4.8.3 Error Recovery in LR Parsing

An LR parser will detect an error when it consults the parsing action table and finds an error entry. Errors are never detected by consulting the goto table. An LR parser will announce an error as soon as there is no valid continuation for the portion of the input thus far scanned. A canonical LR parser will not make even a single reduction before announcing an error. SLR and LALR parsers may make several reductions before announcing an error, but they will never shift an erroneous input symbol onto the stack.

In LR parsing, we can implement panic-mode error recovery as follows. We scan down the stack until a state s with a goto on a particular nonterminal A is found. Zero or more input symbols are then discarded until a symbol a is found that can legitimately follow A. The parser then stacks the state $\text{GOTO}(s, A)$ and resumes normal parsing. There might be more than one choice for the nonterminal A. Normally these would be nonterminals representing major program pieces, such as an expression, statement, or block. For example, if A is the nonterminal *stmt*, a might be semicolon or }, which marks the end of a statement sequence.

This method of recovery attempts to eliminate the phrase containing the syntactic error. The parser determines that a string derivable from A contains an error. Part of that string has already been processed, and the result of this

processing is a sequence of states on top of the stack. The remainder of the string is still in the input, and the parser attempts to skip over the remainder of this string by looking for a symbol on the input that can legitimately follow A. By removing states from the stack, skipping over the input, and pushing GOTO(s, A) on the stack, the parser pretends that it has found an instance of A and resumes normal parsing.

Phrase-level recovery is implemented by examining each error entry in the LR parsing table and deciding on the basis of language usage the most likely programmer error that would give rise to that error. An appropriate recovery procedure can then be constructed; presumably the top of the stack and/or first input symbols would be modified in a way deemed appropriate for each error entry.

In designing specific error-handling routines for an LR parser, we can fill in each blank entry in the action field with a pointer to an error routine that will take the appropriate action selected by the compiler designer. The actions may include insertion or deletion of symbols from the stack or the input or both, or alteration and transposition of input symbols. We must make our choices so that the LR parser will not get into an infinite loop. A safe strategy will assure that at least one input symbol will be removed or shifted eventually, or that the stack will eventually shrink if the end of the input has been reached. Popping a stack state that covers a nonterminal should be avoided, because this modification eliminates from the stack a construct that has already been successfully parsed.

Example 4.68: Consider again the expression grammar

$$E \rightarrow E + E \mid E * E \mid (E) \mid \mathbf{id}$$

Figure 4.53 shows the LR parsing table from Fig. 4.49 for this grammar, modified for error detection and recovery. We have changed each state that calls for a particular reduction on some input symbols by replacing error entries in that state by the reduction. This change has the effect of postponing the error detection until one or more reductions are made, but the error will still be caught before any shift move takes place. The remaining blank entries from Fig. 4.49 have been replaced by calls to error routines.

The error routines are as follows.

e1: This routine is called from states 0, 2, 4 and 5, all of which expect the beginning of an operand, either an **id** or a left parenthesis. Instead, +, *, or the end of the input was found.

> push state 3 (the goto of states 0, 2, 4 and 5 on **id**);
> issue diagnostic "missing operand."

e2: Called from states 0, 1, 2, 4 and 5 on finding a right parenthesis.

> remove the right parenthesis from the input;
> issue diagnostic "unbalanced right parenthesis."

STATE	ACTION						GOTO
	id	+	*	(	)	$	*E*
0	s3	e1	e1	s2	e2	e1	1
1	e3	s4	s5	e3	e2	acc	
2	s3	e1	e1	s2	e2	e1	6
3	r4	r4	r4	r4	r4	r4	
4	s3	e1	e1	s2	e2	e1	7
5	s3	e1	e1	s2	e2	e1	8
6	e3	s4	s5	e3	s9	e4	
7	r1	r1	s5	r1	r1	r1	
8	r2	r2	r2	r2	r2	r2	
9	r3	r3	r3	r3	r3	r3	

Figure 4.53: LR parsing table with error routines

e3: Called from states 1 or 6 when expecting an operator, and an **id** or right parenthesis is found.

> push state 4 (corresponding to symbol +) onto the stack;
> issue diagnostic "missing operator."

e4: Called from state 6 when the end of the input is found.

> push state 9 (for a right parenthesis) onto the stack;
> issue diagnostic "missing right parenthesis."

On the erroneous input **id** +), the sequence of configurations entered by the parser is shown in Fig. 4.54. □