or in the configuration above. In the former case, we choose reduction by production (5); in the latter case by production (7). Notice how the symbol third from the top of the stack determines the reduction to be made, even though it is not involved in the reduction. Shift-reduce parsing can utilize information far down in the stack to guide the parse. □

4.6 OPERATOR-PRECEDENCE PARSING

The largest class of grammars for which shift-reduce parsers can be built successfully – the LR grammars – will be discussed in Section 4.7. However, for a small but important class of grammars we can easily construct efficient shift-reduce parsers by hand. These grammars have the property (among other essential requirements) that no production right side is ϵ or has two adjacent nonterminals. A grammar with the latter property is called an *operator grammar*.

Example 4.27. The following grammar for expressions

$$\begin{array}{l} E \rightarrow EAE \mid (E) \mid -E \mid \textbf{id} \\ A \rightarrow + \mid - \mid * \mid / \mid \uparrow \end{array}$$

is not an operator grammar, because the right side EAE has two (in fact three) consecutive nonterminals. However, if we substitute for A each of its alternatives, we obtain the following operator grammar:

$$E \rightarrow E+E \mid E-E \mid E*E \mid E/E \mid E \uparrow E \mid (E) \mid -E \mid \textbf{id} \qquad (4.17)$$

We now describe an easy-to-implement parsing technique called operator-precedence parsing. Historically, the technique was first described as a manipulation on tokens without any reference to an underlying grammar. In fact, once we finish building an operator-precedence parser from a grammar, we may effectively ignore the grammar, using the nonterminals on the stack only as placeholders for attributes associated with the nonterminals.

As a general parsing technique, operator-precedence parsing has a number of disadvantages. For example, it is hard to handle tokens like the minus sign, which has two different precedences (depending on whether it is unary or binary). Worse, since the relationship between a grammar for the language being parsed and the operator-precedence parser itself is tenuous, one cannot always be sure the parser accepts exactly the desired language. Finally, only a small class of grammars can be parsed using operator-precedence techniques.

Nevertheless, because of its simplicity, numerous compilers using operator-precedence parsing techniques for expressions have been built successfully. Often these parsers use recursive descent, described in Section 4.4, for statements and higher-level constructs. Operator-precedence parsers have even been built for entire languages.

In operator-precedence parsing, we define three disjoint *precedence relations*, $<\cdot$, $\doteq$, and $\cdot>$, between certain pairs of terminals. These precedence relations guide the selection of handles and have the following meanings:

RELATION	MEANING
$a \lessdot b$	a "yields precedence to" b
$a \doteq b$	a "has the same precedence as" b
$a \gtrdot b$	a "takes precedence over" b

We should caution the reader that while these relations may appear similar to the arithmetic relations "less than," "equal to," and "greater than," the precedence relations have quite different properties. For example, we could have $a \lessdot b$ and $a \gtrdot b$ for the same language, or we might have none of $a \lessdot b$, $a \doteq b$, and $a \gtrdot b$ holding for some terminals a and b.

There are two common ways of determining what precedence relations should hold between a pair of terminals. The first method we discuss is intuitive and is based on the traditional notions of associativity and precedence of operators. For example, if $*$ is to have higher precedence than $+$, we make $+ \lessdot *$ and $* \gtrdot +$. This approach will be seen to resolve the ambiguities of grammar (4.17), and it enables us to write an operator-precedence parser for it (although the unary minus sign causes problems).

The second method of selecting operator-precedence relations is first to construct an unambiguous grammar for the language, a grammar that reflects the correct associativity and precedence in its parse trees. This job is not difficult for expressions; the syntax of expressions in Section 2.2 provides the paradigm. For the other common source of ambiguity, the dangling **else**, grammar (4.9) is a useful model. Having obtained an unambiguous grammar, there is a mechanical method for constructing operator-precedence relations from it. These relations may not be disjoint, and they may parse a language other than that generated by the grammar, but with the standard sorts of arithmetic expressions, few problems are encountered in practice. We shall not discuss this construction here; see Aho and Ullman [1972b].

Using Operator-Precedence Relations

The intention of the precedence relations is to delimit the handle of a right-sentential form, with $\lessdot$ marking the left end, $\doteq$ appearing in the interior of the handle, and $\gtrdot$ marking the right end. To be more precise, suppose we have a right-sentential form of an operator grammar. The fact that no adjacent nonterminals appear on the right sides of productions implies that no right-sentential form will have two adjacent nonterminals either. Thus, we may write the right-sentential form as $\beta_0 a_1 \beta_1 \cdots a_n \beta_n$, where each β_i is either ϵ (the empty string) or a single nonterminal, and each a_i is a single terminal.

Suppose that between a_i and a_{i+1} exactly one of the relations $\lessdot$, $\doteq$, and $\gtrdot$ holds. Further, let us use \$ to mark each end of the string, and define $\$ \lessdot b$ and $b \gtrdot \$$ for all terminals b. Now suppose we remove the nonterminals from the string and place the correct relation $\lessdot$, $\doteq$, or $\gtrdot$, between each pair of terminals and between the endmost terminals and the $'s marking the ends of the string. For example, suppose we initially have the right-sentential form **id** + **id** * **id** and the precedence relations are those given in Fig. 4.23. These relations are some of those that we would choose to parse according to grammar (4.17).

	id	+	*	$
id		$\cdot>$	$\cdot>$	$\cdot>$
+	$<\cdot$	$\cdot>$	$<\cdot$	$\cdot>$
*	$<\cdot$	$\cdot>$	$\cdot>$	$\cdot>$
$	$<\cdot$	$<\cdot$	$<\cdot$	

Fig. 4.23. Operator-precedence relations.

Then the string with the precedence relations inserted is:

$$\$ <\cdot \textbf{id} \cdot> + <\cdot \textbf{id} \cdot> * <\cdot \textbf{id} \cdot> \$ \qquad (4.18)$$

For example, $<\cdot$ is inserted between the leftmost $ and **id** since $<\cdot$ is the entry in row $ and column **id**. The handle can be found by the following process.

1. Scan the string from the left end until the first $\cdot>$ is encountered. In (4.18) above, this occurs between the first **id** and +.
2. Then scan backwards (to the left) over any $\doteq$'s until a $<\cdot$ is encountered. In (4.18), we scan backwards to $.
3. The handle contains everything to the left of the first $\cdot>$ and to the right of the $<\cdot$ encountered in step (2), including any intervening or surrounding nonterminals. (The inclusion of surrounding nonterminals is necessary so that two adjacent nonterminals do not appear in a right-sentential form.) In (4.18), the handle is the first **id**.

If we are dealing with grammar (4.17), we then reduce **id** to E. At this point we have the right-sentential form E+**id*****id**. After reducing the two remaining **id**'s to E by the same steps, we obtain the right-sentential form $E+E*E$. Consider now the string $+*$ obtained by deleting the nonterminals. Inserting the precedence relations, we get

$$\$ <\cdot + <\cdot * \cdot> \$$$

indicating that the left end of the handle lies between + and * and the right end between * and $. These precedence relations indicate that, in the right-sentential form $E+E*E$, the handle is $E*E$. Note how the E's surrounding the * become part of the handle.

Since the nonterminals do not influence the parse, we need not worry about distinguishing among them. A single marker "nonterminal" can be kept on the stack of a shift-reduce parser to indicate placeholders for attribute values.

It may appear from the discussion above that the entire right-sentential form must be scanned at each step to find the handle. Such is not the case if we use a stack to store the input symbols already seen and if the precedence relations are used to guide the actions of a shift-reduce parser. If the precedence relation $<\cdot$ or $\doteq$ holds between the topmost terminal symbol on the stack and the next input symbol, the parser shifts; it has not yet found the right end of the handle. If the relation $\cdot>$ holds, a reduction is called for. At this point the parser has found the right end of the handle, and the precedence relations can be used to find the left end of the handle in the stack.

If no precedence relation holds between a pair of terminals (indicated by a blank entry in Fig. 4.23), then a syntactic error has been detected and an error recovery routine must be invoked, as discussed later in this section. The above ideas can be formalized by the following algorithm.

Algorithm 4.5. Operator-precedence parsing algorithm.

Input. An input string w and a table of precedence relations.

Output. If w is well formed, a *skeletal* parse tree, with a placeholder nonterminal E labeling all interior nodes; otherwise, an error indication.

Method. Initially, the stack contains \$ and the input buffer the string $w\$$. To parse, we execute the program of Fig. 4.24. □

```
(1)  set ip to point to the first symbol of w$;
(2)  repeat forever
(3)      if $ is on top of the stack and ip points to $ then
(4)          return
         else begin
(5)          let a be the topmost terminal symbol on the stack
                 and let b be the symbol pointed to by ip;
(6)          if a <· b or a ≐ b then begin
(7)              push b onto the stack;
(8)              advance ip to the next input symbol;
             end;
(9)          else if a ·> b then            /* reduce */
(10)             repeat
(11)                 pop the stack
(12)             until the top stack terminal is related by <·
                     to the terminal most recently popped
(13)         else error()
         end
```

Fig. 4.24. Operator-precedence parsing algorithm.

Operator-Precedence Relations from Associativity and Precedence

We are always free to create operator-precedence relations any way we see fit and hope that the operator-precedence parsing algorithm will work correctly when guided by them. For a language of arithmetic expressions such as that generated by grammar (4.17) we can use the following heuristic to produce a proper set of precedence relations. Note that grammar (4.17) is ambiguous, and right-sentential forms could have many handles. Our rules are designed to select the "proper" handles to reflect a given set of associativity and precedence rules for binary operators.

1. If operator θ_1 has higher precedence than operator θ_2, make $\theta_1 \cdot\!> \theta_2$ and $\theta_2 <\!\cdot\, \theta_1$. For example, if $*$ has higher precedence than $+$, make $* \cdot\!> +$ and $+ <\!\cdot\, *$. These relations ensure that, in an expression of the form $E+E*E+E$, the central $E*E$ is the handle that will be reduced first.

2. If θ_1 and θ_2 are operators of equal precedence (they may in fact be the same operator), then make $\theta_1 \cdot\!> \theta_2$ and $\theta_2 \cdot\!> \theta_1$ if the operators are left-associative, or make $\theta_1 <\!\cdot\, \theta_2$ and $\theta_2 <\!\cdot\, \theta_1$ if they are right-associative. For example, if $+$ and $-$ are left-associative, then make $+ \cdot\!> +$, $+ \cdot\!> -$, $- \cdot\!> -$, and $- \cdot\!> +$. If $\uparrow$ is right associative, then make $\uparrow <\!\cdot\, \uparrow$. These relations ensure that $E-E+E$ will have handle $E-E$ selected and $E \uparrow E \uparrow E$ will have the last $E \uparrow E$ selected.

3. Make $\theta <\!\cdot$ **id**, **id** $\cdot\!> \theta$, $\theta <\!\cdot\, ($, $(<\!\cdot\, \theta$, $) \cdot\!> \theta$, $\theta \cdot\!>)$, $\theta \cdot\!> \$$, and $\$ <\!\cdot\, \theta$ for all operators θ. Also, let

$(\doteq)$	$\$ <\!\cdot\, ($	$\$ <\!\cdot$ **id**
$(<\!\cdot\, ($	**id** $\cdot\!> \$$	$) \cdot\!> \$$
$(<\!\cdot$ **id**	**id** $\cdot\!>)$	$) \cdot\!>)$

These rules ensure that both **id** and (E) will be reduced to E. Also, \$ serves as both the left and right endmarker, causing handles to be found between \$'s wherever possible.

Example 4.28. Figure 4.25 contains the operator-precedence relations for grammar (4.17), assuming

1. $\uparrow$ is of highest precedence and right-associative,
2. $*$ and $/$ are of next highest precedence and left-associative, and
3. $+$ and $-$ are of lowest precedence and left-associative,

(Blanks denote error entries.) The reader should try out the table to see that it works correctly, ignoring problems with unary minus for the moment. Try the table on the input **id** $*$ (**id** $\uparrow$ **id**)$-$**id**$/$**id**, for example. □

	+	−	*	/	↑	**id**	(	)	$
+	·>	·>	<·	<·	<·	<·	<·	·>	·>
−	·>	·>	<·	<·	<·	<·	<·	·>	·>
*	·>	·>	·>	·>	<·	<·	<·	·>	·>
/	·>	·>	·>	·>	<·	<·	<·	·>	·>
↑	·>	·>	·>	·>	<·	<·	<·	·>	·>
id	·>	·>	·>	·>	·>			·>	·>
(	<·	<·	<·	<·	<·	<·	<·	≐	
)	·>	·>	·>	·>	·>			·>	·>
$	<·	<·	<·	<·	<·	<·	<·		

Fig. 4.25. Operator-precedence relations.

Handling Unary Operators

If we have a unary operator such as ¬ (logical negation), which is not also a binary operator, we can incorporate it into the above scheme for creating operator-precedence relations. Supposing ¬ to be a unary prefix operator, we make θ <· ¬ for any operator θ, whether unary or binary. We make ¬ ·> θ if ¬ has higher precedence than θ and ¬ <· θ if not. For example, if ¬ has higher precedence than &, and & is left-associative, we would group *E*&¬*E*&*E* as (*E*&(¬*E*))&*E*, by these rules. The rule for unary postfix operators is analogous.

The situation changes when we have an operator like the minus sign − that is both unary prefix and binary infix. Even if we give unary and binary minus the same precedence, the table of Fig. 4.25 will fail to parse strings like **id**∗−**id** correctly. The best approach in this case is to use the lexical analyzer to distinguish between unary and binary minus, by having it return a different token when it sees unary minus. Unfortunately, the lexical analyzer cannot use lookahead to distinguish the two; it must remember the previous token. In Fortran, for example, a minus sign is unary if the previous token was an operator, a left parenthesis, a comma, or an assignment symbol.

Precedence Functions

Compilers using operator-precedence parsers need not store the table of precedence relations. In most cases, the table can be encoded by two *precedence functions* f and g that map terminal symbols to integers. We attempt to select f and g so that, for symbols a and b,

1. $f(a) < g(b)$ whenever a <· b,
2. $f(a) = g(b)$ whenever $a \doteq b$, and
3. $f(a) > g(b)$ whenever a ·> b.

Thus the precedence relation between a and b can be determined by a numerical comparison between $f(a)$ and $g(b)$. Note, however, that error entries in the precedence matrix are obscured, since one of (1), (2), or (3) holds no matter what $f(a)$ and $g(b)$ are. The loss of error detection capability is generally not considered serious enough to prevent the using of precedence functions where possible; errors can still be caught when a reduction is called for and no handle can be found.

Not every table of precedence relations has precedence functions to encode it, but in practical cases the functions usually exist.

Example 4.29. The precedence table of Fig. 4.25 has the following pair of precedence functions,

	+	−	*	/	↑	(	)	**id**	**$**
f	2	2	4	4	4	0	6	6	0
g	1	1	3	3	5	5	0	5	0

For example, $* <\cdot$ **id**, and $f(*) < g($**id**$)$. Note that $f($**id**$) > g($**id**$)$ suggests that **id** $\cdot>$ **id**; but, in fact, no precedence relation holds between **id** and **id**. Other error entries in Fig. 4.25 are similarly replaced by one or another precedence relation. □

A simple method for finding precedence functions for a table, if such functions exist, is the following.

Algorithm 4.6. Constructing precedence functions.

Input. An operator precedence matrix.

Output. Precedence functions representing the input matrix, or an indication that none exist.

Method.

1. Create symbols f_a and g_a for each a that is a terminal or $.

2. Partition the created symbols into as many groups as possible, in such a way that if $a \doteq b$, then f_a and g_b are in the same group. Note that we may have to put symbols in the same group even if they are not related by $\doteq$. For example, if $a \doteq b$ and $c \doteq b$, then f_a and f_c must be in the same group, since they are both in the same group as g_b. If, in addition, $c \doteq d$, then f_a and g_d are in the same group even though $a \doteq d$ may not hold.

3. Create a directed graph whose nodes are the groups found in (2). For any a and b, if $a <\cdot b$, place an edge from the group of g_b to the group of f_a. If $a \cdot> b$, place an edge from the group of f_a to that of g_b. Note that an edge or path from f_a to g_b means that $f(a)$ must exceed $g(b)$; a path from g_b to f_a means that $g(b)$ must exceed $f(a)$.

4. If the graph constructed in (3) has a cycle, then no precedence functions exist. If there are no cycles, let $f(a)$ be the length of the longest path

beginning at the group of f_a; let $g(a)$ be the length of the longest path from the group of g_a. □

Example 4.30. Consider the matrix of Fig. 4.23. There are no $\doteq$ relationships, so each symbol is in a group by itself. Figure 4.26 shows the graph constructed using Algorithm 4.6.

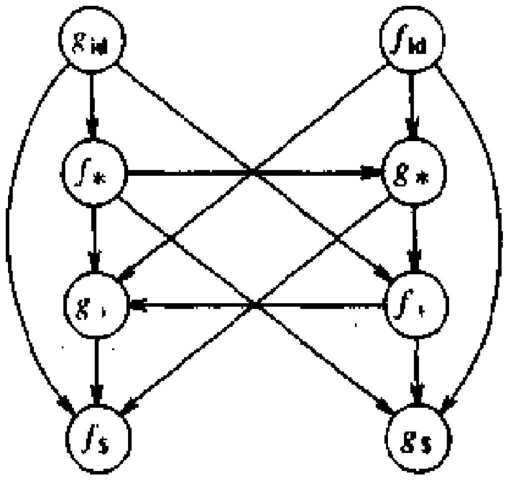

Fig. 4.26. Graph representing precedence functions.

There are no cycles, so precedence functions exist. As $f_\$$ and $g_\$$ have no out-edges, $f(\$) = g(\$) = 0$. The longest path from g_+ has length 1, so $g(+) = 1$. There is a path from g_{id} to f_* to g_* to f_+ to g_+ to $f_\$$, so $g(\mathbf{id}) = 5$. The resulting precedence functions are:

	+	*	**id**	\$
f	2	4	4	0
g	1	3	5	0

□

Error Recovery in Operator-Precedence Parsing

There are two points in the parsing process at which an operator-precedence parser can discover syntactic errors:

1. If no precedence relation holds between the terminal on top of the stack and the current input.[1]
2. If a handle has been found, but there is no production with this handle as a right side.

Recall that the operator-precedence parsing algorithm (Algorithm 4.5) appears to reduce handles composed of terminals only. However, while nonterminals

[1] In compilers using precedence functions to represent the precedence tables, this source of error detection may be unavailable.