NONTERMINAL	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			$F \to (E)$		

Fig. 4.15. Parsing table M for grammar (4.11).

already been scanned, followed by the grammar symbols on the stack (from the top to bottom), make up the left-sentential forms in the derivation. □

STACK	INPUT	OUTPUT
$E	id + id * id$	
$E'T	id + id * id$	$E \to TE'$
$E'T'F	id + id * id$	$T \to FT'$
$E'T'id	id + id * id$	$F \to$ id
$E'T'	+ id * id$	
$E'	+ id * id$	$T' \to \epsilon$
$E'T +	+ id * id$	$E' \to +TE'$
$E'T	id * id$	
$E'T'F	id * id$	$T \to FT'$
$E'T'id	id * id$	$F \to$ id
$E'T'	* id$	
$E'T'F*	* id$	$T' \to *FT'$
$E'T'F	id$	
$E'T'id	id$	$F \to$ id
$E'T'	$	
$E'	$	$T' \to \epsilon$
$	$	$E' \to \epsilon$

Fig. 4.16. Moves made by predictive parser on input id+id*id.

FIRST and FOLLOW

The construction of a predictive parser is aided by two functions associated with a grammar G. These functions, FIRST and FOLLOW, allow us to fill in the entries of a predictive parsing table for G, whenever possible. Sets of tokens yielded by the FOLLOW function can also be used as synchronizing tokens during panic-mode error recovery.

If α is any string of grammar symbols, let $FIRST(\alpha)$ be the set of terminals

that begin the strings derived from α. If $\alpha \overset{*}{\Rightarrow} \epsilon$, then ϵ is also in FIRST(α).

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 $S \overset{*}{\Rightarrow} \alpha A a \beta$ for some α and β. Note that there may, at some time during the derivation, have been symbols between A and a, but if so, they derived ϵ and disappeared. If A can be the rightmost symbol in some sentential form, then \$ is in FOLLOW(A).

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 terminal, then FIRST(X) is $\{X\}$.
2. If $X \rightarrow \epsilon$ is a production, then add ϵ to FIRST(X).
3. If X is nonterminal and $X \rightarrow Y_1 Y_2 \cdots Y_k$ is a production, 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, $Y_1 \cdots Y_{i-1} \overset{*}{\Rightarrow} \epsilon$. 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 $Y_1 \overset{*}{\Rightarrow} \epsilon$, then we add FIRST($Y_2$) and so on.

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 the 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 both 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, FIRST(X_i) contains ϵ.

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 for ϵ is placed in FOLLOW(B).
3. If there is a production $A \rightarrow \alpha B$, or a production $A \rightarrow \alpha B \beta$ where FIRST(β) contains ϵ (i.e., $\beta \overset{*}{\Rightarrow} \epsilon$), then everything in FOLLOW(A) is in FOLLOW(B).

Example 4.17. Consider again grammar (4.11), repeated below:

$$
\begin{aligned}
E &\rightarrow TE' \\
E' &\rightarrow +TE' \mid \epsilon \\
T &\rightarrow FT' \\
T' &\rightarrow *FT' \mid \epsilon \\
F &\rightarrow (\ E\ ) \mid \mathbf{id}
\end{aligned}
$$

Then:

$\text{FIRST}(E) = \text{FIRST}(T) = \text{FIRST}(F) = \{(, \textbf{id}\}$.

$\text{FIRST}(E') = \{+, \epsilon\}$

$\text{FIRST}(T') = \{*, \epsilon\}$

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

$\text{FOLLOW}(T) = \text{FOLLOW}(T') = \{+,), \$\}$

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

For example, **id** and left parenthesis are added to FIRST(F) by rule (3) in the definition of FIRST with $i = 1$ in each case, since FIRST(**id**) = {**id**} and FIRST('(') = { (} by rule (1). Then by rule (3) with $i = 1$, the production $T \rightarrow FT'$ implies that **id** and left parenthesis are in FIRST(T) as well. As another example, ϵ is in FIRST(E') by rule (2).

To compute FOLLOW sets, we put \$ in FOLLOW(E) by rule (1) for FOLLOW. By rule (2) applied to production $F \rightarrow (E)$, the right parenthesis is also in FOLLOW(E). By rule (3) applied to production $E \rightarrow TE'$, \$ and right parenthesis are in FOLLOW(E'). Since $E' \overset{*}{\Rightarrow} \epsilon$, they are also in FOLLOW(T). For a last example of how the FOLLOW rules are applied, the production $E \rightarrow TE'$ implies, by rule (2), that everything other than ϵ in FIRST(E') must be placed in FOLLOW(T). We have already seen that \$ is in FOLLOW(T). □

Construction of Predictive Parsing Tables

The following algorithm can be used to construct a predictive parsing table for a grammar G. The idea behind the algorithm is the following. Suppose $A \rightarrow \alpha$ is a production with a in FIRST(α). Then, the parser will expand A by α when the current input symbol is a. The only complication occurs when $\alpha = \epsilon$ or $\alpha \overset{*}{\Rightarrow} \epsilon$. In this case, we should again expand A by α if the current input symbol is in FOLLOW(A), or if the \$ on the input has been reached and \$ is in FOLLOW(A).

Algorithm 4.4. Construction of a predictive parsing table.

Input. Grammar G.

Output. Parsing table M.

Method.

1. For each production $A \rightarrow \alpha$ of the grammar, do steps 2 and 3.
2. For each terminal a in FIRST(α), add $A \rightarrow \alpha$ to $M[A, a]$.
3. If ϵ is in FIRST(α), add $A \rightarrow \alpha$ to $M[A, b]$ for each terminal b in FOLLOW(A). If ϵ is in FIRST(α) and \$ is in FOLLOW(A), add $A \rightarrow \alpha$ to $M[A, \$]$.
4. Make each undefined entry of M be **error**.

Example 4.18. Let us apply Algorithm 4.4 to grammar (4.11). Since FIRST(TE') = FIRST(T) = {(, **id**}, production $E \to TE'$ causes $M[E, (]$ and $M[E, \mathbf{id}]$ to acquire the entry $E \to TE'$.

Production $E' \to +TE'$ causes $M[E', +]$ to acquire $E' \to +TE'$. Production $E' \to \epsilon$ causes $M[E',)]$ and $M[E', \$]$ to acquire $E' \to \epsilon$ since FOLLOW(E') = {), \$}.

The parsing table produced by Algorithm 4.4 for grammar (4.11) was shown in Fig. 4.15. □

LL(1) Grammars

Algorithm 4.4 can be applied to any grammar G to produce a parsing table M. 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.

Example 4.19. Let us consider grammar (4.13) from Example 4.10 again; it is repeated here for convenience.

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

The parsing table for this grammar is shown in Fig. 4.17.

NONTERMINAL	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		$E \to b$				

Fig. 4.17. Parsing table M for grammar (4.13).

The entry for $M[S', e]$ contains both $S' \to eS$ and $S' \to \epsilon$, since FOLLOW(S') = {e, \$}. 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 the ambiguity if we choose $S' \to eS$. This choice corresponds to associating **else**'s with the closest previous **then**'s. Note that the choice $S' \to \epsilon$ would prevent e from ever being put on the stack or removed from the input, and is therefore surely wrong. □

A grammar whose parsing table has no multiply-defined entries is said to be *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. It can be shown that Algorithm 4.4 produces for every LL(1) grammar G a parsing table that parses all and only the sentences of G.

LL(1) grammars have several distinctive properties. No ambiguous or left-recursive grammar can be LL(1). It can also be shown that 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 $\beta \overset{*}{\Rightarrow} \epsilon$, then α does not derive any string beginning with a terminal in FOLLOW(A).

Clearly, grammar (4.11) for arithmetic expressions is LL(1). Grammar (4.13), modeling if-then-else statements, is not.

There remains the question of what should be done when a parsing table has multiply-defined entries. One recourse is to transform the grammar by eliminating all left recursion and then left factoring whenever possible, hoping to produce a grammar for which the parsing table has no multiply-defined entries. Unfortunately, there are some grammars for which no amount of alteration will yield an LL(1) grammar. Grammar (4.13) is one such example; its language has no LL(1) grammar at all. As we saw, we can still parse (4.13) with a predictive parser by arbitrarily making $M[S', e] = \{S' \rightarrow eS\}$. In general, there are no universal rules by which multiply-defined entries can be made single-valued without affecting the language recognized by the parser.

The main difficulty in using predictive parsing is in writing a grammar for the source language such that a predictive parser can be constructed from the grammar. Although left-recursion elimination and left factoring are easy to do, they make the resulting grammar hard to read and difficult to use for translation purposes. To alleviate some of this difficulty, a common organization for a parser in a compiler is to use a predictive parser for control constructs and to use operator precedence (discussed in Section 4.6) for expressions. However, if an LR parser generator, as discussed in Section 4.9, is available, one can get all the benefits of predictive parsing and operator precedence automatically.

Error Recovery in Predictive Parsing

The stack of a nonrecursive predictive parser makes explicit the terminals and nonterminals that the parser hopes to match with the remainder of the input. We shall therefore refer to symbols on the parser stack in the following discussion. 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 the parsing table entry $M[A, a]$ is empty.

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