(\mathbf{i} - \mathbf{i}) && \text{[using production } E \to E + E\text{]} \\
&\Rightarrow (E + \mathbf{i})/(\mathbf{i} - \mathbf{i}) && \text{[using production } E \to \mathbf{i}\text{]} \\
&\Rightarrow (\mathbf{i} + \mathbf{i})/(\mathbf{i} - \mathbf{i}) && \text{[using production } E \to \mathbf{i}\text{]}
\end{aligned}
$$

This does not reveal the structure of the expression at a glance the way the parse tree does, but it shows how the initial sentence symbol gradually grows into the final sentence as each production is applied. This representation is called a *derivation*. Each step of the derivation shows the result of applying exactly one production, with

the right-hand side of the production written in place of the symbol being replaced. Given a derivation, we can always go through it step by step and construct the corresponding parse tree, and given the tree, we can similarly construct a derivation. Each step shows the result of applying a single production. Occasionally it is useful to indicate the result of applying several productions; in that case, we borrow the Kleene star and write

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

which tells us that we can get from α to β in 0 or more steps—that is, by applying 0 or more productions. The star notation gives us a succinct definition of $L(G)$. Since the language is the set of all strings the grammar G can generate, we have

$$L(G) = \{w \mid S \overset{*}{\underset{G}{\Rightarrow}} w\}$$

That is, $L(G)$ is the set of all strings of terminals derivable from S using G.

Notice that productions are written with single arrows, but derivation steps are separated by double arrows. This is standard. The strings of symbols appearing in the various derivation steps are called *sentential forms*. A sentential form of a grammar G is any sequence of terminals and nonterminals that can occur in a derivation in G; formally, it is any sequence α such that $\alpha \in (N \cup T)^*$ and

$$S \overset{*}{\underset{G}{\Rightarrow}} \alpha.$$

3.1.4 Rightmost and Leftmost Derivations

In every step of the derivation shown above, we singled out the rightmost nonterminal for replacement. Thus in the sentential form $E/(E - E)$, we had a choice of three E's to replace; we chose the rightmost one. Such a derivation is called a *rightmost derivation*. Alternatively, we could have selected the leftmost nonterminal in every step, instead; this would have resulted in a *leftmost derivation*. The leftmost derivation for our sample expression goes

$$\begin{array}{lll}
E & E/E & [\text{using production } E \to E/E] \\
\Rightarrow & (E)/E & [\text{using production } E \to (\ E\ )] \\
\Rightarrow & (E + E)/E & [\text{using production } E \to E + E] \\
\Rightarrow & (\mathbf{i} + E)/E & [\text{using production } E \to \mathbf{i}] \\
\Rightarrow & (\mathbf{i} + \mathbf{i})/E & [\text{using production } E \to \mathbf{i}] \\
\Rightarrow & (\mathbf{i} + \mathbf{i})/(E) & [\text{using production } E \to (\ E\ )] \\
\Rightarrow & (\mathbf{i} + \mathbf{i})/(E - E) & [\text{using production } E \to E - E] \\
\Rightarrow & (\mathbf{i} + \mathbf{i})/(\mathbf{i} - E) & [\text{using production } E \to \mathbf{i}] \\
\Rightarrow & (\mathbf{i} + \mathbf{i})/(\mathbf{i} - \mathbf{i}) & [\text{using production } E \to \mathbf{i}]
\end{array}$$

While it may be possible to construct many different derivations from the same parse tree, the leftmost and rightmost derivations are unique.

Any sentential form occurring in a leftmost derivation is termed a *left sentential form*, and any sentential form occurring in a rightmost derivation is termed a *right sentential form*.

The distinction between leftmost and rightmost derivations is not merely academic. When we come to consider specific parsers, we will discover that one type results in a leftmost derivation and the other in a rightmost derivation, and the differences between these two types directly impact the details of constructing the parsers and of their operation.

3.1.5 Ambiguous Grammars

We can learn something else from this same expression grammar G_E. Consider the expression $\mathbf{i} + \mathbf{i} * \mathbf{i}$. We can parse this by using the production $E \to E * E$, then applying the production $E \to E + E$, and finally applying $E \to \mathbf{i}$ to get the tree shown in part (a) of Figure 3.5. But we could also start by applying $E \to E + E$, then $E \to E * E$, and finally $E \to \mathbf{i}$ to get part (b) of Figure 3.5. Now these are completely different parse trees. Which one is right?

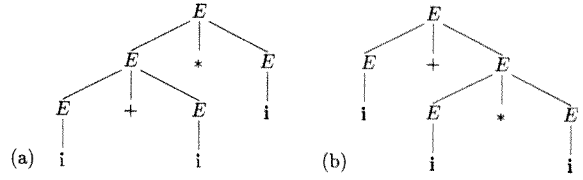

Figure 3.5

That question cannot be answered if we have nothing but the grammar to go on. Both trees were formed by using the productions in the grammar, and there is no reason why both should not be acceptable. A grammar in which it is possible to parse even one sentence in two or more different ways is *ambiguous*. A language for which no unambiguous grammar exists is said to be *inherently ambiguous*. (Natural languages are ambiguous, and we use this ambiguity in constructing plays on words.)

This ambiguity, if it is not resolved somehow, is unacceptable in a compiler. There are two ways of dealing with an ambiguous grammar. First, when we look outside the grammar and consider the operator-precedence rules for Pascal, or for most high-level languages, we see that only one of these trees can be right. The left-hand tree says, in effect, that the expression is basically a product and that one of the factors is a sum. The right-hand tree says that the expression is basically a sum and one of the terms is a product; this is the normal Pascal interpretation of this expression. If we can somehow incorporate into the parser the fact that

multiplication has a higher precedence than addition, this will resolve the ambiguity. We will consider a parser in which we do exactly that.

The second alternative is to rewrite G_E in such a way as to eliminate the ambiguity. For example, if we distinguish among *expressions*, *terms*, and *factors*, we can come up with this alternative grammar:

$$E \rightarrow E + T \mid E - T \mid T$$
$$T \rightarrow T * F \mid T \,/\, F \mid F$$
$$F \rightarrow (\ E\ ) \mid \mathrm{i}$$

Here T stands for ⟨term⟩ and F for ⟨factor⟩. Using this grammar, our example can be parsed in only one way, as shown below. We will see this grammar again in at least one parser.

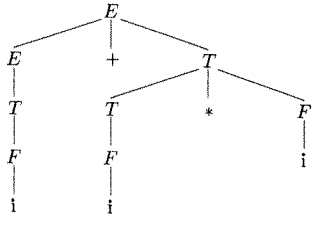

Figure 3.6

3.1.6 The Chomsky Hierarchy

The form of the productions has a profound effect on the kind of language generated by the grammar. If a production is written as $\alpha \rightarrow \beta$, then we may distinguish four general types, depending on what forms α and β are permitted to take. Each of these types generates a different family of languages, and for each family there is a machine that is capable of generating or recognizing the languages. These types form a hierarchy, in the sense that the set of languages generated by Type k grammars is a proper subset of the languages generated by Type $(k-1)$ grammars. This hierarchy is known as the Chomsky hierarchy. The languages generated by Type k grammars are called Type k languages. It is good to know about this hierarchy, because it helps to place the grammars we use for programming languages in a larger context.

In **Type 0** grammars, α and β can be any strings of symbols; the sole restriction is that α may not be the null string. Normally, α always includes at least one nonterminal, although Chomsky [1959, p. 142] does not rule out the possibility of productions in which terminals are replaced by other terminals, as for example $ab \rightarrow cd$. A typical production, however, is of the form $\gamma A \delta \rightarrow \gamma \beta \delta$. Here A is some nonterminal; γ and δ are the left and right contexts, respectively, of A. Since the entire left-hand side is replaced by the entire right-hand side when this production is used, the effect of the contexts γ and δ is that you may replace A by β only if it appears in the indicated context. It takes a Turing machine to recognize Type 0 languages. Type 0 grammars are also known as *unrestricted grammars*, *phrase-structure grammars*, or *semi-Thue grammars* (for Axel Thue, a Norwegian mathematician; Thue is pronounced "two-way").

In **Type 1** grammars, the productions take the same form $\gamma A \delta \rightarrow \gamma \beta \delta$ as in Type 0 grammars, except that β may not be ϵ. As a result of this restriction, the sentential forms in the derivation steps always grow longer (or at any rate, never grow shorter) as you go through the derivation. Type 1 grammars are known as *context-sensitive grammars*; Type 1 languages can be recognized by a special, simpler type of Turing machine known as a linear bounded automaton.

In **Type 2** grammars, the left-hand side of the production is a single nonterminal. Thus productions take the form $A \rightarrow \beta$. Since there is no context specified, this means that we may replace A by β anywhere we like, and Type 2 grammars are the *context-free grammars* (CFGs). These are the grammars we will use in syntactic analysis; all the examples of grammars shown so far in this chapter are context free.

It takes a stack automaton to recognize context-free languages. A stack automaton is a FSA equipped with temporary storage in the form of a pushdown stack. We will see that every parser we study uses a stack, either explicitly as provided in the program or implicitly as a result of recursive procedure or function calls. To recognize the full set of context-free languages, we require a *nondeterministic* stack automaton. We saw that any NFA could be replaced by an equivalent DFA; but we cannot do this in the case of nondeterministic stack automata. There is an important subset of the Type 2 languages, however, known as *deterministic context-free languages*, which can be handled by a deterministic stack automaton. It is our good fortune that practical programming languages can be adequately described by deterministic CFGs.

We said that in Type 1 grammars, the nonterminal on the left-hand side could not be replaced by ϵ. Since these restrictions are cumulative as we go from one level of the hierarchy to the next, this implies that Type 2 grammars may not have productions of the form $A \rightarrow \epsilon$. In fact, we will see many CFGs having such productions; we must allow such productions as exceptions to the Type-1 restriction.

In **Type 3** grammars, the right-hand side of every production may be only (a) a single terminal or (b) a single nonterminal followed by a single terminal. These are the *regular grammars*. The languages they generate, *regular languages*, are exactly the languages generated by the regular expressions we considered in Chapter 2. It follows that regular languages can be recognized by FSAs. There are, in fact, simple ways of constructing a regular grammar G from a machine M, or a machine M from a regular grammar G, such that $L(G) = L(M)$, but these are beyond the scope of this book.

3.1.7 Some Context-Free and Non-Context-Free Languages

We saw in Chapter 2 that L_P, the language consisting of all legal sets of parentheses, was not regular. Here is a CFG that generates L_P; let us call it G_P:

$$\begin{aligned} S &\rightarrow SS \\ &\mid (S) \\ &\mid () \end{aligned}$$

It should be clear that any string derivable from S will begin with a left parenthesis and that all parentheses will be properly paired. The leftmost derivation for the string (())((())())() is as follows:

$$\begin{aligned} S &\Rightarrow SS \\ &\Rightarrow SSS \\ &\Rightarrow (S)SS \\ &\Rightarrow (())SS \\ &\Rightarrow (())(S)S \\ &\Rightarrow (())(SS)S \\ &\Rightarrow (())((S)S)S \\ &\Rightarrow (())((())S)S \\ &\Rightarrow (())((())())S \\ &\Rightarrow (())((())())() \end{aligned}$$

The language $L_{abc} = \{a^n b^n c^n,\ n > 0\}$ is not context-free. A formal proof of this would take us too far afield, but I can provide a plausibility argument. Imagine how the parse tree for such a string would look. It would probably consist of three main parts, as shown:

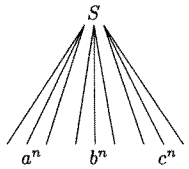

Figure 3.7

Each of these parts would grow as a result of applying productions. But the mechanism of CFGs provides no way of coördinating the growth of these three parts. There is no way of forcing the part generating a^n to produce a string of a's of the same length as the strings of b's and c's generated by the other two parts.

There is a well-known Type 1 (context-sensitive) grammar[1] for this language:

$$\begin{aligned} S &\rightarrow aSBC \\ S &\rightarrow abC \\ CB &\rightarrow BC \\ bB &\rightarrow bb \\ bC &\rightarrow bc \\ cC &\rightarrow cc \end{aligned}$$

A derivation of *aaaabbbbcccc* follows:

$$\begin{aligned} S &\Rightarrow aSBC \quad \Rightarrow aaSBCBC \quad \Rightarrow aaaSBCBCBC \\ &\Rightarrow aaaabCBCBCBC \end{aligned}$$

These first lines generate the required number of a's. We now use the $CB \rightarrow BC$ production to shift the B's toward the left so that they can add to the b's:

$$\begin{aligned} &\Rightarrow aaaabBCCBCBC \Rightarrow aaaabbCCBCBC \Rightarrow aaaabbCBCCBC \\ &\Rightarrow aaaabbBCCCBC \Rightarrow aaaabbbCCCBC \Rightarrow aaaabbbCCBCC \\ &\Rightarrow aaaabbbCBCCC \Rightarrow aaaabbbBCCCC \Rightarrow aaaabbbbCCCC \end{aligned}$$

It now remains only to convert the C nonterminals to c's using the last two productions:

$$\begin{aligned} &\Rightarrow aaaabbbbcCCC \quad \Rightarrow aaaabbbbccCC \quad \Rightarrow aaaabbbbcccC \\ &\Rightarrow aaaabbbbcccc \end{aligned}$$

Another language that is not context-free is $L_2 = \{wcw \mid w \in (T - c)^*\}$, the language in which every sentence consists of two identical copies of a string w separated by a center marker c. Again, the CFG structure has no way of coördinating the growth of the two copies so that they will be identical. At the beginning of this chapter, I said that a CFG could not enforce the requirement that the every variable be declared before it is used. The declaration and use of a variable will always be separated by other code. If we regard the c in wcw as representing that other code, it should not require too great a stretch of the imagination to see that a grammar that enforces this requirement must be at least as powerful as a grammar generating L_2 and therefore cannot be context-free. We must enforce the requirement by means of a collaboration among the parser, the semantic analyzer, and the symbol-table handler.

Notice that if we let w^R represent the reversal of w, then we have $L_R = \{wcw^R \mid w \in (T - c)^*\}$. In Chapter 2, we saw that this language was not regular. But L_R is context-free; if $T = \{a, b, c\}$, a grammar for this language is

[1] Strictly speaking, this grammar is not context-sensitive since the production $CB \rightarrow BC$ violates the definition. This production can be rewritten, in a more lengthy form, so as to make it conform to the definition; see Denning, Dennis, and Qualitz [1978], p. 63.

$S \rightarrow aSa \mid bSb \mid c$

This grammar grows the strings from the outside in, and the matching a's and b's guarantee that the substring to the right of c will be the reverse of the substring to the left.

3.1.8 More about the Chomsky Hierarchy

There is a certain amount of history associated with Chomsky's hierarchy. In 1949, Claude Shannon, in his classic work on information theory, showed that you could get very convincing approaches to natural languages by random choices of letters or words, where the probability of choosing any particular symbol depended on which symbols had been chosen previously. The reader was left with the impression that if you gathered enough statistics over a large enough body of sentences, so that the probabilities could look far enough back into the past history of the sentence, you could generate natural languages this way.

Chomsky, in 1956, pointed out that the Shannon model was equivalent to a NFA and demonstrated that there were constructions in natural languages that could not be generated by such a machine but could be generated by CFGs. Chomsky has had a wide influence in linguistic circles, but his work is not accepted universally (see, for example, Hall [1979]). Furthermore, the finite-state model has been successfully used in programs for computer recognition of spoken sentences [Lowerre, 1976]. But these programs have severe limitations in scope (no 1,000-year sentences for them!). In any case, the usefulness of generative grammars in compiler construction alone would be enough to guarantee their acceptance among computer scientists.

Chomsky's generative grammars steadfastly turn their backs on any question of meaning in the sentence to be parsed. This refusal to consider meaning may well be the central dogma of structural linguistics. (See, for example, Gleason [1961].) Many of us, as children, were taught that a *noun* was the name of a person, place, or thing. This definition is frowned upon by structural linguists: nouns should be defined solely by their structural function in the language, not by their meaning. This dogma is not universally accepted, and surely many of us have found, when laboring through a text in a foreign language, that we cannot tell what the structural function of a word is if we don't know what it means. But in compiler writing, this refusal to consider meaning plays right into our hands. It would be prohibitively complicated to explain to the compiler the meaning of everything in the program, especially the programmer's variables. With parsers based on structural considerations alone, we do not have to do this.

Chomsky believes that the generative model reflects something that is wired into the human brain—that the innate human language-learning capability is in some way based on the generative model. Whether he would go so far as to claim that the unconscious ruminations of the speaker follow the lines given in our dog-bone example is unclear, but the fact that all known languages can be analyzed, up to a point, by generative grammars suggests that the generative model is a universal of

language and hence is built into the human brain. You can read sentences like these on this page, which have never been written before by anybody, and the generative model permits you to understand them immediately.

This is a bold conjecture, but I can offer a bolder one: I suspect that the generative model is not merely wired into the human mind but is an inherent aspect of communication among humans or any other communicating entities. I believe it will eventually be found that if we have a potentially infinite set of messages that must be transmitted in such a way that the receiver will be able to analyze and understand a message that it has never received before, we will be forced to use a system that is based on a generative grammar. If it's wired into the human brain, that's because it's the only way that will work. I have no proof that this is so; but we have yet to see a proof of Chomsky's conjecture, as far as that goes. If you're going to conjecture, you may as well do it on the grand scale.

Of the grammars in this hierarchy, Type 2 are of greatest interest to us. CFGs can define most of the rules required in programming languages, and the few things that can't be defined by CFGs are easily managed by other means. We can handle regular languages in lexical analysis well enough by the DFA model, so we have little need for Type 3 grammars. Types 0 and 1 are less well understood; there are no simple ways of constructing parsers for them, and parsers for these languages are slow.

Furthermore, the lack of restrictions on Type 0 and Type 1 productions makes it difficult to relate the structure of the productions to that of the resulting language. A CFG, on the other hand, tells us a great deal about the structure of the language. For example, G_P tells us all we need to know about the ways in which legal sets of parentheses are formed. The productions tell us that parentheses come in left-right pairs, that we may place a legal set of parentheses inside of parentheses, and that we may concatenate two sets of legal parentheses. Indeed, if you were to explain to someone how legally formed sets of parentheses were constructed, your explanation might well be a paraphrase of our grammar. This is a large part of the appeal of CFGs: in practical situations, the productions give us a good idea of what to expect in the language.

Contrast this with the context-sensitive grammar for L_{abc}. There is no way, short of experimenting with it, of telling what characterizes the language generated by this grammar. The productions give us the strings we want, but it is not clear why they do. We can readily associate a meaning with each production in a CFG, but we can rarely do so with the productions in Type 0 or Type 1 languages.

In addition, we will see that, for practical programming languages, there is a close relationship between the productions in a CFG and the corresponding computations to be carried out by the program being parsed. This is the basis of *syntax-directed translation*, which will be our principal tool in intermediate code generation. There is no such clear correspondence in the case of Type 0 and Type 1 grammars.

The relative clarity of CFGs accounts for their use in lingustics. As in our parentheses example, the productions in a natural-language grammar are paraphrases of

a verbal description of how grammatical sentences are formed; we saw that at the very start with our dog-bone example. CFGs are not, in fact, complete descriptions of natural languages; if we transform that very first example into a question,

Did the dog gnaw the bone?

or into a passive-voice form,

The bone was gnawed by the dog.

we get into difficulties that CFGs cannot handle. It is conceivable that we could construct a Type 0 or Type 1 grammar that would embrace these other forms as well, but in so doing we would lose the clarity of the CFG.

Chomsky's solution to this problem was to extend the CFG with a set of *transformation rules*, which, for example, tell how to transform a declarative sentence into a question. The question and the passive-voice form are *surface structures* and the underlying declarative form is known as the *deep structure*. Such grammars are known as *transformational grammars*; they solve some of the difficulties of CFGs, but not all of them. (See, for example, Schwarcz [1967] or McCawley [1988].) Transformational grammars are better adapted for generating sentences than for parsing, and parsers for transformational grammars tend to proceed by trial and error and to be very slow and of exponential complexity.

One attractive alternative to transformational grammars is the *recursive transition network* of Woods [1970]. This takes the form of a number of FSAs that have the power to call one another recursively. They possess the power of a transformational grammar but are easier to understand and lead to efficient parsing methods. We do not need tools of this complexity in compilers, however.

3.2 TOP-DOWN PARSERS

We will consider two general types of parser, the top-down parser and the bottom-up parser. A top-down parser starts at the root of the parse tree and tries to reconstruct the growth of the tree that led to the given token string. In so doing, it reconstructs a leftmost derivation. A bottom-up parser starts at the leaves of the parse tree and tries to work backward toward the root. In so doing, it reconstructs a rightmost derivation; indeed, the steps in the parse result in the sentential forms of the derivation, in reverse order.

The top-down parser must start at the root of the tree and determine, from the token stream, how to grow the parse tree that results in the observed tokens. It must do this from nothing but a knowledge of the incoming tokens and of the productions in the grammar. Furthermore, as a practical matter, it must scan the incoming tokens from left to right.

This approach runs into several problems; I will address the problems as we come to them and will gradually develop the general method. We will see how

top-down parsing can be done with a set of recursive procedures or functions and will finally study a table-driven top-down parser that is easy to construct and to modify.

3.2.1 Left Recursions

The left-to-right scan gets us into trouble right away. Suppose our grammar is

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow \mathtt{i} \mid (\ E\ )$

and suppose we are parsing the expression i + i + i. The parse tree for this is

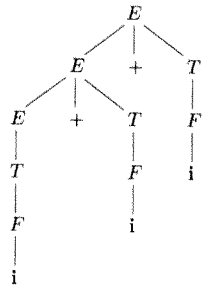

Figure 3.8

A top-down parser for this grammar will start by trying to expand E with the production $E \rightarrow E + T$. It will then try to expand the new E in the same way. This gives us the tree

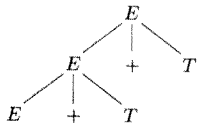

Figure 3.9(a)

which is correct so far. By referring to the original tree, which we constructed by hand, we know that the latest E should be expanded with $E \rightarrow T$ instead of with yet another use of $E \rightarrow E + T$. But how will the parser know this? It has nothing to go on but the grammar and the input token string, and nothing in the input has changed so far. Because of this, there is no way to prevent it from trying to grow the parse tree,

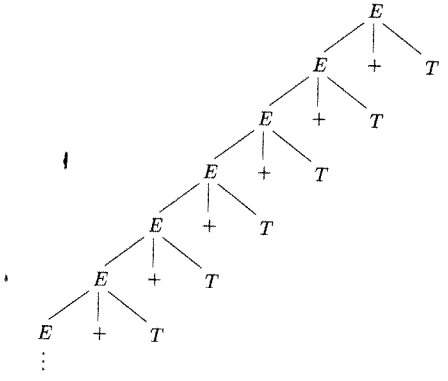

Figure 3.9(b)

There is no solution to this problem as long as the grammar is in its present form. Productions of the form

$$A \to A\alpha$$

are *left-recursive* productions, and no top-down parser can handle them. To see why this must be so, note that the parser proceeds by eating tokens. Each token guides the parser in its choice of actions; when the token is used up, a new token is at hand, and this causes the parser to make a different move. (In this particular case, function E chose to try the right-hand side $E + T$ first. It is left as an exercise to see what happens if the parser tries the right-hand side T first, instead.) Tokens get eaten when they are matched to terminals in the productions. In the present case, the repeated use of $E \to E + T$ uses no tokens; hence on every new move, the parser is presented with the same input string and will make the same move.

This problem afflicts all top-down parsers, however implemented. The solution is to rewrite the grammar in such a way as to eliminate the left recursions. We have two types to worry about: *immediate* left recursions, where the productions are of the form

$$A \to A\alpha$$

and *nonimmediate* left recursions, where there are productions are of the form

$$A \to B\alpha \mid \cdots$$
$$B \to A\beta \mid \cdots$$

In the latter case, A will use $B\alpha$, B will use $A\beta$, and so forth; the parser will try to construct a tree:

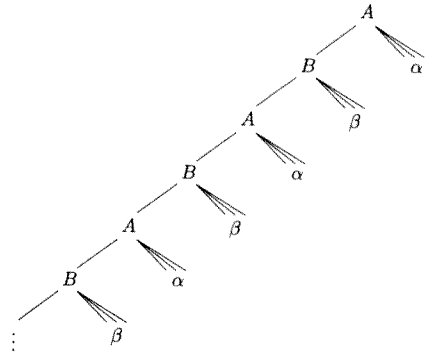

Figure 3.10

I will discuss removing immediate left recursions first, since this is in a sense the basic operation; then I will describe the general technique that removes all kinds of left recursions and uses the immediate technique in the process. To remove immediate left recursions, we use the following procedure. Note that only *left* recursion causes trouble in a top-down parser. A production of the form $B \to \alpha B\gamma$ or $B \to \alpha B$ is not left-recursive and does not require treatment.

For each nonterminal in the grammar,

1. Separate the left-recursive productions from the others. (If there are no left-recursive productions, nothing need be done.) Suppose the productions are
   $$A \to A\alpha_1 \mid A\alpha_2 \mid A\alpha_3 \mid \cdots \quad \text{(left-recursive)}$$
   $$A \to \delta_1 \mid \delta_2 \mid \delta_3 \mid \cdots \quad \text{(non-left-recursive).}$$
2. Introduce a new nonterminal A'.
3. Change non-left-recursive productions on A to
   $$A \to \delta_1 A' \mid \delta_2 A' \mid \delta_3 A' \mid \cdots$$
4. Remove left-recursive productions on A and substitute
   $$A' \to \epsilon \mid \alpha_1 A' \mid \alpha_2 A' \mid \alpha_3 A' \mid \cdots$$
   (What we have done is remove A from α_1 and append A' at the end: from $A\alpha_1$ we went to $\alpha_1 A'$, and similarly for α_2, and so on.) The corresponding left-recursive productions are removed from A.

To see why this works, consider the simple case of a grammar G:

$$S \to Sa \mid b$$

All derivations in this grammar take the form

$$S \Rightarrow Sa \Rightarrow Saa \Rightarrow Saaa \Rightarrow \cdots \Rightarrow ba^*$$

(The process terminates when you choose $S \rightarrow b$.) The transformed grammar G' is

$$\begin{aligned} S &\rightarrow bS' \\ S' &\rightarrow aS' \mid \epsilon \end{aligned}$$

and all derivations in this grammar take the form

$$S \Rightarrow bS' \Rightarrow baS' \Rightarrow baaS' \Rightarrow baaaS' \Rightarrow \cdots \Rightarrow ba^*$$

Here the process terminates when you chose $S \rightarrow \epsilon$. Hence $L(G) = L(G')$ and the grammars are equivalent.

Example 1: Suppose we have

$$A \rightarrow Ac \mid Ad \mid e \mid f$$

Here the left-recursive productions are

$$A \rightarrow Ac \mid Ad$$

and the non-left-recursive ones are

$$A \rightarrow e \mid f$$

Add the nonterminal A'. In the non-left-recursive productions, change e to eA' and f to fA'. Move the left-recursive productions down to A'; right-hand sides of A' are ϵ, cA', and dA'. So the revised version is

$$\begin{aligned} A &\rightarrow eA' \mid fA' \\ A' &\rightarrow \epsilon \mid cA' \mid dA' \end{aligned}$$

Notice that although A originally had two left-recursive right-hand sides, we are able to take care of both of them with only one new nonterminal.

The removal of immediate left recursions entails the introduction of productions of the form $A \rightarrow \epsilon$. When we discussed the Chomsky hierarchy, we mentioned that the Type 1 restriction, that the right-hand side could not be shorter than the left-hand side, would sometimes be violated in CFGs. Here you have the violation and the reason for it.

To remove *all* left recursions in a grammar, we proceed as follows. (This technique is based on Aho, Sethi, and Ullman [1986].)

1. Make a list of all nonterminals (the sequence occurring in the list of productions will usually do).
2. Go through the nonterminals in order. For each nonterminal,

   a. Examine productions. If the right-hand side begins with a nonterminal earlier in the list (e.g., $B \rightarrow A\beta$), look at productions on A. If these are

      $A \rightarrow \gamma_1 \mid \gamma_2 \mid \gamma_3 \mid \cdots$ then substitute
      $B \rightarrow \gamma_1\beta \mid \gamma_2\beta \mid \gamma_3\beta \mid \cdots$

      (If the right-hand side begins with anything else, leave it unchanged.) Continue this step until no right-hand side begins with a nonterminal earlier in the list.

   b. After all productions are examined, remove the *immediate* left recursions (if any) for this nonterminal.

Note: In Step 2a, if the productions for A were revised previously, use the revised versions, not the original versions.

Example 2: Remove all left recursions from this grammar:

$$\begin{aligned} S &\rightarrow aA \mid b \mid cS \\ A &\rightarrow Sd \mid e \end{aligned}$$

The productions from S require no treatment. For the nonterminal A, we have a right-hand side beginning with S. So we change the productions to

$$A \rightarrow aAd \mid bd \mid cSd \mid e$$

The new S appearing in this revised right-hand side is not left-recursive and will give us no trouble.

What we are doing here is embodying the initial steps of all possible leftmost derivations from A via S in the new right-hand sides. We can do this because we can get the following sentential forms from A:

$$A \Rightarrow Sd \Rightarrow \begin{cases} aAd \Rightarrow \cdots \\ bd \\ cSd \Rightarrow \cdots \end{cases}$$

In rewriting the productions from A, we are simply bypassing the step involving Sd.

Example 3: Remove left recursions from this grammar:

$$\begin{aligned} S &\rightarrow A \mid B \mid Sc \mid dS && (1) \\ A &\rightarrow Bd \mid cA \mid f && (2) \\ B &\rightarrow Se \mid Ad \mid g && (3) \end{aligned}$$

1. S: This is the first nonterminal in the list, so there can be no nonimmediate left recursions. Here dS is right-recursive: no problem. But Sc is left-recursive. Let the new nonterminal be P; new productions are

$$S \to AP \mid BP \mid dSP \quad (4a)$$
$$P \to \epsilon \mid cP \quad (4b)$$

2. A: No problems. (No right-hand side begins with S; cA is right-recursive.)
3. B: S and A are earlier in list. Fixing Se gives us
$$B \to APe \mid BPe \mid dSPe \mid Ad \mid g \quad (5)$$
4. Fixing APe and Ad gives us
$$B \to BdPe \mid cAPe \mid fPe \mid BPe \mid dSPe \mid Bdd \mid cAd \mid fd \mid g \quad (6)$$
(Notice that we used the revised right-hand sides from (4a), not the original ones.) Of these, $BdPe$, BPe, and Bdd are immediate left recursions. The new nonterminal is Q; new productions are
$$B \to cAPeQ \mid fPeQ \mid dSPeQ \mid cAdQ \mid fdQ \mid gQ \quad (7a)$$
$$Q \to \epsilon \mid dPeQ \mid PeQ \mid ddQ \quad (7b)$$
5. The final grammar is made of (4a and b), (2), and (7a and b):
$$S \to AP \mid BP \mid dSP \quad (4a)$$
$$P \to \epsilon \mid cP \quad (4b)$$
$$A \to Bd \mid cA \mid f \quad (2)$$
$$B \to cAPeQ \mid fPeQ \mid dSPeQ \mid cAdQ \mid fdQ \mid gQ \quad (7a)$$
$$Q \to \epsilon \mid dPeQ \mid PeQ \mid ddQ \quad (7b)$$

You should satisfy yourself that $Q \to PeQ$ will not get a top-down parser into trouble even though P is earlier in the list.

Example 4: Remove all left recursions from the following grammar:

$$E \to E + T \mid E - T \mid T$$
$$T \to T * F \mid T/F \mid F$$
$$F \to \mathrm{i} \mid (\ E\ )$$

Here we have a practical grammar for generating four-function expressions. In this grammar there are no nonimmediate left recursions. For the E-productions, we introduce a new nonterminal Q:

$$E \to TQ$$
$$Q \to +TQ \mid -TQ \mid \epsilon$$

Similarly, for the T productions, we introduce a new nonterminal R:

$$T \to FR$$
$$R \to *FR \mid /FR \mid \epsilon$$

The productions from F require no change. The revised grammar is

$$E \to TQ$$
$$Q \to +TQ \mid -TQ \mid \epsilon$$
$$T \to FR$$
$$R \to *FR \mid /FR \mid \epsilon$$
$$F \to \mathrm{i} \mid (\ E\ )$$

We will have occasion to use this revised grammar in top-down parsers later.

3.2.2 Backtracking

One way to carry out a top-down parse is simply to have the parser try all applicable productions exhaustively until it finds a tree. This is sometimes called the brute-force method. It identifies the applicable productions, using the forthcoming input token as a guide. This can lead to some tricky problems. For example, consider the grammar $G_1 =$

$$S \to ee \mid bAc \mid bAe$$
$$A \to d \mid cA$$

and the token string, *bcde*. If the parser tries all productions exhaustively, it will start by considering $S \to bAc$, since the initial b in the input rules out the right-hand side ee. The next input symbol is c; this rules out $A \to d$, so it tries $A \to cA$. This will ultimately lead to the tree

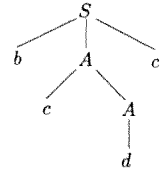

Figure 3.11

At this point, the parser has generated a complete tree; there are no nonterminals left to expand. But this is the wrong tree! It gives us the string *bcdc* instead of *bcde*. Obviously this is because the parser started with the wrong right-hand side. If it had been able to look ahead to the final input symbol, it would not have made this mistake, but parsers scan the input from left to right. To undo the damage, it must now *backtrack* until it can find another alternative production. In this case, it must back up all the way to the root and try the second right-hand side, $S \to bAe$.

Backtracking in this way is a little like a depth-first search of a graph. The search works forward from node to node until it finds its target or it reaches a dead

end. If it reaches a dead end, it must back up until it finds a fork in the road, and follow up that possibility. Similarly, this brute-force method works forward from production to production until it succeeds or reaches a dead end. If it reaches a dead end, it must then work backward until it finds a fork in the road—i.e., a production with an untried right-hand side.

But as the parser proceeds, it uses up input. That is, in generating the tree above, when it chooses $S \rightarrow bAc$, it moves past the b in the input string, and when it chooses $A \rightarrow cA$, it moves past the c. So it must not only dismantle the defective tree but also go back in the input to the token it was examining when it went astray. In some cases, this can be done relatively easily; in other cases, it can be prohibitive. If the lexical scan was a separate pass and the entire program has been tokenized, then it may be simply a matter of backing up the list of tokens. If the scanner is under the control of the parser and tokenizes the program as it goes, backing up can be a messy problem, because the scanner must also work its way back to the corresponding point in the input string. These problems can all be handled, but they slow down the operation of the compiler. The operation is particularly slow if the programmer's code contains an error, since the compiler may have to backtrack repeatedly to try all possibilities and see all of them fail, before it can conclude that the input is faulty. Because of these considerations, any method that entails backtracking is not an attractive approach to parser design.

The problem is partly in parser design and partly one of language design. The example we have used is contrived; it provided a production that looked attractive but had a booby trap at the end. Notice how much less trouble we have with the following grammar G_2 =

$S \rightarrow ee \mid bAQ$
$Q \rightarrow c \mid e$
$A \rightarrow d \mid cA$

Here we have factored out the common prefix bA and used another nonterminal to permit the choice between the final c or e. G_2 defines the same language as G_1, but the parser can now grow the following tree without backtracking:

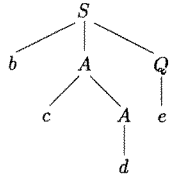

Figure 3.12

The transformation that gave us G_2 from G_1 is known as *left factorization*; it is the first of several techniques that make top-down parsing practical. A more generally

important way of avoiding backtracking is the use of *FIRST* and *FOLLOW* sets, which we will take up in Section 3.4.

3.3 RECURSIVE-DESCENT PARSING

If we can avoid backtracking, then there are a number of possible implementations. One is to embody each set of productions from any one nonterminal in a boolean function. There is one such function for every nonterminal in the grammar. The function will try each right-hand side in turn until a match has been found. If a match is found, then the function returns `true` and skips all the remaining right-hand sides; if no match is found, then the function returns `false`.

A typical right-hand side will be a mixture of nonterminals and terminals. The function will attempt to match the terminals in the right-hand side against the input string. But for nonterminals, it will *call the corresponding function* for that nonterminal. For example, suppose the grammar is

$S \rightarrow bA \mid c$
$A \rightarrow dSd \mid e$

The function for S will check the b in the first right-hand side itself; but for the A, it will call the function for A.

Let us examine this parser in detail. Since there are two nonterminals in this grammar, we need two functions; it is convenient to name them after their corresponding nonterminals. The function `S` takes the form

```
 1   function S: boolean;
 2      begin
 3      S := true;              { Always the optimist   }
 4      if token_is ('b') then
 5         if A then            { Try S --> bA          }
 6            writeln ('S --> bA')
 7         else
 8            S := false        { Failed on A           }
 9      else                    { Try S --> c           }
10         if token_is ('c') then
11            writeln ('S --> c')
12         else
13            begin             { Still no good: fails  }
14            error ('S');      { Break the bad news    }
15            S := false
16            end
17      end;  { S }
```

There must be some control of the lexical scanner in this code, so that once a token has been properly matched, the scanner will advance to the next token. We

have built this into the function Token_Is, which appears in Lines 4 and 10; if a match is found, the scanner advances to the next token; otherwise it stays put. The procedure error copes with the error; it may do no more than display a simple error message, "Unable to expand...".

The main parsing program starts things off by calling S. If this call returns true, and if the input has all been used up, then the parse is successful; otherwise it announces an error.

But notice that S makes no attempt to match A; instead, in Line 5, it turns that problem over to the function A. Function A takes the form

```
1   function A: boolean;
2      begin
3      A := true;
4      if token_is ('d') then
5         begin                { Try A --> dSd          }
6         if S then
7            if token_is ('d') then
8               writeln ('A --> dSd')
9            else
10              begin           { Failed on final d      }
11              error ('A');
12              A := false
13              end
14        else
15           A := false         { Failed on S            }
16        end
17     else                    { Try A --> e            }
18        if token_is ('e') then
19           writeln ('A --> e')
20     else
21        begin                { Still no good: fails   }
22        error ('A');
23        A := false
24        end
25     end;  { A }
```

Function A handles the right-hand sides of A similarly. In trying to expand A with $A \rightarrow dSd$, it checks the first terminal d itself, but to find out whether the subsequent input contains anything expandable from S, it calls function S. (Note that since A was called by S in the first place, this means that all of these functions must be recursive.) If the call to S succeeds, then function A checks the other d.

It is instructive to trace the execution of these functions. Suppose the token stream is $bdcd$. S starts out by testing the production $S \rightarrow bA$. To do this, it first calls Token_Is to find whether the next token is a b. When this call returns successfully, it must then find whether the initial b is followed by anything derivable from A; to do this, it calls A.

A starts out by trying the production $A \rightarrow dSd$. To do this, it tests for an input d. When it finds a match, it calls S to see whether that d is followed by anything derivable from S. So we are back in S again.

S looks for a b in the input and fails to find it. So then it skips to Line 10 and tries $S \rightarrow c$. When it finds this token, it indicates success by displaying the production $S \rightarrow c$; then it returns to A.

A, having satisfied dS, now looks for another d by calling Token_Is again (in Line 7). When it finds a match, it displays the successful production $A \rightarrow dSd$ and returns to S.

We are now back at the top-level call to S. S now has a completed right-hand side and displays the production $S \rightarrow bA$. It returns to the calling program; the calling program finds no leftover input and announces a successful parse.

If we draw a tree diagram to indicate the recursive calls that were made, it closely matches the parse tree for this expression. The parse tree is shown in part (a) of Figure 3.13and the recursion tree in part (b):

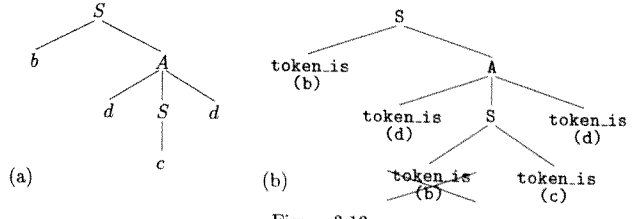

Figure 3.13

The only difference is the unsuccessful call to Token_Is, shown crossed out. This similarity is no coincidence; in general, as they proceed through the parse, S and A will gradually work their way down the parse tree. Because of this, and because the functions must be recursive, this type of parser is known as a *recursive-descent* parser. Recall that recognizing strings generated by a CFG always requires a stack. In this implementation, the stack is invisible; it is the stack used by Pascal to support recursion.

3.4 PREDICTIVE PARSERS

In most of our examples so far, at most one right-hand side of each production began with a nonterminal. Hence our strategy was simple: try the right-hand sides beginning with terminals, and if they fail, try the one beginning with a nonterminal. But suppose we have a grammar

$S \rightarrow Ab \mid Bc$
$A \rightarrow Df \mid CA$
$B \rightarrow gA \mid e$
$C \rightarrow dC \mid c$
$D \rightarrow h \mid i$

Here the right-hand sides of the productions from S and A do not begin with terminals; as a result, the parser has no immediate guidance from the input string. There is no question of a booby trap here; if the input string is *gchfc*, a recursive-descent parser may have to do a good deal of experimenting and backtracking before it finds the derivation, $A \Rightarrow Bc \Rightarrow gAc \Rightarrow gCAc \Rightarrow gcAc \Rightarrow gcDfc \Rightarrow gchfc$. This example is not unrealistic; grammars frequently have productions several of whose right-hand sides begin with nonterminals.

This could be avoided if the parser had the ability to look ahead in the grammar so as to anticipate what terminals are derivable (by leftmost derivations) from each of the various nonterminals on the right-hand sides of productions. For example, if we follow up the right-hand sides of S, considering all possible leftmost derivations, we find the following possibilities:

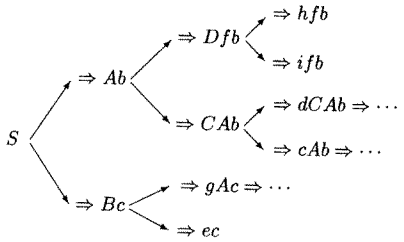

Figure 3.14

From this diagram we see that if the input string begins with c, d, h, or i, we want to choose $S \rightarrow Ab$, and if it begins with e or g, we want to choose $S \rightarrow Bc$. If it begins with anything else, we should announce an error.

This diagram tells us, in fact, what terminals can begin strings derivable from A and what terminals can begin strings derivable from B. These are known as the *FIRST sets*, and we write

$FIRST(Ab) = \{c, d, h, i\}$
$FIRST(Bc) = \{e, g\}$

(We normally associate a *FIRST* set with an entire right-hand side, not just its starting symbol.) Given this information, the function S will try the production $S \rightarrow Ab$ if the input token is in $FIRST(Ab)$ and $S \rightarrow Bc$ if the input token is in

$FIRST(Bc)$. If the input does not begin with a token in $FIRST(S) = FIRST(Ab) \cup FIRST(Bc)$, then it is ungrammatical and can be rejected at once.

The function S thus takes the form

```
if next_token in ['c', 'd', 'h', 'i'] then
   begin
                { Try S --> Ab              }
   end
else if next_token in ['e', 'g'] then
   begin
                { Try S --> Bc              }
   end
else
   begin        { Nothing else will do      }
   error ('S');
   S := false
   end;
```

The function A will similarly try $A \rightarrow Df$ if the next token is in $FIRST(Df)$ and $A \rightarrow CA$ if it is in $FIRST(CA)$.

We use $FIRST(\cdot)$ to denote both the *FIRST* set and the function that finds it for us. We can define the function formally as

$FIRST(\alpha)$:

Argument: α is some sentential form generated by G.

Returns: a set of terminals.

Definition: Consider every string derivable from α by a leftmost derivation. If $\alpha \stackrel{*}{\Rightarrow} \beta$, where β begins with some terminal, then that terminal is in $FIRST(\alpha)$.

In small or moderate-sized grammars, *FIRST* sets can be found by hand along the lines suggested by the diagram in Figure 3.14. Clearly, α must begin with either a terminal or a nonterminal. If it begins with a terminal x, then $FIRST(\alpha) = x$ and we are done, since any subsequent left sentential forms will also begin with x. If α begins with a nonterminal A, we need only follow up leftmost derivations from A, and these normally terminate quickly. For example, in the expression grammar of Example 4, Section 3.2.1, if we want $FIRST(E)$, we see where E can lead. The only possibilities are

$E \Rightarrow TQ$
$\Rightarrow FRQ$
$\Rightarrow (E)RQ$
$\Rightarrow \cdots$

$$\begin{aligned} E &\Rightarrow TQ \\ &\Rightarrow FRQ \\ &\Rightarrow \mathtt{i}RQ \\ &\Rightarrow \cdots \end{aligned}$$

Now, once we have hit $(E)RQ$, we're done, because all further left sentential forms derivable from $(E)RQ$ will begin with (. Similarly, once we have hit i, we are done for the same reason. So $FIRST(E) = \{(, \mathtt{i}\}$.

From the definition of *FIRST*, we can note the following properties:

1. If α begins with a terminal x, then $FIRST(\alpha) = x$.
2. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then $FIRST(\alpha)$ includes ϵ.
3. $FIRST(\epsilon) = \{\epsilon\}$.
4. If α begins with a nonterminal A, then $FIRST(\alpha)$ includes $FIRST(A) - \{\epsilon\}$.

Rule 4 contains a hidden trap, however. Suppose α is $AB\delta$ and $A \stackrel{*}{\Rightarrow} \epsilon$. Then we must follow up the possibilities from B as well. Moreover, if $B \stackrel{*}{\Rightarrow} \epsilon$, then we must follow up the possibilities from δ, too. Here is an example: suppose our grammar includes

$$\begin{aligned} S &\rightarrow ABCd \\ A &\rightarrow e \mid f \mid \epsilon \\ B &\rightarrow g \mid h \mid \epsilon \\ C &\rightarrow p \mid q \end{aligned}$$

and we want to find $FIRST(S) = FIRST(ABCd)$. Exploring this, we find

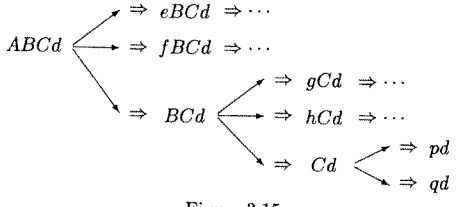

Figure 3.15

So $FIRST(ABCd)$ is $\{e, f, g, h, p, q\}$.

If a nonterminal $A \stackrel{*}{\Rightarrow} \epsilon$, we say that it is *nullable*. In this example, both A and B were nullable, so we had to follow up not only B but C as well. Notice that although $FIRST(A)$ and $FIRST(B)$ included ϵ, $FIRST(ABCd)$ didn't. $FIRST(\alpha)$ includes ϵ only if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, which can happen only if everything in α is nullable.

This hidden trap isn't sprung very often (we won't see it in any of the practical examples in this chapter), but if we want to program this procedure, we must allow for it. Because of the nullability problem, and because of the question of whether to include ϵ or not, there is no nice way of stating the general rule. The least objectionable form seems to be that given by Holub [1990]: Suppose α takes the form $\beta X\delta$, where β is a string of 0 or more nullable nonterminals, X is either a terminal or the first *non*nullable nonterminal, and δ is whatever is left over. Then $FIRST(\alpha) = (FIRST(\beta) - \{\epsilon\}) \cup FIRST(X)$. If everything in α is nullable (that is, if $\alpha = \beta$ alone), then $FIRST(\alpha) = FIRST(\beta)$.

Probably the easiest way to reduce this to an algorithm is to consider two cases:

Case 1: α is a single character or ϵ:

if α is a terminal y then $FIRST(\alpha) = y$
else if α is ϵ then $FIRST(\alpha) = \epsilon$
else if α is a nonterminal and $\alpha \rightarrow \beta_1 \mid \beta_2 \mid \beta_3 \mid \cdots$ then
    $FIRST(\alpha) = \bigcup_k FIRST(\beta_k)$.

Case 2: $\alpha = X_1 X_2 \cdots X_n$:

$FIRST(\alpha) = \{\ \}$;
$j := 0$;
repeat
    $j := j + 1$;
    Include $FIRST(X_j)$ in $FIRST(\alpha)$
until X_j nonnullable or $j = n$;
If X_n is nullable then add $\{\epsilon\}$ to $FIRST(\alpha)$.

The idea is to separate the handling of a single nonterminal from the handling of a lengthy sentential form. In Case 1, we put any terminal or ϵ into $FIRST(\alpha)$; for nonterminals we include the *FIRST* sets of all their right-hand sides. In Case 2, where α is more than one symbol, we include all the *FIRST* sets until we reach something that is nonnullable. At the end of the loop, if X_n is nullable, that means everything in the form was nullable, and in that case only we include ϵ in the *FIRST* set. It is convenient to code these two cases as separate procedures which call each other recursively.

In our example above, C is the first nonnullable nonterminal. So the rule gives us

$$\begin{aligned} FIRST(ABCd) &= \{e, f\} && [\text{that's } FIRST(A) - \{\epsilon\}] \\ &\quad \cup \{g, h\} && [\text{that's } FIRST(B) - \{\epsilon\}] \\ &\quad \cup \{p, q\} && [\text{that's } FIRST(C)] \\ &= \{e, f, g, h, p, q\} \end{aligned}$$

This is in agreement with what we found when we explored the productions by hand.

Parsers that use *FIRST* sets are known as *predictive parsers*. The "predictive" aspect of the parser is its ability to leap forward this way and "predict" that the first derivation step $E \Rightarrow TQ$ will eventually get you to

$$E \stackrel{*}{\Rightarrow} (\cdots$$

or else to

$$E \stackrel{*}{\Rightarrow} \mathtt{i} \cdots$$

This technique cannot always be used. Sometimes the structure of the grammar is such that the next token will not tell you which right-hand side to use. [For example, suppose $FIRST(\alpha)$ and $FIRST(\beta)$ weren't disjoint.] I'll come back to this later.

Furthermore, when grammars acquire ϵ- productions as a result of removing left recursions, the *FIRST* sets will not tell us when to choose $A \rightarrow \epsilon$. To handle these, we need a second function, *FOLLOW*. In defining this, we will assume that a token string, before being passed to the parser, has an end marker appended to it. We will see this end marker frequently from now on; people write it variously as #, $\perp$, $\dashv$, or \$. The last symbol seems most popular, and that's the one we will use.

FOLLOW(A):

Argument: A is some nonterminal in G.

Returns: a set of terminals.

Definition: $FOLLOW(A)$ is the set of all terminals that can come right after A in any sentential form of $L(G)$. If A can come at the end, then $FOLLOW(A)$ includes the end marker \$.

We can compute $FOLLOW(A)$ by the following rules:

1. If A is the starting symbol, then put the end marker \$ into $FOLLOW(A)$.
2. Look through the grammar for all occurrences of A on the right-hand sides of productions. Let a typical production be $Q \rightarrow xAy$. There are three cases:
   a. If y begins with a terminal q, then q is in $FOLLOW(A)$.
   b. Otherwise $FOLLOW(A)$ includes $FIRST(\beta) - \{\epsilon\}$.
   c. If $y = \epsilon$ (that is, A comes at the end), or if y is nullable, then include $FOLLOW(Q)$ in $FOLLOW(A)$.

Note that if Rule 1 gives you \$, you can't stop there; you must go on and try Rule 2.

The reasons for these rules are: For Rule 2a, clearly if y is a terminal, then it must be possible for that terminal to follow A in a sentential form; hence y is in $FOLLOW(A)$. For Rule 2b, if y is a nonterminal, then any initial terminal derivable

from y can follow A; but the set of all such terminals is precisely $FIRST(y)$. We exclude ϵ because ϵ never appears as an explicit token.

Rule 2c requires more explanation. First, note that if A is at the end of the right-hand side, this means that A can come at the very end of a sentential form derivable from Q. Next, if $Q \rightarrow \beta AB$ and B is nullable, that *also* means that A can come at the very end of a sentential form derivable from Q. In those cases, what can come after A? For the answer, consider where Q itself came from. Suppose $Q \rightarrow \beta A$, as shown here:

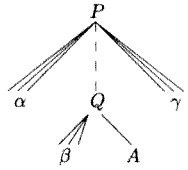

Figure 3.16

If we look farther up the tree, we see that $P \stackrel{*}{\Rightarrow} \alpha Q \gamma \Rightarrow \alpha\beta A\gamma$. So what comes after A turns out to be γ; but γ is what came after Q—and γ must be what gave rise to $FOLLOW(Q)$. This is why, in this case, anything in $FOLLOW(Q)$ must also be in $FOLLOW(A)$, and that is what gives us Rule 2c. In the peculiar case where the left-hand side is *also* A, that is, $A \rightarrow xA$, we ignore this rule, since the left-hand side gives us nothing new.

To show the complete process, we will take our expression grammar as an example. We have

$$\begin{aligned}
&E \rightarrow TQ \\
&Q \rightarrow +\,TQ \mid -TQ \mid \epsilon \\
&T \rightarrow FR \\
&R \rightarrow *\,FR \mid /FR \mid \epsilon \\
&F \rightarrow (\ E\ ) \mid \mathtt{i}
\end{aligned}$$

Then

$$\begin{aligned}
FIRST(E) &= FIRST(T) \\
&= FIRST(F) \\
&= \{(, \mathtt{i}\} \\
FIRST(Q) &= \{+, -, \epsilon\} \\
FIRST(R) &= \{*, /, \epsilon\}
\end{aligned}$$

and, trivially,

$$FIRST(+TQ) = \{+\}$$

$FIRST(-TQ) = \{-\}$
$FIRST(*FR) = \{*\}$
$FIRST(/FR) = \{/\}$
$FIRST((E)) = \{(\}$
$FIRST(\mathrm{i}) = \{\mathrm{i}\}$

For *FOLLOW*, we do this: The starting symbol is E. So $FOLLOW(E)$ includes \$. We now look through the grammar for right-hand sides containing E. There's only one: $F \rightarrow (E)$. Here the symbol coming after E is), so) is in $FOLLOW(E)$. The result is

$$FOLLOW(E) = \{\$,)\}$$

For Q, we find that Q appears only at the end. $E \rightarrow TQ$ tells us that $FOLLOW(Q)$ includes $FOLLOW(E)$, by Rule 2c. The other productions with Q on the right-hand side, $Q \rightarrow +TQ$ and $Q \rightarrow -TQ$, tell us nothing, since the left-hand side is also Q. So

$$FOLLOW(Q) = FOLLOW(E) = \{\$,)\}$$

For T we do the same: look at right-hand sides containing T. We have

$E \rightarrow TQ$
$Q \rightarrow +TQ \mid -TQ \mid \epsilon$

By Rule 2b, $FOLLOW(T)$ includes $FIRST(Q) - \{\epsilon\} = \{+, -\}$. But since Q is nullable [$FIRST(Q)$ includes ϵ], we must also throw in $FOLLOW(E)$ because of Rule 2c. The Q-productions give us nothing new. So we have

$$FOLLOW(T) = \{+, -,), \$\}$$

It is not difficult to show that

$$FOLLOW(R) = FOLLOW(T) = \{+, -,), \$\}$$

Finally, for F we have

$T \rightarrow FR$
$R \rightarrow *FR \mid /FR \mid \epsilon$

By Rule 2b, $FOLLOW(F)$ includes $FIRST(R) - \{\epsilon\} = \{*, /\}$. But again, since R is nullable, we must throw in $FOLLOW(T)$. This gives us

$$FOLLOW(F) = \{+, -, *, /,), \$\}$$

In a predictive parser, *FOLLOW* tells us when to use ϵ-productions. Suppose we are in the function for expanding some nonterminal A. We initially see whether the forthcoming token is in the *FIRST* set for some right-hand side. If it is not, this normally means an error. But if one of the productions from A is $A \rightarrow \epsilon$, then we must see whether the forthcoming token is in $FOLLOW(A)$. If it is, then it may not be an error, and $A \rightarrow \epsilon$ is the indicated production. (We expand $A \rightarrow \epsilon$ by doing nothing and returning.) Thus the function for A would take the general form


```
function A: boolean;
   begin
   A := true;
   if next_token in FIRST(right-hand side1) then
      begin
               { Try A --> right-hand side1                }
      end
   else if next_token in FIRST(right-hand side2) then
      begin
               { Try A --> right-hand side2                }
      end
   else
       {... etc., through all right-hand sides }
   else if not (next_token in FOLLOW(A)) then
      begin
      error ('A');
      A := false
      end
   end;  { A }
```

If all *FIRST* tests fail and the forthcoming token is in $FOLLOW(A)$, then `A` returns its default value of `true` without expanding A or using up any input.

Grammars for which this technique can be used are known as LL(1) grammars, and parsers using this technique are called LL(1) parsers. In this notation, the first L stands for *L*eft-to-right scan of tokens, the second L stands for *L*eftmost derivation, and the (1) stands for one-character lookahead. The one-character lookahead tells us that every incoming token uniquely determines which right-hand side to choose. (In an LL(2) grammar you would have to look at pairs of tokens.) For a grammar to be LL(1), we require that for every pair of productions $A \rightarrow \alpha \mid \beta$,

1. $FIRST(\alpha) - \{\epsilon\}$ and $FIRST(\beta) - \{\epsilon\}$ must be disjoint.
2. If α is nullable, then $FIRST(\beta)$ and $FOLLOW(A)$ must be disjoint.

If Rule 1 is violated, then any token in $FIRST(\alpha) \cap FIRST(\beta)$ will fail to tell us which right-hand side to choose. If Rule 2 is violated, then any token in $FIRST(\beta) \cap FOLLOW(A)$ will fail to tell us whether to choose β or ϵ.

3.4.1 A Predictive Recursive-Descent Parser

We can now incorporate all we have learned about top-down parsing in a sample recursive-descent parser for expressions. We will use this grammar:

$E \rightarrow E + T \mid T$
$T \rightarrow T * F \mid F$
$F \rightarrow (E) \mid \mathrm{i}$

After removing left recursions, we have

$$
\begin{aligned}
E &\rightarrow TQ \\
Q &\rightarrow +TQ \mid \epsilon \\
T &\rightarrow FR \\
R &\rightarrow *FR \mid \epsilon \\
F &\rightarrow (E) \mid \mathrm{i}
\end{aligned}
$$

and this is the form that we use here.

Here is how the program begins:

```
Program REC_DESC;
{
        Simple recursive-descent expression parser
                                                        }
var
   lineimage: string[80];        { Input string  }
   tok_ptr: integer;             { Token pointer }

function TOKEN: char;    { Extracts new token from input }
   begin
   if tok_ptr < length(lineimage) then
      begin
      tok_ptr := tok_ptr + 1;
      token := lineimage[tok_ptr]
      end
   else
      token := #0;       { Null if input exhausted       }
   end;  { Token }

procedure BACKUP;        { Backs up token pointer        }
   begin
   if tok_ptr > 0 then
      tok_ptr := tok_ptr - 1
   end;  { BackUp }
```

We use only the simplest sort of lexical scanner, restricting the user to one-letter variable names and not allowing blanks. In this case, the lexical scanner need only pick up the next character. A global variable `tok_ptr` indexes the input string for this purpose. If the current token is in a *FOLLOW* set, then it is not used up; hence the next time a function is to examine a token, we must give it this same token again. We arrange this by moving the token pointer back one character; that's the reason for the procedure `BackUp`.

We have one function for each nonterminal; I will call them `E`, `Q`, `T`, `R`, and `F`. The parsing routines begin as follows:


```
function Q: boolean; forward;
function T: boolean; forward;
function R: boolean; forward;
function F: boolean; forward;

function E: boolean;     { Expands productions on E      }
   begin
   E := false;                    { Assume the Worst      }
   if T then
      if Q then
         begin
         writeln (' E --> TQ');
         E := true
         end
   end;  { E }
```

`E` starts by assuming that the string is illegal; only if it makes its way past the calls to `T` and `Q` does it accept the string. The other functions are organized similarly. The parser would run faster if `E` tested the forthcoming token for membership in the appropriate *FIRST* sets. Since there is only one production on E, however, we get along without these tests: either this right-hand side can be expanded or the expression is wrong.

The *FIRST* set for $+TQ$ is just $\{+\}$; *FOLLOW*(Q) is $\{), \$\}$. Then `Q` is as follows:

```
function Q: boolean;     { Expands productions on Q      }
   var
      cc: char;
   begin
   Q := false;
   cc := token;
   if cc = '+' then                { Try FIRST set }
      begin
      if T then
         if Q then
            begin
            writeln (' Q --> +TQ');
            Q := true
            end
      end
   else
      if cc in [')', '$'] then     { Try FOLLOW set }
         begin
         backup;
```

```
            writeln (' Q --> eps');
            Q := true
            end
        end;  { Q }
```

The structure of T is virtually the same as that of E:

```
    function T: boolean;     { Expands productions on T      }
       begin
       T := false;
       if F then
          if R then
             begin
             writeln (' T --> FR');
             T := true
             end
       end;  { T }
```

FIRST(∗*FR*) is just {∗} and *FOLLOW*(*R*) is {+,), $}. The remaining functions are as follows:

```
    function R: boolean;     { Expands productions on R      }
       var
          cc: char;
       begin
       R := false;
       cc := token;
       if cc = '*' then               { Try FIRST set }
          begin
          if F then
             if R then
                begin
                writeln (' R --> *FR');
                R := true
                end
          end
       else
          if cc in ['+',')','$'] then { Try FOLLOW set }
             begin
             writeln (' R --> eps');
             backup;
             R := true
             end
       end;  { R }
```



```
    function F: boolean;     { Expands productions on F      }
       var
          cc: char;
       begin
       F := false;
       cc := token;
       if cc in ['a'..'z'] then       { Identifier?   }
          begin
          writeln (' F --> id');
          F := true
          end
       else if cc = '(' then          { --no: try (E) }
          if E then
             begin
             cc := token;
             if cc = ')' then
                begin
                writeln (' F --> (E)');
                F := true
                end
             end
       end;  { F }
```

The main program prompts the user for an expression and calls E:

```
    begin
    write ('Enter expression: ');
    readln (lineimage);
    lineimage := lineimage + '$';     { Append end marker     }
    tok_ptr := 0;
    if E and (token = '$') then       { Must use up input     }
       writeln (' Success.')
    else
       begin
       writeln (' Failure:');         { Locate error  }
       writeln (lineimage);
       writeln ('^':tok_ptr, '---error')
       end
    end.
```

The test (`token = '$'`) keeps the parser from accepting an input like `x=a+b`). We take advantage of the fact that parsing stops when an error is found. This means that the token pointer is still pointing to the faulty input token; hence the "`Failure`" outcome displays the input string with a caret underneath the error.

We could provide much more intelligent error handling by modifying the individual parsing functions.

3.4.2 Table-Driven Predictive Parsers

The removal of left recursions and the construction of the *FIRST* and *FOLLOW* sets make a recursive-descent parser practical. The remaining impracticality of this approach is that for every production we must write a function. If some problem arises that makes it necessary to change the grammar, one or more of the functions will also have to be reprogrammed. (Grammars may have to be debugged, just as programs have to be.)

A nonrecursive form of the predictive parser consists of a simple control procedure which runs off a table. Part of the attraction of this approach is that the control procedure is quite general; if the grammar is to be changed, only the table need be rewritten. This is a much less troublesome process than reprogramming. The table can be constructed by hand for small grammars or by computer for large ones. The recursive predictive parser selected right-hand sides and occasionally used up tokens as it did so. The nonrecursive version's table tells it which right-hand sides to select, and tokens are used up in a natural and easy fashion.

Our example will be another expression parser, using this grammar:

$$E \rightarrow E+T \mid E-T \mid T$$
$$T \rightarrow T*F \mid T/F \mid F$$
$$F \rightarrow (E) \mid \mathbf{i}$$

This was the grammar in Example 4; after removing left recursions, we had

$$E \rightarrow TQ$$
$$Q \rightarrow +TQ \mid -TQ \mid \epsilon$$
$$T \rightarrow FR$$
$$R \rightarrow *FR \mid /FR \mid \epsilon$$
$$F \rightarrow (\ E\ ) \mid \mathrm{i}$$

It is easiest to show the table and how it is used first and to show how the table is constructed afterward. The table for this grammar is

	i	+	−	*	/	(	)	$
E	TQ					TQ		
Q		$+TQ$	$-TQ$				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	$*FR$	$/FR$		ϵ	ϵ
F	i					(E)		

The blanks are error conditions. Some writers put the entire production into each table entry; but since we already know what the left-hand side is (it's the row

index), all we need in the table is the right-hand side. (For large tables, it is even more economical to number the productions and enter the numbers in the table.)

I said that we would always need a stack for Type 2 grammars. In the case of the table-driven predictive parser—and in fact of all the parsers we will consider from here on—we maintain our own stack. Here is how we use it. The rule is:

```
Push $ (an end marker) onto the stack.
Put a similar end marker on the end of the string.
Push the start (sentence) symbol onto the stack.
While (stack not empty) do

    begin
    Let x = top of stack and a = incoming token.
    If x is in T then           { If terminal }
        if x = a then pop x and go to next input token
            else error
    else                        { nonterminal }
        if Table[x, a] nonblank then
            begin
            pop x;
            push Table[x, a] onto stack in reverse order
            end
        else error;
    end;
```

We push the right-hand side in reverse order so that its leftmost symbol will land at the top of the stack, ready for further expansion or for cancellation by popping. An example will show how this works in practice. Suppose our string is

(i + i)*i

Then here is a trace of the parse. This is a step-by-step table showing the moves made by the parser, and for that purpose it is most convenient if we write the stack horizontally so that it grows from left to right. I will write the stack at the left, the unused input in the middle, and the productions on the right, in parallel columns. Since the parse constructs a leftmost derivation, I will add a column showing the derivation steps as well; but for understanding the operation of the parser, we will concentrate on the first three columns.

Stack	Input	Production	Derivation
$E	(i + i)*i$		

This is the starting state. The input has had the end marker added to it, and the stack has had an end marker and the sentence symbol pushed onto it. The parser

now enters its main loop. We will repeat the first line of the trace showing the production selected by the table. The left-hand side is E, since this is the symbol atop the stack, and $Table[E, (] = TQ$. So our production is $E \rightarrow TQ$ and we have

$\$E$	(i + i)*i$	$E \rightarrow TQ$	$E \Rightarrow TQ$
$\$QT$	(i + i)*i$		

Here we popped the E off the stack and pushed the right-hand side TQ onto the stack in reverse order. The parse continues as follows: we have the nonterminal T at the top of the stack and our incoming token is still (. So we have $Table[T, (] = FR$, and the production is $T \rightarrow FR$:

$\$QT$	(i + i)*i$	$T \rightarrow FR$	$\Rightarrow FRQ$
$\$QRF$	(i + i)*i$	$F \rightarrow (E)$	$\Rightarrow (E)RQ$
$\$QR)E($	(i + i)*i$		

Here we have a terminal on the top of the stack. We compare it with the incoming token; since they match, we pop the (and discard it, and we move to the next input token:

$\$QR)E$	i + i) * i$	$E \rightarrow TQ$	$\Rightarrow (TQ)RQ$
$\$QR)QT$	i + i) * i$	$T \rightarrow FR$	$\Rightarrow (FRQ)RQ$
$\$QR)QRF$	i + i) * i$	$F \rightarrow$ i	$\Rightarrow (iRQ)RQ$
$\$QR)QRi$	i + i) * i$	[pop and go to next token]	
$\$QR)QR$	+i)*i$	$R \rightarrow \epsilon$	$\Rightarrow (iQ)RQ$

An ϵ-production replaces the left-hand side with nothing at all; hence in this case when we pop R off the stack, we do not push anything back on:

$\$QR)Q$	+ i)*i$	$Q \rightarrow +TQ$	$\Rightarrow (i+TQ)RQ$
$\$QR)QT+$	+ i)*i$	[pop and go to next token]	
$\$QR)QT$	i)*i$	$T \rightarrow FR$	$\Rightarrow (i+FRQ)RQ$
$\$QR)QRF$	i)*i$	$F \rightarrow$i	$\Rightarrow (i+iRQ)RQ$
$\$QR)QRi$	i)*i$	[pop and go to next token]	
$\$QR)QR$	)*i$	$R \rightarrow \epsilon$	$\Rightarrow (i+iQ)RQ$
$\$QR)Q$	)*i$	$Q \rightarrow \epsilon$	$\Rightarrow (i+i)RQ$
$\$QR)$	)*i$	[pop and go to next token]	
$\$QR$	*i$	$R \rightarrow *FR$	$\Rightarrow (i+i)*FRQ$
$\$QRF*$	*i$	[pop and go to next token]	
$\$QRF$	i$	$F \rightarrow$i	$\Rightarrow (i+i)*iRQ$
$\$Qi$	$	[pop and go to next token]	
$\$QR$	$	$R \rightarrow \epsilon$	$\Rightarrow (i+i)*iQ$
$\$Q$	$	$Q \rightarrow \epsilon$	$\Rightarrow (i+i)*i$
$	$	[pop and go to next token]	

When the input is used up and the stack is empty, we announce a successful parse.

We can construct the parse tree either from the productions in the order in which they appear or from the derivation steps:

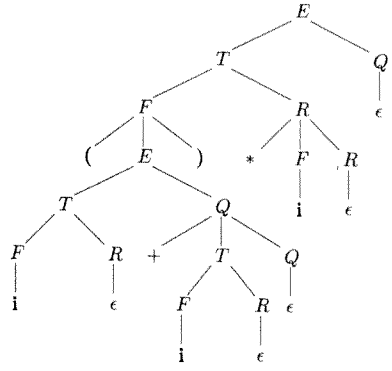

Figure 3.17

Notice how strange this tree looks. (Compare it, for example, with Figure 3.4.) To look ahead for a moment, we will find in Chapter 5 that the parse tree will guide us in generating intermediate code. We can picture this being done in the case of Figure 3.4, but here, where the tree bears so little resemblance to the original expression, we can expect problems.

Next, consider the action of the parser when given an incorrect string, for example, (i*):

Stack	Input	Production	Derivation
$\$E$	(i*)$	$E \rightarrow TQ$	$E \Rightarrow TQ$
$\$QT$	(i*)$	$T \rightarrow FR$	$\Rightarrow FRQ$
$\$QRF$	(i*)$	$F \rightarrow (E)$	$\Rightarrow (E)RQ$
$\$QR)E($	(i*)$	[pop and go to next token]	
$\$QR)E$	i*)$	$E \rightarrow TQ$	$\Rightarrow (TQ)RQ$
$\$QR)QT$	i*)$	$T \rightarrow FR$	$\Rightarrow (FRQ)RQ$
$\$QR)QRF$	i*)$	$F \rightarrow$ i	$\Rightarrow (iRQ)RQ$
$\$QR)QRi$	i*)$	[pop and go to next token]	
$\$QR)QR$	*)$	$R \rightarrow *FR$	$\Rightarrow (i*FRQ)RQ$
$\$QR)QRF*$	*)$	[pop and go to next token]	
$\$QR)QRF$	)$	***Error: no table entry for $[F,)]$.	

Notice that when this happens, the input token pointer is pointing to the close

parenthesis; hence it is a relatively simple matter for the error handler to generate something along the following lines:

```
(i*)
  ^---error: factor expected.
```

3.4.3 Constructing the Predictive Parser Table

Recall that in a top-down parser, if there is to be no backtracking, the next incoming token must always tell us what to do next. The same thing applies in constructing the table. Suppose we have X on the stack and a as our input token. We want to select a right-hand side that either

(a) begins with a or
(b) can *lead* to a sentential form beginning with a.

For example, at the start, we had E on the stack and (as input. We needed a production of the form $E \rightarrow (\cdots$. But there is no such production in the grammar. Since that isn't available, we had to trace a "path" through the productions (i.e., a leftmost derivation) that would *lead* to a sentential form beginning with (. The only such path is

$$E \Rightarrow TQ \Rightarrow FRQ \Rightarrow (E)RQ$$

And when you look at the table, you see that it selects the right-hand side that gives us the first step in this derivation.

This brings us back to familiar territory: we want to select a right-hand side α if the token is in $FIRST(\alpha)$; hence for a row A and a production $A \rightarrow \alpha$, the table must have the right-hand side α in every column headed by a terminal in $FIRST(\alpha)$. This will work in all cases except those where $FIRST(\alpha)$ includes ϵ, since the table will never have a column headed ϵ. For those cases, we use the $FOLLOW$ sets in a manner very nearly the same as in the recursive predictive parser.

The rule for constructing the table is therefore:

Go through all the productions. Let $X \rightarrow \beta$ be a typical production.

1. For all terminals a in $FIRST(\beta)$ except ϵ,
   $Table[X,\ a] = \beta$.
2. If $\beta = \epsilon$ or if ϵ is in $FIRST\ (\beta)$, then
   for all a in $FOLLOW(X)$, $Table[X,\ a] = \epsilon$.

The $FIRST$ entries are simply the old idea that you let $FIRST(\beta)$ tell you the terminals for which the given right-hand side is applicable, which we saw in the recursive version. The significance of the $FOLLOW$ rule is also similar to the recursive case: By Rule 1, we have already accounted for all other right-hand sides

of X. The blank entries that remain are either errors or entries to be filled with ϵ. If some forthcoming terminal $a \notin FOLLOW(X)$, then it's not supposed to come after X, so it must be an error; hence the only places ϵ can go are the columns corresponding to $FOLLOW(X)$.

Let's take our expression grammar as an example. We found the following $FIRST$ and $FOLLOW$ sets previously:

$FIRST(E) = FIRST(T) = FIRST(F) = \{(, \mathrm{i}\}$
$FIRST(Q) = \{+, -, \epsilon\}$
$FIRST(R) = \{*, /, \epsilon\}$
$FIRST(+TQ) = \{+\}$
$FIRST(-TQ) = \{-\}$
$FIRST(*RF) = \{*\}$
$FIRST(/RF) = \{\ /\ \}$
$FOLLOW(E) = \{\$,)\}$
$FOLLOW(Q) = FOLLOW(E) = \{\$,)\ \}$
$FOLLOW(T) = \{+, -,), \$\}$
$FOLLOW(R) = FOLLOW(T) = \{+, -,), \$\}$
$FOLLOW(F) = \{+, -, *, /,), \$\ \}$

Using these, we construct our table as follows:

$E \rightarrow TQ$

and $FIRST(TQ) = FIRST(T) = \{(, \mathrm{i}\}$

So TQ goes in the columns headed (and i; the first row of our table is

	i	+	−	*	/	(	)	$
E	TQ					TQ		

Next, for Q, we have

$Q \rightarrow +TQ \mid -TQ \mid \epsilon$

Clearly, $+TQ$ goes in $FIRST(+TQ) = \{+\}$; similarly, $-TQ$ goes in −. For $Q \rightarrow \epsilon$, we must look at $FOLLOW(Q)$; that comprises $\{\$,)\}$. This gives us the second row:

	i	+	−	*	/	(	)	$
E	TQ					TQ		
Q		$+TQ$	$-TQ$				ϵ	ϵ

For T, we have

$T \rightarrow FR$

and $FIRST(FR) = FIRST(F) = \{(, \mathrm{i}\}$. This gives us Row 3:

	i	+	−	*	/	(	)	$
E	TQ					TQ		
Q		$+TQ$	$-TQ$				ϵ	ϵ
T	FR					FR		

For R, we have

$R \to *FR \mid /FR \mid \epsilon$

We can place $*FR$ in Column $*$ and $/FR$ in Column $/$. For the ϵ-productions, we must use $FOLLOW(R) = \{+, -,), \$\}$. These tell us the columns that get ϵ:

	i	+	−	*	/	(	)	$
E	TQ					TQ		
Q		$+TQ$	$-TQ$				ϵ	ϵ
T	FR					FR		
R		ϵ	ϵ	$*FR$	$/FR$		ϵ	ϵ

The last row is filled in similarly.

Notice that in this grammar, no right-hand side ever begins with a nullable nonterminal. For such a grammar, you need compute the *FIRST* sets for nonterminals only. If β begins with a nonterminal, you use *FIRST* of that nonterminal and don't care what the rest of β is. If it begins with a terminal, $FIRST(\beta)$ is obviously nothing but that initial terminal, and you can identify that on the fly as you fill in the table.

3.4.4 Conflicts

We said, $Table[X, a] = \beta$. Strictly speaking, $Table[X, a]$ *includes* β, because in theory, there could be more than one entry. But in practice, if there is more than one entry, we're in bad trouble, because we have no provision for choosing among multiple entries.

I must qualify that. We can occasionally manipulate the grammar by left factorization to get rid of multiple entries. Suppose we have the productions

$$A \to abE \mid acF \mid \cdots$$

We rely on the first token to tell us what to choose, and here it won't tell us. But if we change this to

$$A \to aQ \mid \cdots$$
$$Q \to bE \mid cF$$

then the a tells us to select aQ, and the next token will tell us which right-hand side of Q to choose.

Occasionally there may also be some reason why you can have two possible entries and automatically rule one out. For a classic example, consider the if-then-else grammar $G_i =$

$$S \to \textbf{if } E \textbf{ then } S \mid \textbf{if } E \textbf{ then } S \textbf{ else } S \mid a \mid b$$
$$E \to x \mid y$$

(Here a, b, x, and y are dummy terminals representing conditions and statements.) After left factorization of the first two productions on S, this grammar is

$$S \to \textbf{if } E \textbf{ then } SQ \mid a \mid b$$
$$E \to x \mid y$$
$$Q \to \textbf{else } S \mid \epsilon$$

Even after factorization, this grammar is ambiguous in a particularly harmful way; here are two distinct parse trees for

if x **then if** y **then** a **else** b

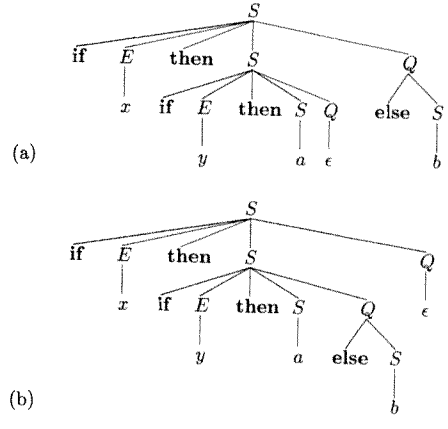

Figure 3.18

Version (a) relates the **else** to the first **then**; version (b) relates it to the second **then**. In most programming languages, it is customary to relate an **else** to the most recent unmatched **then**, and hence we must have the tree of version (b). This ambiguity is reflected in the *FIRST* and *FOLLOW* sets as well; we have, among others,

$FIRST(Q) = \{\textbf{else}, \epsilon\}$
$FOLLOW(S) = \{\$, \textbf{else}\}$
$FOLLOW(Q) = \{\$, \textbf{else}\}$

These give rise to the table

	if	x	y	**then**	a	b	**else**	$
S	**if** E **then** SQ				a	b		
E		x	y					
Q							**else** S	ϵ
							ϵ	

The multiple entry comes about because **else** is in both *FIRST*(Q) and *FOLLOW*(Q). But in fact we can ignore this problem and select **else** S as the table entry. It is left as an exercise to show that if we do this, the parser will correctly associate an **else** with the most recent unmatched **then**.

We can construct a predictive-parser table from any LL(1) grammar. The *FIRST* and *FOLLOW* sets control the selection of a right-hand side in much the same way as they did in the recursive-descent parsers. Being in Row Q of the table is like being in Function `Q` in the recursive parser; in that row, the table essentially represents in compact form what the *FIRST* and *FOLLOW* sets would tell you about choosing the next right-hand side. If the grammar meets the criteria set forth in the definition of an LL(1) grammar at the end of Section 3.4, then there will be no multiple entries in the table.

LL(k) parsers, in which we have k-character lookahead, are possible but rarely used; the table is too big.

3.5 SUMMARY

In this chapter, after a general introduction to grammars, we have considered top-down parsers in detail. We found out about left recursions, left factorization, and the use of *FIRST* and *FOLLOW* sets to make the parser predictive.

Recursive-descent parsers are now probably primarily of historical interest, although they provide a good introduction to top-down parsing generally. The table-driven predictive parser is the one really practical parser we have seen so far. In the next chapter, we will turn our attention to bottom-up parsing, and there we will meet two more practical parsers.

Generative grammars have been applied to other problems besides compilation. Naturally, they are of great interest in linguistics, and they have been applied to the problem of computer recognition of speech. They have also been used in pattern recognition, and automatic typesetting programs like `troff` and TEX use parsers to analyze the input string. In Chapter 2, I mentioned briefly the need to parse regular expressions to write a program like Lex; see Problem 3.18 for the first stages of this procedure.

PROBLEMS

3.1. Write a leftmost derivation for the dog-bone example of Figure 3.1.

3.2. Given the grammar $G =$

$$A \rightarrow Ba \mid bC$$
$$B \rightarrow d \mid eBf$$
$$C \rightarrow gC \mid g$$