be parsed by a recursive-descent parser that needs no backtracking, i.e., a predictive parser, as discussed in Section 2.4. To construct a predictive parser, we must know, given the current input symbol a and the nonterminal A to be expanded, which one of the alternatives of production $A \rightarrow \alpha_1 | \alpha_2 | \cdots | \alpha_n$ is the unique alternative that derives a string beginning with a. That is, the proper alternative must be detectable by looking at only the first symbol it derives. Flow-of-control constructs in most programming languages, with their distinguishing keywords, are usually detectable in this way. For example, if we have the productions

stmt → **if** *expr* **then** *stmt* **else** *stmt*
| **while** *expr* **do** *stmt*
| **begin** *stmt_list* **end**

then the keywords **if**, **while**, and **begin** tell us which alternative is the only one that could possibly succeed if we are to find a statement.

Transition Diagrams for Predictive Parsers

In Section 2.4, we discussed the implementation of predictive parsers by recursive procedures, e.g., those of Fig. 2.15. Just as a transition diagram was seen in Section 3.4 to be a useful plan or flowchart for a lexical analyzer, we can create a transition diagram as a plan for a predictive parser.

Several differences between the transition diagrams for a lexical analyzer and a predictive parser are immediately apparent. In the case of the parser, there is one diagram for each nonterminal. The labels of edges are tokens and nonterminals. A transition on a token (terminal) means we should take that transition if that token is the next input symbol. A transition on a nonterminal A is a call of the procedure for A.

To construct the transition diagram of a predictive parser from a grammar, first eliminate left recursion from the grammar, and then left factor the grammar. Then for each nonterminal A do the following:

1. Create an initial and final (return) state.
2. For each production $A \rightarrow X_1 X_2 \cdots X_n$, create a path from the initial to the final state, with edges labeled $X_1, X_2, \ldots, X_n$.

The predictive parser working off the transition diagrams behaves as follows. It begins in the start state for the start symbol. If after some actions it is in state s with an edge labeled by terminal a to state t, and if the next input symbol is a, then the parser moves the input cursor one position right and goes to state t. If, on the other hand, the edge is labeled by a nonterminal A, the parser instead goes to the start state for A, without moving the input cursor. If it ever reaches the final state for A, it immediately goes to state t, in effect having "read" A from the input during the time it moved from state s to t. Finally, if there is an edge from s to t labeled ϵ, then from state s the parser immediately goes to state t, without advancing the input.

A predictive parsing program based on a transition diagram attempts to match terminal symbols against the input, and makes a potentially recursive procedure call whenever it has to follow an edge labeled by a nonterminal. A nonrecursive implementation can be obtained by stacking the states s when there is a transition on a nonterminal out of s, and popping the stack when the final state for a nonterminal is reached. We shall discuss the implementation of transition diagrams in more detail shortly.

The above approach works if the given transition diagram does not have nondeterminism, in the sense that there is more than one transition from a state on the same input. If ambiguity occurs, we may be able to resolve it in an ad-hoc way, as in the next example. If the nondeterminism cannot be eliminated, we cannot build a predictive parser, but we could build a recursive-descent parser using backtracking to systematically try all possibilities, if that were the best parsing strategy we could find.

Example 4.15. Figure 4.10 contains a collection of transition diagrams for grammar (4.11). The only ambiguities concern whether or not to take an ϵ-edge. If we interpret the edges out of the initial state for E' as saying take the transition on + whenever that is the next input and take the transition on ϵ otherwise, and make the analogous assumption for T', then the ambiguity is removed, and we can write a predictive parsing program for grammar (4.11). □

Transition diagrams can be simplified by substituting diagrams in one another; these substitutions are similar to the transformations on grammars used in Section 2.5. For example, in Fig. 4.11(a), the call of E' on itself has been replaced by a jump to the beginning of the diagram for E'.

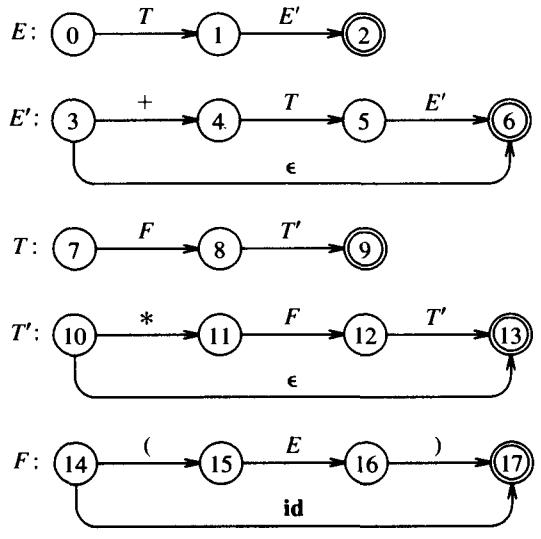

Fig. 4.10. Transition diagrams for grammar (4.11).

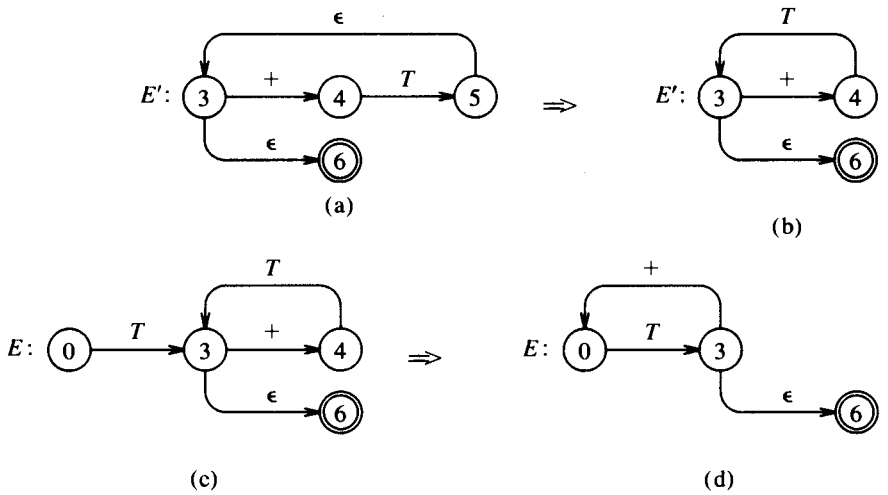

Fig. 4.11. Simplified transition diagrams.

Figure 4.11(b) shows an equivalent transition diagram for E'. We may then substitute the diagram of Fig. 4.11(b) for the transition on E' in the diagram for E in Fig. 4.10, yielding the diagram of Fig. 4.11(c). Lastly, we observe that the first and third nodes in Fig. 4.11(c) are equivalent and we merge them. The result, Fig. 4.11(d), is repeated as the first diagram in Fig. 4.12. The same techniques apply to the diagrams for T and T'. The complete set of resulting diagrams is shown in Fig. 4.12. A C implementation of this predictive parser runs 20-25% faster than a C implementation of Fig. 4.10.

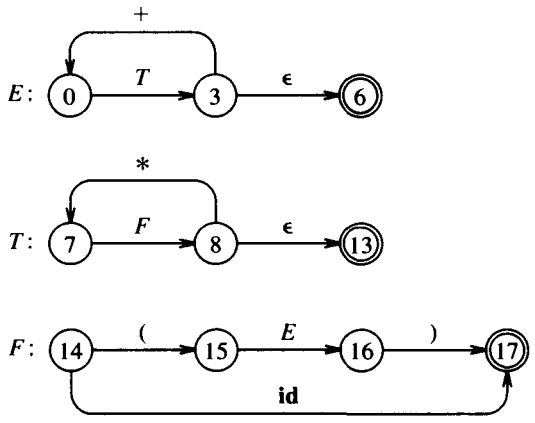

Fig. 4.12. Simplified transition diagrams for arithmetic expressions.