erroneous input **id** +). The first actions taken by the parser are to shift **id**, reduce it to E (we again use E for anonymous nonterminals on the stack), and then to shift the +. We now have configuration

STACK	INPUT
$E +	)$

Since + $\cdot>$) a reduction is called for, and the handle is +. The error checker for reductions is required to inspect for E's to left and right. Finding one missing, it issues the diagnostic

```
missing operand
```

and does the reduction anyway.

Our configuration is now

$$\$E \qquad)\$$$

There is no precedence relation between $ and), and the entry in Fig. 4.28 for this pair of symbols is e2. Routine e2 causes diagnostic

```
unbalanced right parenthesis
```

to be printed and removes the right parenthesis from the input. We are now left with the final configuration for the parser.

$$\$E \qquad \$$$

□

4.7 LR PARSERS

This section presents an efficient, bottom-up syntax analysis technique that can be used to parse a large class of context-free grammars. The technique is called LR(k) parsing; the "L" is for left-to-right scanning of the input, the "R" for constructing a rightmost derivation in reverse, and the k for the number of input symbols of lookahead that are used in making parsing decisions. When (k) is omitted, k is assumed to be 1. LR parsing is attractive for a variety of reasons.

- LR parsers can be constructed to recognize virtually all programming-language constructs for which context-free grammars can be written.
- The LR parsing method is the most general nonbacktracking shift-reduce parsing method known, yet it can be implemented as efficiently as other shift-reduce methods.
- The class of grammars that can be parsed using LR methods is a proper superset of the class of grammars that can be parsed with predictive parsers.
- An LR parser can detect a syntactic error as soon as it is possible to do so on a left-to-right scan of the input.

The principal drawback of the method is that it is too much work to construct an LR parser by hand for a typical programming-language grammar. One needs a specialized tool – an LR parser generator. Fortunately, many such generators are available, and we shall discuss the design and use of one, Yacc, in Section 4.9. With such a generator, one can write a context-free grammar and have the generator automatically produce a parser for that grammar. If the grammar contains ambiguities or other constructs that are difficult to parse in a left-to-right scan of the input, then the parser generator can locate these constructs and inform the compiler designer of their presence.

After discussing the operation of an LR parser, we present three techniques for constructing an LR parsing table for a grammar. The first method, called simple LR (SLR for short), is the easiest to implement, but the least powerful of the three. It may fail to produce a parsing table for certain grammars on which the other methods succeed. The second method, called canonical LR, is the most powerful, and the most expensive. The third method, called lookahead LR (LALR for short), is intermediate in power and cost between the other two. The LALR method will work on most programming-language grammars and, with some effort, can be implemented efficiently. Some techniques for compressing the size of an LR parsing table are considered later in this section.

The LR Parsing Algorithm

The schematic form of an LR parser is shown in Fig. 4.29. It consists of an input, an output, a stack, a driver program, and a parsing table that has two parts (*action* and *goto*). The driver program is the same for all LR parsers; only the parsing table changes from one parser to another. The parsing program reads characters from an input buffer one at a time. The program uses a stack to store a string of the form $s_0X_1s_1X_2s_2 \cdots X_ms_m$, where s_m is on top. Each X_i is a grammar symbol and each s_i is a symbol called a *state*. Each state symbol summarizes the information contained in the stack below it, and the combination of the state symbol on top of the stack and the current input symbol are used to index the parsing table and determine the shift-reduce parsing decision. In an implementation, the grammar symbols need not appear on the stack; however, we shall always include them in our discussions to help explain the behavior of an LR parser.

The parsing table consists of two parts, a parsing action function *action* and a goto function *goto*. The program driving the LR parser behaves as follows. It determines s_m, the state currently on top of the stack, and a_i, the current input symbol. It then consults $action[s_m, a_i]$, the parsing action table entry for state s_m and input a_i, which can have one of four values:

1. shift s, where s is a state,
2. reduce by a grammar production $A \rightarrow \beta$,
3. accept, and
4. error.

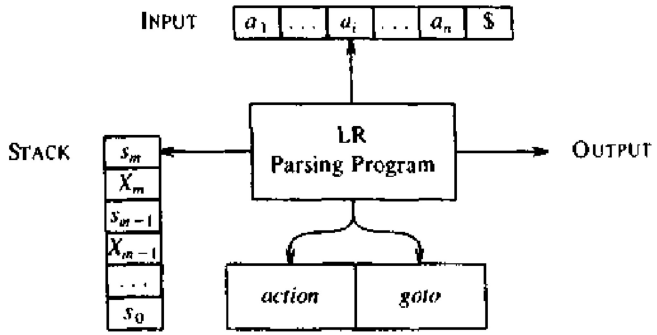

Fig. 4.29. Model of an LR parser.

The function *goto* takes a state and grammar symbol as arguments and produces a state. We shall see that the *goto* function of a parsing table constructed from a grammar G using the SLR, canonical LR, or LALR method is the transition function of a deterministic finite automaton that recognizes the viable prefixes of G. Recall that the viable prefixes of G are those prefixes of right-sentential forms that can appear on the stack of a shift-reduce parser, because they do not extend past the rightmost handle. The initial state of this DFA is the state initially put on top of the LR parser stack.

A *configuration* of an LR parser is a pair whose first component is the stack contents and whose second component is the unexpended input:

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m, \ a_i a_{i+1} \cdots a_n\$)$$

This configuration represents the right-sentential form

$$X_1 X_2 \cdots X_m a_i a_{i+1} \cdots a_n$$

in essentially the same way as a shift-reduce parser would; only the presence of states on the stack is new.

The next move of the parser is determined by reading a_i, the current input symbol, and s_m, the state on top of the stack, and then consulting the parsing action table entry $action[s_m, a_i]$. The configurations resulting after each of the four types of move are as follows:

1. If $action[s_m, a_i]$ = shift s, the parser executes a shift move, entering the configuration

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_m s_m a_i s, \ a_{i+1} \cdots a_n\$)$$

Here the parser has shifted both the current input symbol a_i and the next state s, which is given in $action[s_m, a_i]$, onto the stack; a_{i+1} becomes the current input symbol.

2. If $action[s_m, a_i]$ = reduce $A \to \beta$, then the parser executes a reduce move, entering the configuration

$$(s_0 X_1 s_1 X_2 s_2 \cdots X_{m-r} s_{m-r} A s, \quad a_i a_{i+1} \cdots a_n\$)$$

where $s = goto[s_{m-r}, A]$ and r is the length of β, the right side of the production. Here the parser first popped $2r$ symbols off the stack (r state symbols and r grammar symbols), exposing state s_{m-r}. The parser then pushed both A, the left side of the production, and s, the entry for $goto[s_{m-r}, A]$, onto the stack. The current input symbol is not changed in a reduce move. For the LR parsers we shall construct, $X_{m-r+1} \cdots X_m$, the sequence of grammar symbols popped off the stack, will always match β, the right side of the reducing production.

The output of an LR parser is generated after a reduce move by executing the semantic action associated with the reducing production. For the time being, we shall assume the output consists of just printing the reducing production.

3. If $action[s_m, a_i]$ = accept, parsing is completed.

4. If $action[s_m, a_i]$ = error, the parser has discovered an error and calls an error recovery routine.

The LR parsing algorithm is summarized below. All LR parsers behave in this fashion; the only difference between one LR parser and another is the information in the parsing action and goto fields of the parsing table.

Algorithm 4.7. LR parsing algorithm.

Input. An input string w and an LR parsing table with functions *action* and *goto* for a grammar G.

Output. If w is in $L(G)$, a bottom-up parse for w; otherwise, an error indication.

Method. Initially, the parser has s_0 on its stack, where s_0 is the initial state, and $w\$$ in the input buffer. The parser then executes the program in Fig. 4.30 until an accept or error action is encountered. □

Example 4.33. Figure 4.31 shows the parsing action and goto functions of an LR parsing table for the following grammar for arithmetic expressions with binary operators + and *:

(1) $E \to E + T$
(2) $E \to T$
(3) $T \to T * F$
(4) $T \to F$
(5) $F \to (E)$
(6) $F \to \mathbf{id}$

The codes for the actions are:

```
set ip to point to the first symbol of w$;
repeat forever begin
    let s be the state on top of the stack and
        a the symbol pointed to by ip;
    if action[s, a] = shift s' then begin
        push a then s' on top of the stack;
        advance ip to the next input symbol
    end
    else if action[s, a] = reduce A → β then begin
        pop 2*|β| symbols off the stack;
        let s' be the state now on top of the stack;
        push A then goto[s', A] on top of the stack;
        output the production A → β
    end
    else if action[s, a] = accept then
        return
    else error()
end
```

Fig. 4.30. LR parsing program.

STATE	*action*						*goto*		
	id	+	*	(	)	$	*E*	*T*	*F*
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Fig. 4.31. Parsing table for expression grammar.

1. *si* means shift and stack state *i*,
2. r*j* means reduce by production numbered *j*,
3. acc means accept,
4. blank means error.

Note that the value of *goto*[*s*, *a*] for terminal *a* is found in the action field connected with the shift action on input *a* for state *s*. The goto field gives *goto*[*s*, *A*] for nonterminals *A*. Also, bear in mind that we have not yet explained how the entries for Fig. 4.31 were selected; we shall deal with this issue shortly.

On input **id** * **id** + **id**, the sequence of stack and input contents is shown in Fig. 4.32. For example, at line (1) the LR parser is in state 0 with **id** the first input symbol. The action in row 0 and column **id** of the action field of Fig. 4.31 is s5, meaning shift and cover the stack with state 5. That is what has happened at line (2): the first token **id** and the state symbol 5 have both been pushed onto the stack, and **id** has been removed from the input.

Then, * becomes the current input symbol, and the action of state 5 on input * is to reduce by $F \rightarrow$ **id**. Two symbols are popped off the stack (one state symbol and one grammar symbol). State 0 is then exposed. Since the goto of state 0 on F is 3, F and 3 are pushed onto the stack. We now have the configuration in line (3). Each of the remaining moves is determined similarly. □

	STACK	INPUT	ACTION
(1)	0	**id** * **id** + **id** $	shift
(2)	0 **id** 5	* **id** + **id** $	reduce by $F \rightarrow$ **id**
(3)	0 F 3	* **id** + **id** $	reduce by $T \rightarrow F$
(4)	0 T 2	* **id** + **id** $	shift
(5)	0 T 2 * 7	**id** + **id** $	shift
(6)	0 T 2 * 7 **id** 5	+ **id** $	reduce by $F \rightarrow$ **id**
(7)	0 T 2 * 7 F 10	+ **id** $	reduce by $T \rightarrow T*F$
(8)	0 T 2	+ **id** $	reduce by $E \rightarrow T$
(9)	0 E 1	+ **id** $	shift
(10)	0 E 1 + 6	**id** $	shift
(11)	0 E 1 + 6 **id** 5	$	reduce by $F \rightarrow$ **id**
(12)	0 E 1 + 6 F 3	$	reduce by $T \rightarrow F$
(13)	0 E 1 + 6 T 9	$	$E \rightarrow E+T$
(14)	0 E 1	$	accept

Fig. 4.32. Moves of LR parser on **id** * **id** + **id**.

LR Grammars

How do we construct an LR parsing table for a given grammar? A grammar for which we can construct a parsing table is said to be an *LR grammar*. There are context-free grammars that are not LR, but these can generally be avoided for typical programming-language constructs. Intuitively, in order for a grammar to be LR it is sufficient that a left-to-right shift-reduce parser be able to recognize handles when they appear on top of the stack.