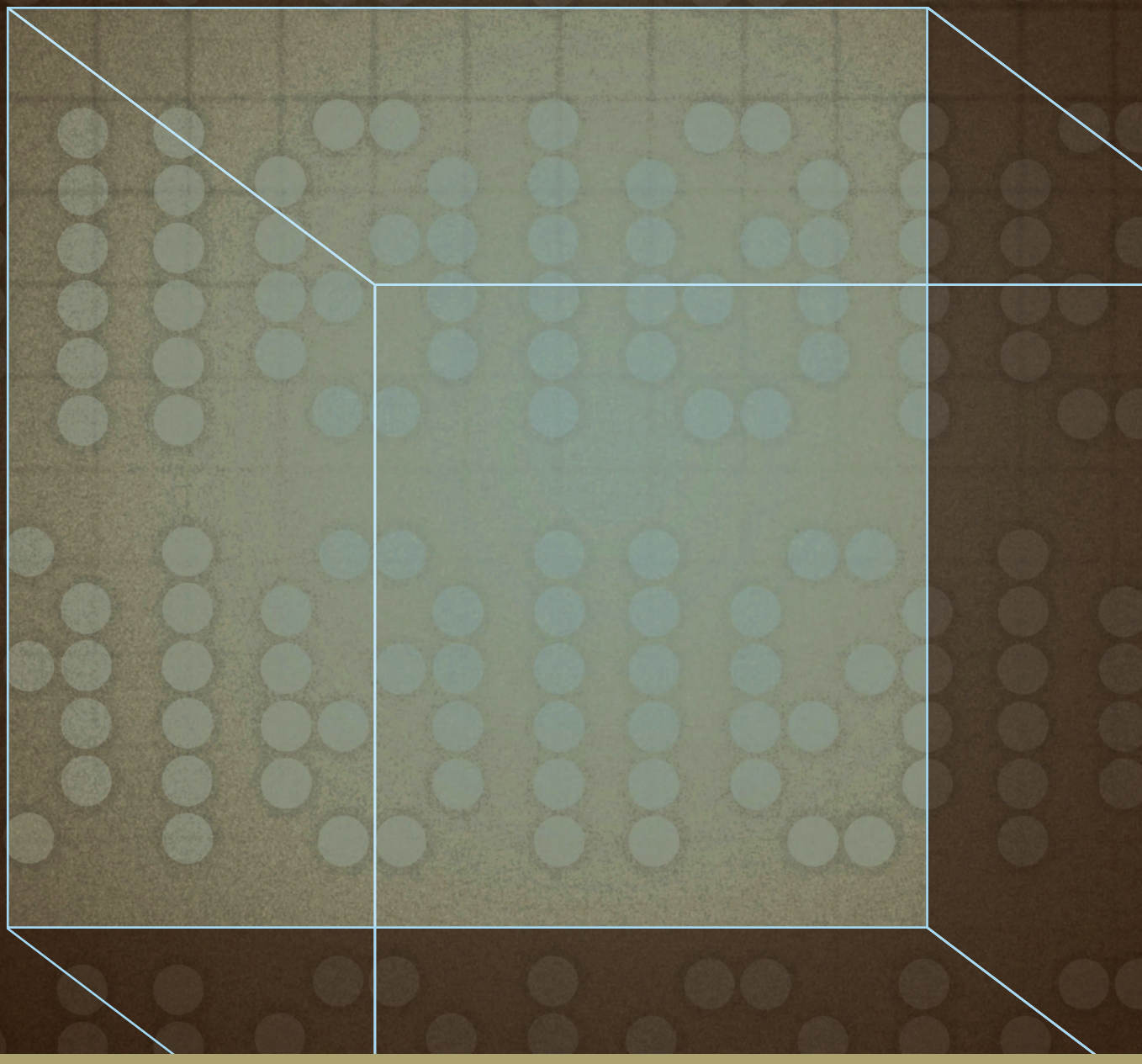
PEARSON NEW INTERNATIONAL EDITION

Compilers
Principles, Techniques, and Tools

Aho Lam Sethi Ullman

Second Edition



ALWAYS LEARNING

PEARSON

Pearson New International Edition

Compilers
Principles, Techniques, and Tools
Aho Lam Sethi Ullman
Second Edition

PEARSON

4.8.1 Precedence and Associativity to Resolve Conflicts

Consider the ambiguous grammar (4.3) for expressions with operators + and $*$, repeated here for convenience:

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

This grammar is ambiguous because it does not specify the associativity or precedence of the operators + and $*$. The unambiguous grammar (4.1), which includes productions $E \to E + T$ and $T \to T * F$, generates the same language, but gives + lower precedence than $*$, and makes both operators left associative. There are two reasons why we might prefer to use the ambiguous grammar. First, as we shall see, we can easily change the associativity and precedence of the operators + and $*$ without disturbing the productions of (4.3) or the number of states in the resulting parser. Second, the parser for the unambiguous grammar will spend a substantial fraction of its time reducing by the productions $E \to T$ and $T \to F$, whose sole function is to enforce associativity and precedence. The parser for the ambiguous grammar (4.3) will not waste time reducing by these *single* productions (productions whose body consists of a single nonterminal).

The sets of LR(0) items for the ambiguous expression grammar (4.3) augmented by $E' \to E$ are shown in Fig. 4.48. Since grammar (4.3) is ambiguous, there will be parsing-action conflicts when we try to produce an LR parsing table from the sets of items. The states corresponding to sets of items I_7 and I_8 generate these conflicts. Suppose we use the SLR approach to constructing the parsing action table. The conflict generated by I_7 between reduction by $E \to E + E$ and shift on + or $*$ cannot be resolved, because + and $*$ are each in FOLLOW(E). Thus both actions would be called for on inputs + and $*$. A similar conflict is generated by I_8, between reduction by $E \to E * E$ and shift on inputs + and $*$. In fact, each of our LR parsing table-construction methods will generate these conflicts.

However, these problems can be resolved using the precedence and associativity information for + and $*$. Consider the input $\mathbf{id} + \mathbf{id} * \mathbf{id}$, which causes a parser based on Fig. 4.48 to enter state 7 after processing $\mathbf{id} + \mathbf{id}$; in particular the parser reaches a configuration

PREFIX	STACK	INPUT
$E + E$	0 1 4 7	$* \mathbf{id}\ \$$

For convenience, the symbols corresponding to the states 1, 4, and 7 are also shown under PREFIX.

If $*$ takes precedence over +, we know the parser should shift $*$ onto the stack, preparing to reduce the $*$ and its surrounding **id** symbols to an expression. This choice was made by the SLR parser of Fig. 4.37, based on an unambiguous grammar for the same language. On the other hand, if + takes precedence over $*$, we know the parser should reduce $E + E$ to E. Thus the relative precedence

I_0: $E' \to \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_1: $E' \to E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_2: $E \to (\cdot E)$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_3: $E \to \mathbf{id}\cdot$

I_4: $E \to E + \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_5: $E \to E * \cdot E$
$E \to \cdot E + E$
$E \to \cdot E * E$
$E \to \cdot (E)$
$E \to \cdot \mathbf{id}$

I_6: $E \to (E\cdot)$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_7: $E \to E + E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_8: $E \to E * E\cdot$
$E \to E\cdot + E$
$E \to E\cdot * E$

I_9: $E \to (E)\cdot$

Figure 4.48: Sets of LR(0) items for an augmented expression grammar

of + followed by $*$ uniquely determines how the parsing action conflict between reducing $E \to E + E$ and shifting on $*$ in state 7 should be resolved.

If the input had been $\mathbf{id} + \mathbf{id} + \mathbf{id}$ instead, the parser would still reach a configuration in which it had stack 0 1 4 7 after processing input $\mathbf{id} + \mathbf{id}$. On input + there is again a shift/reduce conflict in state 7. Now, however, the associativity of the + operator determines how this conflict should be resolved. If + is left associative, the correct action is to reduce by $E \to E + E$. That is, the **id** symbols surrounding the first + must be grouped first. Again this choice coincides with what the SLR parser for the unambiguous grammar would do.

In summary, assuming + is left associative, the action of state 7 on input + should be to reduce by $E \to E + E$, and assuming that $*$ takes precedence over +, the action of state 7 on input $*$ should be to shift. Similarly, assuming that $*$ is left associative and takes precedence over +, we can argue that state 8, which can appear on top of the stack only when $E * E$ are the top three grammar symbols, should have the action reduce $E \to E * E$ on both + and $*$ inputs. In the case of input +, the reason is that $*$ takes precedence over +, while in the case of input $*$, the rationale is that $*$ is left associative.

Proceeding in this way, we obtain the LR parsing table shown in Fig. 4.49. Productions 1 through 4 are $E \rightarrow E + E$, $E \rightarrow E * E$, $\rightarrow (E)$, and $E \rightarrow$ **id**, respectively. It is interesting that a similar parsing action table would be produced by eliminating the reductions by the single productions $E \rightarrow T$ and $T \rightarrow F$ from the SLR table for the unambiguous expression grammar (4.1) shown in Fig. 4.37. Ambiguous grammars like the one for expressions can be handled in a similar way in the context of LALR and canonical LR parsing.

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

Figure 4.49: Parsing table for grammar (4.3)

4.8.2 The "Dangling-Else" Ambiguity

Consider again the following grammar for conditional statements:

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

As we noted in Section 4.3.2, this grammar is ambiguous because it does not resolve the dangling-else ambiguity. To simplify the discussion, let us consider an abstraction of this grammar, where i stands for **if** *expr* **then**, e stands for **else**, and a stands for "all other productions." We can then write the grammar, with augmenting production $S' \rightarrow S$, as

$$\begin{array}{rcl} S' & \rightarrow & S \\ S & \rightarrow & i\ S\ e\ S \ \mid \ i\ S \ \mid \ a \end{array} \tag{4.67}$$

The sets of LR(0) items for grammar (4.67) are shown in Fig. 4.50. The ambiguity in (4.67) gives rise to a shift/reduce conflict in I_4. There, $S \rightarrow iS \cdot eS$ calls for a shift of e and, since FOLLOW$(S) = \{e, \$\}$, item $S \rightarrow iS \cdot$ calls for reduction by $S \rightarrow iS$ on input e.

Translating back to the **if-then-else** terminology, given

I_0: $S' \to \cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_1: $S' \to S\cdot$

I_2: $S \to i\cdot SeS$
$S \to i\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_3: $S \to a\cdot$

I_4: $S \to iS\cdot eS$

I_5: $S \to iSe\cdot S$
$S \to \cdot iSeS$
$S \to \cdot iS$
$S \to \cdot a$

I_6: $S \to iSeS\cdot$

Figure 4.50: LR(0) states for augmented grammar (4.67)

if *expr* **then** *stmt*

on the stack and **else** as the first input symbol, should we shift **else** onto the stack (i.e., shift e) or reduce **if** *expr* **then** *stmt* (i.e, reduce by $S \to iS$)? The answer is that we should shift **else**, because it is "associated" with the previous **then**. In the terminology of grammar (4.67), the e on the input, standing for **else**, can only form part of the body beginning with the iS now on the top of the stack. If what follows e on the input cannot be parsed as an S, completing body $iSeS$, then it can be shown that there is no other parse possible.

We conclude that the shift/reduce conflict in I_4 should be resolved in favor of shift on input e. The SLR parsing table constructed from the sets of items of Fig. 4.50, using this resolution of the parsing-action conflict in I_4 on input e, is shown in Fig. 4.51. Productions 1 through 3 are $S \to iSeS$, $S \to iS$, and $S \to a$, respectively.

STATE	ACTION				GOTO
	i	e	a	\$	S
0	s2		s3		1
1				acc	
2	s2		s3		4
3		r3		r3	
4		s5		r2	
5	s2		s3		6
6		r1		r1	

Figure 4.51: LR parsing table for the "dangling-else" grammar

For example, on input $iiaea$, the parser makes the moves shown in Fig. 4.52, corresponding to the correct resolution of the "dangling-else." At line (5), state 4 selects the shift action on input e, whereas at line (9), state 4 calls for reduction by $S \rightarrow iS$ on input \$.

	STACK	SYMBOLS	INPUT	ACTION
(1)	0		$i\,i\,a\,e\,a\,\$$	shift
(2)	0 2	i	$i\,a\,e\,a\,\$$	shift
(3)	0 2 2	$i\,i$	$a\,e\,a\,\$$	shift
(4)	0 2 2 3	$i\,i\,a$	$e\,a\,\$$	shift
(5)	0 2 2 4	$i\,i\,S$	$e\,a\,\$$	reduce by $S \rightarrow a$
(6)	0 2 2 4 5	$i\,i\,S\,e$	$a\,\$$	shift
(7)	0 2 2 4 5 3	$i\,i\,S\,e\,a$	\$	reduce by $S \rightarrow a$
(8)	0 2 2 4 5 6	$i\,i\,S\,e\,S$	\$	reduce by $S \rightarrow iSeS$
(9)	0 2 4	$i\,S$	\$	reduce by $S \rightarrow iS$
(10)	0 1	S	\$	accept

Figure 4.52: Parsing actions on input $iiaea$

By way of comparison, if we are unable to use an ambiguous grammar to specify conditional statements, then we would have to use a bulkier grammar along the lines of Example 4.16.

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 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."