Chapter 3

Parsers

CHAPTER OVERVIEW

The parser's task is to determine if the input program, represented by the stream of classified words produced by the scanner, is a valid sentence in the programming language. To do so, the parser attempts to build a derivation for the input program, using a grammar for the programming language.

This chapter introduces context-free grammars, a notation used to specify the syntax of programming languages. It develops several techniques for finding a derivation, given a grammar and an input program.

Keywords: Parsing, Grammar, LL(1), LR(1), Recursive Descent

3.1 INTRODUCTION

Parsing is the second stage of the compiler's front end. The parser works with the program as transformed by the scanner; it sees a stream of words where each word is annotated with a syntactic category (analogous to its part of speech). The parser derives a syntactic structure for the program, fitting the words into a grammatical model of the source programming language. If the parser determines that the input stream is a valid program, it builds a concrete model of the program for use by the later phases of compilation. If the input stream is not a valid program, the parser reports the problem and appropriate diagnostic information to the user.

As a problem, parsing has many similarities to scanning. The formal problem has been studied extensively as part of formal language theory; that work forms the theoretical basis for the practical parsing techniques used in most compilers. Speed matters; all of the techniques that we will study take time proportional to the size of the program and its representation. Low-level detail affects performance; the same implementation tradeoffs arise

Engineering a Compiler. DOI: 10.1016/B978-0-12-088478-0.00003-7

in parsing as in scanning. The techniques in this chapter are amenable to implementation as table-driven parsers, direct-coded parsers, and hand-coded parsers. Unlike scanners, where hand-coding is common, tool-generated parsers are more common than hand-coded parsers.

Conceptual Roadmap

The primary task of the parser is to determine whether or not the input program is a syntactically valid sentence in the source language. Before we can build parsers that answer this question, we need both a formal mechanism for specifying the syntax of the source language and a systematic method of determining membership in this formally specified language. By restricting the form of the source language to a set of languages called context-free languages, we can ensure that the parser can efficiently answer the membership question. Section 3.2 introduces context-free grammars (CFGs) as a notation for specifying syntax.

Many algorithms have been proposed to answer the membership question for CFGs. This chapter examines two different approaches to the problem. Section 3.3 introduces top-down parsing in the form of recursive-descent parsers and LL(1) parsers. Section 3.4 examines bottom-up parsing as exemplified by LR(1) parsers. Section 3.4.2 presents the detailed algorithm for generating canonical LR(1) parsers. The final section explores several practical issues that arise in parser construction.

Overview

A compiler's parser has the primary responsibility for recognizing syntax—that is, for determining if the program being compiled is a valid sentence in the syntactic model of the programming language. That model is expressed as a formal grammar G; if some string of words s is in the language defined by G we say that G *derives* s. For a stream of words s and a grammar G, the parser tries to build a constructive proof that s can be derived in G—a process called *parsing*.

Parsing
given a stream s of words and a grammar G, find a derivation in G that produces s

Parsing algorithms fall into two general categories. Top-down parsers try to match the input stream against the productions of the grammar by predicting the next word (at each point). For a limited class of grammars, such prediction can be both accurate and efficient. Bottom-up parsers work from low-level detail—the actual sequence of words—and accumulate context until the derivation is apparent. Again, there exists a restricted class of grammars for which we can generate efficient bottom-up parsers. In practice, these restricted sets of grammars are large enough to encompass most features of interest in programming languages.

3.2 EXPRESSING SYNTAX

The task of the parser is to determine whether or not some stream of words fits into the syntax of the parser's intended source language. Implicit in this description is the notion that we can describe syntax and check it; in practice, we need a notation to describe the syntax of languages that people might use to program computers. In Chapter 2, we worked with one such notation, regular expressions. They provide a concise notation for describing syntax and an efficient mechanism for testing the membership of a string in the language described by an RE. Unfortunately, REs lack the power to describe the full syntax of most programming languages.

For most programming languages, syntax is expressed in the form of a context-free grammar. This section introduces and defines CFGs and explores their use in syntax-checking. It shows how we can begin to encode meaning into syntax and structure. Finally, it introduces the ideas that underlie the efficient parsing techniques described in the following sections.

3.2.1 Why Not Regular Expressions?

To motivate the use of CFGs, consider the problem of recognizing algebraic expressions over variables and the operators +, -, ×, and ÷. We can define "variable" as any string that matches the RE $[a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*$, a simplified, lowercase version of an Algol identifier. Now, we can define an expression as follows:

$$[a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^* \quad ((+\,|\,-\,|\times|\div)\ \ [a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*\,)^*$$

This RE matches "a + b × c" and "fee ÷ fie × foe". Nothing about the RE suggests a notion of operator precedence; in "a + b × c," which operator executes first, the + or the ×? The standard rule from algebra suggests × and ÷ have precedence over + and -. To enforce other evaluation orders, normal algebraic notation includes parentheses.

Adding parentheses to the RE in the places where they need to appear is somewhat tricky. An expression can start with a '(', so we need the option for an initial (. Similarly, we need the option for a final).

We will underline (and) so that they are visually distinct from the (and) used for grouping in RES.

$$(\underline{(}\,|\epsilon)\ [a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*$$
$$((+\,|\,-\,|\times|\ \div)\ \ [a\ldots z]([a\ldots z]\,|\,[0\ldots 9])^*\,)^*\ (\underline{)}\,|\epsilon)$$

This RE can produce an expression enclosed in parentheses, but not one with internal parentheses to denote precedence. The internal instances of (all occur before a variable; similarly, the internal instances of) all occur

after a variable. This observation suggests the following RE:

$$(\underline{(}\,|\epsilon)\ [a\ldots z]\ ([a\ldots z]\,|\,[0\ldots 9])^*$$
$$((+\,|\,-\,|\times|\div)\ \ [a\ldots z]\ ([a\ldots z]\,|\,[0\ldots 9])^*\ (\underline{)}\,|\epsilon)\ )^*$$

Notice that we simply moved the final $\underline{)}$ inside the closure.

This RE matches both "`a+b×c`" and "`(a+b)×c`." It will match any correctly parenthesized expression over variables and the four operators in the RE. Unfortunately, it also matches many syntactically incorrect expressions, such as "`a+(b×c`" and "`a+b)×c)`." In fact, we cannot write an RE that will match all expressions with balanced parentheses. (Paired constructs, such as `begin` and `end` or `then` and `else`, play an important role in most programming languages.) This fact is a fundamental limitation of REs; the corresponding recognizers cannot count because they have only a finite set of states. The language $(\underline{(}^m\ \underline{)}^n$ where $m = n$ is not regular. In principle, DFAs cannot count. While they work well for microsyntax, they are not suitable to describe some important programming language features.

3.2.2 Context-Free Grammars

Context-free grammar
For a language *L*, its CFG defines the sets of strings of symbols that are valid sentences in *L*.

To describe programming language syntax, we need a more powerful notation than regular expressions that still leads to efficient recognizers. The traditional solution is to use a context-free grammar (CFG). Fortunately, large subclasses of the CFGs have the property that they lead to efficient recognizers.

Sentence
a string of symbols that can be derived from the rules of a grammar

A context-free grammar, *G*, is a set of rules that describe how to form sentences. The collection of sentences that can be derived from *G* is called the *language defined by G*, denoted *G*. The set of languages defined by context-free grammars is called the set of context-free languages. An example may help. Consider the following grammar, which we call *SN*:

$$\begin{aligned} \textit{SheepNoise} &\rightarrow \texttt{baa}\ \textit{SheepNoise} \\ &\ \ |\ \ \texttt{baa} \end{aligned}$$

Production
Each rule in a CFG is called a *production*.

Nonterminal symbol
a syntactic variable used in a grammar's productions

Terminal symbol
a word that can occur in a sentence

A word consists of a lexeme and its syntactic category. Words are represented in a grammar by their syntactic category

The first rule, or *production* reads "*SheepNoise* can derive the word `baa` followed by more *SheepNoise*." Here *SheepNoise* is a syntactic variable representing the set of strings that can be derived from the grammar. We call such a syntactic variable a *nonterminal symbol*. Each word in the language defined by the grammar is a *terminal symbol*. The second rule reads "*SheepNoise* can also derive the string `baa`."

To understand the relationship between the *SN* grammar and *L*(*SN*), we need to specify how to apply rules in *SN* to derive sentences in *L*(*SN*). To begin, we must identify the *goal symbol* or *start symbol* of *SN*. The goal symbol

BACKUS-NAUR FORM

The traditional notation used by computer scientists to represent a context-free grammar is called *Backus-Naur form*, or BNF. BNF denoted nonterminal symbols by wrapping them in angle brackets, like ⟨SheepNoise⟩. Terminal symbols were underlined. The symbol ::= means "derives," and the symbol | means "also derives." In BNF, the sheep noise grammar becomes:

⟨SheepNoise⟩ ::= baa ⟨SheepNoise⟩
| baa

This is completely equivalent to our grammar *SN*.

BNF has its origins in the late 1950s and early 1960s [273]. The syntactic conventions of angle brackets, underlining, ::=, and | arose from the limited typographic options available to people writing language descriptions. (For example, see David Gries' book *Compiler Construction for Digital Computers*, which was printed entirely on a standard lineprinter [171].) Throughout this book, we use a typographically updated form of BNF. Nonterminals are written in *italics*. Terminals are written in the `typewriter font`. We use the symbol → for "derives."

represents the set of all strings in $L(SN)$. As such, it cannot be one of the words in the language. Instead, it must be one of the nonterminal symbols introduced to add structure and abstraction to the language. Since *SN* has only one nonterminal, *SheepNoise* must be the goal symbol.

To derive a sentence, we start with a prototype string that contains just the goal symbol, *SheepNoise*. We pick a nonterminal symbol, α, in the prototype string, choose a grammar rule, $\alpha \rightarrow \beta$, and rewrite α with β. We repeat this rewriting process until the prototype string contains no more nonterminals, at which point it consists entirely of words, or terminal symbols, and is a sentence in the language.

Derivation
a sequence of rewriting steps that begins with the grammar's start symbol and ends with a sentence in the language

At each point in this derivation process, the string is a collection of terminal or nonterminal symbols. Such a string is called a *sentential form* if it occurs in some step of a valid derivation. Any sentential form can be derived from the start symbol in zero or more steps. Similarly, from any sentential form we can derive a valid sentence in zero or more steps. Thus, if we begin with *SheepNoise* and apply successive rewrites using the two rules, at each step in the process the string is a sentential form. When we have reached the point where the string contains only terminal symbols, the string is a sentence in $L(SN)$.

Sentential form
a string of symbols that occurs as one step in a valid derivation

CONTEXT-FREE GRAMMARS

Formally, a context-free grammar *G* is a quadruple (*T, NT, S, P*) where:

- *T* is the set of terminal symbols, or words, in the language *L*(*G*). Terminal symbols correspond to syntactic categories returned by the scanner.
- *NT* is the set of nonterminal symbols that appear in the productions of *G*. Nonterminals are syntactic variables introduced to provide abstraction and structure in the productions.
- *S* is a nonterminal designated as the *goal symbol* or *start symbol* of the grammar. *S* represents the set of sentences in *L*(*G*).
- *P* is the set of productions or rewrite rules in *G*. Each rule in *P* has the form $NT \rightarrow (T \cup NT)^{+}$; that is, it replaces a single nonterminal with a string of one or more grammar symbols.

The sets *T* and *NT* can be derived directly from the set of productions, *P*. The start symbol may be unambiguous, as in the *SheepNoise* grammar, or it may not be obvious, as in the following grammar:

```
Paren   →  ( Bracket )        Bracket  →  [ Paren ]
        |  (       )                  |  [       ]
```

In this case, the choice of start symbol determines the shape of the outer brackets. Using *Paren* as *S* ensures that every sentence has an outermost pair of parentheses, while using *Bracket* forces an outermost pair of square brackets. To allow either, we would need to introduce a new symbol *Start* and the productions *Start*→*Paren* | *Bracket*.

Some tools that manipulate grammars require that *S* not appear on the right-hand side of any production, which makes *S* easy to discover.

To derive a sentence in *SN*, we start with the string that consists of one symbol, *SheepNoise*. We can rewrite *SheepNoise* with either rule 1 or rule 2. If we rewrite *SheepNoise* with rule 2, the string becomes `baa` and has no further opportunities for rewriting. The rewrite shows that `baa` is a valid sentence in *L*(*SN*). The other choice, rewriting the initial string with rule 1, leads to a string with two symbols: `baa` *SheepNoise*. This string has one remaining nonterminal; rewriting it with rule 2 leads to the string `baa baa`, which is a sentence in *L*(*SN*). We can represent these derivations in tabular form:

Rule	Sentential Form
	SheepNoise
2	`baa`

Rewrite with Rule 2

Rule	Sentential Form
	SheepNoise
1	`baa` *SheepNoise*
2	`baa baa`

Rewrite with Rules 1 Then 2

As a notational convenience, we will use $\rightarrow^+$ to mean "derives in one or more steps." Thus, *SheepNoise* $\rightarrow^+$ `baa` and *SheepNoise* $\rightarrow^+$ `baa baa`.

Rule 1 lengthens the string while rule 2 eliminates the nonterminal *SheepNoise*. (The string can never contain more than one instance of *SheepNoise*.) All valid strings in *SN* are derived by zero or more applications of rule 1, followed by rule 2. Applying rule 1 k times followed by rule 2 generates a string with $k + 1$ `baa`s.

3.2.3 More Complex Examples

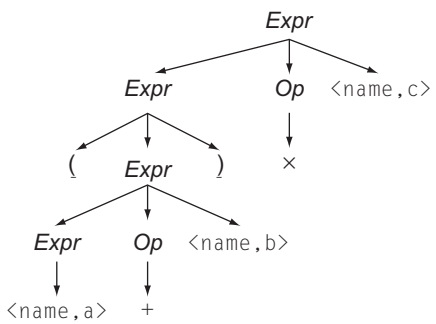
The *SheepNoise* grammar is too simple to exhibit the power and complexity of CFGs. Instead, let's revisit the example that showed the shortcomings of REs: the language of expressions with parentheses.

1	*Expr*	→	(*Expr*)
2		|	*Expr Op* `name`
3		|	`name`
4	*Op*	→	+
5		|	-
6		|	×
7		|	÷

Beginning with the start symbol, *Expr*, we can generate two kinds of subterms: parenthesized subterms, with rule 1, or plain subterms, with rule 2. To generate the sentence "`(a+b)×c`", we can use the following rewrite sequence (2,6,1,2,4,3), shown on the left. Remember that the grammar deals with syntactic categories, such as `name` rather than lexemes such as `a`, `b`, or `c`.

Rule	Sentential Form
	Expr
2	*Expr Op* `name`
6	*Expr* × `name`
1	(*Expr*) × `name`
2	(*Expr Op* `name`) × `name`
4	(*Expr* + `name`) × `name`
3	(`name` + `name`) × `name`

Rightmost Derivation of `( a + b ) × c`



Corresponding Parse Tree

The tree on the right, called a *parse tree*, represents the derivation as a graph.

Parse tree or syntax tree
a graph that represents a derivation

This simple CFG for expressions cannot generate a sentence with unbalanced or improperly nested parentheses. Only rule 1 can generate an open parenthesis; it also generates the matching close parenthesis. Thus, it cannot generate strings such as "a + (b × c" or "a + b) × c)," and a parser built from the grammar will not accept the such strings. (The best RE in Section 3.2.1 matched both of these strings.) Clearly, CFGs provide us with the ability to specify constructs that REs do not.

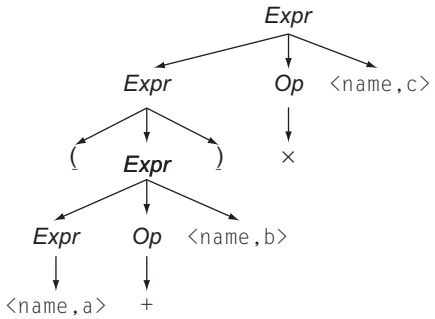
Rightmost derivation
a derivation that rewrites, at each step, the rightmost nonterminal

The derivation of (a + b) × c rewrote, at each step, the rightmost remaining nonterminal symbol. This systematic behavior was a choice; other choices are possible. One obvious alternative is to rewrite the leftmost nonterminal at each step. Using leftmost choices would produce a different derivation sequence for the same sentence. The leftmost derivation of (a + b) × c would be:

Leftmost derivation
a derivation that rewrites, at each step, the leftmost nonterminal

Rule	Sentential Form
	Expr
2	*Expr* *Op* name
1	(*Expr*) *Op* name
2	(*Expr* *Op* name) *Op* name
3	(name *Op* name) *Op* name
4	(name + name) *Op* name
6	(name + name) × name

Leftmost Derivation of (a + b) x c



Corresponding Parse Tree

The leftmost and rightmost derivations use the same set of rules; they apply those rules in a different order. Because a parse tree represents the rules applied, but not the order of their application, the parse trees for the two derivations are identical.

From the compiler's perspective, it is important that each sentence in the language defined by a CFG has a unique rightmost (or leftmost) derivation. If multiple rightmost (or leftmost) derivations exist for some sentence, then, at some point in the derivation, multiple distinct rewrites of the rightmost (or leftmost) nonterminal lead to the same sentence. A grammar in which multiple rightmost (or leftmost) derivations exist for a sentence is called an *ambiguous* grammar. An ambiguous grammar can produce multiple derivations and multiple parse trees. Since later stages of translation will associate meaning with the detailed shape of the parse tree, multiple parse trees imply multiple possible meanings for a single program—a bad property for a programming language to have. If the compiler cannot be sure of the meaning of a sentence, it cannot translate it into a definitive code sequence.

Ambiguity
A grammar *G* is *ambiguous* if some sentence in *L*(*G*) has more than one rightmost (or leftmost) derivation.

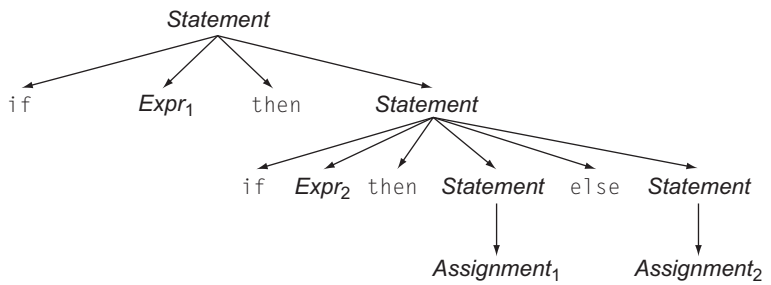
The classic example of an ambiguous construct in the grammar for a programming language is the `if-then-else` construct of many Algol-like languages. The straightforward grammar for `if-then-else` might be

1	*Statement*	→	`if` *Expr* `then` *Statement* `else` *Statement*
2		|	`if` *Expr* `then` *Statement*
3		|	*Assignment*
4		|	. . . *other statements* . . .

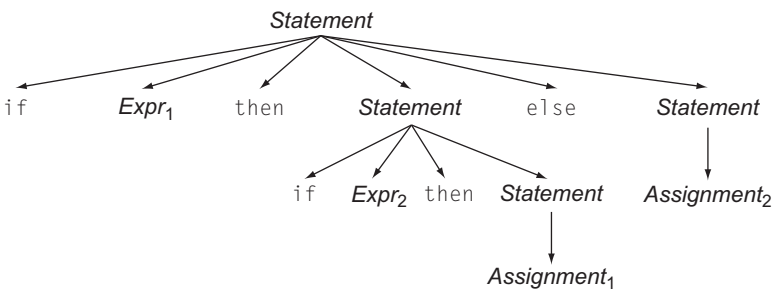
This fragment shows that the `else` is optional. Unfortunately, the code fragment

`if` $Expr_1$ `then` `if` $Expr_2$ `then` $Assignment_1$ `else` $Assignment_2$

has two distinct rightmost derivations. The difference between them is simple. The first derivation has $Assignment_2$ controlled by the inner `if`, so $Assignment_2$ executes when $Expr_1$ is true and $Expr_2$ is false:



The second derivation associates the `else` clause with the first `if`, so that $Assignment_2$ executes when $Expr_1$ is false, independent of the value of $Expr_2$:



Clearly, these two derivations produce different behaviors in the compiled code.

To remove this ambiguity, the grammar must be modified to encode a rule that determines which `if` controls an `else`. To fix the `if-then-else` grammar, we can rewrite it as

1	*Statement*	→	`if` *Expr* `then` *Statement*
2		|	`if` *Expr* `then` *WithElse* `else` *Statement*
3		|	*Assignment*
4	*WithElse*	→	`if` *Expr* `then` *WithElse* `else` *WithElse*
5		|	*Assignment*

The solution restricts the set of statements that can occur in the `then` part of an `if-then-else` construct. It accepts the same set of sentences as the original grammar, but ensures that each `else` has an unambiguous match to a specific `if`. It encodes into the grammar a simple rule—bind each `else` to the innermost unclosed `if`. It has only one rightmost derivation for the example.

Rule	Sentential Form
	Statement
1	`if` *Expr* `then` *Statement*
2	`if` *Expr* `then` `if` *Expr* `then` *WithElse* `else` *Statement*
3	`if` *Expr* `then` `if` *Expr* `then` *WithElse* `else` *Assignment*
5	`if` *Expr* `then` `if` *Expr* `then` *Assignment* `else` *Assignment*

The rewritten grammar eliminates the ambiguity.

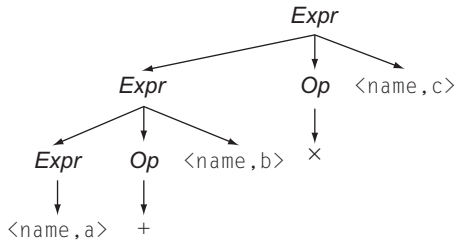
The `if-then-else` ambiguity arises from a shortcoming in the original grammar. The solution resolves the ambiguity in a way by imposing a rule that is easy for the programmer to remember. (To avoid the ambiguity entirely, some language designers have restructured the `if-then-else` construct by introducing `elseif` and `endif`.) In Section 3.5.3, we will look at other kinds of ambiguity and systematic ways of handling them.

3.2.4 Encoding Meaning into Structure

The `if-then-else` ambiguity points out the relationship between meaning and grammatical structure. However, ambiguity is not the only situation where meaning and grammatical structure interact. Consider the parse tree that would be built from a rightmost derivation of the simple expression `a + b x c`.

Rule	Sentential Form
	Expr
2	*Expr Op* name
6	*Expr* x name
2	*Expr Op* name x name
4	*Expr* + name x name
3	name + name x name

Derivation of a + b x c



Corresponding Parse Tree

One natural way to evaluate the expression is with a simple postorder treewalk. It would first compute a + b and then multiply that result by c to produce the result (a + b) x c. This evaluation order contradicts the classic rules of algebraic precedence, which would evaluate it as a + (b x c). Since the ultimate goal of parsing the expression is to produce code that will implement it, the expression grammar should have the property that it builds a tree whose "natural" treewalk evaluation produces the correct result.

The real problem lies in the structure of the grammar. It treats all of the arithmetic operators in the same way, without any regard for precedence. In the parse tree for (a + b) x c, the fact that the parenthetic subexpression was forced to go through an extra production in the grammar adds a level to the parse tree. The extra level, in turn, forces a postorder treewalk to evaluate the parenthetic subexpression before it evaluates the multiplication.

We can use this effect to encode operator precedence levels into the grammar. First, we must decide how many levels of precedence are required. In the simple expression grammar, we have three levels of precedence: highest precedence for ( ), medium precedence for x and ÷, and lowest precedence for + and -. Next, we group the operators at distinct levels and use a nonterminal to isolate the corresponding part of the grammar. Figure 3.1

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* x *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	num
9		|	name

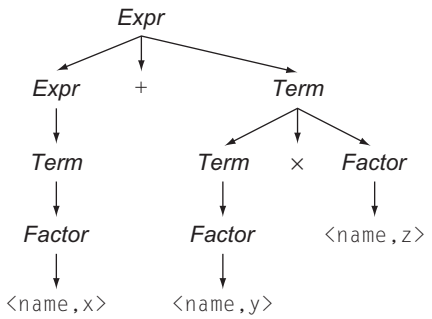
■ **FIGURE 3.1** The Classic Expression Grammar.

shows the resulting grammar; it includes a unique start symbol, *Goal*, and a production for the terminal symbol `num` that we will use in later examples.

In the classic expression grammar, *Expr*, represents the level for + and -, *Term* represents the level for × and ÷, and *Factor* represents the level for `( )`. In this form, the grammar derives a parse tree for `a + b x c` that is consistent with standard algebraic precedence, as shown below.

Rule	Sentential Form
	Expr
1	*Expr* + *Term*
4	*Expr* + *Term* x *Factor*
6	*Expr* + *Term* x `name`
9	*Expr* + *Factor* x `name`
9	*Expr* + `name` x `name`
3	*Term* + `name` x `name`
6	*Factor* + `name` x `name`
9	`name` + `name` x `name`

Derivation of `a + b x c`



Corresponding Parse Tree

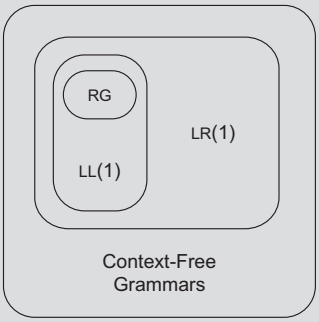
A postorder treewalk over this parse tree will first evaluate `b x c` and then add the result to `a`. This implements the standard rules of arithmetic precedence. Notice that the addition of nonterminals to enforce precedence adds interior nodes to the tree. Similarly, substituting the individual operators for occurrences of *Op* removes interior nodes from the tree.

Other operations require high precedence. For example, array subscripts should be applied before standard arithmetic operations. This ensures, for example, that `a + b[i]` evaluates `b[i]` to a value before adding it to `a`, as opposed to treating `i` as a subscript on some array whose location is computed as `a + b`. Similarly, operations that change the type of a value, known as *type casts* in languages such as C or Java, have higher precedence than arithmetic but lower precedence than parentheses or subscripting operations.

If the language allows assignment inside expressions, the assignment operator should have low precedence. This ensures that the code completely evaluates both the left-hand side and the right-hand side of the assignment before performing the assignment. If assignment (←) had the same precedence as addition, for example, the expression `a` ← `b + c` would assign `b`'s value to `a` before performing the addition, assuming a left-to-right evaluation.

CLASSES OF CONTEXT-FREE GRAMMARS AND THEIR PARSERS

We can partition the universe of context-free grammars into a hierarchy based on the difficulty of parsing the grammars. This hierarchy has many levels. This chapter mentions four of them, namely, arbitrary CFGs, LR(1) grammars, LL(1) grammars, and regular grammars (RGs). These sets nest as shown in the diagram.



Arbitrary CFGs require more time to parse than the more restricted LR(1) or LL(1) grammars. For example, Earley's algorithm parses arbitrary CFGs in $\mathbf{O}(n^3)$ time, worst case, where n is the number of words in the input stream. Of course, the actual running time may be better. Historically, compiler writers have shied away from "universal" techniques because of their perceived inefficiency.

The LR(1) grammars include a large subset of the unambiguous CFGs. LR(1) grammars can be parsed, bottom-up, in a linear scan from left to right, looking at most one word ahead of the current input symbol. The widespread availability of tools that derive parsers from LR(1) grammars has made LR(1) parsers "everyone's favorite parsers."

The LL(1) grammars are an important subset of the LR(1) grammars. LL(1) grammars can be parsed, top-down, in a linear scan from left to right, with a one-word lookahead. LL(1) grammars can be parsed with either a hand-coded recursive-descent parser or a generated LL(1) parser. Many programming languages can be written in an LL(1) grammar.

Regular grammars (RGs) are CFGs that generate regular languages. A regular grammar is a CFG where productions are restricted to two forms, either $A \rightarrow a$ or $A \rightarrow aB$, where $A, B \in NT$ and $a \in T$. Regular grammars are equivalent to regular expressions; they encode precisely those languages that can be recognized by a DFA. The primary use for regular languages in compiler construction is to specify scanners.

Almost all programming-language constructs can be expressed in LR(1) form and, often, in LL(1) form. Thus, most compilers use a fast-parsing algorithm based on one of these two restricted classes of CFG.

3.2.5 Discovering a Derivation for an Input String

We have seen how to use a CFG G as a rewriting system to generate sentences that are in $L(G)$. In contrast, a compiler must infer a derivation for a

given input string, or determine that no such derivation exists. The process of constructing a derivation from a specific input sentence is called *parsing*.

A parser takes, as input, an alleged program written in some source language.

The parser sees the program as it emerges from the scanner: a stream of words annotated with their syntactic categories. Thus, the parser would see `a + b x c` as ⟨name,a⟩ + ⟨name,b⟩ x ⟨name,c⟩. As output, the parser needs to produce either a derivation for the input program or an error message for an invalid program. For an unambiguous language, a parse tree is equivalent to a derivation; thus, we can think of the parser's output as a parse tree.

It is useful to visualize the parser as building a syntax tree for the input program. The parse tree's root is known; it represents the grammar's start symbol. The leaves of the parse tree are known; they must match, in order from left to right, the stream of words returned by the scanner. The hard part of parsing lies in discovering the grammatical connection between the leaves and the root. Two distinct and opposite approaches for constructing the tree suggest themselves:

1. *Top-down parsers* begin with the root and grow the tree toward the leaves. At each step, a top-down parser selects a node for some nonterminal on the lower fringe of the tree and extends it with a subtree that represents the right-hand side of a production that rewrites the nonterminal.
2. *Bottom-up parsers* begin with the leaves and grow the tree toward the root. At each step, a bottom-up parser identifies a contiguous substring of the parse tree's upper fringe that matches the right-hand side of some production; it then builds a node for the rule's left-hand side and connects it into the tree.

In either scenario, the parser makes a series of choices about which productions to apply. Most of the intellectual complexity in parsing lies in the mechanisms for making these choices. Section 3.3 explores the issues and algorithms that arise in top-down parsing, while Section 3.4 examines bottom-up parsing in depth.

3.3 TOP-DOWN PARSING

A top-down parser begins with the root of the parse tree and systematically extends the tree downward until its leaves match the classified words returned by the scanner. At each point, the process considers a partially built parse tree. It selects a nonterminal symbol on the lower fringe of the tree and extends it by adding children that correspond to the right-hand side of

some production for that nonterminal. It cannot extend the frontier from a terminal. This process continues until either

a. the fringe of the parse tree contains only terminal symbols, and the input stream has been exhausted, or
b. a clear mismatch occurs between the fringe of the partially built parse tree and the input stream.

In the first case, the parse succeeds. In the second case, two situations are possible. The parser may have selected the wrong production at some earlier step in the process, in which case it can backtrack, systematically reconsidering earlier decisions. For an input string that is a valid sentence, backtracking will lead the parser to a correct sequence of choices and let it construct a correct parse tree. Alternatively, if the input string is not a valid sentence, backtracking will fail and the parser should report the syntax error to the user.

One key insight makes top-down parsing efficient: *a large subset of the context-free grammars can be parsed without backtracking.* Section 3.3.1 shows transformations that can often convert an arbitrary grammar into one suitable for backtrack-free top-down parsing. The two sections that follow it introduce two distinct techniques for constructing top-down parsers: hand-coded recursive-descent parsers and generated LL(1) parsers.

Figure 3.2 shows a concrete algorithm for a top-down parser that constructs a leftmost derivation. It builds a parse tree, anchored at the variable `root`. It uses a stack, with access functions `push( )` and `pop( )`, to track the unmatched portion of the fringe.

The main portion of the parser consists of a loop that focuses on the leftmost unmatched symbol on the partially-built parse tree's lower fringe. If the focus symbol is a nonterminal, it expands the parse tree downward; it chooses a production, builds the corresponding part of the parse tree, and moves the focus to the leftmost symbol on this new portion of the fringe. If the focus symbol is a terminal, it compares the focus against the next word in the input. A match moves both the focus to the next symbol on the fringe and advances the input stream.

If the focus is a terminal symbol that does not match the input, the parser must backtrack. First, it systematically considers alternatives for the most recently chosen rule. If it exhausts those alternatives, it moves back up the parse tree and reconsiders choices at a higher level in the parse tree. If this process fails to match the input, the parser reports a syntax error. Backtracking increases the asymptotic cost of parsing; in practice, it is an expensive way to discover syntax errors.

```
root ← node for the start symbol, S;
focus ← root;
push(null);
word ← NextWord();

while (true) do;
   if (focus is a nonterminal) then begin;
      pick next rule to expand focus (A → β1,β2,...,βn);
      build nodes for β1,β2...βn as children of focus;
      push(βn, βn−1, ..., β2);
      focus ← β1;
   end;
   else if (word matches focus) then begin;
      word ← NextWord();
      focus ← pop()
   end;
   else if (word = eof and focus = null)
      then accept the input and return root;
      else backtrack;
end;
```

■ **FIGURE 3.2** A Leftmost, Top-Down Parsing Algorithm.

The implementation of "*backtrack*" is straightforward. It sets *focus* to its parent in the partially-built parse tree and disconnects its children. If an untried rule remains with *focus* on its left-hand side, the parser expands *focus* by that rule. It builds children for each symbol on the right-hand side, pushes those symbols onto the stack in right-to-left order, and sets *focus* to point at the first child. If no untried rule remains, the parser moves up another level and tries again. When it runs out of possibilities, it reports a syntax error and quits.

To facilitate finding the "next" rule, the parser can store the rule number in a nonterminal's node when it expands that node.

When it backtracks, the parser must also rewind the input stream. Fortunately, the partial parse tree encodes enough information to make this action efficient. The parser must place each matched terminal in the discarded production back into the input stream, an action it can take as it disconnects them from the parse tree in a left-to-right traversal of the discarded children.

3.3.1 Transforming a Grammar for Top-Down Parsing

The efficiency of a top-down parser depends critically on its ability to pick the correct production each time that it expands a nonterminal. If the parser always makes the right choice, top-down parsing is efficient. If it makes poor choices, the cost of parsing rises. For some grammars, the worst case

behavior is that the parser does not terminate. This section examines two structural issues with CFGs that lead to problems with top-down parsers and presents transformations that the compiler writer can apply to the grammar to avoid these problems.

A Top-Down Parser with Oracular Choice

As an initial exercise, consider the behavior of the parser from Figure 3.2 with the classic expression grammar in Figure 3.1 when applied to the string a + b x c. For the moment, assume that the parser has an oracle that picks the correct production at each point in the parse. With oracular choice, it might proceed as shown in Figure 3.3. The right column shows the input string, with a marker ↑ to indicate the parser's current position in the string. The symbol → in the rule column represents a step in which the parser matches a terminal symbol against the input string and advances the input. At each step, the sentential form represents the lower fringe of the partially-built parse tree.

With oracular choice, the parser should take a number of steps proportional to the length of the derivation plus the length of the input. For a + b x c the parser applied eight rules and matched five words.

Notice, however, that oracular choice means inconsistent choice. In both the first and second steps, the parser considered the nonterminal *Expr*. In the first step, it applied rule 1, *Expr* → *Expr* + *Term*. In the second step, it applied rule 3, *Expr* → *Term*. Similarly, when expanding *Term* in an attempt to match a, it applied rule 6, *Term* → *Factor*, but when expanding *Term* to match b,

Rule	Sentential Form	Input
	Expr	↑ name + name x name
1	*Expr* + *Term*	↑ name + name x name
3	*Term* + *Term*	↑ name + name x name
6	*Factor* + *Term*	↑ name + name x name
9	name + *Term*	↑ name + name x name
→	name + *Term*	name ↑ + name x name
→	name + *Term*	name + ↑ name x name
4	name + *Term* x *Factor*	name + ↑ name x name
6	name + *Factor* x *Factor*	name + ↑ name x name
9	name + name x *Factor*	name + ↑ name x name
→	name + name x *Factor*	name + name ↑ x name
→	name + name x *Factor*	name + name x ↑ name
9	name + name x name	name + name x ↑ name
→	name + name x name	name + name x name ↑

■ **FIGURE 3.3** Leftmost, Top-Down Parse of a + b x c with Oracular Choice.

it applied rule 4, *Term* → *Term* × *Factor*. It would be difficult to make the top-down parser work with consistent, algorithmic choice when using this version of the expression grammar.

Eliminating Left Recursion

One problem with the combination of the classic expression grammar and a leftmost, top-down parser arises from the structure of the grammar. To see the difficulty, consider an implementation that always tries to apply the rules in the order in which they appear in the grammar. Its first several actions would be:

Rule	Sentential Form	Input
	Expr	↑ `name + name × name`
1	*Expr* + *Term*	↑ `name + name × name`
1	*Expr* + *Term* + *Term*	↑ `name + name × name`
1	...	↑ `name + name × name`

It starts with *Expr* and tries to match `a`. It applies rule 1 to create the sentential form *Expr* + *Term* on the fringe. Now, it faces the nonterminal *Expr* and the input word `a`, again. By consistent choice, it applies rule 1 to replace *Expr* with *Expr* + *Term*. Of course, it still faces *Expr* and the input word `a`. With this grammar and consistent choice, the parser will continue to expand the fringe indefinitely because that expansion never generates a leading terminal symbol.

Left recursion
A rule in a CFG is left recursive if the first symbol on its right-hand side is the symbol on its left-hand side or can derive that symbol.

The former case is called *direct* left recursion, while the latter case is called *indirect* left recursion.

This problem arises because the grammar uses *left recursion* in productions 1, 2, 4, and 5. With left-recursion, a top-down parser can loop indefinitely without generating a leading terminal symbol that the parser can match (and advance the input). Fortunately, we can reformulate a left-recursive grammar so that it uses right recursion—any recursion involves the rightmost symbol in a rule.

The translation from left recursion to right recursion is mechanical. For direct left recursion, like the one shown below to the left, we can rewrite the individual productions to use right recursion, shown on the right.

$$\begin{array}{rcl} Fee & \rightarrow & Fee\ \alpha \\ & | & \beta \end{array} \qquad\qquad \begin{array}{rcl} Fee & \rightarrow & \beta\ Fee' \\ Fee' & \rightarrow & \alpha\ Fee' \\ & | & \epsilon \end{array}$$

The transformation introduces a new nonterminal, Fee', and transfers the recursion onto Fee'. It also adds the rule $Fee' \rightarrow \epsilon$, where ϵ represents the empty string. This ϵ-*production* requires careful interpretation in the parsing algorithm. To expand the production $Fee' \rightarrow \epsilon$, the parser simply sets

`focus` ← `pop()`, which advances its attention to the next node, terminal or nonterminal, on the fringe.

In the classic expression grammar, direct left recursion appears in the productions for both *Expr* and *Term*.

Original			Transformed		
Expr	→	*Expr* + *Term*	*Expr*	→	*Term Expr′*
	|	*Expr* - *Term*	*Expr′*	→	+ *Term* Expr′
	|	*Term*		|	- *Term Expr′*
				|	ϵ
Term	→	*Term* x *Factor*	*Term*	→	*Factor Term′*
	|	*Term* ÷ *Factor*	*Term′*	→	x *Factor Term′*
	|	*Factor*		|	÷ *Factor Term′*
				|	ϵ

Plugging these replacements back into the classic expression grammar yields a right-recursive variant of the grammar, shown in Figure 3.4. It specifies the same set of expressions as the classic expression grammar.

The grammar in Figure 3.4 eliminates the problem with nontermination. It does not avoid the need for backtracking. Figure 3.5 shows the behavior of the top-down parser with this grammar on the input `a + b x c`. The example still assumes oracular choice; we will address that issue in the next subsection. It matches all 5 terminals and applies 11 productions—3 more than it did with the left-recursive grammar. All of the additional rule applications involve productions that derive ϵ.

This simple transformation eliminates direct left recursion. We must also eliminate indirect left recursion, which occurs when a chain of rules such as $\alpha \rightarrow \beta$, $\beta \rightarrow \gamma$, and $\gamma \rightarrow \alpha\delta$ creates the situation that $\alpha \rightarrow^{+} \alpha\delta$. Such indirect left recursion is not always obvious; it can be obscured by a long chain of productions.

0	*Goal*	→	*Expr*	6	*Term′*	→	x *Factor Term′*
1	*Expr*	→	*Term Expr′*	7		|	÷ *Factor Term′*
2	*Expr′*	→	+ *Term Expr′*	8		|	ϵ
3		|	- *Term Expr′*	9	*Factor*	→	(*Expr*)
4		|	ϵ	10		|	num
5	*Term*	→	*Factor Term′*	11		|	name

■ **FIGURE 3.4** Right-Recursive Variant of the Classic Expression Grammar.

Rule	Sentential Form	Input
	Expr	↑ name + name x name
1	*Term Expr′*	↑ name + name x name
5	*Factor Term′ Expr′*	↑ name + name x name
11	name *Term′ Expr′*	↑ name + name x name
→	name *Term′ Expr′*	name ↑ + name x name
8	name *Expr′*	name ↑ + name x name
2	name + *Term Expr′*	name ↑ + name x name
→	name + *Term Expr′*	name + ↑ name x name
5	name + *Factor Term′ Expr′*	name + ↑ name x name
11	name + name *Term′ Expr′*	name + ↑ name x name
→	name + name *Term′ Expr′*	name + name ↑ x name
6	name + name x *Factor Term′ Expr′*	name + name ↑ x name
→	name + name x *Factor Term′ Expr′*	name + name x ↑ name
11	name + name x name *Term′ Expr′*	name + name x ↑ name
→	name + name x name *Term′ Expr′*	name + name x name ↑
8	name + name x name *Expr′*	name + name x name ↑
4	name + name x name	name + name x name ↑

■ **FIGURE 3.5** Leftmost, Top-Down Parse of a+b x c with the Right-Recursive Expression Grammar.

To convert indirect left recursion into right recursion, we need a more systematic approach than inspection followed by application of our transformation. The algorithm in Figure 3.6 eliminates all left recursion from a grammar by thorough application of two techniques: forward substitution to convert indirect left recursion into direct left recursion and rewriting direct left recursion as right recursion. It assumes that the original grammar has no cycles ($A \rightarrow^+ A$) and no ϵ-productions.

The algorithm imposes an arbitrary order on the nonterminals. The outer loop cycles through the nonterminals in this order. The inner loop looks for any production that expands A_i into a right-hand side that begins with A_j, for $j < i$. Such an expansion may lead to an indirect left recursion. To avoid this, the algorithm replaces the occurrence of A_j with all the alternative right-hand sides for A_j. That is, if the inner loop discovers a production $A_i \rightarrow A_j\gamma$, and $A_j \rightarrow \delta_1|\delta_2|\cdots|\delta_k$, then the algorithm replaces $A_i \rightarrow A_j\gamma$ with a set of productions $A_i \rightarrow \delta_1\gamma|\delta_2\gamma|\cdots|\delta_k\gamma$. This process eventually converts each indirect left recursion into a direct left recursion. The final step in the outer loop converts any direct left recursion on A_i to right recursion using the simple transformation shown earlier. Because new nonterminals are added at the end and only involve right recursion, the loop can ignore them—they do not need to be checked and converted.

```
impose an order on the nonterminals, A1, A2, ..., An

for i ← 1 to n do;
    for j ← 1 to i - 1 do;
        if ∃ a production Ai→Aj γ
            then replace Ai→Aj γ with one or more
                 productions that expand Aj
    end;
    rewrite the productions to eliminate
         any direct left recursion on Ai
end;
```

■ **FIGURE 3.6** Removal of Indirect Left Recursion.

Considering the loop invariant for the outer loop may make this clearer. At the start of the i^{th} outer loop iteration

$\forall\ k < i$, *no production expanding* A_k *has* A_l *in its rhs, for* $l < k$.

At the end of this process, $(i = n)$, all indirect left recursion has been eliminated through the repetitive application of the inner loop, and all immediate left recursion has been eliminated in the final step of each iteration.

Backtrack-Free Parsing

The major source of inefficiency in the leftmost, top-down parser arises from its need to backtrack. If the parser expands the lower fringe with the wrong production, it eventually encounters a mismatch between that fringe and the parse tree's leaves, which correspond to the words returned by the scanner. When the parser discovers the mismatch, it must undo the actions that built the wrong fringe and try other productions. The act of expanding, retracting, and re-expanding the fringe wastes time and effort.

In the derivation of Figure 3.5, the parser chose the correct rule at each step. With consistent choice, such as considering rules in order of appearance in the grammar, it would have backtracked on each `name`, first trying *Factor* → `(` *Expr* `)` and then *Factor* → `num` before deriving `name`. Similarly, the expansions by rules 4 and 8 would have considered the other alternatives before expanding to ϵ.

For this grammar, the parser can avoid backtracking with a simple modification. When the parser goes to select the next rule, it can consider both the focus symbol and the next input symbol, called the *lookahead symbol*. Using a one symbol lookahead, the parser can disambiguate all of the choices that arise in parsing the right-recursive expression grammar. Thus, we say that the grammar is *backtrack free* with a lookahead of one symbol. A backtrack-free grammar is also called a *predictive grammar*.

Backtrack-free grammar
a CFG for which the leftmost, top-down parser can always predict the correct rule with lookahead of at most one word

```
for each α ∈ (T ∪ eof ∪ ε) do;
    FIRST(α) ← α;
end;

for each A ∈ NT do;
    FIRST(A) ← ∅;
end;

while (FIRST sets are still changing) do;
   for each p ∈ P, where p has the form A→β do;
       if β is β1β2...βk, where βi ∈ T ∪ NT, then begin;
         rhs ← FIRST(β1) − {ε};
         i ← 1;
         while (ε ∈ FIRST(βi) and i ≤ k-1) do;
             rhs ← rhs ∪ (FIRST(βi+1) − {ε});
             i ← i + 1;
         end;
       end;
      if i = k and ε ∈ FIRST(βk)
           then rhs ← rhs ∪ {ε};
      FIRST(A) ← FIRST(A) ∪ rhs;
   end;
end;
```

■ **FIGURE 3.7** Computing FIRST Sets for Symbols in a Grammar.

We can formalize the property that makes the right-recursive expression grammar backtrack free. At each point in the parse, the choice of an expansion is obvious because each alternative for the leftmost nonterminal leads to a distinct terminal symbol. Comparing the next word in the input stream against those choices reveals the correct expansion.

FIRST set
For a grammar symbol α, FIRST(α) is the set of terminals that can appear at the start of a sentence derived from α.

The intuition is clear, but formalizing it will require some notation. For each grammar symbol α, define the set FIRST(α) as the set of terminal symbols that can appear as the first word in some string derived from α. The domain of FIRST is the set of grammar symbols, $T \cup NT \cup \{\epsilon, \texttt{eof}\}$ and its range is $T \cup \{\epsilon, \texttt{eof}\}$. If α is either a terminal, ϵ, or `eof`, then FIRST(α) has exactly one member, α. For a nonterminal A, FIRST(A) contains the complete set of terminal symbols that can appear as the leading symbol in a sentential form derived from A.

`eof` occurs implicitly at the end of every sentence in the grammar. Thus, it is in both the domain and range of FIRST.

Figure 3.7 shows an algorithm that computes the FIRST sets for each symbol in a grammar. As its initial step, the algorithm sets the FIRST sets for the

simple cases, terminals, ϵ, and `eof`. For the right-recursive expression grammar shown in Figure 3.4 on page 101, that initial step produces the following FIRST sets:

	num	name	+	-	×	÷	(	)	eof	ϵ
FIRST	num	name	+	-	x	÷	(	)	eof	ϵ

Next, the algorithm iterates over the productions, using the FIRST sets for the right-hand side of a production to derive the FIRST set for the nonterminal on its left-hand side. This process halts when it reaches a fixed point. For the right-recursive expression grammar, the FIRST sets of the nonterminals are:

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FIRST	(, name, num	+, -, ϵ	(, name, num	x, ÷, ϵ	(, name, num

We defined FIRST sets over single grammar symbols. It is convenient to extend that definition to strings of symbols. For a string of symbols, $s = \beta_1 \beta_2 \beta_3 \ldots \beta_k$, we define FIRST($s$) as the union of the FIRST sets for $\beta_1, \beta_2, \ldots, \beta_n$, where β_n is the first symbol whose FIRST set does not contain ϵ, and $\epsilon \in$ FIRST(s) if and only if it is in the set for each of the β_i, $1 \leq i \leq k$. The algorithm in Figure 3.7 computes this quantity into the variable `rhs`.

Conceptually, FIRST sets simplify implementation of a top-down parser. Consider, for example, the rules for *Expr′* in the right-recursive expression grammar:

2	*Expr′*	→	+	*Term Expr′*
3		|	-	*Term Expr′*
4		|	ϵ	

When the parser tries to expand an *Expr′*, it uses the lookahead symbol and the FIRST sets to choose between rules 2, 3, and 4. With a lookahead of +, the parser expands by rule 2 because + is in FIRST(+ *Term Expr′*) and not in FIRST(- *Term Expr′*) or FIRST(ϵ). Similarly, a lookahead of - dictates a choice of rule 3.

Rule 4, the ϵ-production, poses a slightly harder problem. FIRST(ϵ) is just $\{\epsilon\}$, which matches no word returned by the scanner. Intuitively, the parser should apply the ϵ production when the lookahead symbol is not a member of the FIRST set of any other alternative. To differentiate between legal inputs

```
for each A ∈ NT do;
    FOLLOW(A) ← ∅;
end;
FOLLOW(S) ← {eof};
while (FOLLOW sets are still changing) do;
   for each p ∈ P of the form A → β1β2···βk do;
       TRAILER ← FOLLOW(A);
       for i ← k down to 1 do;
           if βi ∈ NT then begin;
               FOLLOW(βi) ← FOLLOW(βi) ∪ TRAILER;
               if ϵ ∈ FIRST(βi)
                   then TRAILER ← TRAILER ∪ (FIRST(βi) − ϵ);
                   else TRAILER ← FIRST(βi);
           end;
           else TRAILER ← FIRST(βi);     // is {βi}
       end;
   end;
end;
```

■ **FIGURE 3.8** Computing FOLLOW Sets for Non-Terminal Symbols.

and syntax errors, the parser needs to know which words can appear as the leading symbol after a valid application of rule 4—the set of symbols that can follow an *Expr′*.

FOLLOW set
For a nonterminal α, FOLLOW(α) contains the set of words that can occur immediately after α in a sentence.

To capture that knowledge, we define the set FOLLOW(*Expr′*) to contain all of the words that can occur to the immediate right of a string derived from *Expr′*. Figure 3.8 presents an algorithm to compute the FOLLOW set for each nonterminal in a grammar; it assumes the existence of FIRST sets. The algorithm initializes each FOLLOW set to the empty set and then iterates over the productions, computing the contribution of the partial suffixes to the FOLLOW set of each symbol in each right-hand side. The algorithm halts when it reaches a fixed point. For the right-recursive expression grammar, the algorithm produces:

	Expr	*Expr′*	*Term*	*Term′*	*Factor*
FOLLOW	eof,)	eof,)	eof, +, -,)	eof, +, -,)	eof, +, -, x, ÷,)

The parser can use FOLLOW(*Expr′*) when it tries to expand an *Expr′*. If the lookahead symbol is +, it applies rule 2. If the lookahead symbol is -, it applies rule 3. If the lookahead symbol is in FOLLOW(*Expr′*), which contains eof and), it applies rule 4. Any other symbol causes a syntax error.

Using FIRST and FOLLOW, we can specify precisely the condition that makes a grammar backtrack free for a top-down parser. For a production $A \rightarrow \beta$, define its augmented FIRST set, FIRST$^+$, as follows:

$$\text{FIRST}^+(A \rightarrow \beta) = \begin{cases} \text{FIRST}(\beta) & \text{if } \epsilon \notin \text{FIRST}(\beta) \\ \text{FIRST}(\beta) \cup \text{FOLLOW}(A) & \textit{otherwise} \end{cases}$$

Now, a backtrack-free grammar has the property that, for any nonterminal A with multiple right-hand sides, $A \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$

$$\text{FIRST}^+(A \rightarrow \beta_i) \cap \text{FIRST}^+(A \rightarrow \beta_j) = \emptyset, \ \forall \ 1 \leq i,j \leq n, \ i \neq j.$$

Any grammar that has this property is *backtrack free*.

For the right-recursive expression grammar, only productions 4 and 8 have FIRST$^+$ sets that differ from their FIRST sets.

	Production	FIRST set	FIRST$^+$ set
4	$Expr' \rightarrow \epsilon$	$\{\epsilon\}$	$\{\epsilon$, eof,) $\}$
8	$Term' \rightarrow \epsilon$	$\{\epsilon\}$	$\{\epsilon$, eof, +, -,) $\}$

Applying the backtrack-free condition pairwise to each set of alternate right-hand sides proves that the grammar is, indeed, backtrack free.

Left-Factoring to Eliminate Backtracking

Not all grammars are backtrack free. For an example of such a grammar, consider extending the expression grammar to include function calls, denoted with parentheses, (and), and array-element references, denoted with square brackets, [and]. To add these options, we replace production 11, *Factor* → name, with a set of three rules, plus a set of right-recursive rules for argument lists.

11	*Factor*	→	name
12		|	name [*ArgList*]
13		|	name (*ArgList*)
15	*ArgList*	→	*Expr MoreArgs*
16	*MoreArgs*	→	, *Expr MoreArgs*
17		|	ϵ

Because productions 11, 12, and 13 all begin with name, they have identical FIRST$^+$ sets. When the parser tries to expand an instance of *Factor* with a lookahead of name, it has no basis to choose among 11, 12, and 13. The compiler writer can implement a parser that chooses one rule and backtracks when it is wrong. As an alternative, we can transform these productions to create disjoint FIRST$^+$ sets.

A two-word lookahead would handle this case. However, for any finite lookahead we can devise a grammar where that lookahead is insufficient.

The following rewrite of productions 11, 12, and 13 describes the same language but produces disjoint FIRST$^+$ sets:

11	*Factor*	→	`name` *Arguments*
12	*Arguments*	→	`[` *ArgList* `]`
13		|	`(` *ArgList* `)`
14		|	ϵ

Left factoring
the process of extracting and isolating common prefixes in a set of productions

The rewrite breaks the derivation of *Factor* into two steps. The first step matches the common prefix of rules 11, 12, and 13. The second step recognizes the three distinct suffixes: `[` *Expr* `]`, `(` *Expr* `)`, and ϵ. The rewrite adds a new nonterminal, *Arguments*, and pushes the alternate suffixes for *Factor* into right-hand sides for *Arguments*. We call this transformation *left factoring*.

We can left factor any set of rules that has alternate right-hand sides with a common prefix. The transformation takes a nonterminal and its productions:

$$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_j$$

where α is the common prefix and the γ_i's represent right-hand sides that do not begin with α. The transformation introduces a new nonterminal B to represent the alternate suffixes for α and rewrites the original productions according to the pattern:

$$A \rightarrow \alpha B \mid \gamma_1 \mid \gamma_2 \mid \cdots \mid \gamma_j$$
$$B \rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

To left factor a complete grammar, we must inspect each nonterminal, discover common prefixes, and apply the transformation in a systematic way. For example, in the pattern above, we must consider factoring the right-hand sides of B, as two or more of the β_i's could share a prefix. The process stops when all common prefixes have been identified and rewritten.

Left-factoring can often eliminate the need to backtrack. However, some context-free languages have no backtrack-free grammar. Given an arbitrary CFG, the compiler writer can systematically eliminate left recursion and use left-factoring to eliminate common prefixes. These transformations may produce a backtrack-free grammar. In general, however, it is undecidable whether or not a backtrack-free grammar exists for an arbitrary context-free language.

3.3.2 Top-Down Recursive-Descent Parsers

Backtrack-free grammars lend themselves to simple and efficient parsing with a paradigm called *recursive descent*. A recursive-descent parser is

PREDICTIVE PARSERS VERSUS DFAs

Predictive parsing is the natural extension of DFA-style reasoning to parsers. A DFA transitions from state to state based solely on the next input character. A predictive parser chooses an expansion based on the next word in the input stream. Thus, for each nonterminal in the grammar, there must be a unique mapping from the first word in any acceptable input string to a specific production that leads to a derivation for that string. The real difference in power between a DFA and a predictively parsable grammar derives from the fact that one prediction may lead to a right-hand side with many symbols, whereas in a regular grammar, it predicts only a single symbol. This lets predictive grammars include productions such as $p \rightarrow (p)$, which are beyond the power of a regular expression to describe. (Recall that a regular expression can recognize $(^{+}\ \Sigma^{*}\ )^{+}$, but this does not specify that the numbers of opening and closing parentheses must match.)

Of course, a hand-coded, recursive-descent parser can use arbitrary tricks to disambiguate production choices. For example, if a particular left-hand side cannot be predicted with a single-symbol lookahead, the parser could use two symbols. Done judiciously, this should not cause problems.

structured as a set of mutually recursive procedures, one for each nonterminal in the grammar. The procedure corresponding to nonterminal A recognizes an instance of A in the input stream. To recognize a nonterminal B on some right-hand side for A, the parser invokes the procedure corresponding to B. Thus, the grammar itself serves as a guide to the parser's implementation.

Consider the three rules for *Expr′* in the right-recursive expression grammar:

	Production				FIRST^{+}
2	*Expr′*	→	+ *Term Expr′*		{+}
3		|	- *Term Expr′*		{-}
4		|	ϵ		{ϵ, eof,) }

To recognize instances of *Expr′*, we will create a routine `EPrime()`. It follows a simple scheme: choose among the three rules (or a syntax error) based on the FIRST^{+} sets of their right-hand sides. For each right-hand side, the code tests directly for any further symbols.

To test for the presence of a nonterminal, say A, the code invokes the procedure that corresponds to A. To test for a terminal symbol, such as `name`, it performs a direct comparison and, if successful, advances the input stream

```
EPrime()
   /* Expr′ → + Term Expr′ | - Term Expr′ */
   if (word = + or word = -) then begin;
        word ← NextWord();
        if (Term())
             then return EPrime();
             else return false;
   end;
   else if (word = ) or word = eof)    /* Expr′ → ϵ */
      then return true;
      else begin;                      /* no match */
         report a syntax error;
         return false;
      end;
```

■ **FIGURE 3.9** An Implementation of `EPrime()`.

by calling the scanner, `NextWord()`. If it matches an ϵ-production, the code does not call `NextWord()`. Figure 3.9 shows a straightforward implementation of `EPrime()`. It combines rules 2 and 3 because they both end with the same suffix, *Term Expr′*.

The strategy for constructing a complete recursive-descent parser is clear. For each nonterminal, we construct a procedure to recognize its alternative right-hand sides. These procedures call one another to recognize nonterminals. They recognize terminals by direct matching. Figure 3.10 shows a top-down recursive-descent parser for the right-recursive version of the classic expression grammar shown in Figure 3.4 on page 101. The code for similar right-hand sides has been combined.

For a small grammar, a compiler writer can quickly craft a recursive-descent parser. With a little care, a recursive-descent parser can produce accurate, informative error messages. The natural location for generating those messages is when the parser fails to find an expected terminal symbol—inside `EPrime`, `TPrime`, and `Factor` in the example.

3.3.3 Table-Driven LL(1) Parsers

Following the insights that underlie the FIRST$^+$ sets, we can automatically generate top-down parsers for backtrack-free grammars. The tool constructs FIRST, FOLLOW, and FIRST$^+$ sets. The FIRST$^+$ sets completely dictate the parsing decisions, so the tool can then emit an efficient top-down parser. The resulting parser is called an LL(1) parser. The name LL(1) derives from the fact that these parsers scan their input Left to right, construct a Leftmost

```
Main( )
   /* Goal → Expr */
   word ← NextWord( );
   if (Expr( ))
      then if (word = eof )
        then report success;
        else Fail( );

Fail( )
   report syntax error;
   attempt error recovery or exit;

Expr( )
   /* Expr → Term Expr′ */
   if ( Term( ) )
      then return EPrime( );
      else Fail( );

EPrime( )
   /* Expr′→ + Term Expr′ */
   /* Expr′→ - Term Expr′ */
   if (word = + or word = - )
      then begin;
        word ← NextWord( );
        if ( Term( ) )
           then return EPrime( );
           else Fail( );
      end;
   else if (word = ) or word = eof)
      /* Expr′→ ϵ */
      then return true;
      else Fail( );

Term( )
   /* Term → Factor Term′ */
   if ( Factor( ) )
      then return TPrime( );
      else Fail( );

TPrime( )
   /* Term′→ x Factor Term′ */
   /* Term′→ ÷ Factor Term′ */
   if (word = x or word = ÷ )
      then begin;
        word ← NextWord( );
        if ( Factor( ) )
           then return TPrime( );
           else Fail( );
        end;
   else if (word = + or word = - or
            word = ) or word = eof)
     /* Term′→ ϵ */
     then return true;
     else Fail( );

Factor( )
   /* Factor → ( Expr ) */
   if (word = ( ) then begin;
      word ← NextWord( );
      if (not Expr( ))
         then Fail( );
      if (word ≠ ) )
         then Fail( );
      word ← NextWord( );
      return true;
   end;
   /* Factor → num */
   /* Factor → name */
   else if (word = num or
            word = name )
     then begin;
       word ← NextWord( );
       return true;
     end;
   else Fail( );
```

■ **FIGURE 3.10** Recursive-Descent Parser for Expressions.

```
word ← NextWord( );
push eof onto Stack;
push the start symbol, S, onto Stack;
focus ← top of Stack;
loop forever;
   if (focus = eof and word = eof)
       then report success and exit the loop;
   else if (focus ∈ T or focus = eof) then begin;
      if focus matches word then begin;
         pop Stack;
         word ← NextWord( );
      end;
      else report an error looking for symbol at top of stack;
   end;
   else begin; /* focus is a nonterminal */
      if Table[focus,word] is A → B1B2···Bk then begin;
         pop Stack;
         for i ← k to 1 by -1 do;
           if (Bi ≠ ε)
              then push Bi onto Stack;
         end;
      end;
      else report an error expanding focus;
    end;
    focus ← top of Stack;
end;
```

(a) The Skeleton LL(1) Parser

	eof	+	-	×	÷	(	)	name	num
Goal	—	—	—	—	—	0	—	0	0
Expr	—	—	—	—	—	1	—	1	1
Expr′	4	2	3	—	—	—	4	—	—
Term	—	—	—	—	—	5	—	5	5
Term′	8	8	8	6	7	—	8	—	—
Factor	—	—	—	—	—	9	—	11	10

(b) The LL(1) Parse Table for Right-Recursive Expression Grammar

■ **FIGURE 3.11** An LL(1) Parser for Expressions.

```
build FIRST, FOLLOW, and FIRST+ sets;

for each nonterminal A do;
    for each terminal w do;
        Table[A,w] ← error;
    end;
    for each production p of the form A → β do;
       for each terminal w ∈ FIRST+(A → β) do;
           Table[A,w] ← p;
           end;
       if eof ∈ FIRST+(A → β)
          then Table[A,eof] ← p;
   end;
end;
```

■ **FIGURE 3.12** LL(1) Table-Construction Algorithm.

derivation, and use a lookahead of 1 symbol. Grammars that work in an LL(1) scheme are often called LL(1) grammars. LL(1) grammars are, by definition, backtrack free.

To build an LL(1) parser, the compiler writer provides a right-recursive, backtrack-free grammar and a *parser generator* constructs the actual parser. The most common implementation technique for an LL(1) parser generator uses a table-driven skeleton parser, such as the one shown at the top of Figure 3.11. The parser generator constructs the table, `Table`, which codifies the parsing decisions and drives the skeleton parser. The bottom of Figure 3.11 shows the LL(1) table for the right-recursive expression grammar shown in Figure 3.4 on page 101.

Parser generator
a tool that builds a parser from specifications, usually a grammar in a BNF-like notation

Parser generators are also called *compiler compilers*.

In the skeleton parser, the variable `focus` holds the next grammar symbol on the partially built parse tree's lower fringe that must be matched. (`focus` plays the same role in Figure 3.2.) The parse table, `Table`, maps pairs of nonterminals and lookahead symbols (terminals or `eof`) into productions. Given a nonterminal A and a lookahead symbol w, `Table[A,w]` specifies the correct expansion.

The algorithm to build `Table` is straightforward. It assumes that FIRST, FOLLOW, and FIRST$^+$ sets are available for the grammar. It iterates over the grammar symbols and fills in `Table`, as shown in Figure 3.12. If the grammar meets the backtrack free condition (see page 107), the construction will produce a correct table in $\mathbf{O}(|P| \times |T|)$ time, where P is the set of productions and T is the set of terminals.

If the grammar is not backtrack free, the construction will assign more than one production to some elements of `Table`. If the construction assigns to

Rule	Stack	Input
—	eof *Goal*	↑ name + name x name
0	eof *Expr*	↑ name + name x name
1	eof *Expr′ Term*	↑ name + name x name
5	eof *Expr′ Term′ Factor*	↑ name + name x name
11	eof *Expr′ Term′* name	↑ name + name x name
→	eof *Expr′ Term′*	name ↑ + name x name
8	eof *Expr′*	name ↑ + name x name
2	eof *Expr′ Term* +	name ↑ + name x name
→	eof *Expr′ Term*	name + ↑ name x name
5	eof *Expr′ Term′ Factor*	name + ↑ name x name
11	eof *Expr′ Term′* name	name + ↑ name x name
→	eof *Expr′ Term′*	name + name ↑ x name
6	eof *Expr′ Term′ Factor* x	name + name ↑ x name
→	eof *Expr′ Term′ Factor*	name + name x ↑ name
11	eof *Expr′ Term′* name	name + name x ↑ name
→	eof *Expr′ Term′*	name + name x name ↑
8	eof *Expr′*	name + name x name ↑
4	eof	name + name x name ↑

■ **FIGURE 3.13** Actions of the LL(1) Parser on a + b x c.

Table[A,w] multiple times, then two or more alternative right-hand sides for *A* have *w* in their FIRST$^+$ sets, violating the backtrack-free condition. The parser generator can detect this situation with a simple test on the two assignments to Table.

The example in Figure 3.13 shows the actions of the LL(1) expression parser for the input string a + b x c. The central column shows the contents of the parser's stack, which holds the partially completed lower fringe of the parse tree. The parse concludes successfully when it pops *Expr′* from the stack, leaving eof exposed on the stack and eof as the next symbol, implicitly, in the input stream.

Now, consider the actions of the LL(1) parser on the illegal input string x + ÷ y, shown in Figure 3.14 on page 115. It detects the syntax error when it attempts to expand a *Term* with lookahead symbol ÷. *Table*[*Term*,÷] contains "—", indicating a syntax error.

Alternatively, an LL(1) parser generator could emit a direct-coded parser, in the style of the direct-coded scanners discussed in Chapter 2. The parser generator would build FIRST, FOLLOW, and FIRST$^+$ sets. Next, it would iterate through the grammar, following the same scheme used by the table-construction algorithm in Figure 3.12. Rather than emitting table entries, it would generate, for each nonterminal, a procedure to recognize

Rule	Stack	Input
—	eof *Goal*	↑ name + ÷ name
0	eof *Expr*	↑ name + ÷ name
1	eof *Expr′ Term*	↑ name + ÷ name
5	eof *Expr′ Term′ Factor*	↑ name + ÷ name
11	eof *Expr′ Term′* name	↑ name + ÷ name
→	eof *Expr′ Term′*	name ↑ + ÷ name
8	eof *Expr′*	name ↑ + ÷ name
2	eof *Expr′ Term* +	name ↑ + ÷ name
→	eof *Expr′ Term*	name + ↑ ÷ name

syntax error at this point

■ **FIGURE 3.14** Actions of the LL(1) Parser on x + ÷ y.

each of the possible right-hand sides for that nonterminal. This process would be guided by the FIRST$^+$ sets. It would have the same speed and locality advantages that accrue to direct-coded scanners and recursive-descent parsers, while retaining the advantages of a grammar-generated system, such as a concise, high-level specification and reduced implementation effort.

SECTION REVIEW

Predictive parsers are simple, compact, and efficient. They can be implemented in a number of ways, including hand-coded, recursive-descent parsers and generated LL(1) parsers, either table driven or direct coded. Because these parsers know, at each point in the parse, the set of words that can occur as the next symbol in a valid input string, they can produce accurate and useful error messages.

Most programming-language constructs can be expressed in a backtrack-free grammar. Thus, these techniques have widespread application. The restriction that alternate right-hand sides for a nonterminal have disjoint FIRST$^+$ sets does not seriously limit the utility of LL(1) grammars. As we will see in Section 3.5.4, the primary drawback of top-down, predictive parsers lies in their inability to handle left recursion. Left-recursive grammars model the left-to-right associativity of expression operators in a more natural way than right-recursive grammars.

Review Questions

1. To build an efficient top-down parser, the compiler writer must express the source language in a somewhat constrained form. Explain the restrictions on the source-language grammar that are required to make it amenable to efficient top-down parsing.

2. Name two potential advantages of a hand-coded recursive-descent parser over a generated, table-driven LL(1) parser, and two advantages of the LL(1) parser over the recursive-descent implementation.

3.4 BOTTOM-UP PARSING

Bottom-up parsers build a parse tree starting from its leaves and working toward its root. The parser constructs a leaf node in the tree for each word returned by the scanner. These leaves form the lower fringe of the parse tree. To build a derivation, the parser adds layers of nonterminals on top of the leaves in a structure dictated by both the grammar and the partially completed lower portion of the parse tree.

At any stage in the parse, the partially-completed parse tree represents the state of the parse. Each word that the scanner has returned is represented by a leaf. The nodes above the leaves encode all of the knowledge that the parser has yet derived. The parser works along the upper frontier of this partially-completed parse tree; that frontier corresponds to the current sentential form in the derivation being built by the parser.

To extend the frontier upward, the parser looks in the current frontier for a substring that matches the right-hand side of some production $A \rightarrow \beta$. If it finds β in the frontier, with its right end at k, it can replace β with A, to create a new frontier. If replacing β with A at position k is the next step in a valid derivation for the input string, then the pair $\langle A \rightarrow \beta, k \rangle$ is a *handle* in the current derivation and the parser should replace β with A. This replacement is called a *reduction* because it reduces the number of symbols on the frontier, unless $|\beta| = 1$. If the parser is building a parse tree, it builds a node for A, adds that node to the tree, and connects the nodes representing β as A's children.

Handle
a pair, $\langle A \rightarrow \beta, k \rangle$, such that β appears in the frontier with its right end at position k and replacing β with A is the next step in the parse

Reduction
reducing the frontier of a bottom-up parser by $A \rightarrow \beta$ replaces β with A in the frontier

Finding handles is the key issue that arises in bottom-up parsing. The techniques presented in the following sections form a particularly efficient handle-finding mechanism. We will return to this issue periodically throughout Section 3.4. First, however, we will finish our high-level description of bottom-up parsers.

The bottom-up parser repeats a simple process. It finds a handle $\langle A \rightarrow \beta, k \rangle$ on the frontier. It replaces the occurrence of β at k with A. This process continues until either: (1) it reduces the frontier to a single node that represents the grammar's goal symbol, or (2) it cannot find a handle. In the first case, the parser has found a derivation; if it has also consumed all the words in the input stream (i.e. the next word is `eof`), then the parse succeeds. In the

second case, the parser cannot build a derivation for the input stream and it should report that failure.

A successful parse runs through every step of the derivation. When a parse fails, the parser should use the context accumulated in the partial derivation to produce a meaningful error message. In many cases, the parser can recover from the error and continue parsing so that it discovers as many syntactic errors as possible in a single parse (see Section 3.5.1).

The relationship between the derivation and the parse plays a critical role in making bottom-up parsing both correct and efficient. The bottom-up parser works from the final sentence toward the goal symbol, while a derivation starts at the goal symbol and works toward the final sentence. The parser, then, discovers the steps of the derivation in reverse order. For a derivation:

$$Goal = \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_{n-1} \rightarrow \gamma_n = sentence,$$

the bottom-up parser discovers $\gamma_i \rightarrow \gamma_{i+1}$ before it discovers $\gamma_{i-1} \rightarrow \gamma_i$. The way that it builds the parse tree forces this order. The parser must add the node for γ_i to the frontier before it can match γ_i.

The scanner returns classified words in left-to-right order. To reconcile the left-to-right order of the scanner with the reverse derivation constructed by the scanner, a bottom-up parser looks for a rightmost derivation. In a rightmost derivation, the leftmost leaf is considered last. Reversing that order leads to the desired behavior: leftmost leaf first and rightmost leaf last.

At each point, the parser operates on the frontier of the partially constructed parse tree; the current frontier is a prefix of the corresponding sentential form in the derivation. Because each sentential form occurs in a rightmost derivation, the unexamined suffix consists entirely of terminal symbols. When the parser needs more right context, it calls the scanner.

With an unambiguous grammar, the rightmost derivation is unique. For a large class of unambiguous grammars, γ_{i-1} can be determined directly from γ_i (the parse tree's upper frontier) and a limited amount of lookahead in the input stream. In other words, given a frontier γ_i and a limited number of additional classified words, the parser can find the handle that takes γ_i to γ_{i-1}. For such grammars, we can construct an efficient handle-finder, using a technique called LR parsing. This section examines one particular flavor of LR parser, called a *table-driven* LR(1) parser.

An LR(1) parser scans the input from left to right to build a rightmost derivation in reverse. At each step, it makes decisions based on the history of the parse and a lookahead of, at most, one symbol. The name LR(1) derives

from these properties: Left-to-right scan, Reverse rightmost derivation, and 1 symbol of lookahead.

Informally, we will say that a language has the LR(1) property if it can be parsed in a single left-to-right scan, to build a reverse-rightmost derivation, using only one symbol of lookahead to determine parsing actions. In practice, the simplest test to determine if a grammar has the LR(1) property is to let a parser generator attempt to build the LR(1) parser. If that process fails, the grammar lacks the LR(1) property. The remainder of this section introduces LR(1) parsers and their operation. Section 3.4.2 presents an algorithm to build the tables that encode an LR(1) parser.

3.4.1 The LR(1) Parsing Algorithm

The critical step in a bottom-up parser, such as a table-driven LR(1) parser, is to find the next handle. Efficient handle finding is the key to efficient bottom-up parsing. An LR(1) parser uses a handle-finding automaton, encoded into two tables, called `Action` and `Goto`. Figure 3.15 shows a simple table-driven LR(1) parser.

The skeleton LR(1) parser interprets the `Action` and `Goto` tables to find successive handles in the reverse rightmost derivation of the input string. When it finds a handle $\langle A \rightarrow \beta, k \rangle$, it reduces β at k to A in the current sentential form—the upper frontier of the partially completed parse tree. Rather than build an explicit parse tree, the skeleton parser keeps the current upper frontier of the partially constructed tree on a stack, interleaved with states from the handle-finding automaton that let it thread together the reductions into a parse. At any point in the parse, the stack contains a prefix of the current frontier. Beyond this prefix, the frontier consists of leaf nodes. The variable `word` holds the first word in the suffix that lies beyond the stack's contents; it is the *lookahead symbol*.

Using a stack lets the LR(1) parser make the position, k, in the handle be constant and implicit.

To find the next handle, the LR(1) parser shifts symbols onto the stack until the automaton finds the right end of a handle at the stack top. Once it has a handle, the parser reduces by the production in the handle. To do so, it pops the symbols in β from the stack and pushes the corresponding left-hand side, A, onto the stack. The `Action` and `Goto` tables thread together shift and reduce actions in a grammar-driven sequence that finds a reverse rightmost derivation, if one exists.

To make this concrete, consider the grammar shown in Figure 3.16a, which describes the language of properly nested parentheses. Figure 3.16b shows the `Action` and `Goto` tables for this grammar. When used with the skeleton LR(1) parser, they create a parser for the parentheses language.

```
push $;
push start state, s0;
word ← NextWord();
while (true) do;
   state ← top of stack;
   if Action[state,word] = "reduce A → β" then begin;
        pop 2 × |β| symbols;
        state ← top of stack;
        push A;
        push Goto[state, A];
   end;
   else if Action[state,word] = "shift si" then begin;
      push word;
      push si ;
      word ← NextWord();
   end;
   else if Action[state,word] = "accept"
      then break;
   else Fail();
end;
report success;   /* executed break on "accept" case */
```

■ **FIGURE 3.15** The Skeleton LR(1) Parser.

To understand the behavior of the skeleton LR(1) parser, consider the sequence of actions that it takes on the input string "( )".

Iteration	State	word	Stack	Handle	Action
initial	—	(	$ 0	— *none* —	—
1	0	(	$ 0	— *none* —	*shift 3*
2	3	)	$ 0 (3	— *none* —	*shift 7*
3	7	eof	$ 0 (3) 7	( )	*reduce 5*
4	2	eof	$ 0 *Pair* 2	*Pair*	*reduce 3*
5	1	eof	$ 0 *List* 1	*List*	*accept*

The first line shows the parser's initial state. Subsequent lines show its state at the start of the while loop, along with the action that it takes. At the start of the first iteration, the stack does not contain a handle, so the parser shifts the lookahead symbol, (, onto the stack. From the `Action` table, it knows to shift and move to state 3. At the start of the second iteration, the stack still

1	*Goal*	→ *List*
2	*List*	→ *List Pair*
3		| *Pair*
4	*Pair*	→ (*Pair*)
5		| ( )

(a) Parentheses Grammar

	Action Table			Goto Table	
State	eof	(	)	*List*	*Pair*
0		s 3		1	2
1	acc	s 3			4
2	r 3	r 3			
3		s 6	s 7		5
4	r 2	r 2			
5			s 8		
6		s 6	s 10		9
7	r 5	r 5			
8	r 4	r 4			
9			s 11		
10			r 5		
11			r 4		

(b) Action and Goto Tables

■ **FIGURE 3.16** The Parentheses Grammar.

does not contain a handle, so the parser shifts) onto the stack to build more context. It moves to state 7.

In an LR parser, the handle is always positioned at stacktop and the chain of handles produces a reverse rightmost derivation.

In the third iteration, the situation has changed. The stack contains a handle, ⟨*Pair* → ( )⟩,*t*, where *t* is the stack top. The `Action` table directs the parser to reduce ( ) to *Pair*. Using the state beneath *Pair* on the stack, 0, and *Pair*, the parser moves to state 2 (specified by `Goto[0,Pair]`). In state 2, with *Pair* atop the stack and `eof` as its lookahead, the parser finds the handle ⟨*List* → *Pair*,*t*⟩ and reduces, which leaves the parser in state 1 (specified by `Goto[0,List]`). Finally, in state 1, with *List* atop the stack and `eof` as its lookahead, the parser discovers the handle ⟨*Goal* → *List*,*t*⟩. The `Action` table encodes this situation as an *accept* action, so the parse halts.

This parse required two shifts and three reduces. LR(1) parsers take time proportional to the length of the input (one shift per word returned from the scanner) and the length of the derivation (one reduce per step in the derivation). In general, we cannot expect to discover the derivation for a sentence in any fewer steps.

Figure 3.17 shows the parser's behavior on the input string, "(( )) ( )." The parser performs six shifts, five reduces, and one accept on this input. Figure 3.18 shows the state of the partially-built parse tree at the start of each iteration of the parser's while loop. The top of each drawing shows an iteration number and a gray bar that contains the partial parse tree's upper frontier. In the LR(1) parser, this frontier appears on the stack.

Iteration	State	word	Stack	Handle	Action
initial	—	(	$ 0	— none —	—
1	0	(	$ 0	— none —	*shift 3*
2	3	(	$ 0 (3	— none —	*shift 6*
3	6	)	$ 0 (3 (6	— none —	*shift 10*
4	10	)	$ 0 (3 (6) 10	( )	*reduce 5*
5	5	)	$ 0 (3 *Pair* 5	— none —	*shift 8*
6	8	(	$ 0 (3 *Pair* 5) 8	(*Pair*)	*reduce 4*
7	2	(	$ 0 *Pair* 2	*Pair*	*reduce 3*
8	1	(	$ 0 *List* 1	— none —	*shift 3*
9	3	)	$ 0 *List* 1 (3	— none —	*shift 7*
10	7	eof	$ 0 *List* 1 (3) 7	( )	*reduce 5*
11	4	eof	$ 0 *List* 1 *Pair* 4	*List Pair*	*reduce 2*
12	1	eof	$ 0 *List* 1	*List*	*accept*

■ **FIGURE 3.17** States of the LR(1) Parser on (( )) ( ).

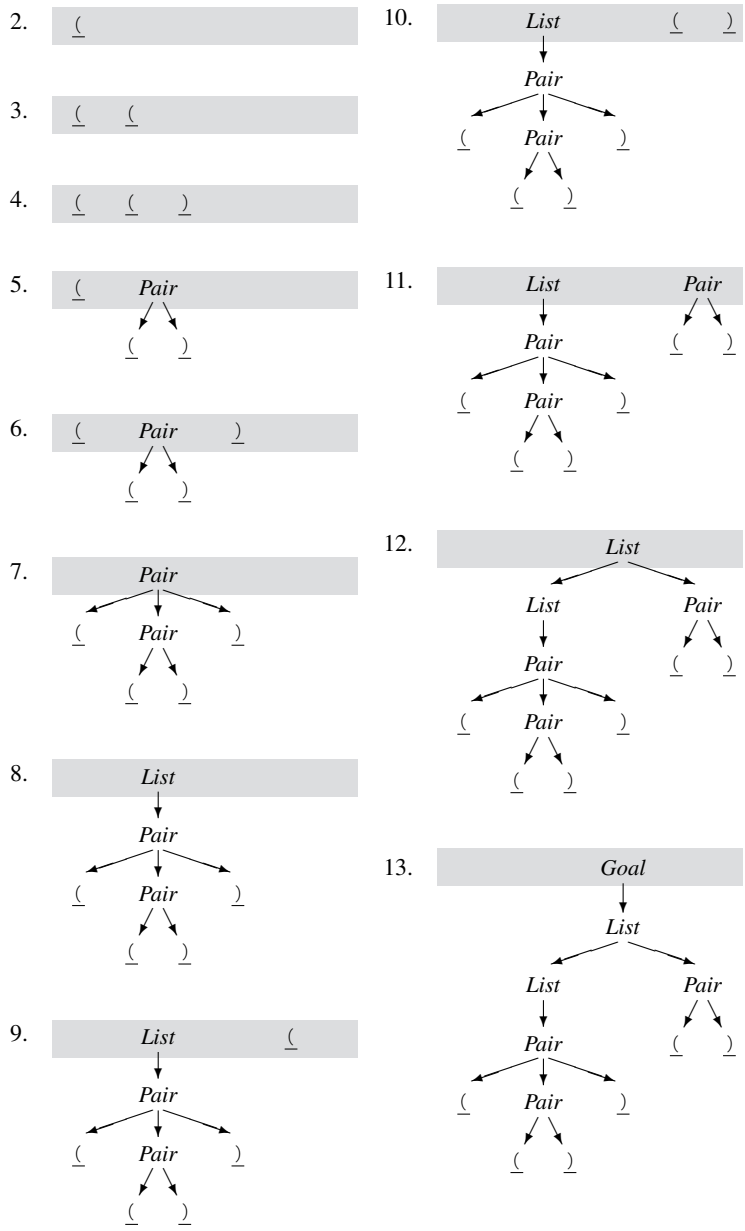
Handle Finding

The parser's actions shed additional light on the process of finding handles. Consider the parser's actions on the string "( )", as shown in the table on page 119. The parser finds a handle in each of iterations 3, 4, and 5. In iteration 3, the frontier of ( ) clearly matches the right-hand side of production 5. From the *Action* table, we see that a lookahead of either eof or (implies a reduce by production 5. Then, in iteration 4, the parser recognizes that *Pair*, followed by a lookahead of either eof or (constitutes a handle for the reduction by *List* → *Pair*. The final handle of the parse, *List* with lookahead of eof in state 1, triggers the accept action.

To understand how the states preserved on the stack change the parser's behavior, consider the parser's actions on our second input string, "(( )) ( )," as shown in Figure 3.17. Initially, the parser shifts (, (, and) onto the stack, in iterations 1 to 3. In iteration 4, the parser reduces by production 5; it replaces the top two symbols on the stack, (and), with *Pair* and moves to state 5.

Between these two examples, the parser recognized the string ( ) at stacktop as a handle three times. It behaved differently in each case, based on the prior left context encoded in the stack. Comparing these three situations exposes how the stacked states control the future direction of the parse.

With the first example, ( ), the parser was in s_7 with a lookahead of eof when it found the handle. The reduction reveals s_0 beneath ( ), and *Goto*[s_0,*Pair*] is s_2. In s_2, a lookahead of eof leads to another reduction followed by an accept action. A lookahead of) in s_2 produces an error.



■ **FIGURE 3.18** The Sequence of Partial Parse Trees Built for (( )) ( ).

The second example, (( )) ( ), encounters a handle for ( ) twice. The first handle occurs in iteration 4. The parser is in s_{10} with a lookahead of). It has previously shifted (, (, and) onto the stack. The *Action* table indicates "r 5," so the parser reduces by *Pair* → ( ). The reduction reveals s_3 beneath ( ) and *Goto*[s_3,*Pair*] is s_5, a state in which further)'s are legal. The second time it finds ( ) as a handle occurs in iteration 10. The reduction reveals s_1 beneath ( ) and takes the parser to s_4. In s_4, a lookahead of either eof or (triggers a reduction of *List Pair* to *List*, while a lookahead of) is an error.

The *Action* and *Goto* tables, along with the stack, cause the parser to track prior left context and let it take different actions based on that context. Thus, the parser handles correctly each of the three instances in which it found a handle for ( ). We will revisit this issue when we examine the construction of *Action* and *Goto*.

Parsing an Erroneous Input String

To see how an LR(1) parser discovers a syntax error, consider the sequence of actions that it takes on the string "( ))", shown below:

Iteration	State	*word*	Stack	Handle	Action
initial	—	(	$ 0	— none —	—
1	0	(	$ 0	— none —	*shift 3*
2	3	)	$ 0 (3	— none —	*shift 7*
3	7	)	$ 0 (3) 7	— none —	*error*

The first two iterations of the parse proceed as in the first example, "( )". The parser shifts (and). In the third iteration of the while loop, it looks at the Action table entry for state 7 and). That entry contains neither shift, reduce, nor accept, so the parser interprets it as an error.

The LR(1) parser detects syntax errors through a simple mechanism: the corresponding table entry is invalid. The parser detects the error as soon as possible, before reading any words beyond those needed to prove the input erroneous. This property allows the parser to localize the error to a specific point in the input. Using the available context and knowledge of the grammar, we can build LR(1) parsers that provide good diagnostic error messages.

Using LR Parsers

The key to LR parsing lies in the construction of the *Action* and *Goto* tables. The tables encode all of the legal reduction sequences that can arise in a

reverse rightmost derivation for the given grammar. While the number of such sequences is huge, the grammar itself constrains the order in which reductions can occur.

The compiler writer can build `Action` and `Goto` tables by hand. However, the table-construction algorithm requires scrupulous bookkeeping; it is a prime example of the kind of task that should be automated and relegated to a computer. Programs that automate this construction are widely available. The next section presents one algorithm that can be used to construct LR(1) parse tables.

With an LR(1) parser generator, the compiler writer's role is to define the grammar and to ensure that the grammar has the LR(1) property. In practice, the LR(1) table generator identifies those productions that are ambiguous or that are expressed in a way that requires more than one word of lookahead to distinguish between a shift action and a reduce action. As we study the table-construction algorithm, we will see how those problems arise, how to cure them, and how to understand the kinds of diagnostic information that LR(1) parser generators produce.

Using More Lookahead

The ideas that underlie LR(1) parsers actually define a family of parsers that vary in the amount of lookahead that they use. An LR(k) parser uses, at most, k lookahead symbols. Additional lookahead allows an LR(2) parser to recognize a larger set of grammars than an LR(1) parsing system. Almost paradoxically, however, the added lookahead does not increase the set of languages that these parsers can recognize. LR(1) parsers accept the same set of languages as LR(k) parsers for $k > 1$. The LR(1) grammar for a language may be more complex than an LR(k) grammar.

3.4.2 Building LR(1) Tables

To construct `Action` and `Goto` tables, an LR(1) parser generator builds a model of the handle-recognizing automaton and uses that model to fill in the tables. The model, called the *canonical collection of sets of LR(1) items*, represents all of the possible states of the parser and the transitions between those states. It is reminiscent of the subset construction from Section 2.4.3.

To illustrate the table-construction algorithm, we will use two examples. The first is the parentheses grammar given in Figure 3.16a. It is small enough to use as a running example, but large enough to exhibit some of the complexities of the process.

```
1  Goal → List
2  List → List Pair
3       | Pair
4  Pair → ( Pair )
5       | ( )
```

Our second example, in Section 3.4.3, is an abstracted version of the classic `if-then-else` ambiguity. The table construction fails on this grammar because of its ambiguity. The example highlights the situations that lead to failures in the table-construction process.

LR(1) Items

In an LR(1) parser, the `Action` and `Goto` tables encode information about the potential handles at each step in the parse. The table-construction algorithm, therefore, needs a concrete representation for both handles and potential handles, and their associated lookahead symbols. We represent each potential handle with an LR(1) item. An LR(1) item $[A \rightarrow \beta \bullet \gamma, \mathtt{a}]$ consists of a production $A \rightarrow \beta\gamma$; a placeholder, $\bullet$, that indicates the position of the stacktop in the production's right-hand side; and a specific terminal symbol, `a`, as a lookahead symbol.

LR(1) item
$[A \rightarrow \beta \bullet \gamma, \mathtt{a}]$ where $A \rightarrow \beta\gamma$ is a grammar production, $\bullet$ represents the position of the parser's stacktop, and `a` is a terminal symbol in the grammar

The table-construction algorithm uses LR(1) items to build a model of the sets of valid states for the parser, the canonical collection of sets of LR(1) items. We designate the canonical collection $\mathcal{CC} = \{\text{CC}_0, \text{CC}_1, \text{CC}_2, \ldots, \text{CC}_n\}$. The algorithm builds $\mathcal{CC}$ by following possible derivations in the grammar; in the final collection, each set CC_i in $\mathcal{CC}$ contains the set of potential handles in some possible parser configuration. Before we delve into the table construction, further explanation of LR(1) items is needed.

For a production $A \rightarrow \beta\gamma$ and a lookahead symbol `a`, the placeholder can generate three distinct items, each with its own interpretation. In each case, the presence of the item in some set CC_i in the canonical collection indicates input that the parser has seen is consistent with the occurrence of an A followed by an `a` in the grammar. The position of $\bullet$ in the item distinguishes between the three cases.

1. $[A \rightarrow \bullet\beta\gamma, \mathtt{a}]$ indicates that an A would be valid and that recognizing a β next would be one step toward discovering an A. We call such an item a *possibility*, because it represents a possible completion for the input already seen.
2. $[A \rightarrow \beta \bullet \gamma, \mathtt{a}]$ indicates that the parser has progressed from the state $[A \rightarrow \bullet\beta\gamma, \mathtt{a}]$ by recognizing β. The β is consistent with recognizing

[*Goal* → • *List*,eof]		
[*Goal* → *List* •,eof]		
[*List* → • *List Pair*,eof]	[*List* → • *List Pair*,( ]	
[*List* → *List* • *Pair*,eof]	[*List* → *List* • *Pair*,( ]	
[*List* → *List Pair* •,eof]	[*List* → *List Pair* •,( ]	
[*List* → • *Pair*,eof]	[*List* → • *Pair*,( ]	
[*List* → *Pair* •,eof]	[*List* → *Pair* •,( ]	
[*Pair* → • (*Pair*),eof]	[*Pair* → • (*Pair*),)]	[*Pair* → • (*Pair*),(]
[*Pair* → (• *Pair*),eof]	[*Pair* → (• *Pair*),)]	[*Pair* → (• *Pair*),(]
[*Pair* → (*Pair* •),eof]	[*Pair* → (*Pair* •),)]	[*Pair* → (*Pair* •),(]
[*Pair* → (*Pair*) •,eof]	[*Pair* → (*Pair*) •,)]	[*Pair* → (*Pair*) •,(]
[*Pair* → • ( ),eof]	[*Pair* → • ( ),(]	[*Pair* → • ( ),)]
[*Pair* → (•),eof]	[*Pair* → (•),(]	[*Pair* → (•),)]
[*Pair* → ( ) •,eof]	[*Pair* → ( ) •,(]	[*Pair* → ( ) •,)]

■ **FIGURE 3.19** LR(1) Items for the Parentheses Grammar.

an *A*. One valid next step would be to recognize a γ. We call such an item *partially complete.*

3. [$A \rightarrow \beta\gamma \bullet$,a] indicates that the parser has found $\beta\gamma$ in a context where an *A* followed by an a would be valid. If the lookahead symbol is a, then the item is a handle and the parser can reduce $\beta\gamma$ to *A*. Such an item is *complete*.

In an LR(1) item, the • encodes some local left context—the portions of the production already recognized. (Recall, from the earlier examples, that the states pushed onto the stack encode a summary of the context to the left of the current LR(1) item—in essence, the history of the parse so far.) The lookahead symbol encodes one symbol of legal right context. When the parser finds itself in a state that includes [$A \rightarrow \beta\gamma \bullet$,a] with a lookahead of a, it has a handle and should reduce $\beta\gamma$ to *A*.

Figure 3.19 shows the complete set of LR(1) items generated by the parentheses grammar. Two items deserve particular notice. The first, [*Goal* → • *List*,eof], represents the initial state of the parser—looking for a string that reduces to *Goal*, followed by eof. Every parse begins in this state. The second, [*Goal* → *List* •,eof], represents the desired final state of the parser—finding a string that reduces to *Goal*, followed by eof. This item represents every successful parse. All of the possible parses result from stringing together parser states in a grammar-directed way, beginning with [*Goal* → • *List*,eof] and ending with [*Goal* → *List* •,eof].

Constructing the Canonical Collection

To build the canonical collection of sets of LR(1) items, $\mathcal{CC}$, a parser generator must start from the parser's initial state, [*Goal* → • *List*,eof], and construct a model of all the potential transitions that can occur. The algorithm represents each possible configuration, or state, of the parser as a set of LR(1) items. The algorithm relies on two fundamental operations on these sets of LR(1) items: taking a closure and computing a transition.

- The closure operation completes a state; given some core set of LR(1) items, it adds to that set any related LR(1) items that they imply. For example, anywhere that *Goal* → *List* is legal, the productions that derive a *List* are legal, too. Thus, the item [*Goal* → • *List*,eof] implies both [*List* → • *List Pair*,eof] and [*List* → • *Pair*,eof]. The `closure` procedure implements this function.
- To model the transition that the parser would make from a given state on some grammar symbol, *x*, the algorithm computes the set of items that would result from recognizing an *x*. To do so, the algorithm selects the subset of the current set of LR(1) items where • precedes *x* and advances the • past the *x* in each of them. The `goto` procedure implements this function.

To simplify the task of finding the goal symbol, we require that the grammar have a unique goal symbol that does not appear on the right-hand side of any production. In the parentheses grammar, that symbol is *Goal*.

The item [*Goal* → • *List*,eof] represents the parser's initial state for the parentheses grammar; every valid parse recognizes *Goal* followed by eof. This item forms the core of the first state in $\mathcal{CC}$, labelled cc_0. If the grammar has multiple productions for the goal symbol, each of them generates an item in the initial core of cc_0.

The `closure` Procedure

To compute the complete initial state of the parser, cc_0, from its core, the algorithm must add to the core all of the items implied by the items in the core. Figure 3.20 shows an algorithm for this computation. `Closure` iterates over all the items in set *s*. If the placeholder • in an item immediately precedes some nonterminal *C*, then `closure` must add one or more items for each production that can derive *C*. Closure places the • at the initial position of each item that it builds this way.

The rationale for `closure` is clear. If $[A \rightarrow \beta \bullet C\delta, a] \in s$, then a string that reduces to *C*, followed by δa will complete the left context. Recognizing a *C* followed by δa should cause a reduction to *A*, since it completes the

```
closure(s)
   while (s is still changing)
      for each item [A→β•Cδ,a] ∈ s
         for each production C→γ ∈ P
            for each b ∈ FIRST(δa)
               s ← s ∪ {[C→•γ,b]}
   return s
```

■ **FIGURE 3.20** The *closure* Procedure.

production's right-hand side ($C\delta$) and follows it with a valid lookahead symbol.

To build the items for a production $C \rightarrow \gamma$, *closure* inserts the placeholder before γ and adds the appropriate lookahead symbols—each terminal that can appear as the initial symbol in δa. This includes every terminal in FIRST(δ). If $\epsilon \in$ FIRST(δ), it also includes a. The notation FIRST(δa) in the algorithm represents this extension of the FIRST set to a string in this way. If δ is ϵ, this devolves into FIRST(a) = { a }.

In our experience, this use of FIRST(δa) is the point in the process where a human is most to likely make a mistake.

For the parentheses grammar, the initial item is [*Goal* → • *List*,eof]. Applying *closure* to that set adds the following items:

[*List* → • *List Pair*,eof], [*List* → • *List Pair*,( ], [*List* → • *Pair*,eof], [*List* → • *Pair*,( ], [*Pair* → • (*Pair*),eof], [*Pair* → • (*Pair*),( ], [*Pair* → • ( ),eof] [*Pair* → • ( ),( ]

These eight items, along with [*Goal* → • *List*,eof], constitute set CC_0 in the canonical collection. The order in which *closure* adds the items will depend on how the set implementation manages the interaction between the "*for each item*" iterator and the set union in the innermost loop.

Closure is another fixed-point computation. The triply-nested loop either adds items to *s* or leaves *s* intact. It never removes an item from *s*. Since the set of LR(1) items is finite, this loop must halt. The triply nested loop looks expensive. However, close examination reveals that each item in *s* needs to be processed only once. A worklist version of the algorithm could capitalize on that fact.

The goto *Procedure*

The second fundamental operation that the construction uses is the *goto* function. *Goto* takes as input a model of a parser state, represented as a set CC_i in the canonical collection, and a grammar symbol *x*. It computes, from CC_i and *x*, a model of the parser state that would result from recognizing an *x* in state *i*.

```
goto(s,x)
  moved ← ∅
  for each item i ∈ s
      if the form of i is [α→β•xδ, a] then
          moved ← moved ∪ {[α→βx•δ, a]}
  return closure(moved)
```

■ **FIGURE 3.21** The *goto* Function.

The *goto* function, shown in Figure 3.21, takes a set of LR(1) items s and a grammar symbol x and returns a new set of LR(1) items. It iterates over the items in s. When it finds an item in which the • immediately precedes x, it creates a new item by moving the • rightward past x. This new item represents the parser's configuration after recognizing x. *Goto* places these new items in a new set, takes its *closure* to complete the parser state, and returns that new state.

Given the initial set for the parentheses grammar,

$$CC_0 = \left\{ \begin{array}{ccc} [Goal \rightarrow \bullet\, List, \texttt{eof}] & [List \rightarrow \bullet\, List\ Pair, \texttt{eof}] & [List \rightarrow \bullet\, List\ Pair, \underline{(}] \\ [List \rightarrow \bullet\, Pair, \texttt{eof}] & [List \rightarrow \bullet\, Pair, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \texttt{eof}] \\ [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

we can derive the state of the parser after it recognizes an initial $\underline{(}$ by computing *goto*(CC_0,$\underline{(}$). The inner loop finds four items that have • before $\underline{(}$. *Goto* creates a new item for each, with the • advanced beyond $\underline{(}$. *Closure* adds two more items, generated from the items with • before *Pair*. These items introduce the lookahead symbol $\underline{)}$. Thus, *goto*(CC_0,$\underline{(}$) returns

$$\left\{ \begin{array}{ccc} [Pair \rightarrow \underline{(} \bullet\, Pair\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \underline{(} \bullet\, Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \underline{(} \bullet\, \underline{)}, \texttt{eof}] \\ [Pair \rightarrow \underline{(} \bullet\, \underline{)}, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{)}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{)}] \end{array} \right\}.$$

To find the set of states that derive directly from some state such as CC_0, the algorithm can compute *goto*(CC_0,x) for each x that occurs after a • in an item in CC_0. This produces all the sets that are one symbol away from CC_0. To compute the complete canonical collection, we simply iterate this process to a fixed point.

The Algorithm

To construct the canonical collection of sets of LR(1) items, the algorithm computes the initial set, CC_0, and then systematically finds all of the sets of LR(1) items that are reachable from CC_0. It repeatedly applies *goto* to the new sets in $\mathcal{CC}$; *goto*, in turn, uses *closure*. Figure 3.22 shows the algorithm.

For a grammar with the goal production $S' \rightarrow S$, the algorithm begins by initializing $\mathcal{CC}$ to contain CC_0, as described earlier. Next, it systematically

```
CC0 ← closure([S′→•S, eof])
CC ← {CC0}
while (new sets are still being added to CC)
  for each unmarked set CCi ∈ CC
      mark CCi as processed
      for each x following a • in an item in CCi
          temp ← goto(CCi, x)
          if temp ∉ CC
              then CC ← CC ∪ {temp}
          record transition from CCi to temp on x
```

■ **FIGURE 3.22** The Algorithm to Build $\mathcal{CC}$.

extends $\mathcal{CC}$ by looking for any transition from a state in $\mathcal{CC}$ to a state not yet in $\mathcal{CC}$. It does this constructively, by building each possible state, *temp*, and testing *temp* for membership in $\mathcal{CC}$. If *temp* is new, it adds *temp* to $\mathcal{CC}$. Whether or not *temp* is new, it records the transition from CC_i to *temp* for later use in building the parser's *Goto* table.

To ensure that the algorithm processes each set CC_i just once, it uses a simple marking scheme. It creates each set in an unmarked condition and marks the set as it is processed. This drastically reduces the number of times that it invokes *goto* and *closure*.

This construction is a fixed-point computation. The canonical collection, $\mathcal{CC}$, is a subset of the powerset of the LR(1) items. The while loop is monotonic; it adds new sets to $\mathcal{CC}$ and never removes them. If the set of LR(1) items has n elements, then $\mathcal{CC}$ can grow no larger than 2^n items, so the computation must halt.

This upper bound on the size of $\mathcal{CC}$ is quite loose. For example, the parentheses grammar has 33 LR(1) items and produces just 12 sets in $\mathcal{CC}$. The upper bound would be 2^{33}, a much larger number. For more complex grammars, $|\mathcal{CC}|$ is a concern, primarily because the *Action* and *Goto* tables grow with $|\mathcal{CC}|$. As described in Section 3.6, both the compiler writer and the parser-generator writer can take steps to reduce the size of those tables.

The Canonical Collection for the Parentheses Grammar

As a first complete example, consider the problem of building $\mathcal{CC}$ for the parentheses grammar. The initial set, CC_0, is computed as *closure*([*Goal* → • *List*, eof]).

Iteration	Item	*Goal*	*List*	*Pair*	(	)	eof
0	CC_0	Ø	CC_1	CC_2	CC_3	Ø	Ø
1	CC_1	Ø	Ø	CC_4	CC_3	Ø	Ø
	CC_2	Ø	Ø	Ø	Ø	Ø	Ø
	CC_3	Ø	Ø	CC_5	CC_6	CC_7	Ø
2	CC_4	Ø	Ø	Ø	Ø	Ø	Ø
	CC_5	Ø	Ø	Ø	Ø	CC_8	Ø
	CC_6	Ø	Ø	CC_9	CC_6	CC_{10}	Ø
	CC_7	Ø	Ø	Ø	Ø	Ø	Ø
3	CC_8	Ø	Ø	Ø	Ø	Ø	Ø
	CC_9	Ø	Ø	Ø	Ø	CC_{11}	Ø
	CC_{10}	Ø	Ø	Ø	Ø	Ø	Ø
4	CC_{11}	Ø	Ø	Ø	Ø	Ø	Ø

■ **FIGURE 3.23** Trace of the LR(1) Construction on the Parentheses Grammar.

$$CC_0 = \left\{ \begin{array}{ccc} [Goal \rightarrow \bullet\, List, \texttt{eof}] & [List \rightarrow \bullet\, List\ Pair, \texttt{eof}] & [List \rightarrow \bullet\, List\ Pair, \underline{(}] \\ [List \rightarrow \bullet\, Pair, \texttt{eof}] & [List \rightarrow \bullet\, Pair, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \texttt{eof}] \\ [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}] \end{array} \right\}$$

Since each item has the • at the start of its right-hand side, CC_0 contains only possibilities. This is appropriate, since it is the parser's initial state. The first iteration of the *while* loop produces three sets, CC_1, CC_2, and CC_3. All of the other combinations in the first iteration produce empty sets, as indicated in Figure 3.23, which traces the construction of $\mathcal{CC}$.

goto(CC_0, *List*) is CC_1.

$$CC_1 = \left\{ \begin{array}{ccc} [Goal \rightarrow List \bullet, \texttt{eof}] & [List \rightarrow List \bullet\, Pair, \texttt{eof}] & [List \rightarrow List \bullet\, Pair, \underline{(}] \\ [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \texttt{eof}] & [Pair \rightarrow \bullet\, \underline{(}\ Pair\ \underline{)}, \underline{(}] & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \texttt{eof}] \\ & [Pair \rightarrow \bullet\, \underline{(}\ \underline{)}, \underline{(}] & \end{array} \right\}$$

CC_1 represents the parser configurations that result from recognizing a *List*. All of the items are possibilities that lead to another pair of parentheses, except for the item [*Goal* → *List* •, eof]. It represents the parser's accept state—a reduction by *Goal* → *List*, with a lookahead of eof.

goto(CC_0, *Pair*) is CC_2.

$$CC_2 = \left\{ [List \rightarrow Pair \bullet, \texttt{eof}] \quad [List \rightarrow Pair \bullet, \underline{(}] \right\}$$

CC_2 represents the parser configurations after it has recognized an initial *Pair*. Both items are handles for a reduction by *List* → *Pair*.

goto(CC_0,() is CC_3.

$$CC_3 = \left\{ \begin{array}{ccc} [Pair \rightarrow \bullet \text{ (} Pair \text{) }, \text{)}] & [Pair \rightarrow \text{ (} \bullet Pair \text{) }, \texttt{eof}] & [Pair \rightarrow \text{ (} \bullet Pair \text{) }, \text{ (}] \\ [Pair \rightarrow \bullet \text{ ( ) }, \text{)}] & [Pair \rightarrow \text{ (} \bullet \text{) }, \texttt{eof}] & [Pair \rightarrow \text{ (} \bullet \text{) }, \text{ (}] \end{array} \right\}$$

CC_3 represents the parser's configuration after it recognizes an initial (. When the parser enters state 3, it must recognize a matching) at some point in the future.

The second iteration of the *while* loop tries to derive new sets from CC_1, CC_2, and CC_3. Five of the combinations produce nonempty sets, four of which are new.

goto(CC_1, *Pair*) is CC_4.

$$CC_4 = \left\{ [List \rightarrow List\ Pair \bullet, \texttt{eof}] \quad [List \rightarrow List\ Pair \bullet, \text{ (}] \right\}$$

The left context for this set is CC_1, which represents a state where the parser has recognized one or more occurrences of *List*. When it then recognizes a *Pair*, it enters this state. Both items represent a reduction by *List* → *List Pair*.

goto(CC_1,() is CC_3, which represents the future need to find a matching).

goto(CC_3, *Pair*) is CC_5.

$$CC_5 = \left\{ [Pair \rightarrow \text{ (} Pair \bullet \text{) }, \texttt{eof}] \quad [Pair \rightarrow \text{ (} Pair \bullet \text{) }, \text{ (}] \right\}$$

CC_5 consists of two partially complete items. The parser has recognized a (followed by a *Pair*; it now must find a matching). If the parser finds a), it will reduce by rule 4, *Pair* → (*Pair*).

goto(CC_3,() is CC_6.

$$CC_6 = \left\{ \begin{array}{cc} [Pair \rightarrow \bullet \text{ (} Pair \text{) }, \text{)}] & [Pair \rightarrow \text{ (} \bullet Pair \text{) }, \text{)}] \\ [Pair \rightarrow \bullet \text{ ( ) }, \text{)}] & [Pair \rightarrow \text{ (} \bullet \text{) }, \text{)}] \end{array} \right\}$$

The parser arrives in CC_6 when it encounters a (and it already has at least one (on the stack. The items show that either a (or a) lead to valid states.

goto(CC_3,)) is CC_7.

$$CC_7 = \left\{ [Pair \rightarrow \text{ ( ) } \bullet, \texttt{eof}] \quad [Pair \rightarrow \text{ ( ) } \bullet, \text{ (}] \right\}$$

If, in state 3, the parser finds a), it takes the transition to CC_7. Both items specify a reduction by *Pair* → ( ).

The third iteration of the `while` loop tries to derive new sets from CC_4, CC_5, CC_6, and CC_7. Three of the combinations produce new sets, while one produces a transition to an existing state.

goto(CC_5,) is CC_8.

$$CC_8 = \{[Pair \rightarrow \underline{(}\ Pair\ \underline{)}\ \bullet, \texttt{eof}] \quad [Pair \rightarrow \underline{(}\ Pair\ \underline{)}\ \bullet, \underline{(}]\}$$

When it arrives in state 8, the parser has recognized an instance of rule 4, $Pair \rightarrow \underline{(}\ Pair\ \underline{)}$. Both items specify the corresponding reduction.

goto(CC_6, *Pair*) is CC_9.

$$CC_9 = \{[Pair \rightarrow \underline{(}\ Pair \bullet\ \underline{)}, \underline{)}]\}$$

In CC_9, the parser needs to find a $\underline{)}$ to complete rule 4.

goto(CC_6,) is CC_6. In CC_6, another $\underline{(}$ will cause the parser to stack another state 6 to represent the need for a matching $\underline{)}$.

goto(CC_6,) is CC_{10}.

$$CC_{10} = \{[Pair \rightarrow \underline{(}\ \underline{)}\ \bullet, \underline{)}]\}$$

This set contains one item, which specifies a reduction to *Pair*.

The fourth iteration of the *while* loop tries to derive new sets from CC_8, CC_9, and CC_{10}. Only one combination creates a nonempty set.

goto(CC_9,) is CC_{11}.

$$CC_{11} = \{[Pair \rightarrow \underline{(}\ Pair\ \underline{)}\ \bullet, \underline{)}]\}$$

State 11 calls for a reduction by $Pair \rightarrow \underline{(}\ Pair\ \underline{)}$.

The final iteration of the `while` loop tries to derive new sets from CC_{11}. It finds only empty sets, so the construction halts with 12 sets, CC_0 through CC_{11}.

Filling in the Tables

Given the canonical collection of sets of LR(1) items for a grammar, the parser generator can fill in the *Action* and *Goto* tables by iterating through $\mathcal{CC}$ and examining the items in each $CC_j \in \mathcal{CC}$. Each CC_j becomes a parser state. Its items generate the nonempty elements of one row of *Action*; the corresponding transitions recorded during construction of $\mathcal{CC}$ specify the nonempty elements of *Goto*. Three cases generate entries in the *Action* table:

1. An item of the form $[A \rightarrow \beta \bullet \mathtt{c}\gamma, \mathtt{a}]$ indicates that encountering the terminal symbol `c` would be a valid next step toward discovering the nonterminal A. Thus, it generates a *shift* item on `c` in the current state. The next state for the recognizer is the state generated by computing `goto` on the current state with the terminal `c`. Either β or γ can be ϵ.
2. An item of the form $[A \rightarrow \beta \bullet, \mathtt{a}]$ indicates that the parser has recognized a β, and if the lookahead is `a`, then the item is a handle. Thus, it generates a *reduce* item for the production $A \rightarrow \beta$ on `a` in the current state.
3. An item of the form $[S' \rightarrow S \bullet, \mathtt{eof}]$ where S' is the goal symbol indicates the accepting state for the parser; the parser has recognized an input stream that reduces to the goal symbol and the lookahead symbol is `eof`. This item generates an *accept* action on `eof` in the current state.

Figure 3.24 makes this concrete. For an LR(1) grammar, it should uniquely define the nonerror entries in the `Action` and `Goto` tables.

The table-filling actions can be integrated into the construction of $\mathcal{CC}$.

Notice that the table-filling algorithm essentially ignores items where the $\bullet$ precedes a nonterminal symbol. Shift actions are generated when $\bullet$ precedes a terminal. Reduce and accept actions are generated when $\bullet$ is at the right end of the production. What if CC_i contains an item $[A \rightarrow \beta \bullet \gamma\delta, \mathtt{a}]$, where $\gamma \in NT$? While this item does not generate any table entries itself, its presence in the set forces the `closure` procedure to include items that generate table entries. When `closure` finds a $\bullet$ that immediately precedes a nonterminal symbol γ, it adds productions that have γ as their left-hand side, with a $\bullet$ preceding their right-hand sides. This process instantiates $\text{FIRST}(\gamma)$ in CC_i. The `closure` procedure will find each $x \in \text{FIRST}(\gamma)$ and add the items into CC_i to generate shift items for each x.

```
for each CC_i ∈ CC
    for each item I ∈ CC_i
      if I is [A→β•cγ,a] and goto(CC_i ,c) = CC_j then
          Action[i ,c] ← "shift j"
      else if I is [A→β•,a] then
          Action[i ,a] ← "reduce A→β"
      else if I is [S'→S•,eof] then
          Action[i ,eof ] ← "accept"
    for each n ∈ NT
      if goto(CC_i ,n) = CC_j then
          Goto[i ,n] ← j
```

■ **FIGURE 3.24** LR(1) Table-Filling Algorithm.

For the parentheses grammar, the construction produces the `Action` and `Goto` tables shown in Figure 3.16b on page 120. As we saw, combining the tables with the skeleton parser in Figure 3.15 creates a functional parser for the language.

In practice, an LR(1) parser generator must produce other tables needed by the skeleton parser. For example, when the skeleton parser in Figure 3.15 on page 119 reduces by $A \rightarrow \beta$, it pops "$2 \times |\beta|$" symbols from the stack and pushes A onto the stack. The table generator must produce data structures that map a production from the reduce entry in the `Action` table, say $A \rightarrow \beta$, into both $|\beta|$ and A. Other tables, such as a map from the integer representing a grammar symbol into its textual name, are needed for debugging and for diagnostic messages.

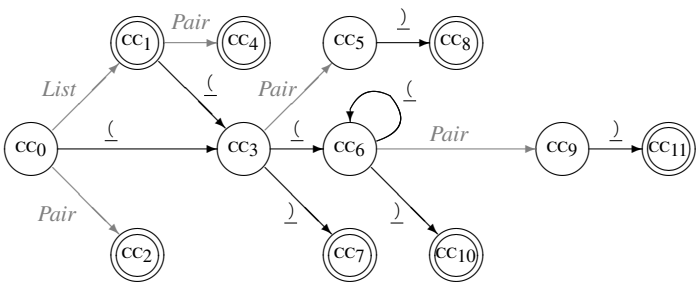
Handle Finding, Revisited

LR(1) parsers derive their efficiency from a fast handle-finding mechanism embedded in the `Action` and `Goto` tables. The canonical collection, $\mathcal{CC}$, represents a handle-finding DFA for the grammar. Figure 3.25 shows the DFA for our example, the parentheses grammar.

How can the LR(1) parser use a DFA to find the handles, when we know that the language of parentheses is not a regular language? The LR(1) parser relies on a simple observation: *the set of handles is finite.* The set of handles is precisely the set of complete LR(1) items—those with the placeholder • at the right end of the item's production. Any language with a finite set of sentences can be recognized by a DFA. Since the number of productions and the number of lookahead symbols are both finite, the number of complete items is finite, and the language of handles is a regular language.

The LR(1) parser makes the handle's position implicit, at stacktop. This design decision drastically reduces the number of possible handles.

When the LR(1) parser executes, it interleaves two kinds of actions: shifts and reduces. The shift actions simulate steps in the handle-finding DFA. The



■ **FIGURE 3.25** Handle-Finding DFA for the Parentheses Grammar.

parser performs one shift action per word in the input stream. When the handle-finding DFA reaches a final state, the LR(1) parser performs a reduce action. The reduce actions reset the state of the handle-finding DFA to reflect the fact that the parser has recognized a handle and replaced it with a nonterminal. To accomplish this, the parser pops the handle and its state off the stack, revealing an older state. The parser uses that older state, the lookahead symbol, and the `Goto` table to discover the state in the DFA from which handle-finding should continue.

The reduce actions tie together successive handle-finding phases. The reduction uses left context—the state revealed by the reduction summarizes the prior history of the parse—to restart the handle-finding DFA in a state that reflects the nonterminal that the parser just recognized. For example, in the parse of "(( )) ( )", the parser stacked an instance of state 3 for every (that it encounters. These stacked states allow the algorithm to match up the opening and closing parentheses.

Notice that the handle-finding DFA has transitions on both terminal and nonterminal symbols. The parser traverses the nonterminal edges only on a reduce action. Each of these transitions, shown in gray in Figure 3.25, corresponds to a valid entry in the `Goto` table. The combined effect of the terminal and nonterminal actions is to invoke the DFA recursively each time it must recognize a nonterminal.

3.4.3 Errors in the Table Construction

As a second example of the LR(1) table construction, consider the ambiguous grammar for the classic `if-then-else` construct. Abstracting away the details of the controlling expression and all other statements (by treating them as terminal symbols) produces the following four-production grammar:

1	*Goal*	→	*Stmt*
2	*Stmt*	→	`if expr then` *Stmt*
3		|	`if expr then` *Stmt* `else` *Stmt*
4		|	`assign`

It has two nonterminal symbols, *Goal* and *Stmt*, and six terminal symbols, `if`, `expr`, `then`, `else`, `assign`, and the implicit `eof`.

The construction begins by initializing CC_0 to the item [*Goal* → • *Stmt*, `eof`] and taking its `closure` to produce the first set.

	Item	*Goal*	*Stmt*	if	expr	then	else	assign	eof
0	CC_0	Ø	CC_1	CC_2	Ø	Ø	Ø	CC_3	Ø
1	CC_1	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
	CC_2	Ø	Ø	Ø	CC_4	Ø	Ø	Ø	Ø
	CC_3	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
2	CC_4	Ø	Ø	Ø	Ø	CC_5	Ø	Ø	Ø
3	CC_5	Ø	CC_6	CC_7	Ø	Ø	Ø	CC_8	Ø
4	CC_6	Ø	Ø	Ø	Ø	Ø	CC_9	Ø	Ø
	CC_7	Ø	Ø	Ø	CC_{10}	Ø	Ø	Ø	Ø
	CC_8	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
5	CC_9	Ø	CC_{11}	CC_2	Ø	Ø	Ø	CC_3	Ø
	CC_{10}	Ø	Ø	Ø	Ø	CC_{12}	Ø	Ø	Ø
6	CC_{11}	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø
	CC_{12}	Ø	CC_{13}	CC_7	Ø	Ø	Ø	CC_8	Ø
7	CC_{13}	Ø	Ø	Ø	Ø	Ø	CC_{14}	Ø	Ø
8	CC_{14}	Ø	CC_{15}	CC_7	Ø	Ø	Ø	CC_8	Ø
9	CC_{15}	Ø	Ø	Ø	Ø	Ø	Ø	Ø	Ø

■ **FIGURE 3.26** Trace of the LR(1) Construction on the `If-Then-Else` Grammar.

$$CC_0 = \left\{ \begin{array}{ll} [Goal \rightarrow \bullet\ Stmt, \texttt{eof}] & [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt, \texttt{eof}] \\ [Stmt \rightarrow \bullet\ \texttt{assign}, \texttt{eof}] & [Stmt \rightarrow \bullet\ \texttt{if expr then}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}] \end{array} \right\}$$

From this set, the construction begins deriving the remaining members of the canonical collection of sets of LR(1) items.

Figure 3.26 shows the progress of the construction. The first iteration examines the transitions out of CC_0 for each grammar symbol. It produces three new sets for the canonical collection from CC_0: CC_1 for *Stmt*, CC_2 for `if`, and CC_3 for `assign`. These sets are:

$$CC_1 = \left\{ [Goal \rightarrow Stmt\ \bullet, \texttt{eof}] \right\}$$

$$CC_2 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if} \bullet\ \texttt{expr then}\ Stmt, \texttt{eof}], \\ [Stmt \rightarrow \texttt{if} \bullet\ \texttt{expr then}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}] \end{array} \right\}$$

$$CC_3 = \left\{ [Stmt \rightarrow \texttt{assign}\ \bullet, \texttt{eof}] \right\}$$

The second iteration examines transitions out of these three new sets. Only one combination produces a new set, looking at CC_2 with the symbol `expr`.

$$CC_4 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr} \bullet\ \texttt{then}\ Stmt, \texttt{eof}], \\ [Stmt \rightarrow \texttt{if expr} \bullet\ \texttt{then}\ Stmt\ \texttt{else}\ Stmt, \texttt{eof}] \end{array} \right\}$$

The next iteration computes transitions from CC_4; it creates CC_5 as *goto*(CC_4, then).

$$CC_5 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then} \bullet Stmt, \texttt{eof}], \\ [Stmt \rightarrow \texttt{if expr then} \bullet Stmt \texttt{ else } Stmt, \texttt{eof}], \\ [Stmt \rightarrow \bullet \texttt{ if expr then } Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{ assign}, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{ if expr then } Stmt \texttt{ else } Stmt, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

The fourth iteration examines transitions out of CC_5. It creates new sets for *Stmt*, for `if`, and for `assign`.

$$CC_6 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then } Stmt \bullet, \texttt{eof}], \\ [Stmt \rightarrow \texttt{if expr then } Stmt \bullet \texttt{else } Stmt, \texttt{eof}] \end{array} \right\}$$

$$CC_7 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if} \bullet \texttt{ expr then } Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if} \bullet \texttt{ expr then } Stmt \texttt{ else } Stmt, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

$$CC_8 = \{[Stmt \rightarrow \texttt{assign} \bullet, \{\texttt{eof}, \texttt{else}\}]\}$$

The fifth iteration examines CC_6, CC_7, and CC_8. While most of the combinations produce the empty set, two combinations lead to new sets. The transition on `else` from CC_6 leads to CC_9, and the transition on `expr` from CC_7 creates CC_{10}.

$$CC_9 = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then } Stmt \texttt{ else} \bullet Stmt, \texttt{eof}], \\ [Stmt \rightarrow \bullet \texttt{ if expr then } Stmt, \texttt{eof}], \\ [Stmt \rightarrow \bullet \texttt{ if expr then } Stmt \texttt{ else } Stmt, \texttt{eof}], \\ [Stmt \rightarrow \bullet \texttt{ assign}, \texttt{eof}] \end{array} \right\}$$

$$CC_{10} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr} \bullet \texttt{ then } Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if expr} \bullet \texttt{ then } Stmt \texttt{ else } Stmt, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

When the sixth iteration examines the sets produced in the fifth iteration, it creates two new sets, CC_{11} from CC_9 on *Stmt* and CC_{12} from CC_{10} on `then`. It also creates duplicate sets for CC_2 and CC_3 from CC_9.

$$CC_{11} = \{[Stmt \rightarrow \texttt{if expr then } Stmt \texttt{ else } Stmt \bullet, \texttt{eof}]\}$$

$$CC_{12} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then} \bullet Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \texttt{if expr then} \bullet Stmt \texttt{ else } Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{if expr then } Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{if expr then } Stmt \texttt{ else } Stmt, \{\texttt{eof}, \texttt{else}\}], \\ [Stmt \rightarrow \bullet \texttt{assign}, \{\texttt{eof}, \texttt{else}\}] \end{array} \right\}$$

Iteration seven creates CC_{13} from CC_{12} on *Stmt*. It recreates CC_7 and CC_8.

$$CC_{13} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then } Stmt \bullet \texttt{ , \{eof, else\}}], \\ [Stmt \rightarrow \texttt{if expr then } Stmt \bullet \texttt{ else } Stmt, \texttt{\{eof, else\}}] \end{array} \right\}$$

Iteration eight finds one new set, CC_{14} from CC_{13} on the transition for `else`.

$$CC_{14} = \left\{ \begin{array}{l} [Stmt \rightarrow \texttt{if expr then } Stmt \texttt{ else} \bullet Stmt, \texttt{\{eof, else\}}], \\ [Stmt \rightarrow \bullet \texttt{ if expr then } Stmt, \texttt{\{eof, else\}}], \\ [Stmt \rightarrow \bullet \texttt{ if expr then } Stmt \texttt{ else } Stmt, \texttt{\{eof, else\}}], \\ [Stmt \rightarrow \bullet \texttt{ assign, \{eof, else\}}] \end{array} \right\}$$

Iteration nine generates CC_{15} from CC_{14} on the transition for *Stmt*, along with duplicates of CC_7 and CC_8.

$$CC_{15} = \{[Stmt \rightarrow \texttt{if expr then } Stmt \texttt{ else } Stmt \bullet, \texttt{\{eof, else\}}]\}$$

The final iteration looks at CC_{15}. Since the • lies at the end of every item in CC_{15}, it can only generate empty sets. At this point, no additional sets of items can be added to the canonical collection, so the algorithm has reached a fixed point. It halts.

The ambiguity in the grammar becomes apparent during the table-filling algorithm. The items in states CC_0 through CC_{12} generate no conflicts. State CC_{13} contains four items:

1. [*Stmt* → `if expr then` *Stmt* • , `else`]
2. [*Stmt* → `if expr then` *Stmt* • , `eof`]
3. [*Stmt* → `if expr then` *Stmt* • `else` *Stmt*, `else`]
4. [*Stmt* → `if expr then` *Stmt* • `else` *Stmt*, `eof`]

Item 1 generates a reduce entry for CC_{13} and the lookahead `else`. Item 3 generates a shift entry for the same location in the table. Clearly, the table entry cannot hold both actions. This *shift-reduce conflict* indicates that the grammar is ambiguous. Items 2 and 4 generate a similar shift-reduce conflict with a lookahead of `eof`. When the table-filling algorithm encounters such a conflict, the construction has failed. The table generator should report the problem—a fundamental ambiguity between the productions in the specific LR(1) items—to the compiler writer.

A typical error message from a parser generator includes the LR(1) items that generate the conflict; another reason to study the table construction.

In this case, the conflict arises because production 2 in the grammar is a prefix of production 3. The table generator could be designed to resolve this conflict in favor of shifting; that forces the parser to recognize the longer production and binds the `else` to the innermost `if`.

An ambiguous grammar can also produce a *reduce-reduce conflict*. Such a conflict can occur if the grammar contains two productions $A \rightarrow \gamma\delta$ and $B \rightarrow \gamma\delta$, with the same right-hand side $\gamma\delta$. If a state contains the items $[A \rightarrow \gamma\delta \bullet, \mathtt{a}]$ and $[B \rightarrow \gamma\delta \bullet, \mathtt{a}]$, then it will generate two conflicting reduce actions for the lookahead `a`—one for each production. Again, this conflict reflects a fundamental ambiguity in the underlying grammar; the compiler writer must reshape the grammar to eliminate it (see Section 3.5.3).

Since parser generators that automate this process are widely available, the method of choice for determining whether a grammar has the LR(1) property is to invoke an LR(1) parser generator on it. If the process succeeds, the grammar has the LR(1) property.

Exercise 12 shows an LR(1) grammar that has no equivalent LL(1) grammar.

SECTION REVIEW

LR(1) parsers are widely used in compilers built in both industry and academia. These parsers accept a large class of languages. They use time proportional to the size of the derivation that they construct. Tools that generate an LR(1) parser are widely available in a broad variety of implementation languages.

The LR(1) table-construction algorithm is an elegant application of theory to practice. It systematically builds up a model of the handle-recognizing DFA and then translates that model into a pair of tables that drive the skeleton parser. The table construction is a complex undertaking that requires painstaking attention to detail. It is precisely the kind of task that should be automated—parser generators are better at following these long chains of computations than are humans. That notwithstanding, a skilled compiler writer should understand the table-construction algorithms because they provide insight into how the parsers work, what kinds of errors the parser generator can encounter, how those errors arise, and how they can be remedied.

As a final example, the LR tables for the classic expression grammar appear in Figures 3.31 and 3.32 on pages 151 and 152.

Review Questions

1. Show the steps that the skeleton LR(1) parser, with the tables for the parentheses grammar, would take on the input string "`( ( ) ( ) ) ( )`."
2. Build the LR(1) tables for the *SheepNoise* grammar, given in Section 3.2.2 on page 86, and show the skeleton parser's actions on the input "`baa baa baa`."

3.5 PRACTICAL ISSUES

Even with automatic parser generators, the compiler writer must manage several issues to produce a robust, efficient parser for a real programming language. This section addresses several issues that arise in practice.

3.5.1 Error Recovery

Programmers often compile code that contains syntax errors. In fact, compilers are widely accepted as the fastest way to discover such errors. In this application, the compiler must find as many syntax errors as possible in a single attempt at parsing the code. This requires attention to the parser's behavior in error states.

All of the parsers shown in this chapter have the same behavior when they encounter a syntax error: they report the problem and halt. This behavior prevents the compiler from wasting time trying to translate an incorrect program. However, it ensures that the compiler finds at most one syntax error per compilation. Such a compiler would make finding all the syntax errors in a file of program text a potentially long and painful process.

A parser should find as many syntax errors as possible in each compilation. This requires a mechanism that lets the parser recover from an error by moving to a state where it can continue parsing. A common way of achieving this is to select one or more words that the parser can use to synchronize the input with its internal state. When the parser encounters an error, it discards input symbols until it finds a synchronizing word and then resets its internal state to one consistent with the synchronizing word.

In an Algol-like language, with semicolons as statement separators, the semicolon is often used as a synchronizing word. When an error occurs, the parser calls the scanner repeatedly until it finds a semicolon. It then changes state to one that would have resulted from successful recognition of a complete statement, rather than an error.

In a recursive-descent parser, the code can simply discard words until it finds a semicolon. At that point, it can return control to the point where the routine that parses statements reports success. This may involve manipulating the runtime stack or using a nonlocal jump like C's `setjmp` and `longjmp`.

In an LR(1) parser, this kind of resynchronization is more complex. The parser discards input until it finds a semicolon. Next, it scans backward down the parse stack until it finds a state s such that `Goto`[s,*Statement*] is a valid, nonerror entry. The first such state on the stack represents the statement that

contains the error. The error recovery routine then discards entries on the stack above that state, pushes the state Goto[*s*,*Statement*] onto the stack and resumes normal parsing.

In a table-driven parser, either LL(1) or LR(1), the compiler needs a way of telling the parser generator where to synchronize. This can be done using error productions—a production whose right-hand side includes a reserved word that indicates an error synchronization point and one or more synchronizing tokens. With such a construct, the parser generator can construct error-recovery routines that implement the desired behavior.

Of course, the error-recovery routines should take steps to ensure that the compiler does not try to generate and optimize code for a syntactically invalid program. This requires simple handshaking between the error-recovery apparatus and the high-level driver that invokes the various parts of the compiler.

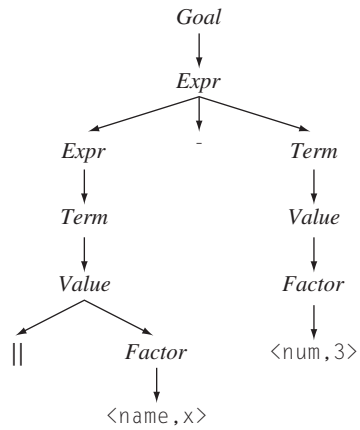
3.5.2 Unary Operators

The classic expression grammar includes only binary operators. Algebraic notation, however, includes unary operators, such as unary minus and absolute value. Other unary operators arise in programming languages, including autoincrement, autodecrement, address-of, dereference, boolean complement, and typecasts. Adding such operators to the expression grammar requires some care.

Consider adding a unary absolute-value operator, ||, to the classic expression grammar. Absolute value should have higher precedence than either x or ÷.

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* x *Value*
5		|	*Term* ÷ *Value*
6		|	*Value*
7	*Value*	→	|| *Factor*
8		|	*Factor*
9	*Factor*	→	(*Expr*)
10		|	num
11		|	name

(a) The Grammar



(b) Parse Tree for ||x - 3

■ **FIGURE 3.27** Adding Unary Absolute Value to the Classic Expression Grammar.

However, it needs a lower precedence than *Factor* to force evaluation of parenthetic expressions before application of ||. One way to write this grammar is shown in Figure 3.27. With these additions, the grammar is still LR(1). It lets the programmer form the absolute value of a number, an identifier, or a parenthesized expression.

Figure 3.27b shows the parse tree for the string `||x - 3`. It correctly shows that the code must evaluate `||x` before performing the subtraction. The grammar does not allow the programmer to write `|||x`, as that makes little mathematical sense. It does, however, allow `||(||x)`, which makes as little sense as `|||x`.

The inability to write `|||x` hardly limits the expressiveness of the language. With other unary operators, however, the issue seems more serious. For example, a C programmer might need to write `**p` to dereference a variable declared as `char **p;`. We can add a dereference production for *Value* as well: *Value* → `*` *Value*. The resulting grammar is still an LR(1) grammar, even if we replace the `x` operator in *Term* → *Term* `x` *Value* with `*`, overloading the operator "`*`" in the way that C does. This same approach works for unary minus.

3.5.3 Handling Context-Sensitive Ambiguity

Using one word to represent two different meanings can create a syntactic ambiguity. One example of this problem arose in the definitions of several early programming languages, including FORTRAN, PL/I, and Ada. These languages used parentheses to enclose both the subscript expressions of an array reference and the argument list of a subroutine or function. Given a textual reference, such as `fee(i,j)`, the compiler cannot tell if `fee` is a two-dimensional array or a procedure that must be invoked. Differentiating between these two cases requires knowledge of `fee`'s declared type. This information is not syntactically obvious. The scanner undoubtedly classifies `fee` as a `name` in either case. A function call and an array reference can appear in many of the same situations.

Neither of these constructs appears in the classic expression grammar. We can add productions that derive them from *Factor*.

Factor	→	*FunctionReference*
	|	*ArrayReference*
	|	`(` *Expr* `)`
	|	`num`
	|	`name`
FunctionReference	→	`name` `(` *ArgList* `)`
ArrayReference	→	`name` `(` *ArgList* `)`

Since the last two productions have identical right-hand sides, this grammar is ambiguous, which creates a reduce-reduce conflict in an LR(1) table builder.

Resolving this ambiguity requires extra-syntactic knowledge. In a recursive-descent parser, the compiler writer can combine the code for *FunctionReference* and *ArrayReference* and add the extra code required to check the `name`'s declared type. In a table-driven parser built with a parser generator, the solution must work within the framework provided by the tools.

Two different approaches have been used to solve this problem. The compiler writer can rewrite the grammar to combine both the function invocation and the array reference into a single production. In this scheme, the issue is deferred until a later step in translation, when it can be resolved with information from the declarations. The parser must construct a representation that preserves all the information needed by either resolution; the later step will then rewrite the reference to its appropriate form as an array reference or as a function invocation.

Alternatively, the scanner can classify identifiers based on their declared types, rather than their microsyntactic properties. This classification requires some hand-shaking between the scanner and the parser; the coordination is not hard to arrange as long as the language has a define-before-use rule. Since the declaration is parsed before the use occurs, the parser can make its internal symbol table available to the scanner to resolve identifiers into distinct classes, such as `variable-name` and `function-name`. The relevant productions become:

$$\begin{array}{lcl} \textit{FunctionReference} & \rightarrow & \texttt{function-name}\ \underline{(}\ \textit{ArgList}\ \underline{)} \\ \textit{ArrayReference} & \rightarrow & \texttt{variable-name}\ \underline{(}\ \textit{ArgList}\ \underline{)} \end{array}$$

Rewritten in this way, the grammar is unambiguous. Since the scanner returns a distinct syntactic category in each case, the parser can distinguish the two cases.

3.5.4 Left versus Right Recursion

As we have seen, top-down parsers need right-recursive grammars rather than left-recursive ones. Bottom-up parsers can accommodate either left or right recursion. Thus, the compiler writer must choose between left and right recursion in writing the grammar for a bottom-up parser. Several factors play into this decision.

Stack Depth

In general, left recursion can lead to smaller stack depths. Consider two alternate grammars for a simple list construct, shown in Figures 3.28a and 3.28b. (Notice the similarity to the *SheepNoise* grammar.) Using these grammars to produce a five-element list leads to the derivations shown in Figures 3.28c and 3.28d, respectively. An LR(1) parser would construct these sequences in reverse. Thus, if we read the derivation from the bottom line to the top line, we can follow the parsers's actions with each grammar.

1. *Left-recursive grammar* This grammar shifts $\texttt{elt}_1$ onto its stack and immediately reduces it to *List*. Next, it shifts $\texttt{elt}_2$ onto the stack and reduces it to *List*. It proceeds until it has shifted each of the five $\texttt{elt}_i$s onto the stack and reduced them to *List*. Thus, the stack reaches a maximum depth of two and an average depth of $\frac{10}{6} = 1\frac{2}{3}$.
2. *Right-recursive grammar* This version shifts all five $\texttt{elt}_i$s onto its stack. Next, it reduces $\texttt{elt}_5$ to *List* using rule two, and the remaining

List → *List* elt
| elt

(a) Left-Recursive Grammar

List → elt *List*
| elt

(b) Right-Recursive Grammar

List

List $\texttt{elt}_5$

List $\texttt{elt}_4$ $\texttt{elt}_5$

List $\texttt{elt}_3$ $\texttt{elt}_4$ $\texttt{elt}_5$

List $\texttt{elt}_2$ $\texttt{elt}_3$ $\texttt{elt}_4$ $\texttt{elt}_5$

$\texttt{elt}_1$ $\texttt{elt}_2$ $\texttt{elt}_3$ $\texttt{elt}_4$ $\texttt{elt}_5$

(c) Derivation with Left Recursion

List

$\texttt{elt}_1$ *List*

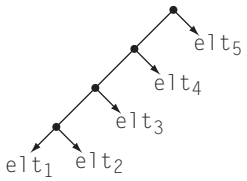
$\texttt{elt}_1$ $\texttt{elt}_2$ *List*

$\texttt{elt}_1$ $\texttt{elt}_2$ $\texttt{elt}_3$ *List*

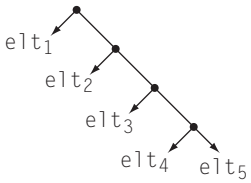
$\texttt{elt}_1$ $\texttt{elt}_2$ $\texttt{elt}_3$ $\texttt{elt}_4$ *List*

$\texttt{elt}_1$ $\texttt{elt}_2$ $\texttt{elt}_3$ $\texttt{elt}_4$
$\texttt{elt}_5$ *List*

(d) Derivation with Right Recursion



(e) AST with Left Recursion



(f) AST with Right Recursion

■ **FIGURE 3.28** Left- and Right-Recursive List Grammars.

elt$_i$s using rule one. Thus, its maximum stack depth will be five and its average will be $\frac{20}{6} = 3\frac{1}{3}$.

The right-recursive grammar requires more stack space; its maximum stack depth is bounded only by the length of the list. In contrast, the maximum stack depth with the left-recursive grammar depends on the grammar rather than the input stream.

For short lists, this is not a problem. If, however, the list represents the statement list in a long run of straight-line code, it might have hundreds of elements. In this case, the difference in space can be dramatic. If all other issues are equal, the smaller stack height is an advantage.

Associativity

Abstract syntax tree
An AST is a contraction of the parse tree. See Section 5.2.1 on page 227.

Left recursion naturally produces left associativity, and right recursion naturally produces right associativity. In some cases, the order of evaluation makes a difference. Consider the abstract syntax trees (ASTs) for the two five-element lists, shown in Figures 3.28e and 3.28f. The left-recursive grammar reduces elt$_1$ to a *List*, then reduces *List* elt$_2$, and so on. This produces the AST shown on the left. Similarly, the right-recursive grammar produces the AST shown on the right.

For a list, neither of these orders is obviously incorrect, although the right-recursive AST may seem more natural. Consider, however, the result if we replace the list constructor with arithmetic operations, as in the grammars

Expr	→	*Expr* + Operand	*Expr*	→	Operand + *Expr*
	|	*Expr* - Operand		|	Operand - *Expr*
	|	Operand		|	Operand

For the string x$_1$ + x$_2$ + x$_3$ + x$_4$ + x$_5$ the left-recursive grammar implies a left-to-right evaluation order, while the right-recursive grammar implies a right-to-left evaluation order. With some number systems, such as floating-point arithmetic, these two evaluation orders can produce different results.

Since the mantissa of a floating-point number is small relative to the range of the exponent, addition can become an identity operation with two numbers that are far apart in magnitude. If, for example, x$_4$ is much smaller than x$_5$, the processor may compute x$_4$ + x$_5$ = x$_5$ With well-chosen values, this effect can cascade and yield different answers from left-to-right and right-to-left evaluations.

Similarly, if any of the terms in the expression is a function call, then the order of evaluation may be important. If the function call changes the value

of a variable in the expression, then changing the evaluation order might change the result.

In a string with subtractions, such as $x_1 - x_2 + x_3$, changing the evaluation order can produce incorrect results. Left associativity evaluates, in a postorder tree walk, to $(x_1 - x_2) + x_3$, the expected result. Right associativity, on the other hand, implies an evaluation order of $x_1 - (x_2 + x_3)$. The compiler must, of course, preserve the evaluation order dictated by the language definition. The compiler writer can either write the expression grammar so that it produces the desired order or take care to generate the intermediate representation to reflect the correct order and associativity, as described in Section 4.5.2.

SECTION REVIEW

Building a compiler involves more than just transcribing the grammar from some language definition. In writing down the grammar, many choices arise that have an impact on both the function and the utility of the resulting compiler. This section dealt with a variety of issues, ranging from how to perform error recovery through the tradeoff between left recursion and right recursion.

Review Questions

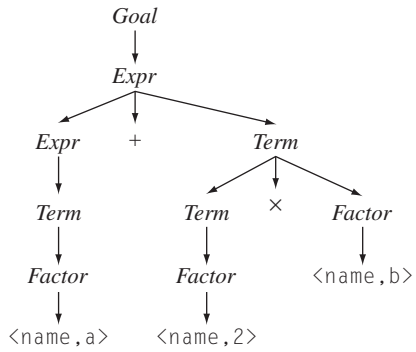
1. The programming language C uses square brackets to indicate an array subscript and parentheses to indicate a procedure or function argument list. How does this simplify the construction of a parser for C?
2. The grammar for unary absolute value introduced a new terminal symbol as the unary operator. Consider adding a unary minus to the classic expression grammar. Does the fact that the same terminal symbol occurs as either a unary minus or a binary minus introduce complications? Justify your answer.

3.6 ADVANCED TOPICS

To build a satisfactory parser, the compiler writer must understand the basics of engineering a grammar and a parser. Given a working parser, there are often ways of improving its performance. This section looks at two specific issues in parser construction. First, we examine transformations on the grammar that reduce the length of a derivation to produce a faster parse. These

0	*Goal*	→	*Expr*
1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* x *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	num
9		|	name

(a) The Classic Expression Grammar



(b) Parse Tree for a + 2 x b

■ **FIGURE 3.29** The Classic Expression Grammar, Revisited.

ideas apply to both top-down and bottom-up parsers. Second, we discuss transformations on the grammar and the `Action` and `Goto` tables that reduce table size. These techniques apply only to LR parsers.

3.6.1 Optimizing a Grammar

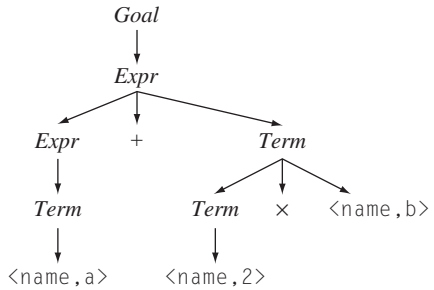
While syntax analysis no longer consumes a major share of compile time, the compiler should not waste undue time in parsing. The actual form of a grammar has a direct effect on the amount of work required to parse it. Both top-down and bottom-up parsers construct derivations. A top-down parser performs an expansion for every production in the derivation. A bottom-up parser performs a reduction for every production in the derivation. A grammar that produces shorter derivations takes less time to parse.

The compiler writer can often rewrite the grammar to reduce the parse tree height. This reduces the number of expansions in a top-down parser and the number of reductions in a bottom-up parser. Optimizing the grammar cannot change the parser's asymptotic behavior; after all, the parse tree must have a leaf node for each symbol in the input stream. Still, reducing the constants in heavily used portions of the grammar, such as the expression grammar, can make enough difference to justify the effort.

Consider, again, the classic expression grammar from Section 3.2.4. (The LR(1) tables for the grammar appear in Figures 3.31 and 3.32.) To enforce the desired precedence among operators, we added two nonterminals, *Term* and *Factor*, and reshaped the grammar into the form shown in Figure 3.29a. This grammar produces rather large parse trees, even for simple expressions. For example, the expression a + 2 x b, the parse tree has 14 nodes, as shown

```
4   Term → Term x ( Expr )
5        | Term x name
6        | Term x num
7        | Term ÷ ( Expr )
8        | Term ÷ name
9        | Term ÷ num
10       | ( Expr )
11       | name
12       | num
```

(a) New Productions for *Term*



(b) Parse Tree for a + 2 x b

■ **FIGURE 3.30** Replacement Productions for *Term*.

in Figure 3.29b. Five of these nodes are leaves that we cannot eliminate. (Changing the grammar cannot shorten the input program.)

Any interior node that has only one child is a candidate for optimization. The sequence of nodes *Expr* to *Term* to *Factor* to ⟨name,a⟩ uses four nodes for a single word in the input stream. We can eliminate at least one layer, the layer of *Factor* nodes, by folding the alternative expansions for *Factor* into *Term*, as shown in Figure 3.30a. It multiplies by three the number of alternatives for *Term*, but shrinks the parse tree by one layer, shown in Figure 3.30b.

In an LR(1) parser, this change eliminates three of nine reduce actions, and leaves the five shifts intact. In a top-down recursive-descent parser for an equivalent predictive grammar, it would eliminate 3 of 14 procedure calls.

In general, any production that has a single symbol on its right-hand side can be folded away. These productions are sometimes called *useless productions*. Sometimes, useless productions serve a purpose—making the grammar more compact and, perhaps, more readable, or forcing the derivation to assume a particular shape. (Recall that the simplest of our expression grammars accepts a + 2 x b but does not encode any notion of precedence into the parse tree.) As we shall see in Chapter 4, the compiler writer may include a useless production simply to create a point in the derivation where a particular action can be performed.

Folding away useless productions has its costs. In an LR(1) parser, it can make the tables larger. In our example, eliminating *Factor* removes one column from the Goto table, but the extra productions for *Term* increase the size of CC from 32 sets to 46 sets. Thus, the tables have one fewer column, but an extra 14 rows. The resulting parser performs fewer reductions (and runs faster), but has larger tables.

In a hand-coded, recursive-descent parser, the larger grammar may increase the number of alternatives that must be compared before expanding some left-hand side. The compiler writer can sometimes compensate for the increased cost by combining cases. For example, the code for both nontrivial expansions of *Expr′* in Figure 3.10 is identical. The compiler writer could combine them with a test that matches `word` against either + or `-`. Alternatively, the compiler writer could assign both + and - to the same syntactic category, have the parser inspect the syntactic category, and use the lexeme to differentiate between the two when needed.

3.6.2 Reducing the Size of LR(1) Tables

Unfortunately, the LR(1) tables generated for relatively small grammars can be large. Figures 3.31 and 3.32 show the canonical LR(1) tables for the classic expression grammar. Many techniques exist for shrinking such tables, including the three approaches to reducing table size described in this section.

Combining Rows or Columns

If the table generator can find two rows, or two columns, that are identical, it can combine them. In Figure 3.31, the rows for states 0 and 7 through 10 are identical, as are rows 4, 14, 21, 22, 24, and 25. The table generator can implement each of these sets once, and remap the states accordingly. This would remove nine rows from the table, reducing its size by 28 percent. To use this table, the skeleton parser needs a mapping from a parser state to a row index in the `Action` table. The table generator can combine identical columns in the analogous way. A separate inspection of the `Goto` table will yield a different set of state combinations—in particular, all of the rows containing only zeros should condense to a single row.

In some cases, the table generator can prove that two rows or two columns differ only in cases where one of the two has an "error" entry (denoted by a blank in our figures). In Figure 3.31, the columns for `eof` and for `num` differ only where one or the other has a blank. Combining such columns produces the same behavior on correct inputs. It does change the parser's behavior on erroneous inputs and may impede the parser's ability to provide accurate and helpful error messages.

Combining rows and columns produces a direct reduction in table size. If this space reduction adds an extra indirection to every table access, the cost of those memory operations must trade off directly against the savings in memory. The table generator could also use other techniques to represent sparse matrices—again, the implementor must consider the tradeoff of memory size against any increase in access costs.

	Action **Table**								
State	eof	+	−	×	÷	(	)	num	name
0						s 4		s 5	s 6
1	acc	s 7	s 8						
2	r 4	r 4	r 4	s 9	s 10				
3	r 7	r 7	r 7	r 7	r 7				
4						s 14		s 15	s 16
5	r 9	r 9	r 9	r 9	r 9				
6	r 10	r 10	r 10	r 10	r 10				
7						s 4		s 5	s 6
8						s 4		s 5	s 6
9						s 4		s 5	s 6
10						s 4		s 5	s 6
11		s 21	s 22				s 23		
12		r 4	r 4	s 24	s 25		r 4		
13		r 7	r 7	r 7	r 7		r 7		
14						s 14		s 15	s 16
15		r 9	r 9	r 9	r 9		r 9		
16		r 10	r 10	r 10	r 10		r 10		
17	r 2	r 2	r 2	s 9	s 10				
18	r 3	r 3	r 3	s 9	s 10				
19	r 5	r 5	r 5	r 5	r 5				
20	r 6	r 6	r 6	r 6	r 6				
21						s 14		s 15	s 16
22						s 14		s 15	s 16
23	r 8	r 8	r 8	r 8	r 8				
24						s 14		s 15	s 16
25						s 14		s 15	s 16
26		s 21	s 22				s 31		
27		r 2	r 2	s 24	s 25		r 2		
28		r 3	r 3	s 24	s 25		r 3		
29		r 5	r 5	r 5	r 5		r 5		
30		r 6	r 6	r 6	r 6		r 6		
31		r 8	r 8	r 8	r 8		r 8		

■ **FIGURE 3.31** Action Table for the Classic Expression Grammar.

Shrinking the Grammar

In many cases, the compiler writer can recode the grammar to reduce the number of productions it contains. This usually leads to smaller tables. For example, in the classic expression grammar, the distinction between a number and an identifier is irrelevant to the productions for *Goal*, *Expr*, *Term*, and *Factor*. Replacing the two productions *Factor* → num and *Factor* →

State	Goto Table		
	Expr	*Term*	*Factor*
0	1	2	3
1			
2			
3			
4	11	12	13
5			
6			
7		17	3
8		18	3
9			19
10			20
11			
12			
13			
14	26	12	13
15			

State	Goto Table		
	Expr	*Term*	*Factor*
16			
17			
18			
19			
20			
21		27	13
22		28	13
23			
24			29
25			30
26			
27			
28			
29			
30			
31			

■ **FIGURE 3.32** *Goto* Table for the Classic Expression Grammar.

`name` with a single production *Factor* → `val` shrinks the grammar by a production. In the *Action* table, each terminal symbol has its own column. Folding `num` and `name` into a single symbol, `val`, removes a column from the *Action* table. To make this work, in practice, the scanner must return the same syntactic category, or word, for both `num` and `name`.

Similar arguments can be made for combining `x` and ÷ into a single terminal `muldiv`, and for combining + and - into a single terminal `addsub`. Each of these replacements removes a terminal symbol and a production. These three changes produce the reduced expression grammar shown in Figure 3.33a. This grammar produces a smaller $\mathcal{CC}$, removing rows from the table. Because it has fewer terminal symbols, it has fewer columns as well.

The resulting *Action* and *Goto* tables are shown in Figure 3.33b. The *Action* table contains 132 entries and the *Goto* table contains 66 entries, for a total of 198 entries. This compares favorably with the tables for the original grammar, with their 384 entries. Changing the grammar produced a 48 percent reduction in table size. The tables still contain opportunities for further reductions. For example, rows 0, 6, and 7 in the *Action* table are identical, as are rows 4, 11, 15, and 17. Similarly, the *Goto* table has many

```
1  Goal   → Expr
2  Expr   → Expr addsub Term
3         |  Term
4  Term   → Term muldiv Factor
5         |  Factor
6  Factor → ( Expr )
7         |  val
```

(a) The Reduced Expression Grammar

	Action **Table**						*Goto* **Table**		
	eof	addsub	muldiv	(	)	val	***Expr***	***Term***	***Factor***
0				s 4		s 5	1	2	3
1	acc	s 6							
2	r 3	r 3	s 7						
3	r 5	r 5	r 5						
4				s 11		s 12	8	9	10
5	r 7	r 7	r 7						
6				s 4		s 5		13	3
7				s 4		s 5			14
8		s 15			s 16				
9		r 3	s 17		r 3				
10		r 5	r 5		r 5				
11				s 11		s 12	18	9	10
12		r 7	r 7		r 7				
13	r 2	r 2	s 7						
14	r 4	r 4	r 4						
15				s 11		s 12		19	10
16	r 6	r 6	r 6						
17				s 11		s 12			20
18		s 15			s 21				
19		r 2	s 17		r 2				
20		r 4	r 4		r 4				
21		r 6	r 6		r 6				

(b) *Action* and *Goto* Tables for the Reduced Expression Grammar

■ **FIGURE 3.33** The Reduced Expression Grammar and its Tables.

rows that only contain the error entry. If table size is a serious concern, rows and columns can be combined after shrinking the grammar.

Other considerations may limit the compiler writer's ability to combine productions. For example, the x operator might have multiple uses that make combining it with ÷ impractical. Similarly, the parser might use separate

productions to let the parser handle two syntactically similar constructs in different ways.

Directly Encoding the Table

As a final improvement, the parser generator can abandon the table-driven skeleton parser in favor of a hard-coded implementation. Each state becomes a small case statement or a collection of `if–then–else` statements that test the type of the next symbol and either shift, reduce, accept, or report an error. The entire contents of the *Action* and *Goto* tables can be encoded in this way. (A similar transformation for scanners is discussed in Section 2.5.2.)

The resulting parser avoids directly representing all of the "don't care" states in the *Action* and *Goto* tables, shown as blanks in the figures. This space savings may be offset by larger code size, since each state now includes more code. The new parser, however, has no parse table, performs no table lookups, and lacks the outer loop found in the skeleton parser. While its structure makes it almost unreadable by humans, it should execute more quickly than the corresponding table-driven parser. With appropriate code-layout techniques, the resulting parser can exhibit strong locality in both the instruction cache and the paging system. For example, we should place all the routines for the expression grammar together on a single page, where they cannot conflict with one another.

Using Other Construction Algorithms

Several other algorithms to construct LR-style parsers exist. Among these techniques are the SLR(1) construction, for simple LR(1), and the LALR(1) construction, for lookahead LR(1). Both of these constructions produce smaller tables than the canonical LR(1) algorithm.

The SLR(1) algorithm accepts a smaller class of grammars than the canonical LR(1) construction. These grammars are restricted so that the lookahead symbols in the LR(1) items are not needed. The algorithm uses FOLLOW sets to distinguish between cases in which the parser should shift and those in which it should reduce. This mechanism is powerful enough to resolve many grammars of practical interest. By using FOLLOW sets, the algorithm eliminates the need for lookahead symbols. This produces a smaller canonical collection and a table with fewer rows.

The LALR(1) algorithm capitalizes on the observation that some items in the set representing a state are critical and that the remaining ones can be derived from the critical items. The LALR(1) table construction only represents the

critical items; again, this produces a canonical collection that is equivalent to the one produced by the SLR(1) construction. The details differ, but the table sizes are the same.

The canonical LR(1) construction presented earlier in the chapter is the most general of these table-construction algorithms. It produces the largest tables, but accepts the largest class of grammars. With appropriate table reduction techniques, the LR(1) tables can approximate the size of those produced by the more limited techniques. However, in a mildly counterintuitive result, any language that has an LR(1) grammar also has an LALR(1) grammar and an SLR(1) grammar. The grammars for these more restrictive forms will be shaped in a way that allows their respective construction algorithms to resolve the situations in which the parser should shift and those in which it should reduce.

3.7 SUMMARY AND PERSPECTIVE

Almost every compiler contains a parser. For many years, parsing was a subject of intense interest. This led to the development of many different techniques for building efficient parsers. The LR(1) family of grammars includes all of the context-free grammars that can be parsed in a deterministic fashion. The tools produce efficient parsers with provably strong error-detection properties. This combination of features, coupled with the widespread availability of parser generators for LR(1), LALR(1), and SLR(1) grammars, has decreased interest in other automatic parsing techniques such as operator precedence parsers.

Top-down, recursive-descent parsers have their own set of advantages. They are, arguably, the easiest hand-coded parsers to construct. They provide excellent opportunities to detect and repair syntax errors. They are efficient; in fact, a well-constructed top-down, recursive-descent parser can be faster than a table-driven LR(1) parser. (The direct encoding scheme for LR(1) may overcome this speed advantage.) In a top-down, recursive-descent parser, the compiler writer can more easily finesse ambiguities in the source language that might trouble an LR(1) parser—such as a language in which keyword names can appear as identifiers. A compiler writer who wants to construct a hand-coded parser, for whatever reason, is well advised to use the top-down, recursive-descent method.

In choosing between LR(1) and LL(1) grammars, the choice becomes one of available tools. In practice, few, if any, programming-language constructs fall in the gap between LR(1) grammars and LL(1) grammars. Thus, starting with an available parser generator is always better than implementing a parser generator from scratch.

More general parsing algorithms are available. In practice, however, the restrictions placed on context-free grammars by the LR(1) and LL(1) classes do not cause problems for most programming languages.

■ CHAPTER NOTES

The earliest compilers used hand-coded parsers [27, 227, 314]. The syntactic richness of Algol 60 challenged early compiler writers. They tried a variety of schemes to parse the language; Randell and Russell give a fascinating overview of the methods used in a variety of Algol 60 compilers [293, Chapter 1].

Irons was one of the first to separate the notion of syntax from translation [202]. Lucas appears to have introduced the notion of recursive-descent parsing [255]. Conway applies similar ideas to an efficient single-pass compiler for COBOL [96].

The ideas behind LL and LR parsing appeared in the 1960s. Lewis and Stearns introduced LL(k) grammars [245]; Rosenkrantz and Stearns described their properties in more depth [305]. Foster developed an algorithm to transform a grammar into LL(1) form [151]. Wood formalized the notion of left-factoring a grammar and explored the theoretical issues involved in transforming a grammar to LL(1) form [353, 354, 355].

Knuth laid out the theory behind LR(1) parsing [228]. DeRemer and others developed techniques, the SLR and LALR table-construction algorithms, that made the use of LR parser generators practical on the limited-memory computers of the day [121, 122]. Waite and Goos describe a technique for automatically eliminating useless productions during the LR(1) table-construction algorithm [339]. Penello suggested direct encoding of the tables into executable code [282]. Aho and Ullman [8] is a definitive reference on both LL and LR parsing. Bill Waite provided the example grammar in exercise 3.7.

Several algorithms for parsing arbitrary context-free grammars appeared in the 1960s and early 1970s. Algorithms by Cocke and Schwartz [91], Younger [358], Kasami [212], and Earley [135] all had similar computational complexity. Earley's algorithm deserves particular note because of its similarity to the LR(1) table-construction algorithm. Earley's algorithm derives the set of possible parse states at parse time, rather than at runtime, where the LR(1) techniques precompute these in a parser generator. From a high-level view, the LR(1) algorithms might appear as a natural optimization of Earley's algorithm.

■ EXERCISES

1. Write a context-free grammar for the syntax of regular expressions. Section 3.2

2. Write a context-free grammar for the Backus-Naur form (BNF) notation for context-free grammars.

3. When asked about the definition of an *unambiguous context-free grammar* on an exam, two students gave different answers. The first defined it as "a grammar where each sentence has a unique syntax tree by leftmost derivation." The second defined it as "a grammar where each sentence has a unique syntax tree by any derivation." Which one is correct?

4. The following grammar is not suitable for a top-down predictive parser. Identify the problem and correct it by rewriting the grammar. Show that your new grammar satisfies the LL(1) condition. Section 3.3

$$
\begin{array}{lcl}
L & \rightarrow & R\,\mathtt{a} \\
 & | & Q\,\mathtt{ba}
\end{array}
\qquad
\begin{array}{lcl}
R & \rightarrow & \mathtt{aba} \\
 & | & \mathtt{caba} \\
 & | & R\,\mathtt{bc}
\end{array}
\qquad
\begin{array}{lcl}
Q & \rightarrow & \mathtt{bbc} \\
 & | & \mathtt{bc}
\end{array}
$$

5. Consider the following grammar:

$$
\begin{array}{lcl}
A & \rightarrow & B\,\mathtt{a} \\
B & \rightarrow & \mathtt{dab} \\
 & | & C\,\mathtt{b}
\end{array}
\qquad
\begin{array}{lcl}
C & \rightarrow & \mathtt{c}\,B \\
 & | & A\,\mathtt{c}
\end{array}
$$

Does this grammar satisfy the LL(1) condition? Justify your answer. If it does not, rewrite it as an LL(1) grammar for the same language.

6. Grammars that can be parsed top-down, in a linear scan from left to right, with a k word lookahead are called LL(k) grammars. In the text, the LL(1) condition is described in terms of FIRST sets. How would you define the FIRST sets necessary to describe an LL(k) condition?

7. Suppose an elevator is controlled by two commands: ↑ to move the elevator up one floor and ↓ to move the elevator down one floor. Assume that the building is arbitrarily tall and that the elevator starts at floor x.

Write an LL(1) grammar that generates arbitrary command sequences that (1) never cause the elevator to go below floor x and (2) always return the elevator to floor x at the end of the sequence. For example, ↑↑↓↓ and ↑↓↑↓ are valid command sequences, but ↑↓↓↑ and ↑↓↓ are not. For convenience, you may consider a null sequence as valid. Prove that your grammar is LL(1).

Section 3.4

8. Top-down and bottom-up parsers build syntax trees in different orders. Write a pair of programs, `TopDown` and `BottomUp`, that take a syntax tree and print out the nodes in order of construction. `TopDown` should display the order for a top-down parser, while `BottomUp` should show the order for a bottom-up parser.

9. The *ClockNoise* language (*CN*) is represented by the following grammar:

Goal	→	*ClockNoise*
ClockNoise	→	*ClockNoise* `tick tock`
	|	`tick tock`

a. What are the LR(1) items of *CN*?
b. What are the FIRST sets of *CN*?
c. Construct the Canonical Collection of Sets of LR(1) Items for *CN*.
d. Derive the `Action` and `Goto` tables.

10. Consider the following grammar:

Start	→	*S*
S	→	*A* `a`
A	→	*B C*
	|	*B C* `f`
B	→	`b`
C	→	`c`

a. Construct the canonical collection of sets of LR(1) items for this grammar.
b. Derive the `Action` and `Goto` tables.
c. Is the grammar LR(1)?

11. Consider a robot arm that accepts two commands: ▽ puts an apple in the bag and △ takes an apple out of the bag. Assume the robot arm starts with an empty bag.

A valid command sequence for the robot arm should have no prefix that contains more △ commands than ▽ commands. As examples, ▽▽△△ and ▽△▽ are valid command sequences, but ▽△△▽ and ▽△▽△△ are not.

a. Write an LR(1) grammar that represents all the value command sequences for the robot arm.
b. Prove that the grammar is LR(1).

12. The following grammar has no known LL(1) equivalent:

0	*Start*	→	*A*		
1		|	*B*		
2	*A*	→	(	*A*	)
3		|	a		
4	*B*	→	(	*B*	>
5		|	b		

Show that the grammar is LR(1).

13. Write a grammar for expressions that can include binary operators (+ and x), unary minus (-), autoincrement (++), and autodecrement (- -) with their customary precedence. Assume that repeated unary minuses are not allowed, but that repeated autoincrement and autodecrement operators are allowed.

Section 3.6

14. Consider the task of building a parser for the programming language Scheme. Contrast the effort required for a top-down recursive-descent parser with that needed for a table-driven LR(1) parser. (Assume that you already have an LR(1) table generator.)

Section 3.7

15. The text describes a manual technique for eliminating useless productions in a grammar.

a. Can you modify the LR(1) table-construction algorithm so that it automatically eliminates the overhead from useless productions?

b. Even though a production is syntactically useless, it may serve a practical purpose. For example, the compiler writer might associate a syntax-directed action (see Chapter 4) with the useless production. How should your modified table-construction algorithm handle an action associated with a useless production?