1. Separating the syntactic structure of a language into lexical and nonlexical parts provides a convenient way of modularizing the front end of a compiler into two manageable-sized components.

2. The lexical rules of a language are frequently quite simple, and to describe them we do not need a notation as powerful as grammars.

3. Regular expressions generally provide a more concise and easier-to-understand notation for tokens than grammars.

4. More efficient lexical analyzers can be constructed automatically from regular expressions than from arbitrary grammars.

There are no firm guidelines as to what to put into the lexical rules, as opposed to the syntactic rules. Regular expressions are most useful for describing the structure of constructs such as identifiers, constants, keywords, and white space. Grammars, on the other hand, are most useful for describing nested structures such as balanced parentheses, matching begin-end's, corresponding if-then-else's, and so on. These nested structures cannot be described by regular expressions.

4.3.2 Eliminating Ambiguity

Sometimes an ambiguous grammar can be rewritten to eliminate the ambiguity. As an example, we shall eliminate the ambiguity from the following "dangling-else" grammar:

$$\begin{array}{lcl} \mathit{stmt} & \rightarrow & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \\ & | & \textbf{if } \mathit{expr} \textbf{ then } \mathit{stmt} \textbf{ else } \mathit{stmt} \\ & | & \textbf{other} \end{array} \tag{4.14}$$

Here "**other**" stands for any other statement. According to this grammar, the compound conditional statement

$$\textbf{if } E_1 \textbf{ then } S_1 \textbf{ else if } E_2 \textbf{ then } S_2 \textbf{ else } S_3$$

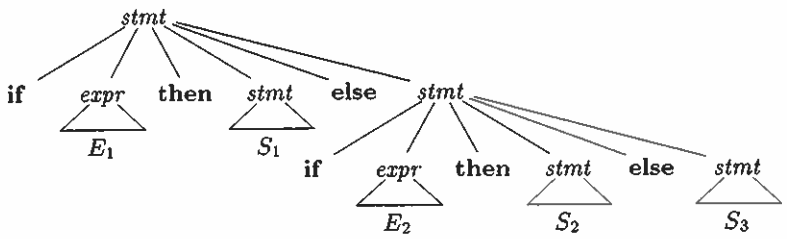

Figure 4.8: Parse tree for a conditional statement

has the parse tree shown in Fig. 4.8.[1] Grammar (4.14) is ambiguous since the string

$$\textbf{if } E_1 \textbf{ then if } E_2 \textbf{ then } S_1 \textbf{ else } S_2 \tag{4.15}$$

has the two parse trees shown in Fig. 4.9.

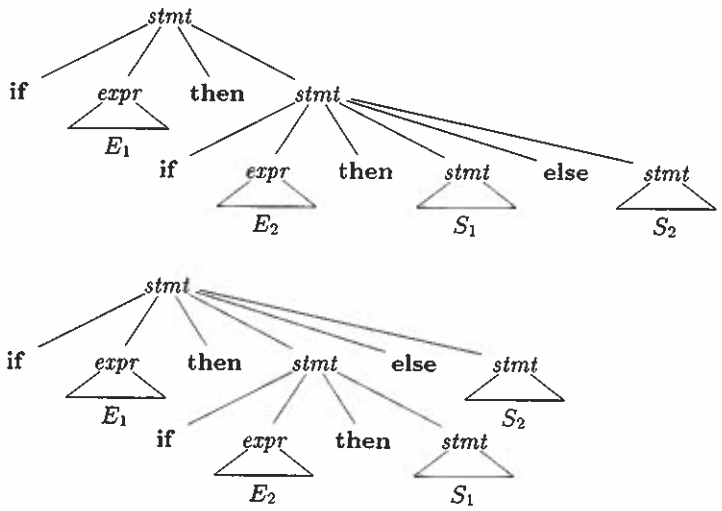

Figure 4.9: Two parse trees for an ambiguous sentence

In all programming languages with conditional statements of this form, the first parse tree is preferred. The general rule is, "Match each **else** with the closest unmatched **then**."[2] This disambiguating rule can theoretically be incorporated directly into a grammar, but in practice it is rarely built into the productions.

Example 4.16: We can rewrite the dangling-else grammar (4.14) as the following unambiguous grammar. The idea is that a statement appearing between a **then** and an **else** must be "matched"; that is, the interior statement must not end with an unmatched or open **then**. A matched statement is either an **if-then-else** statement containing no open statements or it is any other kind of unconditional statement. Thus, we may use the grammar in Fig. 4.10. This grammar generates the same strings as the dangling-else grammar (4.14), but it allows only one parsing for string (4.15); namely, the one that associates each **else** with the closest previous unmatched **then**. □

[1] The subscripts on E and S are just to distinguish different occurrences of the same nonterminal, and do not imply distinct nonterminals.

[2] We should note that C and its derivatives are included in this class. Even though the C family of languages do not use the keyword **then**, its role is played by the closing parenthesis for the condition that follows **if**.

$$
\begin{array}{rcl}
\textit{stmt} & \rightarrow & \textit{matched\_stmt} \\
 & | & \textit{open\_stmt} \\
\textit{matched\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{matched\_stmt} \\
 & | & \textbf{other} \\
\textit{open\_stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\
 & | & \textbf{if } \textit{expr} \textbf{ then } \textit{matched\_stmt} \textbf{ else } \textit{open\_stmt}
\end{array}
$$

Figure 4.10: Unambiguous grammar for if-then-else statements

4.3.3 Elimination of Left Recursion

A grammar is *left recursive* if it has a nonterminal A such that there is a derivation $A \stackrel{+}{\Rightarrow} A\alpha$ for some string α. Top-down parsing methods cannot handle left-recursive grammars, so a transformation is needed to eliminate left recursion. In Section 2.4.5, we discussed *immediate left recursion*, where there is a production of the form $A \rightarrow A\alpha$. Here, we study the general case. In Section 2.4.5, we showed how the left-recursive pair of productions $A \rightarrow A\alpha \mid \beta$ could be replaced by the non-left-recursive productions:

$$
\begin{array}{l}
A \rightarrow \beta A' \\
A' \rightarrow \alpha A' \mid \epsilon
\end{array}
$$

without changing the strings derivable from A. This rule by itself suffices for many grammars.

Example 4.17: The non-left-recursive expression grammar (4.2), repeated here,

$$
\begin{array}{l}
E \rightarrow T\ E' \\
E' \rightarrow +\ T\ E' \\
T \rightarrow F\ T' \\
T' \rightarrow *\ F\ T' \\
F \rightarrow (\ E\ ) \mid \textbf{id}
\end{array}
$$

is obtained by eliminating immediate left recursion from the expression grammar (4.1). The left-recursive pair of productions $E \rightarrow E + T \mid T$ are replaced by $E \rightarrow T\ E'$ and $E' \rightarrow +\ T\ E' \mid \epsilon$. The new productions for T and T' are obtained similarly by eliminating immediate left recursion. □

Immediate left recursion can be eliminated by the following technique, which works for any number of A-productions. First, group the productions as

$$
A \rightarrow A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_m \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n
$$

where no β_i begins with an A. Then, replace the A-productions by

$$A \to \beta_1 A' \mid \beta_2 A' \mid \cdots \mid \beta_n A'$$
$$A' \to \alpha_1 A' \mid \alpha_2 A' \mid \cdots \mid \alpha_m A' \mid \epsilon$$

The nonterminal A generates the same strings as before but is no longer left recursive. This procedure eliminates all left recursion from the A and A' productions (provided no α_i is ϵ), but it does not eliminate left recursion involving derivations of two or more steps. For example, consider the grammar

$$\begin{array}{l} S \to A\,a \mid b \\ A \to A\,c \mid S\,d \mid \epsilon \end{array} \tag{4.18}$$

The nonterminal S is left recursive because $S \Rightarrow Aa \Rightarrow Sda$, but it is not immediately left recursive.

Algorithm 4.19, below, systematically eliminates left recursion from a grammar. It is guaranteed to work if the grammar has no cycles (derivations of the form $A \stackrel{+}{\Rightarrow} A$) or ϵ-productions (productions of the form $A \to \epsilon$). Cycles can be eliminated systematically from a grammar, as can ϵ-productions (see Exercises 4.4.6 and 4.4.7).

Algorithm 4.19: Eliminating left recursion.

INPUT: Grammar G with no cycles or ϵ-productions.

OUTPUT: An equivalent grammar with no left recursion.

METHOD: Apply the algorithm in Fig. 4.11 to G. Note that the resulting non-left-recursive grammar may have ϵ-productions. □

```
1)  arrange the nonterminals in some order A1, A2, ..., An.
2)  for ( each i from 1 to n ) {
3)      for ( each j from 1 to i - 1 ) {
4)          replace each production of the form Ai → Aj γ by the
                productions Ai → δ1 γ | δ2 γ | ··· | δk γ, where
                Aj → δ1 | δ2 | ··· | δk are all current Aj-productions
5)      }
6)      eliminate the immediate left recursion among the Ai-productions
7)  }
```

Figure 4.11: Algorithm to eliminate left recursion from a grammar

The procedure in Fig. 4.11 works as follows. In the first iteration for $i = 1$, the outer for-loop of lines (2) through (7) eliminates any immediate left recursion among A_1-productions. Any remaining A_1 productions of the form $A_1 \to A_l\alpha$ must therefore have $l > 1$. After the $i-1$st iteration of the outer for-loop, all nonterminals A_k, where $k < i$, are "cleaned"; that is, any production $A_k \to A_l\alpha$, must have $l > k$. As a result, on the ith iteration, the inner loop

of lines (3) through (5) progressively raises the lower limit in any production $A_i \rightarrow A_m\alpha$, until we have $m \geq i$. Then, eliminating immediate left recursion for the A_i productions at line (6) forces m to be greater than i.

Example 4.20: Let us apply Algorithm 4.19 to the grammar (4.18). Technically, the algorithm is not guaranteed to work, because of the ϵ-production, but in this case, the production $A \rightarrow \epsilon$ turns out to be harmless.

We order the nonterminals S, A. There is no immediate left recursion among the S-productions, so nothing happens during the outer loop for $i = 1$. For $i = 2$, we substitute for S in $A \rightarrow S\,d$ to obtain the following A-productions.

$$A \rightarrow A\,c \mid A\,a\,d \mid b\,d \mid \epsilon$$

Eliminating the immediate left recursion among these A-productions yields the following grammar.

$$\begin{aligned} S &\rightarrow A\,a \mid b \\ A &\rightarrow b\,d\,A' \mid A' \\ A' &\rightarrow c\,A' \mid a\,d\,A' \mid \epsilon \end{aligned}$$

□

4.3.4 Left Factoring

Left factoring is a grammar transformation that is useful for producing a grammar suitable for predictive, or top-down, parsing. When the choice between two alternative A-productions is not clear, we may be able to rewrite the productions to defer the decision until enough of the input has been seen that we can make the right choice.

For example, if we have the two productions

$$\begin{aligned} \mathit{stmt} \quad &\rightarrow \quad \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ &\mid \quad \textbf{if}\ \mathit{expr}\ \textbf{then}\ \mathit{stmt} \end{aligned}$$

on seeing the input **if**, we cannot immediately tell which production to choose to expand *stmt*. In general, if $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2$ are two A-productions, and the input begins with a nonempty string derived from α, we do not know whether to expand A to $\alpha\beta_1$ or $\alpha\beta_2$. However, we may defer the decision by expanding A to $\alpha A'$. Then, after seeing the input derived from α, we expand A' to β_1 or to β_2. That is, left-factored, the original productions become

$$\begin{aligned} A &\rightarrow \alpha A' \\ A' &\rightarrow \beta_1 \mid \beta_2 \end{aligned}$$

Algorithm 4.21: Left factoring a grammar.

INPUT: Grammar G.

OUTPUT: An equivalent left-factored grammar.

METHOD: For each nonterminal A, find the longest prefix α common to two or more of its alternatives. If $\alpha \neq \epsilon$ — i.e., there is a nontrivial common prefix — replace all of the A-productions $A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \cdots \mid \alpha\beta_n \mid \gamma$, where γ represents all alternatives that do not begin with α, by

$$\begin{aligned} A &\rightarrow \alpha A' \mid \gamma \\ A' &\rightarrow \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n \end{aligned}$$

Here A' is a new nonterminal. Repeatedly apply this transformation until no two alternatives for a nonterminal have a common prefix. □

Example 4.22: The following grammar abstracts the "dangling-else" problem:

$$\begin{aligned} S &\rightarrow i\ E\ t\ S \mid i\ E\ t\ S\ e\ S \mid a \\ E &\rightarrow b \end{aligned} \tag{4.23}$$

Here, i, t, and e stand for **if**, **then**, and **else**; E and S stand for "conditional expression" and "statement." Left-factored, this grammar becomes:

$$\begin{aligned} S &\rightarrow i\ E\ t\ S\ S' \mid a \\ S' &\rightarrow e\ S \mid \epsilon \\ E &\rightarrow b \end{aligned} \tag{4.24}$$

Thus, we may expand S to $iEtSS'$ on input i, and wait until $iEtS$ has been seen to decide whether to expand S' to eS or to ϵ. Of course, these grammars are both ambiguous, and on input e, it will not be clear which alternative for S' should be chosen. Example 4.33 discusses a way out of this dilemma. □

4.3.5 Non-Context-Free Language Constructs

A few syntactic constructs found in typical programming languages cannot be specified using grammars alone. Here, we consider two of these constructs, using simple abstract languages to illustrate the difficulties.

Example 4.25: The language in this example abstracts the problem of checking that identifiers are declared before they are used in a program. The language consists of strings of the form wcw, where the first w represents the declaration of an identifier w, c represents an intervening program fragment, and the second w represents the use of the identifier.

The abstract language is $L_1 = \{wcw \mid w \text{ is in } (\mathbf{a}|\mathbf{b})^*\}$. L_1 consists of all words composed of a repeated string of a's and b's separated by c, such as $aabcaab$. While it is beyond the scope of this book to prove it, the non-context-freedom of L_1 directly implies the non-context-freedom of programming languages like C and Java, which require declaration of identifiers before their use and which allow identifiers of arbitrary length.

For this reason, a grammar for C or Java does not distinguish among identifiers that are different character strings. Instead, all identifiers are represented

by a token such as **id** in the grammar. In a compiler for such a language, the semantic-analysis phase checks that identifiers are declared before they are used. □

Example 4.26: The non-context-free language in this example abstracts the problem of checking that the number of formal parameters in the declaration of a function agrees with the number of actual parameters in a use of the function. The language consists of strings of the form $a^n b^m c^n d^m$. (Recall a^n means a written n times.) Here a^n and b^m could represent the formal-parameter lists of two functions declared to have n and m arguments, respectively, while c^n and d^m represent the actual-parameter lists in calls to these two functions.

The abstract language is $L_2 = \{a^n b^m c^n d^m \mid n \geq 1 \text{ and } m \geq 1\}$. That is, L_2 consists of strings in the language generated by the regular expression $\mathbf{a^*b^*c^*d^*}$ such that the number of a's and c's are equal and the number of b's and d's are equal. This language is not context free.

Again, the typical syntax of function declarations and uses does not concern itself with counting the number of parameters. For example, a function call in C-like language might be specified by

$$\begin{array}{rcl} \textit{stmt} & \rightarrow & \textbf{id (} \textit{expr\_list} \textbf{)} \\ \textit{expr\_list} & \rightarrow & \textit{expr\_list , expr} \\ & | & \textit{expr} \end{array}$$

with suitable productions for *expr*. Checking that the number of parameters in a call is correct is usually done during the semantic-analysis phase. □

4.3.6 Exercises for Section 4.3

Exercise 4.3.1: The following is a grammar for regular expressions over symbols a and b only, using + in place of | for union, to avoid conflict with the use of vertical bar as a metasymbol in grammars:

$$\begin{array}{lcl} \textit{rexpr} & \rightarrow & \textit{rexpr} + \textit{rterm} \mid \textit{rterm} \\ \textit{rterm} & \rightarrow & \textit{rterm rfactor} \mid \textit{rfactor} \\ \textit{rfactor} & \rightarrow & \textit{rfactor} * \mid \textit{rprimary} \\ \textit{rprimary} & \rightarrow & \mathbf{a} \mid \mathbf{b} \end{array}$$

a) Left factor this grammar.

b) Does left factoring make the grammar suitable for top-down parsing?

c) In addition to left factoring, eliminate left recursion from the original grammar.

d) Is the resulting grammar suitable for top-down parsing?

Exercise 4.3.2: Repeat Exercise 4.3.1 on the following grammars:

a) The grammar of Exercise 4.2.1.

b) The grammar of Exercise 4.2.2(a).

c) The grammar of Exercise 4.2.2(c).

d) The grammar of Exercise 4.2.2(e).

e) The grammar of Exercise 4.2.2(g).

! **Exercise 4.3.3:** The following grammar is proposed to remove the "dangling-else ambiguity" discussed in Section 4.3.2:

$$\begin{array}{lcl} \textit{stmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{stmt} \\ & | & \textit{matchedStmt} \\ \textit{matchedStmt} & \rightarrow & \textbf{if } \textit{expr} \textbf{ then } \textit{matchedStmt} \textbf{ else } \textit{stmt} \\ & | & \textbf{other} \end{array}$$

Show that this grammar is still ambiguous.

4.4 Top-Down Parsing

Top-down parsing can be viewed as the problem of constructing a parse tree for the input string, starting from the root and creating the nodes of the parse tree in preorder (depth-first, as discussed in Section 2.3.4). Equivalently, top-down parsing can be viewed as finding a leftmost derivation for an input string.

Example 4.27: The sequence of parse trees in Fig. 4.12 for the input **id+id*id** is a top-down parse according to grammar (4.2), repeated here:

$$\begin{array}{lcl} E & \rightarrow & T\ E' \\ E' & \rightarrow & +\ T\ E' \mid \epsilon \\ T & \rightarrow & F\ T' \\ T' & \rightarrow & *\ F\ T' \mid \epsilon \\ F & \rightarrow & (\ E\ ) \mid \textbf{id} \end{array} \tag{4.28}$$

This sequence of trees corresponds to a leftmost derivation of the input. □

At each step of a top-down parse, the key problem is that of determining the production to be applied for a nonterminal, say A. Once an A-production is chosen, the rest of the parsing process consists of "matching" the terminal symbols in the production body with the input string.

The section begins with a general form of top-down parsing, called recursive-descent parsing, which may require backtracking to find the correct A-production to be applied. Section 2.4.2 introduced predictive parsing, a special case of recursive-descent parsing, where no backtracking is required. Predictive parsing chooses the correct A-production by looking ahead at the input a fixed number of symbols, typically we may look only at one (that is, the next input symbol).

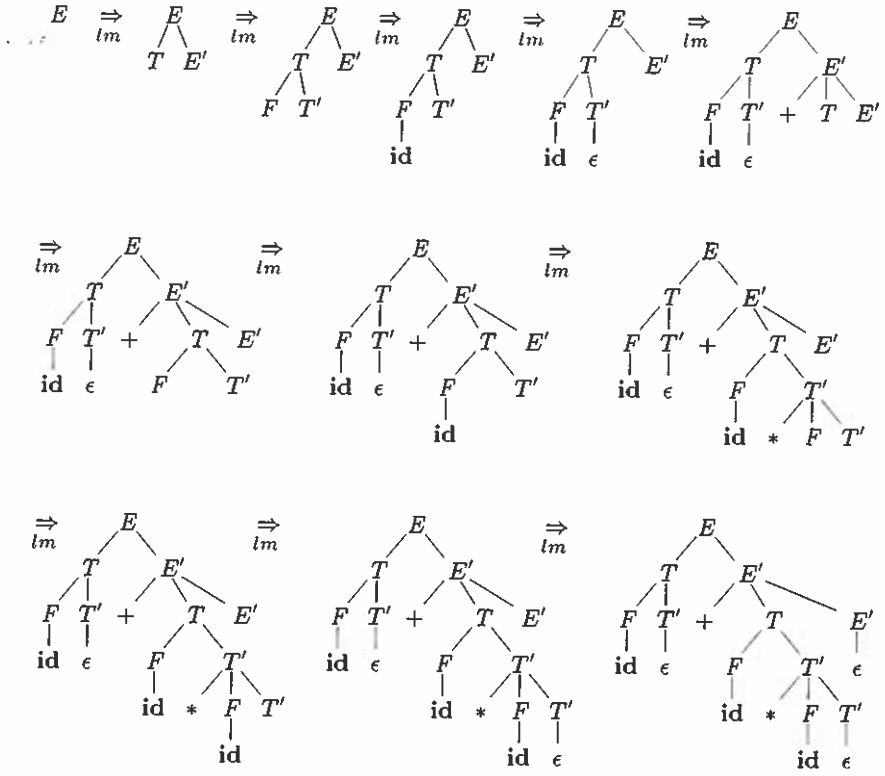

Figure 4.12: Top-down parse for **id** + **id** * **id**

For example, consider the top-down parse in Fig. 4.12, which constructs a tree with two nodes labeled E'. At the first E' node (in preorder), the production $E' \rightarrow +TE'$ is chosen; at the second E' node, the production $E' \rightarrow \epsilon$ is chosen. A predictive parser can choose between E'-productions by looking at the next input symbol.

The class of grammars for which we can construct predictive parsers looking k symbols ahead in the input is sometimes called the $LL(k)$ class. We discuss the LL(1) class in Section 4.4.3, but introduce certain computations, called FIRST and FOLLOW, in a preliminary Section 4.4.2. From the FIRST and FOLLOW sets for a grammar, we shall construct "predictive parsing tables," which make explicit the choice of production during top-down parsing. These sets are also useful during bottom-up parsing,

In Section 4.4.4 we give a nonrecursive parsing algorithm that maintains a stack explicitly, rather than implicitly via recursive calls. Finally, in Section 4.4.5 we discuss error recovery during top-down parsing.

4.4.1 Recursive-Descent Parsing

```
    void A() {
1)      Choose an A-production, A → X1 X2 ··· Xk;
2)      for ( i = 1 to k ) {
3)          if ( Xi is a nonterminal )
4)              call procedure Xi();
5)          else if ( Xi equals the current input symbol a )
6)              advance the input to the next symbol;
7)          else /* an error has occurred */;
        }
    }
```

Figure 4.13: A typical procedure for a nonterminal in a top-down parser

A recursive-descent parsing program consists of a set of procedures, one for each nonterminal. Execution begins with the procedure for the start symbol, which halts and announces success if its procedure body scans the entire input string. Pseudocode for a typical nonterminal appears in Fig. 4.13. Note that this pseudocode is nondeterministic, since it begins by choosing the A-production to apply in a manner that is not specified.

General recursive-descent may require backtracking; that is, it may require repeated scans over the input. However, backtracking is rarely needed to parse programming language constructs, so backtracking parsers are not seen frequently. Even for situations like natural language parsing, backtracking is not very efficient, and tabular methods such as the dynamic programming algorithm of Exercise 4.4.9 or the method of Earley (see the bibliographic notes) are preferred.

To allow backtracking, the code of Fig. 4.13 needs to be modified. First, we cannot choose a unique A-production at line (1), so we must try each of several productions in some order. Then, failure at line (7) is not ultimate failure, but suggests only that we need to return to line (1) and try another A-production. Only if there are no more A-productions to try do we declare that an input error has been found. In order to try another A-production, we need to be able to reset the input pointer to where it was when we first reached line (1). Thus, a local variable is needed to store this input pointer for future use.

Example 4.29: Consider the grammar

$$\begin{array}{lcl} S & \rightarrow & c\,A\,d \\ A & \rightarrow & a\,b \mid a \end{array}$$

To construct a parse tree top-down for the input string $w = cad$, begin with a tree consisting of a single node labeled S, and the input pointer pointing to c, the first symbol of w. S has only one production, so we use it to expand S and

obtain the tree of Fig. 4.14(a). The leftmost leaf, labeled c, matches the first symbol of input w, so we advance the input pointer to a, the second symbol of w, and consider the next leaf, labeled A.

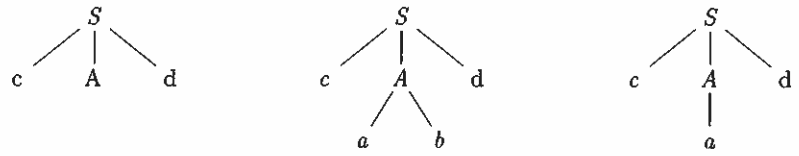

Figure 4.14: Steps in a top-down parse

Now, we expand A using the first alternative $A \rightarrow a\ b$ to obtain the tree of Fig. 4.14(b). We have a match for the second input symbol, a, so we advance the input pointer to d, the third input symbol, and compare d against the next leaf, labeled b. Since b does not match d, we report failure and go back to A to see whether there is another alternative for A that has not been tried, but that might produce a match.

In going back to A, we must reset the input pointer to position 2, the position it had when we first came to A, which means that the procedure for A must store the input pointer in a local variable.

The second alternative for A produces the tree of Fig. 4.14(c). The leaf a matches the second symbol of w and the leaf d matches the third symbol. Since we have produced a parse tree for w, we halt and announce successful completion of parsing. □

A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop. That is, when we try to expand a nonterminal A, we may eventually find ourselves again trying to expand A without having consumed any input.

4.4.2 FIRST and FOLLOW

The construction of both top-down and bottom-up parsers is aided by two functions, FIRST and FOLLOW, associated with a grammar G. During top-down parsing, FIRST and FOLLOW allow us to choose which production to apply, based on the next input symbol. During panic-mode error recovery, sets of tokens produced by FOLLOW can be used as synchronizing tokens.

Define $FIRST(\alpha)$, where α is any string of grammar symbols, to be the set of terminals that begin strings derived from α. If $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α). For example, in Fig. 4.15, $A \stackrel{*}{\Rightarrow} c\gamma$, so c is in FIRST(A).

For a preview of how FIRST can be used during predictive parsing, consider two A-productions $A \rightarrow \alpha \mid \beta$, where FIRST($\alpha$) and FIRST($\beta$) are disjoint sets. We can then choose between these A-productions by looking at the next input

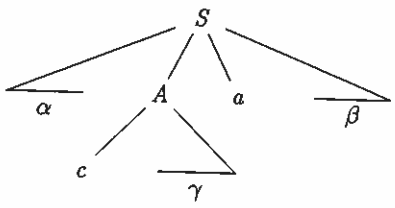

Figure 4.15: Terminal c is in FIRST(A) and a is in FOLLOW(A)

symbol a, since a can be in at most one of FIRST(α) and FIRST(β), not both. For instance, if a is in FIRST(β) choose the production $A \to \beta$. This idea will be explored when LL(1) grammars are defined in Section 4.4.3.

Define *FOLLOW*(A), for nonterminal A, to be the set of terminals a that can appear immediately to the right of A in some sentential form; that is, the set of terminals a such that there exists a derivation of the form $S \stackrel{*}{\Rightarrow} \alpha A a \beta$, for some α and β, as in Fig. 4.15. Note that there may have been symbols between A and a, at some time during the derivation, but if so, they derived ϵ and disappeared. In addition, if A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A); recall that \$ is a special "endmarker" symbol that is assumed not to be a symbol of any grammar.

To compute FIRST(X) for all grammar symbols X, apply the following rules until no more terminals or ϵ can be added to any FIRST set.

1. If X is a terminal, then FIRST(X) = $\{X\}$.

2. If X is a nonterminal and $X \to Y_1 Y_2 \cdots Y_k$ is a production for some $k \geq 1$, then place a in FIRST(X) if for some i, a is in FIRST(Y_i), and ϵ is in all of FIRST(Y_1), ..., FIRST(Y_{i-1}); that is, $Y_1 \cdots Y_{i-1} \stackrel{*}{\Rightarrow} \epsilon$. If ϵ is in FIRST(Y_j) for all $j = 1, 2, \ldots, k$, then add ϵ to FIRST(X). For example, everything in FIRST(Y_1) is surely in FIRST(X). If Y_1 does not derive ϵ, then we add nothing more to FIRST(X), but if $Y_1 \stackrel{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$), and so on.

3. If $X \to \epsilon$ is a production, then add ϵ to FIRST(X).

Now, we can compute FIRST for any string $X_1 X_2 \cdots X_n$ as follows. Add to FIRST($X_1 X_2 \cdots X_n$) all non-ϵ symbols of FIRST(X_1). Also add the non-ϵ symbols of FIRST(X_2), if ϵ is in FIRST(X_1); the non-ϵ symbols of FIRST(X_3), if ϵ is in FIRST(X_1) and FIRST(X_2); and so on. Finally, add ϵ to FIRST($X_1 X_2 \cdots X_n$) if, for all i, ϵ is in FIRST(X_i).

To compute FOLLOW(A) for all nonterminals A, apply the following rules until nothing can be added to any FOLLOW set.

1. Place \$ in FOLLOW(S), where S is the start symbol, and \$ is the input right endmarker.

2. If there is a production $A \rightarrow \alpha B \beta$, then everything in FIRST(β) except ϵ is in FOLLOW(B).

3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$, where FIRST($\beta$) contains ϵ, then everything in FOLLOW(A) is in FOLLOW(B).

Example 4.30: Consider again the non-left-recursive grammar (4.28). Then:

1. FIRST(F) = FIRST(T) = FIRST(E) = {(, **id**}. To see why, note that the two productions for F have bodies that start with these two terminal symbols, **id** and the left parenthesis. T has only one production, and its body starts with F. Since F does not derive ϵ, FIRST(T) must be the same as FIRST(F). The same argument covers FIRST(E).

2. FIRST(E') = {+, ϵ}. The reason is that one of the two productions for E' has a body that begins with terminal +, and the other's body is ϵ. Whenever a nonterminal derives ϵ, we place ϵ in FIRST for that nonterminal.

3. FIRST(T') = {*, ϵ}. The reasoning is analogous to that for FIRST(E').

4. FOLLOW(E) = FOLLOW(E') = {), \$}. Since E is the start symbol, FOLLOW(E) must contain \$. The production body (E) explains why the right parenthesis is in FOLLOW(E). For E', note that this nonterminal appears only at the ends of bodies of E-productions. Thus, FOLLOW(E') must be the same as FOLLOW(E).

5. FOLLOW(T) = FOLLOW(T') = {+,), \$}. Notice that T appears in bodies only followed by E'. Thus, everything except ϵ that is in FIRST(E') must be in FOLLOW(T); that explains the symbol +. However, since FIRST(E') contains ϵ (i.e., $E' \stackrel{*}{\Rightarrow} \epsilon$), and E' is the entire string following T in the bodies of the E-productions, everything in FOLLOW(E) must also be in FOLLOW(T). That explains the symbols \$ and the right parenthesis. As for T', since it appears only at the ends of the T-productions, it must be that FOLLOW(T') = FOLLOW(T).

6. FOLLOW(F) = {+, *,), \$}. The reasoning is analogous to that for T in point (5).

□

4.4.3 LL(1) Grammars

Predictive parsers, that is, recursive-descent parsers needing no backtracking, can be constructed for a class of grammars called LL(1). The first "L" in LL(1) stands for scanning the input from left to right, the second "L" for producing a leftmost derivation, and the "1" for using one input symbol of lookahead at each step to make parsing action decisions.

Transition Diagrams for Predictive Parsers

Transition diagrams are useful for visualizing predictive parsers. For example, the transition diagrams for nonterminals E and E' of grammar (4.28) appear in Fig. 4.16(a). To construct the transition diagram from a grammar, first eliminate left recursion and then left factor the grammar. Then, for each nonterminal A,

1. Create an initial and final (return) state.
2. For each production $A \rightarrow X_1 X_2 \cdots X_k$, create a path from the initial to the final state, with edges labeled $X_1, X_2, \ldots, X_k$. If $A \rightarrow \epsilon$, the path is an edge labeled ϵ.

Transition diagrams for predictive parsers differ from those for lexical analyzers. Parsers have one diagram for each nonterminal. The labels of edges can be tokens or nonterminals. A transition on a token (terminal) means that we take that transition if that token is the next input symbol. A transition on a nonterminal A is a call of the procedure for A.

With an LL(1) grammar, the ambiguity of whether or not to take an ϵ-edge can be resolved by making ϵ-transitions the default choice.

Transition diagrams can be simplified, provided the sequence of grammar symbols along paths is preserved. We may also substitute the diagram for a nonterminal A in place of an edge labeled A. The diagrams in Fig. 4.16(a) and (b) are equivalent: if we trace paths from E to an accepting state and substitute for E', then, in both sets of diagrams, the grammar symbols along the paths make up strings of the form $T + T + \cdots + T$. The diagram in (b) can be obtained from (a) by transformations akin to those in Section 2.5.4, where we used tail-recursion removal and substitution of procedure bodies to optimize the procedure for a nonterminal.

The class of LL(1) grammars is rich enough to cover most programming constructs, although care is needed in writing a suitable grammar for the source language. For example, no left-recursive or ambiguous grammar can be LL(1).

A grammar G is LL(1) if and only if whenever $A \rightarrow \alpha \mid \beta$ are two distinct productions of G, the following conditions hold:

1. For no terminal a do both α and β derive strings beginning with a.
2. At most one of α and β can derive the empty string.
3. If $\beta \stackrel{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A). Likewise, if $\alpha \stackrel{*}{\Rightarrow} \epsilon$, then β does not derive any string beginning with a terminal in FOLLOW(A).

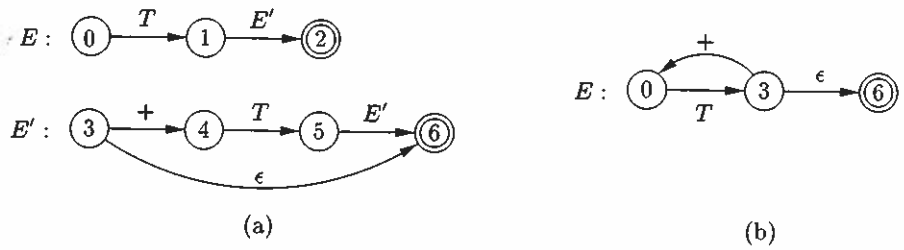

Figure 4.16: Transition diagrams for nonterminals E and E' of grammar 4.28

The first two conditions are equivalent to the statement that FIRST(α) and FIRST(β) are disjoint sets. The third condition is equivalent to stating that if ϵ is in FIRST(β), then FIRST(α) and FOLLOW(A) are disjoint sets, and likewise if ϵ is in FIRST(α).

Predictive parsers can be constructed for LL(1) grammars since the proper production to apply for a nonterminal can be selected by looking only at the current input symbol. Flow-of-control constructs, with their distinguishing keywords, generally satisfy the LL(1) constraints. For instance, if we have the productions

$$\begin{array}{lcl} \mathit{stmt} & \rightarrow & \textbf{if } (\ \mathit{expr}\ )\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\ & | & \textbf{while } (\ \mathit{expr}\ )\ \mathit{stmt} \\ & | & \{\ \mathit{stmt\_list}\ \} \end{array}$$

then the keywords **if**, **while**, and the symbol { tell us which alternative is the only one that could possibly succeed if we are to find a statement.

The next algorithm collects the information from FIRST and FOLLOW sets into a predictive parsing table $M[A, a]$, a two-dimensional array, where A is a nonterminal, and a is a terminal or the symbol \$, the input endmarker. The algorithm is based on the following idea: the production $A \rightarrow \alpha$ is chosen if the next input symbol a is in FIRST(α). The only complication occurs when $\alpha = \epsilon$ or, more generally, $\alpha \stackrel{*}{\Rightarrow} \epsilon$. In this case, we should again choose $A \rightarrow \alpha$, if the current input symbol is in FOLLOW(A), or if the \$ on the input has been reached and \$ is in FOLLOW(A).

Algorithm 4.31: Construction of a predictive parsing table.

INPUT: Grammar G.

OUTPUT: Parsing table M.

METHOD: For each production $A \rightarrow \alpha$ of the grammar, do the following:

1. For each terminal a in FIRST(A), add $A \rightarrow \alpha$ to $M[A, a]$.
2. If ϵ is in FIRST(α), then for each terminal b in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, b]$. If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$ as well.

If, after performing the above, there is no production at all in $M[A, a]$, then set $M[A, a]$ to **error** (which we normally represent by an empty entry in the table). □

Example 4.32: For the expression grammar (4.28), Algorithm 4.31 produces the parsing table in Fig. 4.17. Blanks are error entries; nonblanks indicate a production with which to expand a nonterminal.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$		
E'		$E' \to +TE'$			$E' \to \epsilon$	$E' \to \epsilon$
T	$T \to FT'$			$T \to FT'$		
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ **id**			$F \to (E)$		

Figure 4.17: Parsing table M for Example 4.32

Consider production $E \to TE'$. Since

$$\text{FIRST}(TE') = \text{FIRST}(T) = \{(, \textbf{id}\}$$

this production is added to $M[E, (]$ and $M[E, \textbf{id}]$. Production $E' \to +TE'$ is added to $M[E', +]$ since $\text{FIRST}(+TE') = \{+\}$. Since $\text{FOLLOW}(E') = \{), \$\}$, production $E' \to \epsilon$ is added to $M[E',)]$ and $M[E', \$]$. □

Algorithm 4.31 can be applied to any grammar G to produce a parsing table M. For every LL(1) grammar, each parsing-table entry uniquely identifies a production or signals an error. For some grammars, however, M may have some entries that are multiply defined. For example, if G is left-recursive or ambiguous, then M will have at least one multiply defined entry. Although left-recursion elimination and left factoring are easy to do, there are some grammars for which no amount of alteration will produce an LL(1) grammar.

The language in the following example has no LL(1) grammar at all.

Example 4.33: The following grammar, which abstracts the dangling-else problem, is repeated here from Example 4.22:

$$\begin{aligned} S &\to iEtSS' \mid a \\ S' &\to eS \mid \epsilon \\ E &\to b \end{aligned}$$

The parsing table for this grammar appears in Fig. 4.18. The entry for $M[S', e]$ contains both $S' \to eS$ and $S' \to \epsilon$.

The grammar is ambiguous and the ambiguity is manifested by a choice in what production to use when an e (**else**) is seen. We can resolve this ambiguity

NON-TERMINAL	INPUT SYMBOL					
	a	b	e	i	t	\$
S	$S \to a$			$S \to iEtSS'$		
S'			$S' \to \epsilon$ $S' \to eS$			$S' \to \epsilon$
E		$E \to b$				

Figure 4.18: Parsing table M for Example 4.33

by choosing $S' \to eS$. This choice corresponds to associating an **else** with the closest previous **then**. Note that the choice $S' \to \epsilon$ would prevent e from ever being put on the stack or removed from the input, and is surely wrong. □

4.4.4 Nonrecursive Predictive Parsing

A nonrecursive predictive parser can be built by maintaining a stack explicitly, rather than implicitly via recursive calls. The parser mimics a leftmost derivation. If w is the input that has been matched so far, then the stack holds a sequence of grammar symbols α such that

$$S \underset{lm}{\overset{*}{\Rightarrow}} w\alpha$$

The table-driven parser in Fig. 4.19 has an input buffer, a stack containing a sequence of grammar symbols, a parsing table constructed by Algorithm 4.31, and an output stream. The input buffer contains the string to be parsed, followed by the endmarker \$. We reuse the symbol \$ to mark the bottom of the stack, which initially contains the start symbol of the grammar on top of \$.

The parser is controlled by a program that considers X, the symbol on top of the stack, and a, the current input symbol. If X is a nonterminal, the parser chooses an X-production by consulting entry $M[X, a]$ of the parsing table M. (Additional code could be executed here, for example, code to construct a node in a parse tree.) Otherwise, it checks for a match between the terminal X and current input symbol a.

The behavior of the parser can be described in terms of its *configurations*, which give the stack contents and the remaining input. The next algorithm describes how configurations are manipulated.

Algorithm 4.34: Table-driven predictive parsing.

INPUT: A string w and a parsing table M for grammar G.

OUTPUT: If w is in $L(G)$, a leftmost derivation of w; otherwise, an error indication.

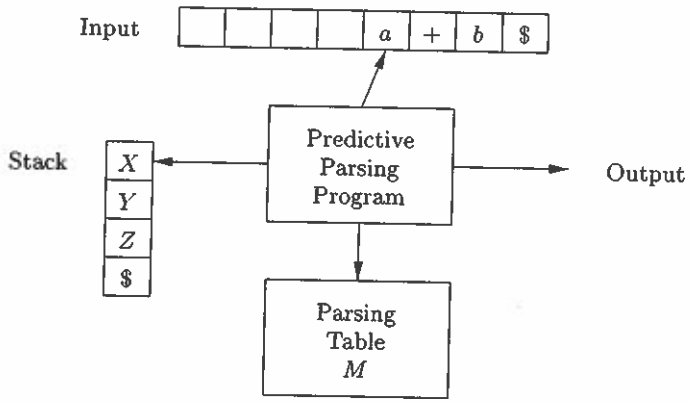

Figure 4.19: Model of a table-driven predictive parser

METHOD: Initially, the parser is in a configuration with $w\$$ in the input buffer and the start symbol S of G on top of the stack, above \$. The program in Fig. 4.20 uses the predictive parsing table M to produce a predictive parse for the input. □

```
set ip to point to the first symbol of w;
set X to the top stack symbol;
while ( X ≠ $ ) { /* stack is not empty */
        if ( X is a ) pop the stack and advance ip;
        else if ( X is a terminal ) error();
        else if ( M[X, a] is an error entry ) error();
        else if ( M[X, a] = X → Y1 Y2 ··· Yk ) {
                output the production X → Y1 Y2 ··· Yk;
                pop the stack;
                push Yk, Yk-1, ..., Y1 onto the stack, with Y1 on top;
        }
        set X to the top stack symbol;
}
```

Figure 4.20: Predictive parsing algorithm

Example 4.35 : Consider grammar (4.28); we have already seen its the parsing table in Fig. 4.17. On input **id** + **id** * **id**, the nonrecursive predictive parser of Algorithm 4.34 makes the sequence of moves in Fig. 4.21. These moves correspond to a leftmost derivation (see Fig. 4.12 for the full derivation):

$$E \underset{lm}{\Rightarrow} TE' \underset{lm}{\Rightarrow} FT'E' \underset{lm}{\Rightarrow} \mathbf{id}\,T'E' \underset{lm}{\Rightarrow} \mathbf{id}\,E' \underset{lm}{\Rightarrow} \mathbf{id} + TE' \underset{lm}{\Rightarrow} \cdots$$

MATCHED	STACK	INPUT	ACTION
	$E\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	
	$TE'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $E \to TE'$
	$FT'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
	$\mathbf{id}\ T'E'\$$	$\mathbf{id} + \mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id}$	$T'E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id}$	$E'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $T' \to \epsilon$
$\mathbf{id}$	$+\ TE'\$$	$+\ \mathbf{id} * \mathbf{id}\$$	output $E' \to +\ TE'$
$\mathbf{id} +$	$TE'\$$	$\mathbf{id} * \mathbf{id}\$$	match $+$
$\mathbf{id} +$	$FT'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $T \to FT'$
$\mathbf{id} +$	$\mathbf{id}\ T'E'\$$	$\mathbf{id} * \mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$T'E'\$$	$* \mathbf{id}\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id}$	$* FT'E'\$$	$* \mathbf{id}\$$	output $T' \to * FT'$
$\mathbf{id} + \mathbf{id} *$	$FT'E'\$$	$\mathbf{id}\$$	match $*$
$\mathbf{id} + \mathbf{id} *$	$\mathbf{id}\ T'E'\$$	$\mathbf{id}\$$	output $F \to \mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$T'E'\$$	$\$$	match $\mathbf{id}$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$E'\$$	$\$$	output $T' \to \epsilon$
$\mathbf{id} + \mathbf{id} * \mathbf{id}$	$\$$	$\$$	output $E' \to \epsilon$

Figure 4.21: Moves made by a predictive parser on input $\mathbf{id} + \mathbf{id} * \mathbf{id}$

Note that the sentential forms in this derivation correspond to the input that has already been matched (in column MATCHED) followed by the stack contents. The matched input is shown only to highlight the correspondence. For the same reason, the top of the stack is to the left; when we consider bottom-up parsing, it will be more natural to show the top of the stack to the right. The input pointer points to the leftmost symbol of the string in the INPUT column. □

4.4.5 Error Recovery in Predictive Parsing

This discussion of error recovery refers to the stack of a table-driven predictive parser, since it makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input; the techniques can also be used with recursive-descent parsing.

An error is detected during predictive parsing when the terminal on top of the stack does not match the next input symbol or when nonterminal A is on top of the stack, a is the next input symbol, and $M[A, a]$ is **error** (i.e., the parsing-table entry is empty).

Panic Mode

Panic-mode error recovery is based on the idea of skipping symbols on the the input until a token in a selected set of synchronizing tokens appears. Its

effectiveness depends on the choice of synchronizing set. The sets should be chosen so that the parser recovers quickly from errors that are likely to occur in practice. Some heuristics are as follows:

1. As a starting point, place all symbols in FOLLOW(A) into the synchronizing set for nonterminal A. If we skip tokens until an element of FOLLOW(A) is seen and pop A from the stack, it is likely that parsing can continue.

2. It is not enough to use FOLLOW(A) as the synchronizing set for A. For example, if semicolons terminate statements, as in C, then keywords that begin statements may not appear in the FOLLOW set of the nonterminal representing expressions. A missing semicolon after an assignment may therefore result in the keyword beginning the next statement being skipped. Often, there is a hierarchical structure on constructs in a language; for example, expressions appear within statements, which appear within blocks, and so on. We can add to the synchronizing set of a lower-level construct the symbols that begin higher-level constructs. For example, we might add keywords that begin statements to the synchronizing sets for the nonterminals generating expressions.

3. If we add symbols in FIRST(A) to the synchronizing set for nonterminal A, then it may be possible to resume parsing according to A if a symbol in FIRST(A) appears in the input.

4. If a nonterminal can generate the empty string, then the production deriving ϵ can be used as a default. Doing so may postpone some error detection, but cannot cause an error to be missed. This approach reduces the number of nonterminals that have to be considered during error recovery.

5. If a terminal on top of the stack cannot be matched, a simple idea is to pop the terminal, issue a message saying that the terminal was inserted, and continue parsing. In effect, this approach takes the synchronizing set of a token to consist of all other tokens.

Example 4.36: Using FIRST and FOLLOW symbols as synchronizing tokens works reasonably well when expressions are parsed according to the usual grammar (4.28). The parsing table for this grammar in Fig. 4.17 is repeated in Fig. 4.22, with "synch" indicating synchronizing tokens obtained from the FOLLOW set of the nonterminal in question. The FOLLOW sets for the nonterminals are obtained from Example 4.30.

The table in Fig. 4.22 is to be used as follows. If the parser looks up entry $M[A, a]$ and finds that it is blank, then the input symbol a is skipped. If the entry is "synch," then the nonterminal on top of the stack is popped in an attempt to resume parsing. If a token on top of the stack does not match the input symbol, then we pop the token from the stack, as mentioned above.

NON-TERMINAL	INPUT SYMBOL					
	id	+	*	(	)	$
E	$E \to TE'$			$E \to TE'$	synch	synch
E'		$E \to +TE'$			$E \to \epsilon$	$E \to \epsilon$
T	$T \to FT'$	synch		$T \to FT'$	synch	synch
T'		$T' \to \epsilon$	$T' \to *FT'$		$T' \to \epsilon$	$T' \to \epsilon$
F	$F \to$ id	synch	synch	$F \to (E)$	synch	synch

Figure 4.22: Synchronizing tokens added to the parsing table of Fig. 4.17

On the erroneous input) **id** $*$ $+$ **id**, the parser and error recovery mechanism of Fig. 4.22 behave as in Fig. 4.23. □

STACK	INPUT	REMARK
E \$	) **id** $*$ $+$ **id** \$	error, skip)
E \$	**id** $*$ $+$ **id** \$	**id** is in FIRST(E)
TE' \$	**id** $*$ $+$ **id** \$	
$FT'E'$ \$	**id** $*$ $+$ **id** \$	
id $T'E'$\$	**id** $*$ $+$ **id** \$	
$T'E'$ \$	$*$ $+$ **id** \$	
$*FT'E'$ \$	$*$ $+$ **id** \$	
$FT'E'$ \$	$+$ **id** \$	error, $M[F, +]$ = synch
$T'E'$ \$	$+$ **id** \$	F has been popped
E' \$	$+$ **id** \$	
$+TE'$ \$	$+$ **id** \$	
TE' \$	**id** \$	
$FT'E'$ \$	**id** \$	
id $T'E'$ \$	**id** \$	
$T'E'$ \$	\$	
E' \$	\$	
\$	\$	

Figure 4.23: Parsing and error recovery moves made by a predictive parser

The above discussion of panic-mode recovery does not address the important issue of error messages. The compiler designer must supply informative error messages that not only describe the error, they must draw attention to where the error was discovered.