1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	`num`
9		|	`ident`

FIGURE 3.1 The Classic Expression Grammar

A postorder tree walk over this parse tree will first evaluate $\texttt{num}_2$ × $\texttt{num}_3$ and then subtract the result from $\texttt{num}_1$. 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.

To add parentheses to the grammar requires another level of precedence. Parentheses have a higher level of precedence than either × or ÷; this forces evaluation of an expression enclosed in parentheses before any operator that either precedes or follows the parenthetic expression. Thus, `a × (b - c)` evaluates `b - c` to a value and multiplies that value by `a`. To work this into the grammar, we add another nonterminal, *Factor*. The resulting grammar, shown in Figure 3.1, correctly represents the relative precedence of +, -, ×, ÷, and parenthetic expressions. We refer to this grammar as the *classic expression grammar*.

Other operations require high precedence. For example, array subscripts should be applied before standard arithmetic operations. This ensures, for example, that `x + y[i]` evaluates `y[i]` to a value before adding it to ×, as opposed to treating `i` as a subscript on some array whose location is computed as `x + y`. 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 `x ← y + z` would assign y's value to × before performing the addition, assuming a left-to-right evaluation.

3.4 Bottom-Up Parsing

A bottom-up parser builds the parse tree starting with its leaves and working toward its root. As it encounters each word in the input stream, it constructs a leaf node. These form the base of the parse tree. To construct a derivation, it adds layers of nonterminals on top of the leaves, in a structure dictated by both the grammar and the input stream. The upper edge of this partially constructed parse tree is called its *upper frontier*. This process extends the frontier upward, toward the tree's root.

At each step, the parser looks for a section of the upper frontier that matches the right-hand side of some production in the grammar. When it finds a match, the parser builds a node to represent the nonterminal symbol on the production's left-hand side and adds edges to the nodes that represent the symbols on the right-hand side. Since productions have only one nonterminal on their left-hand sides, these upward extensions replace one or more symbols on the frontier with a single symbol. The parser repeats this process until one of two conditions occurs:

1. The upper frontier reduces to a single node that represents the grammar's start symbol. If the parser has matched all the words in the input, then the input is a valid sentence in the language. If some words remain, then the input is not a valid sentence. Instead, it is a valid sentence followed by extra words, and the parser should report this error to the user.

2. No match can be found. Since the parser has been unable to build a derivation for the input stream, the input is not a valid sentence. The parser should report the failure to the user. The upper fringe of the parse tree contains information that can be used to construct a diagnostic message.

A successful parse runs through every step of the derivation. A failed parse halts when it can find no further steps, at which point it can use the context accumulated in the tree to produce a meaningful error message. In many cases, it can recover from the error and continue parsing so that it discovers as many syntactic errors as possible in a single parse (see Section 3.6.1).

Derivations in a bottom-up parser begin with the goal symbol and work toward a sentence. Because the parser builds the parse tree bottom up, it discovers derivation steps in reverse order. If the derivation consists of a series of steps that produces the sentential forms

1	*Expr*	→	*Expr* + *Term*
2		|	*Expr* - *Term*
3		|	*Term*
4	*Term*	→	*Term* × *Factor*
5		|	*Term* ÷ *Factor*
6		|	*Factor*
7	*Factor*	→	(*Expr*)
8		|	`num`
9		|	`ident`

FIGURE 3.1 The Classic Expression Grammar

A postorder tree walk over this parse tree will first evaluate $\texttt{num}_2$ × $\texttt{num}_3$ and then subtract the result from $\texttt{num}_1$. 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.

To add parentheses to the grammar requires another level of precedence. Parentheses have a higher level of precedence than either × or ÷; this forces evaluation of an expression enclosed in parentheses before any operator that either precedes or follows the parenthetic expression. Thus, `a × (b - c)` evaluates `b - c` to a value and multiplies that value by `a`. To work this into the grammar, we add another nonterminal, *Factor*. The resulting grammar, shown in Figure 3.1, correctly represents the relative precedence of +, -, ×, ÷, and parenthetic expressions. We refer to this grammar as the *classic expression grammar*.

Other operations require high precedence. For example, array subscripts should be applied before standard arithmetic operations. This ensures, for example, that `x + y[i]` evaluates `y[i]` to a value before adding it to ×, as opposed to treating `i` as a subscript on some array whose location is computed as `x + y`. 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 `x ← y + z` would assign y's value to × before performing the addition, assuming a left-to-right evaluation.

$$S_0 = \gamma_0 \rightarrow \gamma_1 \rightarrow \gamma_2 \rightarrow \cdots \rightarrow \gamma_{n-1} \rightarrow \gamma_n = \textit{sentence},$$

the bottom-up parser will discover $\gamma_{n-1} \rightarrow \gamma_n$ before it discovers $\gamma_{n-2} \rightarrow \gamma_{n-1}$. The bottom-up construction of the tree forces this order. The parser must add the nodes implied by $\gamma_{n-1} \rightarrow \gamma_n$ to the frontier before it can discover any matches that involve those nodes. Thus, it can only discover the nodes in an order consistent with the reverse derivation.

Because the scanner finds the words in the input stream in left-to-right order, the parser should look at the leaves from left to right. This suggests a derivation order that produces terminals from right to left, so that its reverse order matches the scanner's behavior. This leads, rather naturally, to bottom-up parsers that construct, in reverse, a rightmost derivation. At each point, the parser will operate 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.

Bottom-up parsing is easier if the grammar is unambiguous. With an unambiguous grammar, the rightmost derivation is unique. For a large class of unambiguous grammars, γ_{i-1} can be determined from γ_i and a limited amount of context. This leads to an efficient class of bottom-up parsers.

In this section, we consider a specific class of bottom-up parsers called LR(1) parsers. These parsers scan the input from left to right, the order in which scanners return classified words. These parsers build a rightmost derivation, in reverse. LR(1) parsers make decisions, at each step in the parse, based on the history of the parse so far and a lookahead of, at most, one symbol. The name LR(1) derives from these three 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.[2]

3.4.1 Shift-Reduce Parsing

The key to constructing efficient top-down parsers is discovering the correct right-hand side to use at each step. In bottom-up parsing, the critical step is developing an efficient mechanism that finds matches along the tree's current

2. The theory of LR parsing defines a family of parsing techniques, the LR(k) parsers, for arbitrary $k \geq 0$. Here, k denotes the amount of lookahead that the parser needs. LR(1) parsers accept the same set of languages as LR(k) parsers for any $k > 1$; however, the LR(1) grammar for a language may be more complicated than a grammar that requires more lookahead.

upper frontier. Formally, the parser must find some substring, β, of the upper frontier where

1. β is the right-hand side of some production $A \rightarrow \beta$, and
2. $A \rightarrow \beta$ is one step in the rightmost derivation of the input stream.

For the sake of efficiency, we want the parser to accomplish this while looking no more than one word beyond the right end of β.

We can represent each potential match as a pair $\langle A \rightarrow \beta, k \rangle$, where $A \rightarrow \beta$ is a production in G and k is the position on the tree's current frontier of the right end of β. If replacing this occurrence of β with A is the next step in the reverse-rightmost derivation of the input string, then $\langle A \rightarrow \beta, k \rangle$ is called a *handle* of the bottom-up parse. A handle concisely represents the next step in building the reverse rightmost derivation.

A bottom-up parser operates by repeatedly locating handles on the frontier of the current partial parse tree and performing the reductions that they specify. When the frontier does not contain a handle, the parser calls the scanner to obtain the next word, builds the corresponding leaf, and makes it the rightmost leaf in the partially constructed tree. This extends the frontier by one leaf node.

To see how this works, consider parsing the string `x - 2 × y` with the classic expression grammar from Figure 3.1. The state of the parser, at each step, is summarized in Figure 3.9. Figure 3.10 shows the corresponding partial parse tree for each step in the process; the trees are drawn with their frontier elements justified along the top of each drawing. At each step, the parser either finds a handle on the frontier, or it adds to the frontier.

As the example shows, the parser only needs to examine the upper frontier of the partially constructed parse tree. Using this fact, we can build a particularly clean form of bottom-up parser called a *shift-reduce* parser. These parsers use a stack to hold the frontier; this simplifies the algorithm in two ways. First, the stack trivializes the problem of managing space for the frontier. To extend the frontier, the parser simply pushes the current input symbol onto the top of the stack. Second, it ensures that all handles occur with their right end at the top of the stack. This eliminates the need explicitly to represent the handle's position—simplifying the representation and making the set of handles finite.

Figure 3.11 shows a simple shift-reduce parser. To begin, it shifts a designated symbol, *invalid*, onto the stack and gets the first input symbol from the scanner. Then, it follows a simple discipline: it shifts symbols from the input onto the stack until it discovers a handle, and it reduces handles as soon as they are found. It halts when the top of the stack contains *Goal* and the parser has consumed all the input.

	Next Word	Frontier	Handle	Action
1	ident		— none —	*extend*
2	-	ident	⟨*Factor*→ident,1⟩	*reduce*
3	-	*Factor*	⟨*Term*→*Factor*,1⟩	*reduce*
4	-	*Term*	⟨*Expr*→*Term*,1⟩	*reduce*
5	-	*Expr*	— none —	*extend*
6	num	*Expr* -	— none —	*extend*
7	×	*Expr* - num	⟨*Factor*→num,3⟩	*reduce*
8	×	*Expr* - *Factor*	⟨*Term*→*Factor*,3⟩	*reduce*
9	×	*Expr* - *Term*	— none —	*extend*
10	ident	*Expr* - *Term* ×	— none —	*extend*
11	*eof*	*Expr* - *Term* × ident	⟨*Factor*→ident,5⟩	*reduce*
12	*eof*	*Expr* - *Term* × *Factor*	⟨*Term*→*Term* × *Factor*,5⟩	*reduce*
13	*eof*	*Expr* - *Term*	⟨*Expr*→*Expr* - *Term*,3⟩	*reduce*
14	*eof*	*Expr*	⟨*Goal*→*Expr*,1⟩	*reduce*
15	*eof*	*Goal*	— none —	*accept*

FIGURE 3.9 States of the Bottom-Up Parser on x - 2 × y

The algorithm discovers syntax errors when the handle-discovery mechanism fails. Since Figure 3.11 describes handle-finding only in an abstract way, "*if a handle for A→β is on top of the stack*," the details of error detection are not clear. In fact, the figure suggests that an error can occur only when all the input has been shifted onto the stack. As we explore the mechanism for handle-finding, we will see that it discovers syntax errors much earlier in the process. For example, the input string × + ÷ y is not in the language described by the classic expression grammar. The handle-finder should recognize this as soon as it sees that the ÷ follows a +.

Using the algorithm in Figure 3.11, we can reinterpret Figure 3.9 to show the actions of our shift-reduce parser on the input stream x - 2 × y. The column labelled "Next Word" shows the contents of the variable *word* in the algorithm. The column labelled "Frontier" depicts the contents of the stack at each step; the stack top is to the right. Finally, the action *extend* indicates a shift; *reduce* still indicates a reduction.

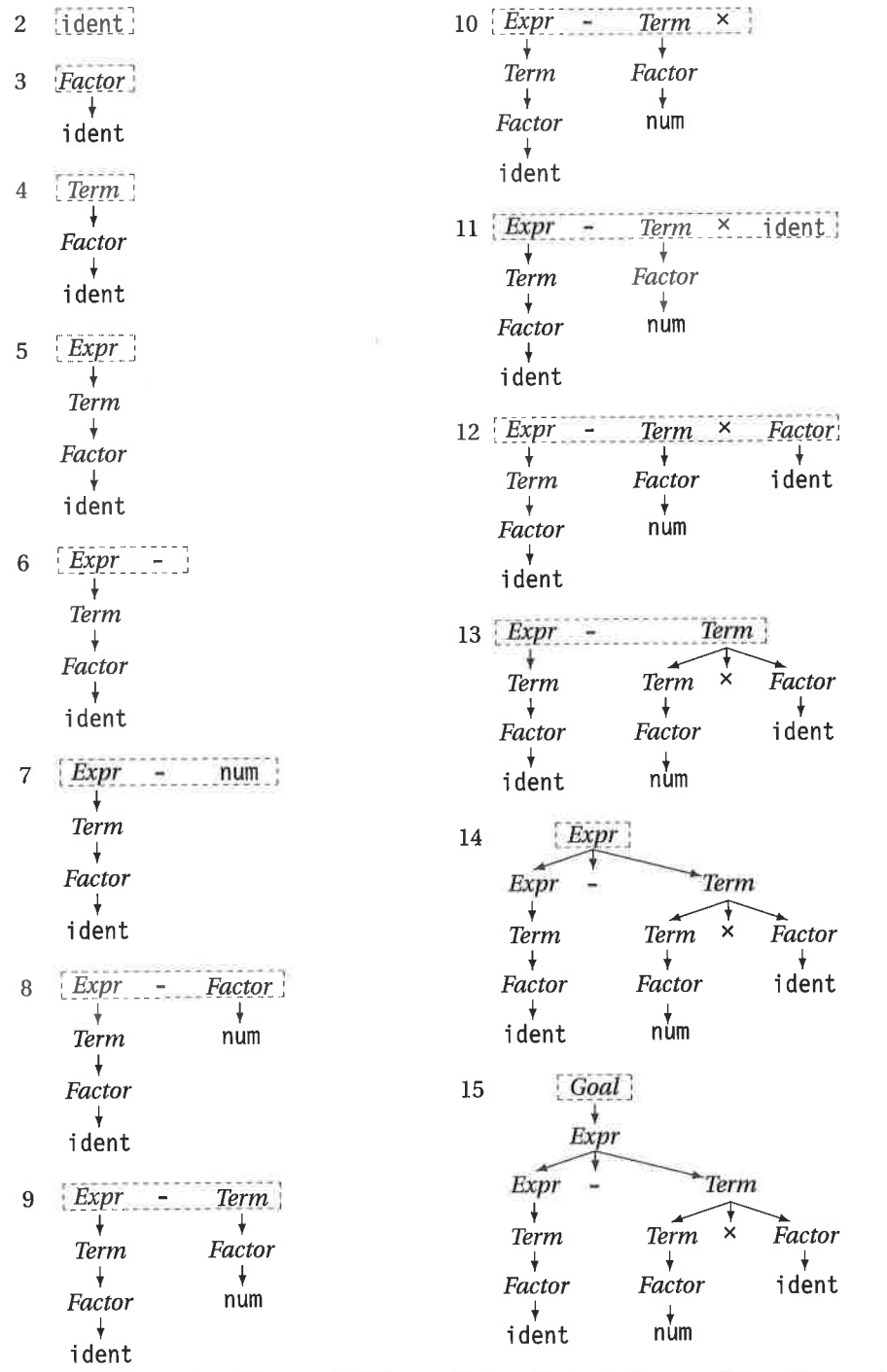

FIGURE 3.10 Partial Parse Trees for a Bottom-Up Parse of x - 2 × y

```
push invalid
word ← NextWord()

repeat until (word = eof & the stack contains
      exactly Goal on top of invalid)
   if a handle for A→β is on top of the stack then
      /* reduce by A→β */
      pop | β | symbols off the stack
      push A onto the stack
   else if (word ≠ eof) then
      /* shift word onto the stack */
      push word
      word ← NextWord()
   else /* no handle, no input */
      report syntax error & halt
```

FIGURE 3.11 Shift-Reduce Parsing Algorithm

For an input stream of length s, this shift-reduce parser performs s shifts. It performs a reduction for each step in the derivation, for r steps. It looks for a handle on each iteration of the *repeat until* loop, so it must perform $s + r$ handle-finding operations. This is equivalent to the number of nodes in the parse tree; each shift and each reduce creates one new node in the parse tree.

For a fixed grammar, r must be $\mathbf{O}(s)$, so the number of handle-finding operations is proportional to the length of the input string. If we can keep the cost of handle-finding to a small constant, the parser will operate in time proportional to $s + r$. Of course, constraining the cost of handle-finding in this way rules out any technique that traverses the entire stack on each handle-finding operation. It places a premium on efficient handle-finding.

3.4.2 Finding Handles

The handle-finding mechanism is the key to efficient bottom-up parsing. As it processes an input string, the parser must find and track all the potential handles. For example, every legal input eventually reduces the entire frontier to the grammar's goal symbol. In the classic expression grammar, *Goal→Expr* is the only production that reduces to *Goal*. It must be the last reduction in any successful parse. If the entire frontier is reduced to *Goal*, then the position in the handle must be 1. Thus, ⟨*Goal→Expr*,1⟩ is a potential handle at the start of every parse.

As the parser builds a derivation, it discovers other handles. At each step, the set of potential handles should represent the different suffixes that, if seen, lead to a reduction. Given the part of the derivation already constructed, each potential handle represents a string of grammar symbols that, if seen, would complete the right-hand side of some production in the grammar. Figure 3.9 shows the nine complete handles that the shift-reduce parser found while processing `x - 2 × y`.

The handles at steps three and eight in that parse both specify a reduction by *Term*→*Factor*. The handles have different position fields, specifying where on the frontier the symbol *Factor* occurs. In a shift-reduce parser, however, that portion of the frontier shown in the table always resides on the stack, with the most recently recognized symbol on top. In both steps three and eight, the position field in the handle specifies the symbol on the top of the stack. In fact, for each handle that the parser recognizes, the position field specifies the top of the stack. If we treat the position field as relative to the top of the stack, we can drastically reduce the number of distinct handles—to one handle per production in the grammar.

Pushing this notion further, we can represent the potential handles that the shift-reduce parser should track. If we use the placeholder • to represent the top of the stack, then the nine handles in Figure 3.9 become:

⟨*Factor*→`ident` •⟩	⟨*Term*→*Factor* •⟩	⟨*Expr*→*Term* •⟩
⟨*Factor*→`num` •⟩	⟨*Term*→*Factor* •⟩	⟨*Factor*→`ident` •⟩
⟨*Term*→*Term* × *Factor* •⟩	⟨*Expr*→*Expr* - *Term* •⟩	⟨*Goal*→*Expr* •⟩

This notation shows that the second and fifth handles are identical, as are the first and sixth. It also creates a way to represent the potential of discovering a handle in the future.

Consider the parser's state in step six, with the input symbol `num` and the frontier *Expr* -. From the table, we know that the next reduction will be ⟨*Factor*→`num` •⟩, followed by further shifts and reductions to recognize the symbols × and `ident`. What, however, of the current frontier? The parser has recognized *Expr* -. It needs to recognize, eventually, a *Term* before it can reduce this part of the frontier. Using this stack-relative notation, we can represent the parser's state as *Expr*→*Expr* - • *Term*. The parser has already recognized an *Expr* and a -, with the - on top of the stack. If it reaches a state where it shifts a *Term* on top of *Expr*-, it will complete the handle *Expr*→*Expr* - *Term* •.

With a concrete notation for representing both handles and potential handles, we can ask the question, how many potential handles must the parser recognize? The right-hand side of each production can have a placeholder at its start, at its end, and between any two consecutive symbols. If the right-hand side has k symbols, it has $k + 1$ placeholder positions. The number of

potential handles for the grammar is simply the sum of the lengths of the right-hand sides of all the productions. The number of complete handles is simply the number of productions. These two facts lead to the critical insight behind LR(1) parsers.

A given grammar generates a finite set of handles (and potential handles) that the parser must recognize.

From Chapter 2, we have an appropriate tool to recognize finite collections of words — the DFA. The LR(1) parsers use a handle-recognizing DFA to efficiently find handles on the top of the parse stack. The table-construction algorithm builds a model of this DFA and encodes it into a pair of tables.

Careful examination of the parse in Figure 3.9 reveals one flaw in this reasoning. Consider the parser's action at step nine. The frontier is *Expr - Term*, suggesting a handle $\langle Expr \rightarrow Expr - Term \bullet \rangle$. However, the parser decides to extend the frontier by shifting × onto the stack, rather than reducing the frontier to *Expr*. Clearly, this is the correct move for the parser. No potential handle contains *Expr* followed by × .

To determine the correct action, the parser can recognize the distinct actions required by different right contexts—the symbols that come later in the input stream. At step nine, the set of potential handles is

$\langle Expr \rightarrow Expr - Term \bullet \rangle$ $\langle Term \rightarrow Term \bullet \times Factor \rangle$ $\langle Term \rightarrow Term \bullet \div Factor \rangle$

The next input symbol × clearly matches the second choice and rules out the third choice. The parser needs a basis for deciding between the first and second choices. This requires more context than the parser has in the frontier.

To choose between reducing *Expr - Term* to *Expr* and shifting × in an attempt to recognize *Term* × *Factor*, the parser must recognize which symbols can occur to the right of *Expr* and *Term* in valid parses. Looking at the grammar, + and - immediately follow *Expr* in productions one and two. From productions four and five, × and ÷ can follow *Term*. Thus, in step nine, the parser can choose the correct action by looking at the next symbol: On + or -, it should reduce; on × or ÷ , it should shift. Since the next symbol is × , the parser shifts.

The LR(1) parsers can recognize precisely those languages in which a one-symbol lookahead suffices to determine whether to shift or reduce. The LR(1) construction algorithm builds a handle-recognizing DFA; the parsing algorithm uses this DFA to recognize handles and potential handles on the parse stack. It uses a shift-reduce parsing framework to guide the application of the DFA. The framework can invoke the DFA recursively; to accomplish this, it

stores information about the DFA's internal state on the stack, interleaved with the grammar symbols that represent the upper frontier of the parse tree.

3.4.3 LR(1) Parsers

The LR(1) parsers, including the restricted forms known as SLR(1) and LALR(1) parsers, are the most widely used family of parsers. Table-driven LR(1) parsers are efficient. Tools that automate construction of the tables are widely available. The grammars that these tools accept allow most programming-language constructs to be expressed in a natural way. This section explains how LR(1) parsers work and shows how to construct the parse tables for one kind of LR(1) parser, namely, a canonical LR(1) parser.

Figure 3.12 shows the structure of a typical LR(1) parser-generator system. The compiler writer creates a grammar that describes the source language. The parser generator consumes that grammar and produces a pair of tables that drive the LR(1) parser. The tables encode all of the grammatical knowledge needed for parsing; in some sense, the parser generator precompiles into the tables all the knowledge required to identify handles, to decide when to shift, and to decide when to reduce and which rule to use in the reduction. In most such systems, the compiler writer can also provide a snippet of code for each production that will execute on a reduction. These ad hoc "actions" provide

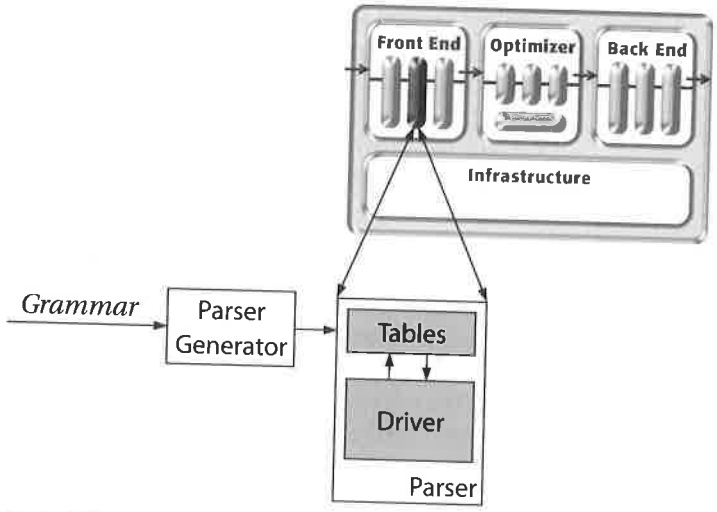

FIGURE 3.12 Structure of an LR(1) Parser Generator System