action for each of the first two productions. When executed during a postorder traversal of the parse tree, the actions in Fig. 2.14 print 95-2+. •

Note that although the schemes in Fig. 2.10 and Fig. 2.15 produce the same translation, they construct it differently; Fig. 2.10 attaches strings as attributes to the nodes in the parse tree, while the scheme in Fig. 2.15 prints the translation incrementally, through semantic actions.

The semantic actions in the parse tree in Fig. 2.14 translate the infix expression 9-5+2 into 95-2+ by printing each character in 9-5+2 exactly once, without using any storage for the translation of subexpressions. When the output is created incrementally in this fashion, the order in which the characters are printed is significant.

The implementation of a translation scheme must ensure that semantic actions are performed in the order they would appear during a postorder traversal of a parse tree. The implementation need not actually construct a parse tree (often it does not), as long as it ensures that the semantic actions are performed as if we constructed a parse tree and then executed the actions during a postorder traversal.

2.3.6 Exercises for Section 2.3

Exercise 2.3.1: Construct a syntax-directed translation scheme that translates arithmetic expressions from infix notation into prefix notation in which an operator appears before its operands; e.g., $-xy$ is the prefix notation for $x - y$. Give annotated parse trees for the inputs 9-5+2 and 9-5*2.

Exercise 2.3.2: Construct a syntax-directed translation scheme that translates arithmetic expressions from postfix notation into infix notation. Give annotated parse trees for the inputs 95-2* and 952*-.

Exercise 2.3.3: Construct a syntax-directed translation scheme that translates integers into roman numerals.

Exercise 2.3.4: Construct a syntax-directed translation scheme that translates roman numerals into integers.

Exercise 2.3.5: Construct a syntax-directed translation scheme that translates postfix arithmetic expressions into equivalent infix arithmetic expressions.

2.4 Parsing

Parsing is the process of determining how a string of terminals can be generated by a grammar. In discussing this problem, it is helpful to think of a parse tree being constructed, even though a compiler may not construct one, in practice. However, a parser must be capable of constructing the tree in principle, or else the translation cannot be guaranteed correct.

This section introduces a parsing method called "recursive descent," which can be used both to parse and to implement syntax-directed translators. A complete Java program, implementing the translation scheme of Fig. 2.15, appears in the next section. A viable alternative is to use a software tool to generate a translator directly from a translation scheme. Section 4.9 describes such a tool — Yacc; it can implement the translation scheme of Fig. 2.15 without modification.

For any context-free grammar there is a parser that takes at most $O(n^3)$ time to parse a string of n terminals. But cubic time is generally too expensive. Fortunately, for real programming languages, we can generally design a grammar that can be parsed quickly. Linear-time algorithms suffice to parse essentially all languages that arise in practice. Programming-language parsers almost always make a single left-to-right scan over the input, looking ahead one terminal at a time, and constructing pieces of the parse tree as they go.

Most parsing methods fall into one of two classes, called the *top-down* and *bottom-up* methods. These terms refer to the order in which nodes in the parse tree are constructed. In top-down parsers, construction starts at the root and proceeds towards the leaves, while in bottom-up parsers, construction starts at the leaves and proceeds towards the root. The popularity of top-down parsers is due to the fact that efficient parsers can be constructed more easily by hand using top-down methods. Bottom-up parsing, however, can handle a larger class of grammars and translation schemes, so software tools for generating parsers directly from grammars often use bottom-up methods.

2.4.1 Top-Down Parsing

We introduce top-down parsing by considering a grammar that is well-suited for this class of methods. Later in this section, we consider the construction of top-down parsers in general. The grammar in Fig. 2.16 generates a subset of the statements of C or Java. We use the boldface terminals **if** and **for** for the keywords "if" and "for", respectively, to emphasize that these character sequences are treated as units, i.e., as single terminal symbols. Further, the terminal **expr** represents expressions; a more complete grammar would use a nonterminal *expr* and have productions for nonterminal *expr*. Similarly, **other** is a terminal representing other statement constructs.

The top-down construction of a parse tree like the one in Fig. 2.17, is done by starting with the root, labeled with the starting nonterminal *stmt,* and repeatedly performing the following two steps.

1. At node JV, labeled with nonterminal *A,* select one of the productions for *A* and construct children at *N* for the symbols in the production body.

2. Find the next node at which a subtree is to be constructed, typically the leftmost unexpanded nonterminal of the tree.

For some grammars, the above steps can be implemented during a single left-to-right scan of the input string. The current terminal being scanned in the

```
stmt  —y  expr ;
          if ( expr ) stmt
          for ( optexpr ; optexpr ; optexpr ) stmt
          other

optexpr —>  e
            expr
```

Figure 2.16: A grammar for some statements in C and Java

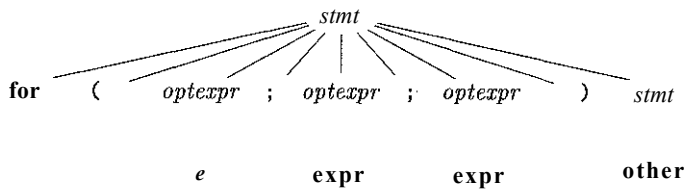

Figure 2.17: A parse tree according to the grammar in Fig. 2.16

input is frequently referred to as the *lookahead* symbol. Initially, the lookahead symbol is the first, i.e., leftmost, terminal of the input string. Figure 2.18 illustrates the construction of the parse tree in Fig. 2.17 for the input string

for (; expr ; expr) other

Initially, the terminal **for** is the lookahead symbol, and the known part of the parse tree consists of the root, labeled with the starting nonterminal *stmt* in Fig. 2.18(a). The objective is to construct the remainder of the parse tree in such a way that the string generated by the parse tree matches the input string.

For a match to occur, the nonterminal *stmt* in Fig. 2.18(a) must derive a string that starts with the lookahead symbol **for.** In the grammar of Fig. 2.16, there is just one production for *stmt that* can derive such a string, so we select it, and construct the children of the root labeled with the symbols in the production body. This expansion of the parse tree is shown in Fig. 2.18(b).

Each of the three snapshots in Fig. 2.18 has arrows marking the lookahead symbol in the input and the node in the parse tree that is being considered. Once children are constructed at a node, we next consider the leftmost child. In Fig. 2.18(b), children have just been constructed at the root, and the leftmost child labeled with **for** is being considered.

When the node being considered in the parse tree is for a terminal, and the terminal matches the lookahead symbol, then we advance in both the parse tree and the input. The next terminal in the input becomes the new lookahead symbol, and the next child in the parse tree is considered. In Fig. 2.18(c), the arrow in the parse tree has advanced to the next child of the root, and the arrow

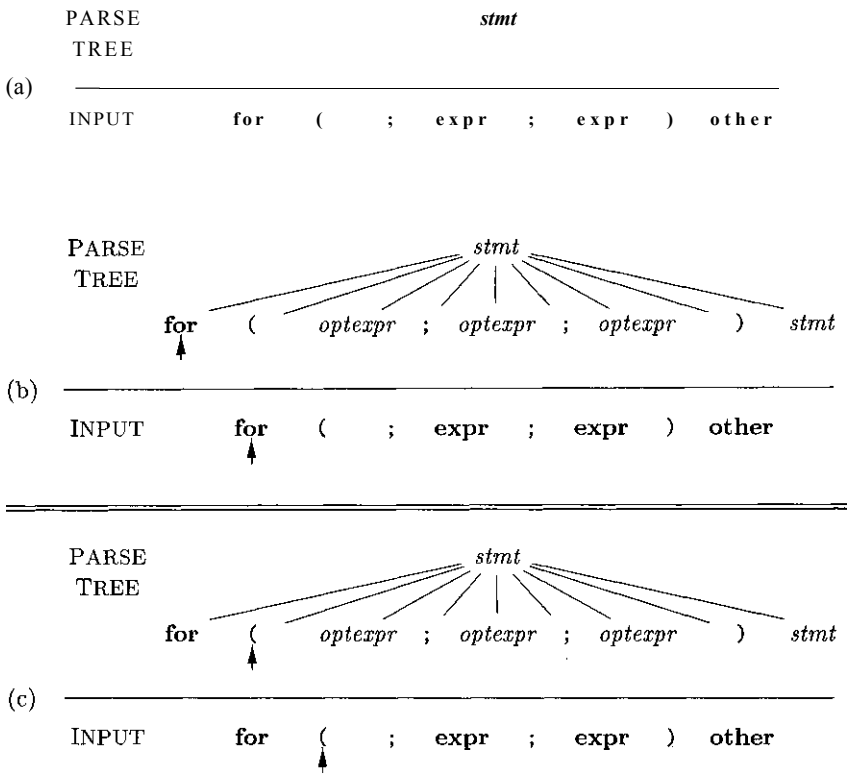

Figure 2.18: Top-down parsing while scanning the input from left to right

in the input has advanced to the next terminal, which is (. A further advance will take the arrow in the parse tree to the child labeled with nonterminal *optexpr* and take the arrow in the input to the terminal ;.

At the nonterminal node labeled *optexpr,* we repeat the process of selecting a production for a nonterminal. Productions with e as the body ("e-productions") require special treatment. For the moment, we use them as a default when no other production can be used; we return to them in Section 2.4.3. With nonterminal *optexpr* and lookahead ;, the e-production is used, since ; does not match the only other production for *optexpr,* which has terminal **expr** as its body.

In general, the selection of a production for a nonterminal may involve trial-and-error; that is, we may have to try a production and backtrack to try another production if the first is found to be unsuitable. A production is unsuitable if, after using the production, we cannot complete the tree to match the input

string. Backtracking is not needed, however, in an important special case called predictive parsing, which we discuss next.

2.4.2 Predictive Parsing

Recursive-descent parsing is a top-down method of syntax analysis in which a set of recursive procedures is used to process the input. One procedure is associated with each nonterminal of a grammar. Here, we consider a simple form of recursive-descent parsing, called *predictive parsing,* in which the lookahead symbol unambiguously determines the flow of control through the procedure body for each nonterminal. The sequence of procedure calls during the analysis of an input string implicitly defines a parse tree for the input, and can be used to build an explicit parse tree, if desired.

The predictive parser in Fig. **2.19** consists of procedures for the nonterminals *stmt* and *optexpr* of the grammar in Fig. **2.16** and an additional procedure *match,* used to simplify the code for *stmt* and *optexpr.* Procedure *match(t)* compares its argument *t* with the lookahead symbol and advances to the next input terminal if they match. Thus *match* changes the value of variable *lookahead,* a global variable that holds the currently scanned input terminal.

Parsing begins with a call of the procedure for the starting nonterminal *stmt.* With the same input as in Fig. **2.18,** *lookahead* is initially the first terminal **for.** Procedure *stmt* executes code corresponding to the production

stmt → **for** **(** *optexpr* ; *optexpr* ; *optexpr*) *stmt*

In the code for the production body — that is, the **for** case of procedure *stmt* — each terminal is matched with the lookahead symbol, and each nonterminal leads to a call of its procedure, in the following sequence of calls:

```
match(for);     matched);
optexprQ;   match(';');   optexprQ;   match(';');   optexprQ;
match(')');     stmtQ;
```

Predictive parsing relies on information about the first symbols that can be generated by a production body. More precisely, let *a* be a string of grammar symbols (terrninals and/or nonterminals). We define **FIRST**(a) to be the set of terminals that appear as the first symbols of one or more strings of terminals generated from *a.* If *a* is e or can generate e, then e is also in FIRST *(a).*

The details of how one computes FIRST(o;) are in Section 4.4.2. Here, we shall just use ad hoc reasoning to deduce the symbols in FIRST (a); typically, *a* will either begin with a terminal, which is therefore the only symbol in FIRST(Q;), or *a* will begin with a nonterminal whose production bodies begin with terminals, in which case these terminals are the only members of **FIRST**(a).

For example, with respect to the grammar of Fig. **2.16,** the following are correct calculations of FIRST.

```
void stmt() {
        switch ( lookahead ) {
        case expr:
                match(expr);    match(';');    break;
        case if:
                match(it);    match(' ');    match (expr);    match(')');    stmtQ;
                break;
        case for:
                match(for);    match(' ');
                optexprQ;    match(';');    optexpr();    match(';');    optexprQ;
                match(')');    stmtQ;    break;
        case other;
                match (other);  break;
        default:
                report ("syntax  error");
        }
}

void optexpr() {
        if ( lookahead == expr ) match(expr);
}

void match(terminal £) {
        if ( lookahead == £ ) lookahead — next Terminal;
        else report("syntax  error");
}
```

Figure 2.19: Pseudocode for a predictive parser

FIRST(stmt) = {expr, if, for, other}
FIRST(expr ;) = {expr}

The **FIRST** sets must be considered if there are two productions *A -» a* and *A —> 0.* Ignoring e-productions for the moment, predictive parsing requires **FIRST**(a) and **FIRST**(/3) to be disjoint. The lookahead symbol can then be used to decide which production to use; if the lookahead symbol is in **FIRST(OJ)**, then *a* is used. Otherwise, if the lookahead symbol is in **FIRST**(/3), then *0* is used.

2.4.3 When to Use e-Productions

Our predictive parser uses an e-production as a default when no other production can be used. With the input of Fig. 2.18, after the terminals **for** and (are matched, the lookahead symbol is ;. At this point procedure *optexpr* is called, and the code

```
if ( lookahead === expr ) match(expr);
```

in its body is executed. Nonterminal *optexpr* has two productions, with bodies **expr** and e. The lookahead symbol ";" does not match the terminal **expr,** so the production with body **expr** cannot apply. In fact, the procedure returns without changing the lookahead symbol or doing anything else. Doing nothing corresponds to applying an e-production.

More generally, consider a variant of the productions in Fig. 2.16 where *optexpr* generates an expression nonterminal instead of the terminal **expr:**

$$optexpr \longrightarrow \begin{array}{|} \\ \mathrm{I} \end{array} \begin{array}{l} \bullet \\ e \end{array} expr$$

Thus, *optexpr* either generates an expression using nonterminal *expr* or it generates e. While parsing *optexpr,* if the lookahead symbol is not in FIRST (espr), then the e-production is used.

For more on when to use e-productions, see the discussion of LL(1) grammars in Section 4.4.3.

2.4.4 Designing a Predictive Parser

We can generalize the technique introduced informally in Section 2.4.2, to apply to any grammar that has disjoint FIRST sets for the production bodies belonging to any nonterminal. We shall also see that when we have a translation scheme — that is, a grammar with embedded actions — it is possible to execute those actions as part of the procedures designed for the parser.

Recall that a *predictive parser* is a program consisting of a procedure for every nonterminal. The procedure for nonterminal *A* does two things.

1. It decides which A-production to use by examining the lookahead symbol. The production with body *a* (where *a* is not e, the empty string) is used if the lookahead symbol is in FiRST(a). If there is a conflict between two nonempty bodies for any lookahead symbol, then we cannot use this parsing method on this grammar. In addition, the e-production for *A,* if it exists, is used if the lookahead symbol is not in the FIRST set for any other production body for *A.*

2. The procedure then mimics the body of the chosen production. That is, the symbols of the body are "executed" in turn, from the left. A nonterminal is "executed" by a call to the procedure for that nonterminal, and a terminal matching the lookahead symbol is "executed" by reading the next input symbol. If at some point the terminal in the body does not match the lookahead symbol, a syntax error is reported.

Figure 2.19 is the result of applying these rules to the grammar in Fig. 2.16.

Just as a translation scheme is formed by extending a grammar, a syntax-directed translator can be formed by extending a predictive parser. An algorithm for this purpose is given in Section 5.4. The following limited construction suffices for the present:

1. Construct a predictive parser, ignoring the actions in productions.
2. Copy the actions from the translation scheme into the parser. If an action appears after grammar symbol X in production p, then it is copied after the implementation of X in the code for p. Otherwise, if it appears at the beginning of the production, then it is copied just before the code for the production body.

We shall construct such a translator in Section 2.5.

2.4.5 Left Recursion

It is possible for a recursive-descent parser to loop forever. A problem arises with "left-recursive" productions like

$$expr \quad —y \quad expr + term$$

where the leftmost symbol of the body is the same as the nonterminal at the head of the production. Suppose the procedure for *expr* decides to apply this production. The body begins with *expr* so the procedure for *expr* is called recursively. Since the lookahead symbol changes only when a terminal in the body is matched, no change to the input took place between recursive calls of *expr*. As a result, the second call to *expr* does exactly what the first call did, which means a third call to *expr*, and so on, forever.

A left-recursive production can be eliminated by rewriting the offending production. Consider a nonterminal A with two productions

$$A \ \text{->•} \ Aa \mid /3$$

where a and $/3$ are sequences of terminals and nonterminals that do not start with A. For example, in

$$expr \quad —y \quad expr + term \quad \backslash \quad term$$

nonterminal A — *expr*, string a = + *term*, and string $(3$ = *term*.

The nonterminal A and its production are said to be *left recursive*, because the production A -> Aa has A itself as the leftmost symbol on the right side.[4] Repeated application of this production builds up a sequence of a's to the right of A, as in Fig. 2.20(a). When A is finally replaced by $/3$, we have a $/3$ followed by a sequence of zero or more a's.

The same effect can be achieved, as in Fig. 2.20(b), by rewriting the productions for A in the following manner, using a new nonterminal R:

[4] In a general left-recursive grammar, instead of a production $A \ A \ a$, the nonterminal A may derive Aa through intermediate productions.

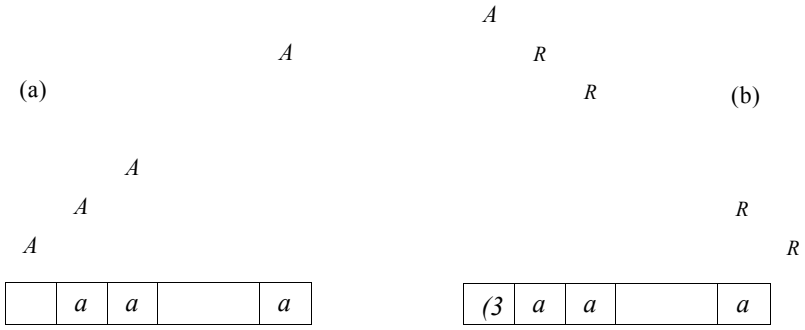

Figure 2.20: Left- and right-recursive ways of generating a string

$$A \rightarrow /3R$$
$$R \rightarrow\bullet\ aR \mid e$$

Nonterminal R and its production $R \rightarrow aR$ are *recursive* because this production for R has R itself as the last symbol on the right side. Right-recursive productions lead to trees that grow down towards the right, as in Fig. 2.20(b). Trees growing down to the right make it harder to translate expressions containing left-associative operators, such as minus. In Section 2.5.2, however, we shall see that the proper translation of expressions into postfix notation can still be attained by a careful design of the translation scheme.

In Section 4.3.3, we shall consider more general forms of left recursion and show how all left recursion can be eliminated from a grammar.

2.4.6 Exercises for Section 2.4

Exercise **2.4.1**: Construct recursive-descent parsers, starting with the following grammars:

a) $S \rightarrow +\ SS \mid -SS \mid \mathbf{a}$

b) $S \rightarrow 5\ (\ 5\ )\ 5 \mid e$

c) $S \quad 0\ 5\ 1 \mid 0\ 1$

2.5 A Translator for Simple Expressions

Using the techniques of the last three sections, we now construct a syntax-directed translator, in the form of a working Java program, that translates arithmetic expressions into postfix form. To keep the initial program manageably small, we start with expressions consisting of digits separated by binary plus and minus signs. We extend the program in Section 2.6 to translate expressions that include numbers and other operators. It is worth studying the