Chapter 2

A Simple Syntax-Directed Translator

This chapter is an introduction to the compiling techniques in Chapters 3 through 6 of this book. It illustrates the techniques by developing a working Java program that translates representative programming language statements into three-address code, an intermediate representation. In this chapter, the emphasis is on the front end of a compiler, in particular on lexical analysis, parsing, and intermediate code generation. Chapters 7 and 8 show how to generate machine instructions from three-address code.

We start small by creating a syntax-directed translator that maps infix arithmetic expressions into postfix expressions. We then extend this translator to map code fragments as shown in Fig. 2.1 into three-address code of the form in Fig. 2.2.

The working Java translator appears in Appendix A. The use of Java is convenient, but not essential. In fact, the ideas in this chapter predate the creation of both Java and C.

```
{
      int i; int j; float[100] a; float v; float x;

      while ( true ) {
            do i = i+1; while ( a[i] < v );
            do j = j-1; while ( a[j] > v );
            if ( i >= j ) break;
            x = a[i]; a[i] = a[j]; a[j] = x;
      }
}
```

Figure 2.1: A code fragment to be translated

```
 1:  i = i + 1
 2:  t1 = a [ i ]
 3:  if t1 < v goto 1
 4:  j = j - 1
 5:  t2 = a [ j ]
 6:  if t2 > v goto 4
 7:  ifFalse i >= j goto 9
 8:  goto 14
 9:  x = a [ i ]
10:  t3 = a [ j ]
11:  a [ i ] = t3
12:  a [ j ] = x
13:  goto 1
14:
```

Figure 2.2: Simplified intermediate code for the program fragment in Fig. 2.1

2.1 Introduction

The analysis phase of a compiler breaks up a source program into constituent pieces and produces an internal representation for it, called intermediate code. The synthesis phase translates the intermediate code into the target program.

Analysis is organized around the "syntax" of the language to be compiled. The *syntax* of a programming language describes the proper form of its programs, while the *semantics* of the language defines what its programs mean; that is, what each program does when it executes. For specifying syntax, we present a widely used notation, called context-free grammars or BNF (for Backus-Naur Form) in Section 2.2. With the notations currently available, the semantics of a language is much more difficult to describe than the syntax. For specifying semantics, we shall therefore use informal descriptions and suggestive examples.

Besides specifying the syntax of a language, a context-free grammar can be used to help guide the translation of programs. In Section 2.3, we introduce a grammar-oriented compiling technique known as *syntax-directed translation.* Parsing or syntax analysis is introduced in Section 2.4.

The rest of this chapter is a quick tour through the model of a compiler front end in Fig. 2.3. We begin with the parser. For simplicity, we consider the syntax-directed translation of infix expressions to postfix form, a notation in which operators appear after their operands. For example, the postfix form of the expression $9 - 5 + 2$ is $95 - 2+$. Translation into postfix form is rich enough to illustrate syntax analysis, yet simple enough that the translator is shown in full in Section 2.5. The simple translator handles expressions like $9 - 5 + 2$, consisting of digits separated by plus and minus signs. One reason for starting with such simple expressions is that the syntax analyzer can work directly with the individual characters for operators and operands.

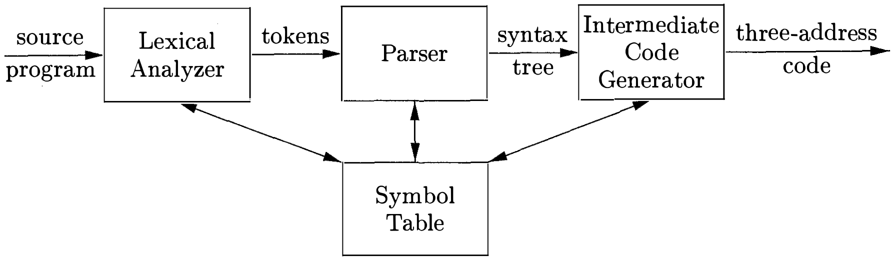
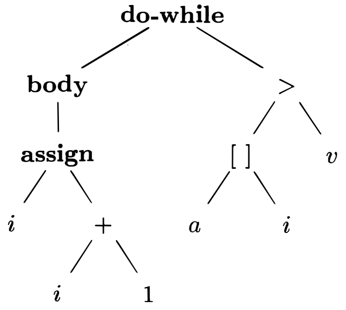


Figure 2.3: A model of a compiler front end

A lexical analyzer allows a translator to handle multicharacter constructs like identifiers, which are written as sequences of characters, but are treated as units called *tokens* during syntax analysis; for example, in the expression `count + 1`, the identifier `count` is treated as a unit. The lexical analyzer in Section 2.6 allows numbers, identifiers, and "white space" (blanks, tabs, and newlines) to appear within expressions.

Next, we consider intermediate-code generation. Two forms of intermediate code are illustrated in Fig. 2.4. One form, called *abstract syntax trees* or simply *syntax trees*, represents the hierarchical syntactic structure of the source program. In the model in Fig. 2.3, the parser produces a syntax tree, that is further translated into three-address code. Some compilers combine parsing and intermediate-code generation into one component.



(a)

```
1: i = i + 1
2: t1 = a [ i ]
3: if t1 < v goto 1
```

(b)

Figure 2.4: Intermediate code for "`do i = i + 1; while ( a[i] < v);`"

The root of the abstract syntax tree in Fig. 2.4(a) represents an entire do-while loop. The left child of the root represents the body of the loop, which consists of only the assignment `i = i + 1;`. The right child of the root represents the condition `a[i] < v`. An implementation of syntax trees appears in Section 2.8(a).

The other common intermediate representation, shown in Fig. 2.4(b), is a sequence of "three-address" instructions; a more complete example appears in Fig. 2.2. This form of intermediate code takes its name from instructions of the form $x = y$ **op** z, where **op** is a binary operator, y and z the are addresses for the operands, and x is the address for the result of the operation. A three-address instruction carries out at most one operation, typically a computation, a comparison, or a branch.

In Appendix A, we put the techniques in this chapter together to build a compiler front end in Java. The front end translates statements into assembly-level instructions.

2.2 Syntax Definition

In this section, we introduce a notation — the "context-free grammar," or "grammar" for short — that is used to specify the syntax of a language. Grammars will be used throughout this book to organize compiler front ends.

A grammar naturally describes the hierarchical structure of most programming language constructs. For example, an if-else statement in Java can have the form

if (expression) statement **else** statement

That is, an if-else statement is the concatenation of the keyword **if**, an opening parenthesis, an expression, a closing parenthesis, a statement, the keyword **else**, and another statement. Using the variable *expr* to denote an expression and the variable *stmt* to denote a statement, this structuring rule can be expressed as

stmt → **if** (*expr*) *stmt* **else** *stmt*

in which the arrow may be read as "can have the form." Such a rule is called a *production.* In a production, lexical elements like the keyword **if** and the parentheses are called *terminals.* Variables like *expr* and *stmt* represent sequences of terminals and are called *nonterminals.*

2.2.1 Definition of Grammars

A *context-free grammar* has four components:

1. A set of *terminal* symbols, sometimes referred to as "tokens." The terminals are the elementary symbols of the language defined by the grammar.

2. A set of *nonterminals*, sometimes called "syntactic variables." Each nonterminal represents a set of strings of terminals, in a manner we shall describe.

3. A set of *productions*, where each production consists of a nonterminal, called the *head* or *left side* of the production, an arrow, and a sequence of

Tokens Versus Terminals

In a compiler, the lexical analyzer reads the characters of the source program, groups them into lexically meaningful units called lexemes, and produces as output tokens representing these lexemes. A token consists of two components, a token name and an attribute value. The token names are abstract symbols that are used by the parser for syntax analysis. Often, we shall call these token names *terminals*, since they appear as terminal symbols in the grammar for a programming language. The attribute value, if present, is a pointer to the symbol table that contains additional information about the token. This additional information is not part of the grammar, so in our discussion of syntax analysis, often we refer to tokens and terminals synonymously.

terminals and/or nonterminals, called the *body* or *right side* of the production. The intuitive intent of a production is to specify one of the written forms of a construct; if the head nonterminal represents a construct, then the body represents a written form of the construct.

4. A designation of one of the nonterminals as the *start* symbol.

We specify grammars by listing their productions, with the productions for the start symbol listed first. We assume that digits, signs such as < and <=, and boldface strings such as **while** are terminals. An italicized name is a nonterminal, and any nonitalicized name or symbol may be assumed to be a terminal.[1] For notational convenience, productions with the same nonterminal as the head can have their bodies grouped, with the alternative bodies separated by the symbol |, which we read as "or."

Example 2.1: Several examples in this chapter use expressions consisting of digits and plus and minus signs; e.g., strings such as 9-5+2, 3-1, or 7. Since a plus or minus sign must appear between two digits, we refer to such expressions as "lists of digits separated by plus or minus signs." The following grammar describes the syntax of these expressions. The productions are:

$$
\begin{array}{rcll}
list & \rightarrow & list \;\texttt{+}\; digit & (2.1)\\
list & \rightarrow & list \;\texttt{-}\; digit & (2.2)\\
list & \rightarrow & digit & (2.3)\\
digit & \rightarrow & \texttt{0} \mid \texttt{1} \mid \texttt{2} \mid \texttt{3} \mid \texttt{4} \mid \texttt{5} \mid \texttt{6} \mid \texttt{7} \mid \texttt{8} \mid \texttt{9} & (2.4)
\end{array}
$$

[1] Individual italic letters will be used for additional purposes, especially when grammars are studied in detail in Chapter 4. For example, we shall use X, Y, and Z to talk about a symbol that is either a terminal or a nonterminal. However, any italicized name containing two or more characters will continue to represent a nonterminal.

The bodies of the three productions with nonterminal *list* as head equivalently can be grouped:

$$list \quad \rightarrow \quad list \; \texttt{+} \; digit \mid list \; \texttt{-} \; digit \mid digit$$

According to our conventions, the terminals of the grammar are the symbols

`+ - 0 1 2 3 4 5 6 7 8 9`

The nonterminals are the italicized names *list* and *digit*, with *list* being the start symbol because its productions are given first. □

We say a production is *for* a nonterminal if the nonterminal is the head of the production. A string of terminals is a sequence of zero or more terminals. The string of zero terminals, written as ϵ, is called the *empty* string.[2]

2.2.2 Derivations

A grammar derives strings by beginning with the start symbol and repeatedly replacing a nonterminal by the body of a production for that nonterminal. The terminal strings that can be derived from the start symbol form the *language* defined by the grammar.

Example 2.2: The language defined by the grammar of Example 2.1 consists of lists of digits separated by plus and minus signs. The ten productions for the nonterminal *digit* allow it to stand for any of the terminals `0`, `1`, ... , `9`. From production (2.3), a single digit by itself is a list. Productions (2.1) and (2.2) express the rule that any list followed by a plus or minus sign and then another digit makes up a new list.

Productions (2.1) to (2.4) are all we need to define the desired language. For example, we can deduce that `9-5+2` is a *list* as follows.

a) `9` is a *list* by production (2.3), since `9` is a *digit*.

b) `9-5` is a *list* by production (2.2), since `9` is a *list* and `5` is a *digit*.

c) `9-5+2` is a *list* by production (2.1), since `9-5` is a *list* and `2` is a *digit*.

□

Example 2.3: A somewhat different sort of list is the list of parameters in a function call. In Java, the parameters are enclosed within parentheses, as in the call `max(x,y)` of function `max` with parameters `x` and `y`. One nuance of such lists is that an empty list of parameters may be found between the terminals `(` and `)`. We may start to develop a grammar for such sequences with the productions:

[2] Technically, ϵ can be a string of zero symbols from any alphabet (collection of symbols).

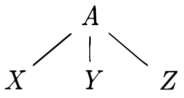
$$\begin{array}{rcl} \mathit{call} & \rightarrow & \textbf{id} \; \texttt{(} \; \mathit{optparams} \; \texttt{)} \\ \mathit{optparams} & \rightarrow & \mathit{params} \mid \epsilon \\ \mathit{params} & \rightarrow & \mathit{params} \; \texttt{,} \; \mathit{param} \mid \mathit{param} \end{array}$$

Note that the second possible body for *optparams* ("optional parameter list") is ϵ, which stands for the empty string of symbols. That is, *optparams* can be replaced by the empty string, so a *call* can consist of a function name followed by the two-terminal string `()`. Notice that the productions for *params* are analogous to those for *list* in Example 2.1, with comma in place of the arithmetic operator `+` or `-`, and *param* in place of *digit*. We have not shown the productions for *param*, since parameters are really arbitrary expressions. Shortly, we shall discuss the appropriate productions for the various language constructs, such as expressions, statements, and so on. □

Parsing is the problem of taking a string of terminals and figuring out how to derive it from the start symbol of the grammar, and if it cannot be derived from the start symbol of the grammar, then reporting syntax errors within the string. Parsing is one of the most fundamental problems in all of compiling; the main approaches to parsing are discussed in Chapter 4. In this chapter, for simplicity, we begin with source programs like `9-5+2` in which each character is a terminal; in general, a source program has multicharacter lexemes that are grouped by the lexical analyzer into tokens, whose first components are the terminals processed by the parser.

2.2.3 Parse Trees

A parse tree pictorially shows how the start symbol of a grammar derives a string in the language. If nonterminal A has a production $A \rightarrow XYZ$, then a parse tree may have an interior node labeled A with three children labeled X, Y, and Z, from left to right:



Formally, given a context-free grammar, a *parse tree* according to the grammar is a tree with the following properties:

1. The root is labeled by the start symbol.

2. Each leaf is labeled by a terminal or by ϵ.

3. Each interior node is labeled by a nonterminal.

4. If A is the nonterminal labeling some interior node and $X_1, X_2, \ldots, X_n$ are the labels of the children of that node from left to right, then there must be a production $A \rightarrow X_1 X_2 \cdots X_n$. Here, $X_1, X_2, \ldots, X_n$ each stand

Tree Terminology

Tree data structures figure prominently in compiling.

- A tree consists of one or more *nodes*. Nodes may have *labels*, which in this book typically will be grammar symbols. When we draw a tree, we often represent the nodes by these labels only.

- Exactly one node is the *root*. All nodes except the root have a unique *parent*; the root has no parent. When we draw trees, we place the parent of a node above that node and draw an edge between them. The root is then the highest (top) node.

- If node N is the parent of node M, then M is a *child* of N. The children of one node are called *siblings*. They have an order, *from the left*, and when we draw trees, we order the childen of a given node in this manner.

- A node with no children is called a *leaf*. Other nodes — those with one or more children — are *interior nodes*.

- A *descendant* of a node N is either N itself, a child of N, a child of a child of N, and so on, for any number of levels. We say node N is an *ancestor* of node M if M is a descendant of N.

for a symbol that is either a terminal or a nonterminal. As a special case, if $A \rightarrow \epsilon$ is a production, then a node labeled A may have a single child labeled ϵ.

Example 2.4: The derivation of 9-5+2 in Example 2.2 is illustrated by the tree in Fig. 2.5. Each node in the tree is labeled by a grammar symbol. An interior node and its children correspond to a production; the interior node corresponds to the head of the production, the children to the body.

In Fig. 2.5, the root is labeled *list*, the start symbol of the grammar in Example 2.1. The children of the root are labeled, from left to right, *list*, +, and *digit*. Note that

$$list \rightarrow list + digit$$

is a production in the grammar of Example 2.1. The left child of the root is similar to the root, with a child labeled - instead of +. The three nodes labeled *digit* each have one child that is labeled by a digit. □

From left to right, the leaves of a parse tree form the *yield* of the tree, which is the string *generated* or *derived* from the nonterminal at the root of the parse

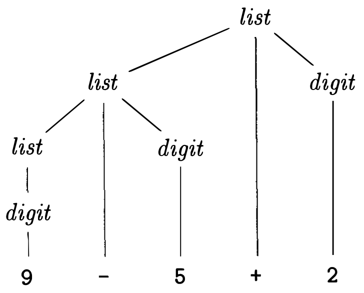


Figure 2.5: Parse tree for 9-5+2 according to the grammar in Example 2.1

tree. In Fig. 2.5, the yield is 9-5+2; for convenience, all the leaves are shown at the bottom level. Henceforth, we shall not necessarily line up the leaves in this way. Any tree imparts a natural left-to-right order to its leaves, based on the idea that if X and Y are two children with the same parent, and X is to the left of Y, then all descendants of X are to the left of descendants of Y.

Another definition of the language generated by a grammar is as the set of strings that can be generated by some parse tree. The process of finding a parse tree for a given string of terminals is called *parsing* that string.

2.2.4 Ambiguity

We have to be careful in talking about *the* structure of a string according to a grammar. A grammar can have more than one parse tree generating a given string of terminals. Such a grammar is said to be *ambiguous.* To show that a grammar is ambiguous, all we need to do is find a terminal string that is the yield of more than one parse tree. Since a string with more than one parse tree usually has more than one meaning, we need to design unambiguous grammars for compiling applications, or to use ambiguous grammars with additional rules to resolve the ambiguities.

Example 2.5 : Suppose we used a single nonterminal *string* and did not distinguish between digits and lists, as in Example 2.1. We could have written the grammar

$$\mathit{string} \rightarrow \mathit{string} + \mathit{string} \mid \mathit{string} - \mathit{string} \mid 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9$$

Merging the notion of *digit* and *list* into the nonterminal *string* makes superficial sense, because a single *digit* is a special case of a *list.*

However, Fig. 2.6 shows that an expression like 9-5+2 has more than one parse tree with this grammar. The two trees for 9-5+2 correspond to the two ways of parenthesizing the expression: (9-5)+2 and 9-(5+2). This second parenthesization gives the expression the unexpected value 2 rather than the customary value 6. The grammar of Example 2.1 does not permit this interpretation. □

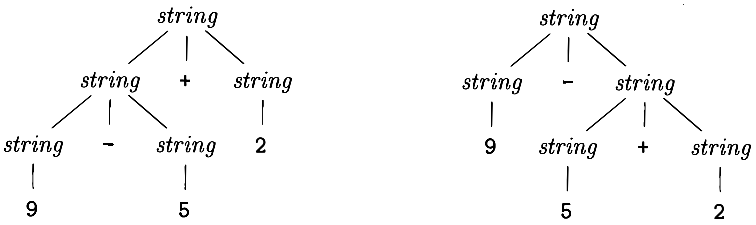


Figure 2.6: Two parse trees for 9-5+2

2.2.5 Associativity of Operators

By convention, 9+5+2 is equivalent to (9+5)+2 and 9-5-2 is equivalent to (9-5)-2. When an operand like 5 has operators to its left and right, conventions are needed for deciding which operator applies to that operand. We say that the operator + *associates* to the left, because an operand with plus signs on both sides of it belongs to the operator to its left. In most programming languages the four arithmetic operators, addition, subtraction, multiplication, and division are left-associative.

Some common operators such as exponentiation are right-associative. As another example, the assignment operator = in C and its descendants is right-associative; that is, the expression a=b=c is treated in the same way as the expression a=(b=c).

Strings like a=b=c with a right-associative operator are generated by the following grammar:

$$\begin{array}{lll} right & \rightarrow & letter = right \mid letter \\ letter & \rightarrow & \texttt{a} \mid \texttt{b} \mid \cdots \mid \texttt{z} \end{array}$$

The contrast between a parse tree for a left-associative operator like - and a parse tree for a right-associative operator like = is shown by Fig. 2.7. Note that the parse tree for 9-5-2 grows down towards the left, whereas the parse tree for a=b=c grows down towards the right.

2.2.6 Precedence of Operators

Consider the expression 9+5*2. There are two possible interpretations of this expression: (9+5)*2 or 9+(5*2). The associativity rules for + and * apply to occurrences of the same operator, so they do not resolve this ambiguity. Rules defining the relative precedence of operators are needed when more than one kind of operator is present.

We say that * has *higher precedence* than + if * takes its operands before + does. In ordinary arithmetic, multiplication and division have higher precedence than addition and subtraction. Therefore, 5 is taken by * in both 9+5*2 and 9*5+2; i.e., the expressions are equivalent to 9+(5*2) and (9*5)+2, respectively.

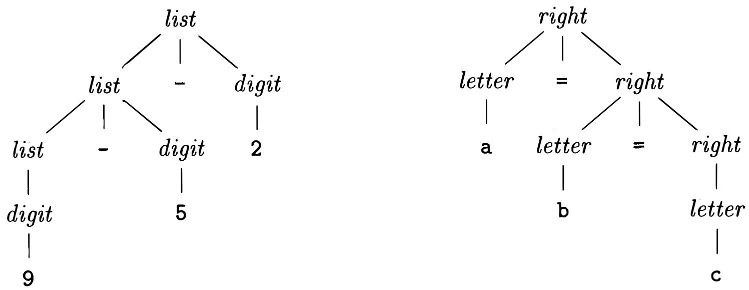


Figure 2.7: Parse trees for left- and right-associative grammars

Example 2.6: A grammar for arithmetic expressions can be constructed from a table showing the associativity and precedence of operators. We start with the four common arithmetic operators and a precedence table, showing the operators in order of increasing precedence. Operators on the same line have the same associativity and precedence:

left-associative: + -
left-associative: * /

We create two nonterminals *expr* and *term* for the two levels of precedence, and an extra nonterminal *factor* for generating basic units in expressions. The basic units in expressions are presently digits and parenthesized expressions.

$$factor \rightarrow \textbf{digit} \mid (\ expr\ )$$

Now consider the binary operators, * and /, that have the highest precedence. Since these operators associate to the left, the productions are similar to those for lists that associate to the left.

$$\begin{array}{rcl} term & \rightarrow & term * factor \\ & \mid & term\ /\ factor \\ & \mid & factor \end{array}$$

Similarly, *expr* generates lists of terms separated by the additive operators.

$$\begin{array}{rcl} expr & \rightarrow & expr + term \\ & \mid & expr - term \\ & \mid & term \end{array}$$

The resulting grammar is therefore

$$\begin{array}{rcl} expr & \rightarrow & expr + term \mid expr - term \mid term \\ term & \rightarrow & term * factor \mid term\ /\ factor \mid factor \\ factor & \rightarrow & \textbf{digit} \mid (\ expr\ ) \end{array}$$

Generalizing the Expression Grammar of Example 2.6

We can think of a factor as an expression that cannot be "torn apart" by any operator. By "torn apart," we mean that placing an operator next to any factor, on either side, does not cause any piece of the factor, other than the whole, to become an operand of that operator. If the factor is a parenthesized expression, the parentheses protect against such "tearing," while if the factor is a single operand, it cannot be torn apart.

A term (that is not also a factor) is an expression that can be torn apart by operators of the highest precedence: $*$ and /, but not by the lower-precedence operators. An expression (that is not a term or factor) can be torn apart by any operator.

We can generalize this idea to any number n of precedence levels. We need $n+1$ nonterminals. The first, like *factor* in Example 2.6, can never be torn apart. Typically, the production bodies for this nonterminal are only single operands and parenthesized expressions. Then, for each precedence level, there is one nonterminal representing expressions that can be torn apart only by operators at that level or higher. Typically, the productions for this nonterminal have bodies representing uses of the operators at that level, plus one body that is just the nonterminal for the next higher level.

With this grammar, an expression is a list of terms separated by either + or - signs, and a term is a list of factors separated by $*$ or / signs. Notice that any parenthesized expression is a factor, so with parentheses we can develop expressions that have arbitrarily deep nesting (and arbitrarily deep trees). □

Example 2.7: Keywords allow us to recognize statements, since most statement begin with a keyword or a special character. Exceptions to this rule include assignments and procedure calls. The statements defined by the (ambiguous) grammar in Fig. 2.8 are legal in Java.

In the first production for *stmt*, the terminal **id** represents any identifier. The productions for *expression* are not shown. The assignment statements specified by the first production are legal in Java, although Java treats = as an assignment operator that can appear within an expression. For example, Java allows a = b = c, which this grammar does not.

The nonterminal *stmts* generates a possibly empty list of statements. The second production for *stmts* generates the empty list ϵ. The first production generates a possibly empty list of statements followed by a statement.

The placement of semicolons is subtle; they appear at the end of every body that does not end in *stmt*. This approach prevents the build-up of semicolons after statements such as if- and while-, which end with nested substatements. When the nested substatement is an assignment or a do-while, a semicolon will be generated as part of the substatement. □

$$
\begin{array}{lcl}
\mathit{stmt} & \rightarrow & \textbf{id} = \mathit{expression}\ ; \\
& | & \textbf{if}\ (\ \mathit{expression}\ )\ \mathit{stmt} \\
& | & \textbf{if}\ (\ \mathit{expression}\ )\ \mathit{stmt}\ \textbf{else}\ \mathit{stmt} \\
& | & \textbf{while}\ (\ \mathit{expression}\ )\ \mathit{stmt} \\
& | & \textbf{do}\ \mathit{stmt}\ \textbf{while}\ (\ \mathit{expression}\ )\ ; \\
& | & \{\ \mathit{stmts}\ \} \\
\\
\mathit{stmts} & \rightarrow & \mathit{stmts}\ \mathit{stmt} \\
& | & \epsilon
\end{array}
$$

Figure 2.8: A grammar for a subset of Java statements

2.2.7 Exercises for Section 2.2

Exercise 2.2.1: Consider the context-free grammar

$$S \ \rightarrow\ S\ S\ +\ \mid\ S\ S\ *\ \mid\ \mathbf{a}$$

a) Show how the string aa+a* can be generated by this grammar.

b) Construct a parse tree for this string.

c) What language does this grammar generate? Justify your answer.

Exercise 2.2.2: What language is generated by the following grammars? In each case justify your answer.

a) $S \ \rightarrow\ 0\ S\ 1\ \mid\ 0\ 1$

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

c) $S \ \rightarrow\ S\ (\ S\ )\ S\ \mid\ \epsilon$

d) $S \ \rightarrow\ \mathbf{a}\ S\ \mathbf{b}\ S\ \mid\ \mathbf{b}\ S\ \mathbf{a}\ S\ \mid\ \epsilon$

e) $S \ \rightarrow\ \mathbf{a}\ \mid\ S + S\ \mid\ S\ S\ \mid S\ *\ \mid (\ S\ )$

Exercise 2.2.3: Which of the grammars in Exercise 2.2.2 are ambiguous?

Exercise 2.2.4: Construct unambiguous context-free grammars for each of the following languages. In each case show that your grammar is correct.

a) Arithmetic expressions in postfix notation.

b) Left-associative lists of identifiers separated by commas.

c) Right-associative lists of identifiers separated by commas.

d) Arithmetic expressions of integers and identifiers with the four binary operators +, -, *, /.

! e) Add unary plus and minus to the arithmetic operators of (d).

Exercise 2.2.5:

a) Show that all binary strings generated by the following grammar have values divisible by 3. *Hint.* Use induction on the number of nodes in a parse tree.

$$num \;\rightarrow\; \texttt{11} \;|\; 1001 \;|\; num\ \texttt{0} \;|\; num\ num$$

b) Does the grammar generate all binary strings with values divisible by 3?

Exercise 2.2.6: Construct a context-free grammar for roman numerals.

2.3 Syntax-Directed Translation

Syntax-directed translation is done by attaching rules or program fragments to productions in a grammar. For example, consider an expression *expr* generated by the production

$$expr \;\rightarrow\; expr_1 \texttt{ + } term$$

Here, *expr* is the sum of the two subexpressions $expr_1$ and *term*. (The subscript in $expr_1$ is used only to distinguish the instance of *expr* in the production body from the head of the production). We can translate *expr* by exploiting its structure, as in the following pseudo-code:

```
translate $expr_1$;
translate $term$;
handle +;
```

Using a variant of this pseudocode, we shall build a syntax tree for *expr* in Section 2.8 by building syntax trees for $expr_1$ and *term* and then handling + by constructing a node for it. For convenience, the example in this section is the translation of infix expressions into postfix notation.

This section introduces two concepts related to syntax-directed translation:

- *Attributes.* An *attribute* is any quantity associated with a programming construct. Examples of attributes are data types of expressions, the number of instructions in the generated code, or the location of the first instruction in the generated code for a construct, among many other possibilities. Since we use grammar symbols (nonterminals and terminals) to represent programming constructs, we extend the notion of attributes from constructs to the symbols that represent them.

- (*Syntax-directed*) *translation schemes.* A *translation scheme* is a notation for attaching program fragments to the productions of a grammar. The program fragments are executed when the production is used during syntax analysis. The combined result of all these fragment executions, in the order induced by the syntax analysis, produces the translation of the program to which this analysis/synthesis process is applied.

Syntax-directed translations will be used throughout this chapter to translate infix expressions into postfix notation, to evaluate expressions, and to build syntax trees for programming constructs. A more detailed discussion of syntax-directed formalisms appears in Chapter 5.

2.3.1 Postfix Notation

The examples in this section deal with translation into postfix notation. The *postfix notation* for an expression E can be defined inductively as follows:

1. If E is a variable or constant, then the postfix notation for E is E itself.

2. If E is an expression of the form E_1 **op** E_2, where **op** is any binary operator, then the postfix notation for E is E'_1 E'_2 **op**, where E'_1 and E'_2 are the postfix notations for E_1 and E_2, respectively.

3. If E is a parenthesized expression of the form (E_1), then the postfix notation for E is the same as the postfix notation for E_1.

Example 2.8: The postfix notation for `(9-5)+2` is `95-2+`. That is, the translations of `9`, `5`, and `2` are the constants themselves, by rule (1). Then, the translation of `9-5` is `95-` by rule (2). The translation of `(9-5)` is the same by rule (3). Having translated the parenthesized subexpression, we may apply rule (2) to the entire expression, with `(9-5)` in the role of E_1 and `2` in the role of E_2, to get the result `95-2+`.

As another example, the postfix notation for `9-(5+2)` is `952+-`. That is, `5+2` is first translated into `52+`, and this expression becomes the second argument of the minus sign. □

No parentheses are needed in postfix notation, because the position and *arity* (number of arguments) of the operators permits only one decoding of a postfix expression. The "trick" is to repeatedly scan the postfix string from the left, until you find an operator. Then, look to the left for the proper number of operands, and group this operator with its operands. Evaluate the operator on the operands, and replace them by the result. Then repeat the process, continuing to the right and searching for another operator.

Example 2.9: Consider the postfix expression `952+-3*`. Scanning from the left, we first encounter the plus sign. Looking to its left we find operands `5` and `2`. Their sum, 7, replaces `52+`, and we have the string `97-3*`. Now, the leftmost operator is the minus sign, and its operands are 9 and 7. Replacing these by the result of the subtraction leaves 23*. Last, the multiplication sign applies to 2 and 3, giving the result 6. □

2.3.2 Synthesized Attributes

The idea of associating quantities with programming constructs—for example, values and types with expressions—can be expressed in terms of grammars. We associate attributes with nonterminals and terminals. Then, we attach rules to the productions of the grammar; these rules describe how the attributes are computed at those nodes of the parse tree where the production in question is used to relate a node to its children.

A *syntax-directed definition* associates

1. With each grammar symbol, a set of attributes, and

2. With each production, a set of *semantic rules* for computing the values of the attributes associated with the symbols appearing in the production.

Attributes can be evaluated as follows. For a given input string x, construct a parse tree for x. Then, apply the semantic rules to evaluate attributes at each node in the parse tree, as follows.

Suppose a node N in a parse tree is labeled by the grammar symbol X. We write $X.a$ to denote the value of attribute a of X at that node. A parse tree showing the attribute values at each node is called an *annotated* parse tree. For example, Fig. 2.9 shows an annotated parse tree for 9-5+2 with an attribute t associated with the nonterminals *expr* and *term*. The value 95-2+ of the attribute at the root is the postfix notation for 9-5+2. We shall see shortly how these expressions are computed.

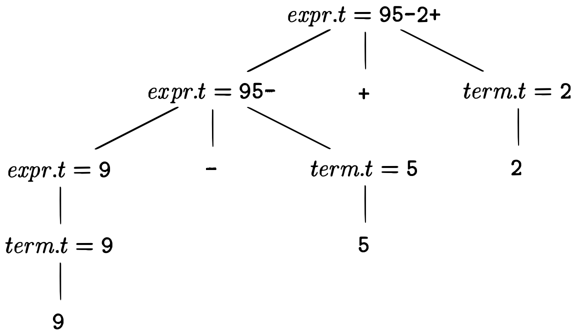


Figure 2.9: Attribute values at nodes in a parse tree

An attribute is said to be *synthesized* if its value at a parse-tree node N is determined from attribute values at the children of N and at N itself. Synthesized attributes have the desirable property that they can be evaluated during a single bottom-up traversal of a parse tree. In Section 5.1.1 we shall discuss another important kind of attribute: the "inherited" attribute. Informally, inherited attributes have their value at a parse-tree node determined from attribute values at the node itself, its parent, and its siblings in the parse tree.

Example 2.10: The annotated parse tree in Fig. 2.9 is based on the syntax-directed definition in Fig. 2.10 for translating expressions consisting of digits separated by plus or minus signs into postfix notation. Each nonterminal has a string-valued attribute t that represents the postfix notation for the expression generated by that nonterminal in a parse tree. The symbol $||$ in the semantic rule is the operator for string concatenation.

PRODUCTION	SEMANTIC RULES
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \;\|\|\; term.t \;\|\|\; '+'$
$expr \rightarrow expr_1 - term$	$expr.t = expr_1.t \;\|\|\; term.t \;\|\|\; '-'$
$expr \rightarrow term$	$expr.t = term.t$
$term \rightarrow 0$	$term.t = '0'$
$term \rightarrow 1$	$term.t = '1'$
...	...
$term \rightarrow 9$	$term.t = '9'$

Figure 2.10: Syntax-directed definition for infix to postfix translation

The postfix form of a digit is the digit itself; e.g., the semantic rule associated with the production $term \rightarrow 9$ defines $term.t$ to be 9 itself whenever this production is used at a node in a parse tree. The other digits are translated similarly. As another example, when the production $expr \rightarrow term$ is applied, the value of $term.t$ becomes the value of $expr.t$.

The production $expr \rightarrow expr_1 + term$ derives an expression containing a plus operator.[3] The left operand of the plus operator is given by $expr_1$ and the right operand by $term$. The semantic rule

$$expr.t = expr_1.t \;||\; term.t \;||\; '+'$$

associated with this production constructs the value of attribute $expr.t$ by concatenating the postfix forms $expr_1.t$ and $term.t$ of the left and right operands, respectively, and then appending the plus sign. This rule is a formalization of the definition of "postfix expression." □

[3] In this and many other rules, the same nonterminal ($expr$, here) appears several times. The purpose of the subscript 1 in $expr_1$ is to distinguish the two occurrences of $expr$ in the production; the "1" is not part of the nonterminal. See the box on "Convention Distinguishing Uses of a Nonterminal" for more details.

Convention Distinguishing Uses of a Nonterminal

In rules, we often have a need to distinguish among several uses of the same nonterminal in the head and/or body of a production; e.g., see Example 2.10. The reason is that in the parse tree, different nodes labeled by the same nonterminal usually have different values for their translations. We shall adopt the following convention: the nonterminal appears unsubscripted in the head and with distinct subscripts in the body. These are all occurrences of the same nonterminal, and the subscript is not part of its name. However, the reader should be alert to the difference between examples of specific translations, where this convention is used, and generic productions like $A \rightarrow X_1 X_2, \dots, X_n$, where the subscripted X's represent an arbitrary list of grammar symbols, and are *not* instances of one particular nonterminal called X.

2.3.3 Simple Syntax-Directed Definitions

The syntax-directed definition in Example 2.10 has the following important property: the string representing the translation of the nonterminal at the head of each production is the concatenation of the translations of the nonterminals in the production body, in the same order as in the production, with some optional additional strings interleaved. A syntax-directed definition with this property is termed *simple.*

Example 2.11 : Consider the first production and semantic rule from Fig. 2.10:

PRODUCTION	SEMANTIC RULE	
$expr \rightarrow expr_1 + term$	$expr.t = expr_1.t \mathbin{\|\|} term.t \mathbin{\|\|} '+'$	(2.5)

Here the translation *expr.t* is the concatenation of the translations of $expr_1$ and *term*, followed by the symbol +. Notice that $expr_1$ and *term* appear in the same order in both the production body and the semantic rule. There are no additional symbols before or between their translations. In this example, the only extra symbol occurs at the end. □

When translation schemes are discussed, we shall see that a simple syntax-directed definition can be implemented by printing only the additional strings, in the order they appear in the definition.

2.3.4 Tree Traversals

Tree traversals will be used for describing attribute evaluation and for specifying the execution of code fragments in a translation scheme. A *traversal* of a tree starts at the root and visits each node of the tree in some order.

A *depth-first* traversal starts at the root and recursively visits the children of each node in any order, not necessarily from left to right. It is called "depth-first" because it visits an unvisited child of a node whenever it can, so it visits nodes as far away from the root (as "deep") as quickly as it can.

The procedure *visit*(N) in Fig. 2.11 is a depth first traversal that visits the children of a node in left-to-right order, as shown in Fig. 2.12. In this traversal, we have included the action of evaluating translations at each node, just before we finish with the node (that is, after translations at the children have surely been computed). In general, the actions associated with a traversal can be whatever we choose, or nothing at all.

```
procedure visit(node N) {
        for ( each child C of N, from left to right ) {
                visit(C);
        }
        evaluate semantic rules at node N;
}
```

Figure 2.11: A depth-first traversal of a tree

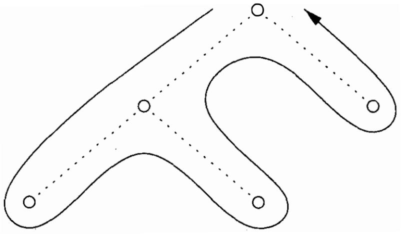


Figure 2.12: Example of a depth-first traversal of a tree

A syntax-directed definition does not impose any specific order for the evaluation of attributes on a parse tree; any evaluation order that computes an attribute a after all the other attributes that a depends on is acceptable. Synthesized attributes can be evaluated during any *bottom-up* traversal, that is, a traversal that evaluates attributes at a node after having evaluated attributes at its children. In general, with both synthesized and inherited attributes, the matter of evaluation order is quite complex; see Section 5.2.

2.3.5 Translation Schemes

The syntax-directed definition in Fig. 2.10 builds up a translation by attaching strings as attributes to the nodes in the parse tree. We now consider an alternative approach that does not need to manipulate strings; it produces the same translation incrementally, by executing program fragments.

Preorder and Postorder Traversals

Preorder and postorder traversals are two important special cases of depth-first traversals in which we visit the children of each node from left to right.

Often, we traverse a tree to perform some particular action at each node. If the action is done when we first visit a node, then we may refer to the traversal as a *preorder traversal.* Similarly, if the action is done just before we leave a node for the last time, then we say it is a *postorder traversal* of the tree. The procedure *visit*(N) in Fig. 2.11 is an example of a postorder traversal.

Preorder and postorder traversals define corresponding orderings on nodes, based on when the action at a node would be performed. The *preorder* of a (sub)tree rooted at node N consists of N, followed by the preorders of the subtrees of each of its children, if any, from the left. The *postorder* of a (sub)tree rooted at N consists of the postorders of each of the subtrees for the children of N, if any, from the left, followed by N itself.

A syntax-directed translation scheme is a notation for specifying a translation by attaching program fragments to productions in a grammar. A translation scheme is like a syntax-directed definition, except that the order of evaluation of the semantic rules is explicitly specified.

Program fragments embedded within production bodies are called *semantic actions.* The position at which an action is to be executed is shown by enclosing it between curly braces and writing it within the production body, as in

$$rest \;\rightarrow\; +\; term\; \{\text{print}('+')\}\; rest_1$$

We shall see such rules when we consider an alternative form of grammar for expressions, where the nonterminal *rest* represents "everything but the first term of an expression." This form of grammar is discussed in Section 2.4.5. Again, the subscript in $rest_1$ distinguishes this instance of nonterminal *rest* in the production body from the instance of *rest* at the head of the production.

When drawing a parse tree for a translation scheme, we indicate an action by constructing an extra child for it, connected by a dashed line to the node that corresponds to the head of the production. For example, the portion of the parse tree for the above production and action is shown in Fig. 2.13. The node for a semantic action has no children, so the action is performed when that node is first seen.

Example 2.12: The parse tree in Fig. 2.14 has print statements at extra leaves, which are attached by dashed lines to interior nodes of the parse tree. The translation scheme appears in Fig. 2.15. The underlying grammar generates expressions consisting of digits separated by plus and minus signs. The

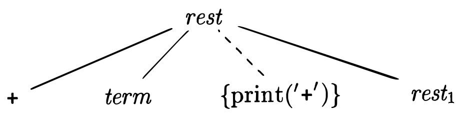


Figure 2.13: An extra leaf is constructed for a semantic action

actions embedded in the production bodies translate such expressions into postfix notation, provided we perform a left-to-right depth-first traversal of the tree and execute each print statement when we visit its leaf.

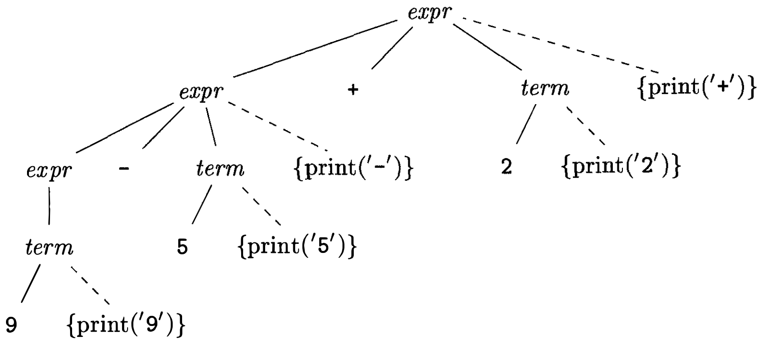


Figure 2.14: Actions translating 9-5+2 into 95-2+

expr	→	$expr_1$ + *term*	{print('+')}
expr	→	$expr_1$ - *term*	{print('-')}
expr	→	*term*	
term	→	0	{print('0')}
term	→	1	{print('1')}
		. . .	
term	→	9	{print('9')}

Figure 2.15: Actions for translating into postfix notation

The root of Fig. 2.14 represents the first production in Fig. 2.15. In a postorder traversal, we first perform all the actions in the leftmost subtree of the root, for the left operand, also labeled *expr* like the root. We then visit the leaf + at which there is no action. We next perform the actions in the subtree for the right operand *term* and, finally, the semantic action { print('+') } at the extra node.

Since the productions for *term* have only a digit on the right side, that digit is printed by the actions for the productions. No output is necessary for the production *expr* → *term*, and only the operator needs to be printed in the 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 N, 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

$$\begin{array}{lcl} stmt & \rightarrow & \textbf{expr} \; ; \\ & | & \textbf{if} \; (\; \textbf{expr} \;) \; stmt \\ & | & \textbf{for} \; (\; optexpr \; ; \; optexpr \; ; \; optexpr \;) \; stmt \\ & | & \textbf{other} \\ & & \\ optexpr & \rightarrow & \epsilon \\ & | & \textbf{expr} \end{array}$$

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

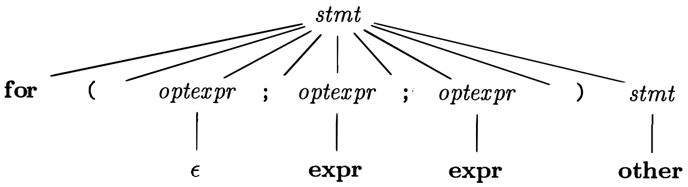


Figure 2.17: A parse tree according to the grammar in Fig. 2.16

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

$$\textbf{for} \; (\; ; \; \textbf{expr} \; ; \; \textbf{expr} \;) \; \textbf{other}$$

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

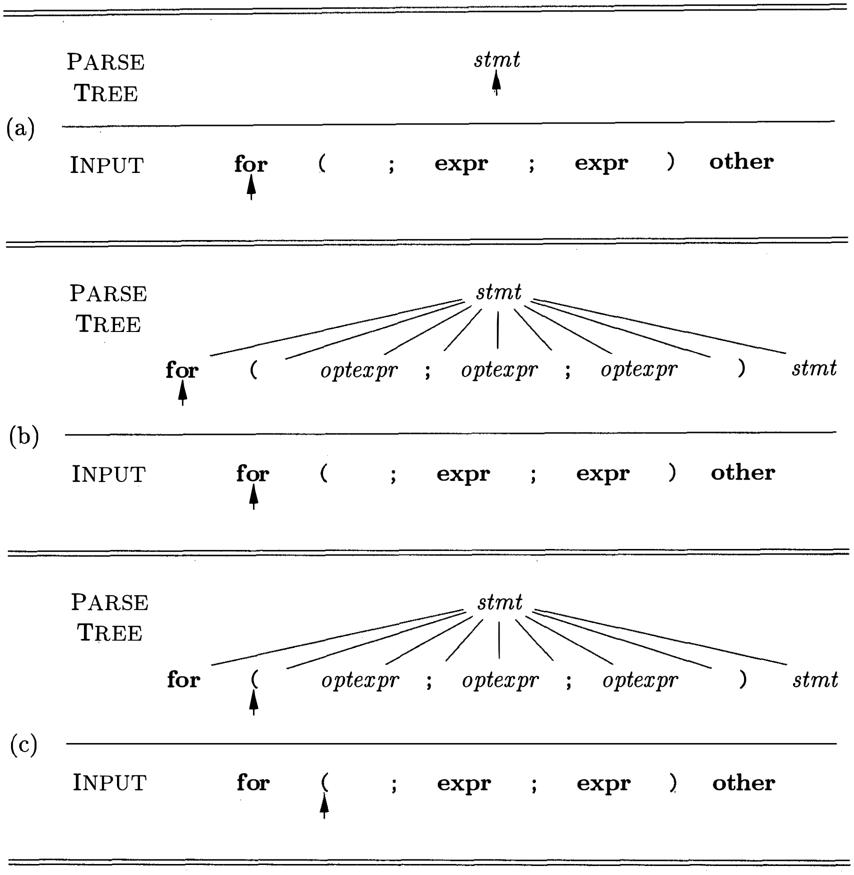

(a)
PARSE TREE
stmt
INPUT
for (; expr ; expr) other
(b)
PARSE TREE
stmt
for (optexpr ; optexpr ; optexpr) stmt
INPUT
for (; expr ; expr) other
(c)
PARSE TREE
stmt
for (optexpr ; optexpr ; optexpr) stmt
INPUT
for (; expr ; expr) other

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 ϵ as the body ("ϵ-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 ϵ-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

$$stmt \rightarrow \textbf{for} \texttt{ (} optexpr \texttt{ ; } optexpr \texttt{ ; } optexpr \texttt{) } stmt$$

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:

$match(\textbf{for})$; $match('\texttt{(}')$;
$optexpr()$; $match('\texttt{;}')$; $optexpr()$; $match('\texttt{;}')$; $optexpr()$;
$match('\texttt{)}')$; $stmt()$;

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

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

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(if); match('('); match(expr); match(')'); stmt();
                break;
        case for:
                match(for); match('(');
                optexpr(); match(';'); optexpr(); match(';'); optexpr();
                match(')'); stmt(); break;
        case other;
                match(other); break;
        default:
                report("syntax error");
        }
}

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

void match(terminal t) {
        if ( lookahead == t ) lookahead = nextTerminal;
        else report("syntax error");
}
```

Figure 2.19: Pseudocode for a predictive parser

$$\begin{aligned} \text{FIRST}(\mathit{stmt}) &= \{\textbf{expr},\ \textbf{if},\ \textbf{for},\ \textbf{other}\} \\ \text{FIRST}(\textbf{expr}\ ;) &= \{\textbf{expr}\} \end{aligned}$$

The FIRST sets must be considered if there are two productions $A \rightarrow \alpha$ and $A \rightarrow \beta$. Ignoring ϵ-productions for the moment, predictive parsing requires FIRST(α) and FIRST(β) to be disjoint. The lookahead symbol can then be used to decide which production to use; if the lookahead symbol is in FIRST(α), then α is used. Otherwise, if the lookahead symbol is in FIRST(β), then β is used.

2.4.3 When to Use ϵ-Productions

Our predictive parser uses an ϵ-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 ϵ. 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 ϵ-production.

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

$$\begin{array}{lcl} optexpr & \rightarrow & expr \\ & | & \epsilon \end{array}$$

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

For more on when to use ϵ-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 α (where α is not ϵ, the empty string) is used if the lookahead symbol is in FIRST(α). 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 ϵ-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

$$expr \quad \rightarrow \quad expr + term$$

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

$$A \rightarrow A\alpha \mid \beta$$

where α and β are sequences of terminals and nonterminals that do not start with A. For example, in

$$expr \quad \rightarrow \quad expr + term \;\mid\; term$$

nonterminal $A = expr$, string $\alpha = + \, term$, and string $\beta = term$.

The nonterminal A and its production are said to be *left recursive*, because the production $A \rightarrow A\alpha$ has A itself as the leftmost symbol on the right side.[4] Repeated application of this production builds up a sequence of α's to the right of A, as in Fig. 2.20(a). When A is finally replaced by β, we have a β followed by a sequence of zero or more α'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 $A \rightarrow A\alpha$, the nonterminal A may derive $A\alpha$ through intermediate productions.

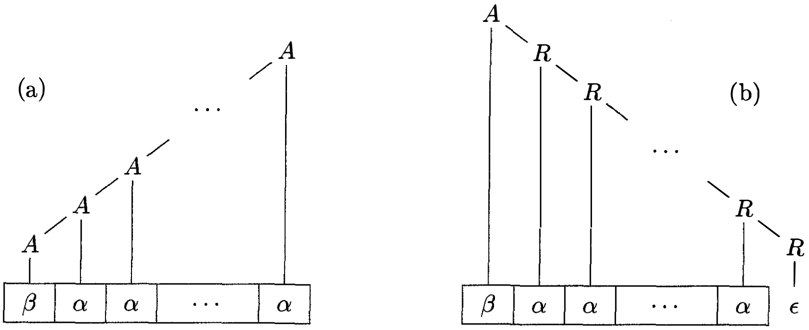


Figure 2.20: Left- and right-recursive ways of generating a string

$$\begin{array}{lcl} A & \rightarrow & \beta R \\ R & \rightarrow & \alpha R \mid \epsilon \end{array}$$

Nonterminal R and its production $R \rightarrow \alpha R$ are *right 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) $S \rightarrow$ + S S | - S S | **a**

b) $S \rightarrow$ S (S) S | ϵ

c) $S \rightarrow$ 0 S 1 | 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 translation of expressions in detail, since they appear as a construct in so many languages.

A syntax-directed translation scheme often serves as the specification for a translator. The scheme in Fig. 2.21 (repeated from Fig. 2.15) defines the translation to be performed here.

$$\begin{array}{lcll}
\mathit{expr} & \rightarrow & \mathit{expr} + \mathit{term} & \{\ \text{print}('+')\ \} \\
 & | & \mathit{expr} - \mathit{term} & \{\ \text{print}('-')\ \} \\
 & | & \mathit{term} & \\
 & & & \\
\mathit{term} & \rightarrow & 0 & \{\ \text{print}('0')\ \} \\
 & | & 1 & \{\ \text{print}('1')\ \} \\
 & & \cdots & \\
 & | & 9 & \{\ \text{print}('9')\ \}
\end{array}$$

Figure 2.21: Actions for translating into postfix notation

Often, the underlying grammar of a given scheme has to be modified before it can be parsed with a predictive parser. In particular, the grammar underlying the scheme in Fig. 2.21 is left recursive, and as we saw in the last section, a predictive parser cannot handle a left-recursive grammar.

We appear to have a conflict: on the one hand we need a grammar that facilitates translation, on the other hand we need a significantly different grammar that facilitates parsing. The solution is to begin with the grammar for easy translation and carefully transform it to facilitate parsing. By eliminating the left recursion in Fig. 2.21, we can obtain a grammar suitable for use in a predictive recursive-descent translator.

2.5.1 Abstract and Concrete Syntax

A useful starting point for designing a translator is a data structure called an abstract syntax tree. In an *abstract syntax tree* for an expression, each interior node represents an operator; the children of the node represent the operands of the operator. More generally, any programming construct can be handled by making up an operator for the construct and treating as operands the semantically meaningful components of that construct.

In the abstract syntax tree for 9-5+2 in Fig. 2.22, the root represents the operator +. The subtrees of the root represent the subexpressions 9-5 and 2. The grouping of 9-5 as an operand reflects the left-to-right evaluation of operators at the same precedence level. Since - and + have the same precedence, 9-5+2 is equivalent to (9-5)+2.

Abstract syntax trees, or simply *syntax trees*, resemble parse trees to an extent. However, in the syntax tree, interior nodes represent programming constructs while in the parse tree, the interior nodes represent nonterminals. Many nonterminals of a grammar represent programming constructs, but others

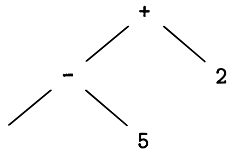


Figure 2.22: Syntax tree for 9-5+2

are "helpers" of one sort of another, such as those representing terms, factors, or other variations of expressions. In the syntax tree, these helpers typically are not needed and are hence dropped. To emphasize the contrast, a parse tree is sometimes called a *concrete syntax tree*, and the underlying grammar is called a *concrete syntax* for the language.

In the syntax tree in Fig. 2.22, each interior node is associated with an operator, with no "helper" nodes for *single productions* (a production whose body consists of a single nonterminal, and nothing else) like $expr \rightarrow term$ or for ϵ-productions like $rest \rightarrow \epsilon$.

It is desirable for a translation scheme to be based on a grammar whose parse trees are as close to syntax trees as possible. The grouping of subexpressions by the grammar in Fig. 2.21 is similar to their grouping in syntax trees. For example, subexpressions of the addition operator are given by *expr* and *term* in the production body *expr* + *term*.

2.5.2 Adapting the Translation Scheme

The left-recursion-elimination technique sketched in Fig. 2.20 can also be applied to productions containing semantic actions. First, the technique extends to multiple productions for A. In our example, A is *expr*, and there are two left-recursive productions for *expr* and one that is not left recursive. The technique transforms the productions $A \rightarrow A\alpha \mid A\beta \mid \gamma$ into

$$\begin{array}{lcl} A & \rightarrow & \gamma R \\ R & \rightarrow & \alpha R \mid \beta R \mid \epsilon \end{array}$$

Second, we need to transform productions that have embedded actions, not just terminals and nonterminals. Semantic actions embedded in the productions are simply carried along in the transformation, as if they were terminals.

Example 2.13: Consider the translation scheme of Fig. 2.21. Let

$$\begin{array}{lcl} A & = & expr \\ \alpha & = & \texttt{+}\ term\ \{\ \text{print}('\texttt{+}')\ \} \\ \beta & = & \texttt{-}\ term\ \{\ \text{print}('\texttt{-}')\ \} \\ \gamma & = & term \end{array}$$

Then the left-recursion-eliminating transformation produces the translation scheme in Fig. 2.23. The *expr* productions in Fig. 2.21 have been transformed into the productions for *expr*, and a new nonterminal *rest* plays the role of R. The productions for *term* are repeated from Fig. 2.21. Figure 2.24 shows how `9-5+2` is translated using the grammar in Fig. 2.23. □

$$\begin{array}{rcl}
\mathit{expr} & \rightarrow & \mathit{term}\ \mathit{rest} \\
\\
\mathit{rest} & \rightarrow & \texttt{+}\ \mathit{term}\ \ \{\ \mathrm{print}('\texttt{+}')\ \}\ \ \mathit{rest} \\
 & | & \texttt{-}\ \mathit{term}\ \ \{\ \mathrm{print}('\texttt{-}')\ \}\ \ \mathit{rest} \\
 & | & \epsilon \\
\\
\mathit{term} & \rightarrow & \texttt{0}\ \{\ \mathrm{print}('\texttt{0}')\ \} \\
 & | & \texttt{1}\ \{\ \mathrm{print}('\texttt{1}')\ \} \\
 & & \dots \\
 & | & \texttt{9}\ \{\ \mathrm{print}('\texttt{9}')\ \}
\end{array}$$

Figure 2.23: Translation scheme after left-recursion elimination

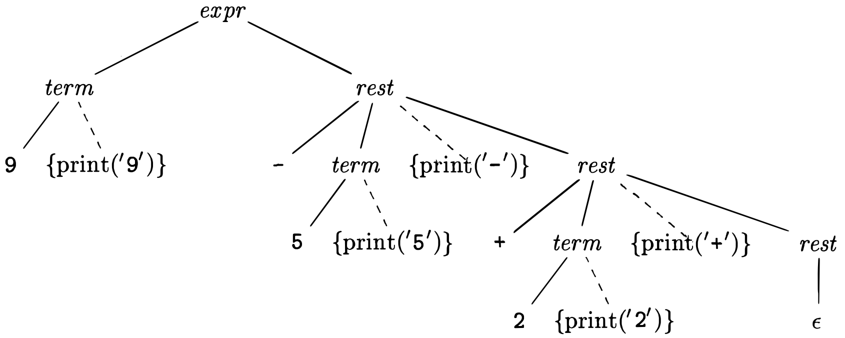


Figure 2.24: Translation of `9-5+2` to `95-2+`

Left-recursion elimination must be done carefully, to ensure that we preserve the ordering of semantic actions. For example, the transformed scheme in Fig. 2.23 has the actions { print('+') } and { print('-') } in the middle of a production body, in each case between nonterminals *term* and *rest*. If the actions were to be moved to the end, after *rest*, then the translations would become incorrect. We leave it to the reader to show that `9-5+2` would then be translated incorrectly into `952+-`, the postfix notation for `9-(5+2)`, instead of the desired `95-2+`, the postfix notation for `(9-5)+2`.

2.5.3 Procedures for the Nonterminals

Functions *expr*, *rest*, and *term* in Fig. 2.25 implement the syntax-directed translation scheme in Fig. 2.23. These functions mimic the production bodies of the corresponding nonterminals. Function *expr* implements the production *expr* → *term rest* by the calls *term*() followed by *rest*().

```
void expr() {
        term(); rest();
}

void rest() {
        if ( lookahead == '+' ) {
                match('+'); term(); print('+'); rest();
        }
        else if ( lookahead == '-' ) {
                match('-'); term(); print('-'); rest();
        }
        else { } /* do nothing with the input */ ;
}

void term() {
        if ( lookahead is a digit ) {
                t = lookahead; match(lookahead); print(t);
        }
        else report("syntax error");
}
```

Figure 2.25: Pseudocode for nonterminals *expr*, *rest*, and *term*.

Function *rest* implements the three productions for nonterminal *rest* in Fig. 2.23. It applies the first production if the lookahead symbol is a plus sign, the second production if the lookahead symbol is a minus sign, and the production *rest* → ϵ in all other cases. The first two productions for *rest* are implemented by the first two branches of the if-statement in procedure *rest*. If the lookahead symbol is +, the plus sign is matched by the call *match*('+'). After the call *term*(), the semantic action is implemented by writing a plus character. The second production is similar, with - instead of +. Since the third production for *rest* has ϵ as its right side, the last else-clause in function *rest* does nothing.

The ten productions for *term* generate the ten digits. Since each of these productions generates a digit and prints it, the same code in Fig. 2.25 implements them all. If the test succeeds, variable *t* saves the digit represented by *lookahead* so it can be written after the call to *match*. Note that *match* changes the lookahead symbol, so the digit needs to be saved for later printing.[5]

2.5.4 Simplifying the Translator

Before showing a complete program, we shall make two simplifying transformations to the code in Fig. 2.25. The simplifications will fold procedure *rest* into procedure *expr*. When expressions with multiple levels of precedence are translated, such simplifications reduce the number of procedures needed.

First, certain recursive calls can be replaced by iterations. When the last statement executed in a procedure body is a recursive call to the same procedure, the call is said to be ***tail recursive***. For example, in function *rest*, the calls of *rest*() with lookahead + and - are tail recursive because in each of these branches, the recursive call to *rest* is the last statement executed by the given call of *rest*.

For a procedure without parameters, a tail-recursive call can be replaced simply by a jump to the beginning of the procedure. The code for *rest* can be rewritten as the pseudocode of Fig. 2.26. As long as the lookahead symbol is a plus or a minus sign, procedure *rest* matches the sign, calls *term* to match a digit, and continues the process. Otherwise, it breaks out of while loop and returns from *rest*.

```
void rest() {
    while( true ) {
        if( lookahead == '+' ) {
            match('+'); term(); print('+'); continue;
        }
        else if ( lookahead == '-' ) {
            match('-'); term(); print('-'); continue;
        }
        break ;
    }
}
```

Figure 2.26: Eliminating tail recursion in the procedure *rest* of Fig. 2.25.

Second, the complete Java program will include one more change. Once the tail-recursive calls to *rest* in Fig. 2.25 are replaced by iterations, the only remaining call to *rest* is from within procedure *expr*. The two procedures can therefore be integrated into one, by replacing the call *rest*() by the body of procedure *rest*.

[5] As a minor optimization, we could print before calling *match* to avoid the need to save the digit. In general, changing the order of actions and grammar symbols is risky, since it could change what the translation does.

2.5.5 The Complete Program

The complete Java program for our translator appears in Fig. 2.27. The first line of Fig. 2.27, beginning with `import`, provides access to the package `java.io` for system input and output. The rest of the code consists of the two classes `Parser` and `Postfix`. Class `Parser` contains variable `lookahead` and functions `Parser`, `expr`, `term`, and `match`.

Execution begins with function `main`, which is defined in class `Postfix`. Function `main` creates an instance `parse` of class `Parser` and calls its function `expr` to parse an expression.

The function `Parser`, with the same name as its class, is a *constructor*; it is called automatically when an object of the class is created. Notice from its definition at the beginning of class `Parser` that the constructor `Parser` initializes variable `lookahead` by reading a token. Tokens, consisting of single characters, are supplied by the system input routine `read`, which reads the next character from the input file. Note that `lookahead` is declared to be an integer, rather than a character, to anticipate the fact that additional tokens other than single characters will be introduced in later sections.

Function `expr` is the result of the simplifications discussed in Section 2.5.4; it implements nonterminals *expr* and *rest* in Fig. 2.23. The code for `expr` in Fig. 2.27 calls `term` and then has a while-loop that forever tests whether `lookahead` matches either `'+'` or `'-'`. Control exits from this while-loop when it reaches the return statement. Within the loop, the input/output facilities of the `System` class are used to write a character.

Function `term` uses the routine `isDigit` from the Java class `Character` to test if the lookahead symbol is a digit. The routine `isDigit` expects to be applied to a character; however, `lookahead` is declared to be an integer, anticipating future extensions. The construction `(char)lookahead` *casts* or coerces `lookahead` to be a character. In a small change from Fig. 2.25, the semantic action of writing the lookahead character occurs before the call to `match`.

The function `match` checks terminals; it reads the next input terminal if the lookahead symbol is matched and signals an error otherwise by executing

```
throw new Error("syntax error");
```

This code creates a new exception of class `Error` and supplies it the string `syntax error` as an error message. Java does not require `Error` exceptions to be declared in a `throws` clause, since they are meant to be used only for abnormal events that should never occur.[6]

[6] Error handling can be streamlined using the exception-handling facilities of Java. One approach is to define a new exception, say **`SyntaxError`**, that extends the system class **`Exception`**. Then, throw **`SyntaxError`** instead of **`Error`** when an error is detected in either **`term`** or **`match`**. Further, handle the exception in **`main`** by enclosing the call **`parse.expr()`** within a **`try`** statement that catches exception **`SyntaxError`**, writes a message, and terminates. We would need to add a class **`SyntaxError`** to the program in Fig. 2.27. To complete the extension, in addition to **`IOException`**, functions **`match`** and **`term`** must now declare that they can throw **`SyntaxError`**. Function **`expr`**, which calls them, must also declare that it can throw **`SyntaxError`**.

```
import java.io.*;
class Parser {
    static int lookahead;

    public Parser() throws IOException {
        lookahead = System.in.read();
    }

    void expr() throws IOException {
        term();
        while(true) {
            if( lookahead == ’+’ ) {
                match(’+’); term(); System.out.write(’+’);
            }
            else if( lookahead == ’-’ ) {
                match(’-’); term(); System.out.write(’-’);
            }
            else return;
        }
    }

    void term() throws IOException {
        if( Character.isDigit((char)lookahead) ) {
            System.out.write((char)lookahead); match(lookahead);
        }
        else throw new Error("syntax error");
    }

    void match(int t) throws IOException {
        if( lookahead == t ) lookahead = System.in.read();
        else throw new Error("syntax error");
    }
}

public class Postfix {
    public static void main(String[] args) throws IOException {
        Parser parse = new Parser();
        parse.expr(); System.out.write(’\n’);
    }
}
```

Figure 2.27: Java program to translate infix expressions into postfix form

A Few Salient Features of Java

Those unfamiliar with Java may find the following notes on Java helpful in reading the code in Fig. 2.27:

- A class in Java consists of a sequence of variable and function definitions.
- Parentheses enclosing function parameter lists are needed even if there are no parameters; hence we write `expr()` and `term()`. These functions are actually procedures, because they do not return values, signified by the keyword `void` before the function name.
- Functions communicate either by passing parameters "by value" or by accessing shared data. For example, the functions `expr()` and `term()` examine the lookahead symbol using the class variable `lookahead` that they can all access since they all belong to the same class `Parser`.
- Like C, Java uses = for assignment, == for equality, and != for inequality.
- The clause "`throws IOException`" in the definition of `term()` declares that an exception called `IOException` can occur. Such an exception occurs if there is no input to be read when the function `match` uses the routine `read`. Any function that calls `match` must also declare that an `IOException` can occur during its own execution.

2.6 Lexical Analysis

A lexical analyzer reads characters from the input and groups them into "token objects." Along with a terminal symbol that is used for parsing decisions, a token object carries additional information in the form of attribute values. So far, there has been no need to distinguish between the terms "token" and "terminal," since the parser ignores the attribute values that are carried by a token. In this section, a token is a terminal along with additional information.

A sequence of input characters that comprises a single token is called a *lexeme*. Thus, we can say that the lexical analyzer insulates a parser from the lexeme representation of tokens.

The lexical analyzer in this section allows numbers, identifiers, and "white space" (blanks, tabs, and newlines) to appear within expressions. It can be used to extend the expression translator of the previous section. Since the expression grammar of Fig. 2.21 must be extended to allow numbers and identifiers, we shall take this opportunity to allow multiplication and division as well. The extended translation scheme appears in Fig. 2.28.

$$
\begin{array}{rcll}
\mathit{expr} & \rightarrow & \mathit{expr} + \mathit{term} & \{\ \text{print}('+')\ \} \\
 & | & \mathit{expr} - \mathit{term} & \{\ \text{print}('-')\ \} \\
 & | & \mathit{term} & \\
 & & & \\
\mathit{term} & \rightarrow & \mathit{term} * \mathit{factor} & \{\ \text{print}('*')\ \} \\
 & | & \mathit{term}\ /\ \mathit{factor} & \{\ \text{print}('/')\ \} \\
 & | & \mathit{factor} & \\
 & & & \\
\mathit{factor} & \rightarrow & (\ \mathit{expr}\ ) & \\
 & | & \textbf{num} & \{\ \text{print}(\textbf{num}.\mathit{value})\ \} \\
 & | & \textbf{id} & \{\ \text{print}(\textbf{id}.\mathit{lexeme})\ \}
\end{array}
$$

Figure 2.28: Actions for translating into postfix notation

In Fig. 2.28, the terminal **num** is assumed to have an attribute **num**.*value*, which gives the integer value corresponding to this occurrence of **num**. Terminal **id** has a string-valued attribute written as **id**.*lexeme*; we assume this string is the actual lexeme comprising this instance of the token **id**.

The pseudocode fragments used to illustrate the workings of a lexical analyzer will be assembled into Java code at the end of this section. The approach in this section is suitable for hand-written lexical analyzers. Section 3.5 describes a tool called Lex that generates a lexical analyzer from a specification. Symbol tables or data structures for holding information about identifiers are considered in Section 2.7.

2.6.1 Removal of White Space and Comments

The expression translator in Section 2.5 sees every character in the input, so extraneous characters, such as blanks, will cause it to fail. Most languages allow arbitrary amounts of white space to appear between tokens. Comments are likewise ignored during parsing, so they may also be treated as white space.

If white space is eliminated by the lexical analyzer, the parser will never have to consider it. The alternative of modifying the grammar to incorporate white space into the syntax is not nearly as easy to implement.

The pseudocode in Fig. 2.29 skips white space by reading input characters as long as it sees a blank, a tab, or a newline. Variable *peek* holds the next input character. Line numbers and context are useful within error messages to help pinpoint errors; the code uses variable *line* to count newline characters in the input.

```
for ( ; ; peek = next input character ) {
        if ( peek is a blank or a tab ) do nothing;
        else if ( peek is a newline ) line = line+1;
        else break;
}
```

Figure 2.29: Skipping white space

2.6.2 Reading Ahead

A lexical analyzer may need to read ahead some characters before it can decide on the token to be returned to the parser. For example, a lexical analyzer for C or Java must read ahead after it sees the character `>`. If the next character is `=`, then `>` is part of the character sequence `>=`, the lexeme for the token for the "greater than or equal to" operator. Otherwise `>` itself forms the "greater than" operator, and the lexical analyzer has read one character too many.

A general approach to reading ahead on the input, is to maintain an input buffer from which the lexical analyzer can read and push back characters. Input buffers can be justified on efficiency grounds alone, since fetching a block of characters is usually more efficient than fetching one character at a time. A pointer keeps track of the portion of the input that has been analyzed; pushing back a character is implemented by moving back the pointer. Techniques for input buffering are discussed in Section 3.2.

One-character read-ahead usually suffices, so a simple solution is to use a variable, say *peek*, to hold the next input character. The lexical analyzer in this section reads ahead one character while it collects digits for numbers or characters for identifiers; e.g., it reads past `1` to distinguish between `1` and `10`, and it reads past `t` to distinguish between `t` and `true`.

The lexical analyzer reads ahead only when it must. An operator like `*` can be identified without reading ahead. In such cases, *peek* is set to a blank, which will be skipped when the lexical analyzer is called to find the next token. The invariant assertion in this section is that when the lexical analyzer returns a token, variable *peek* either holds the character beyond the lexeme for the current token, or it holds a blank.

2.6.3 Constants

Anytime a single digit appears in a grammar for expressions, it seems reasonable to allow an arbitrary integer constant in its place. Integer constants can be allowed either by creating a terminal symbol, say **num**, for such constants or by incorporating the syntax of integer constants into the grammar. The job of collecting characters into integers and computing their collective numerical value is generally given to a lexical analyzer, so numbers can be treated as single units during parsing and translation.

When a sequence of digits appears in the input stream, the lexical analyzer passes to the parser a token consisting of the terminal **num** along with an integer-valued attribute computed from the digits. If we write tokens as tuples enclosed between $\langle\ \rangle$, the input `31 + 28 + 59` is transformed into the sequence

$$\langle \textbf{num}, 31\rangle \quad \langle \texttt{+} \rangle \quad \langle \textbf{num}, 28\rangle \quad \langle \texttt{+} \rangle \quad \langle \textbf{num}, 59\rangle$$

Here, the terminal symbol `+` has no attributes, so its tuple is simply $\langle \texttt{+} \rangle$. The pseudocode in Fig. 2.30 reads the digits in an integer and accumulates the value of the integer using variable v.

```
if ( peek holds a digit ) {
      $v$ = 0;
      do {
            $v$ = $v * 10$ + integer value of digit peek;
            peek = next input character;
      } while ( peek holds a digit );
      return token ⟨num, $v$⟩;
}
```

Figure 2.30: Grouping digits into integers

2.6.4 Recognizing Keywords and Identifiers

Most languages use fixed character strings such as `for`, `do`, and `if`, as punctuation marks or to identify constructs. Such character strings are called *keywords*.

Character strings are also used as identifiers to name variables, arrays, functions, and the like. Grammars routinely treat identifiers as terminals to simplify the parser, which can then expect the same terminal, say **id**, each time any identifier appears in the input. For example, on input

$$\texttt{count = count + increment;} \tag{2.6}$$

the parser works with the terminal stream **id** `=` **id** `+` **id**. The token for **id** has an attribute that holds the lexeme. Writing tokens as tuples, we see that the tuples for the input stream (2.6) are

$$\langle \textbf{id}, \texttt{"count"}\rangle\ \langle \texttt{=} \rangle\ \langle \textbf{id}, \texttt{"count"}\rangle\ \langle \texttt{+} \rangle\ \langle \textbf{id}, \texttt{"increment"}\rangle\ \langle \texttt{;} \rangle$$

Keywords generally satisfy the rules for forming identifiers, so a mechanism is needed for deciding when a lexeme forms a keyword and when it forms an identifier. The problem is easier to resolve if keywords are *reserved*; i.e., if they cannot be used as identifiers. Then, a character string forms an identifier only if it is not a keyword.

The lexical analyzer in this section solves two problems by using a table to hold character strings:

- *Single Representation.* A string table can insulate the rest of the compiler from the representation of strings, since the phases of the compiler can work with references or pointers to the string in the table. References can also be manipulated more efficiently than the strings themselves.

- *Reserved Words.* Reserved words can be implemented by initializing the string table with the reserved strings and their tokens. When the lexical analyzer reads a string or lexeme that could form an identifier, it first checks whether the lexeme is in the string table. If so, it returns the token from the table; otherwise, it returns a token with terminal **id**.

In Java, a string table can be implemented as a hash table using a class called *Hashtable*. The declaration

Hashtable words = **new** *Hashtable*();

sets up *words* as a default hash table that maps keys to values. We shall use it to map lexemes to tokens. The pseudocode in Fig. 2.31 uses the operation *get* to look up reserved words.

```
if ( peek holds a letter ) {
        collect letters or digits into a buffer b;
        s = string formed from the characters in b;
        w = token returned by words.get(s);
        if ( w is not null ) return w;
        else {
                Enter the key-value pair (s, ⟨id, s⟩) into words
                return token ⟨id, s⟩;
        }
}
```

Figure 2.31: Distinguishing keywords from identifiers

This pseudocode collects from the input a string *s* consisting of letters and digits beginning with a letter. We assume that *s* is made as long as possible; i.e., the lexical analyzer will continue reading from the input as long as it encounters letters and digits. When something other than a letter or digit, e.g., white space, is encountered, the lexeme is copied into a buffer *b*. If the table has an entry for *s*, then the token retrieved by *words.get* is returned. Here, *s* could be either a keyword, with which the *words* table was initially seeded, or it could be an identifier that was previously entered into the table. Otherwise, token **id** and attribute *s* are installed in the table and returned.

2.6.5 A Lexical Analyzer

The pseudocode fragments so far in this section fit together to form a function *scan* that returns token objects, as follows:

```
Token scan() {
        skip white space, as in Section 2.6.1;
        handle numbers, as in Section 2.6.3;
        handle reserved words and identifiers, as in Section 2.6.4;
        /* if we get here, treat read-ahead character peek as a token */
        Token t = new Token(peek);
        peek = blank /* initialization, as discussed in Section 2.6.2 */ ;
        return t;
}
```

The rest of this section implements function *scan* as part of a Java package for lexical analysis. The package, called `lexer` has classes for tokens and a class `Lexer` containing function `scan`.

The classes for tokens and their fields are illustrated in Fig. 2.32; their methods are not shown. Class `Token` has a field `tag` that is used for parsing decisions. Subclass `Num` adds a field `value` for an integer value. Subclass `Word` adds a field `lexeme` that is used for reserved words and identifiers.

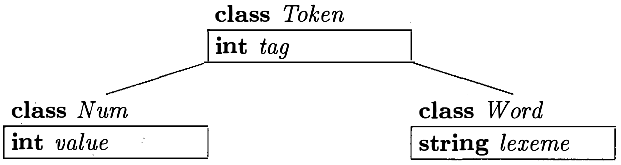


Figure 2.32: Class *Token* and subclasses *Num* and *Word*

Each class is in a file by itself. The file for class `Token` is as follows:

```
1) package lexer;                    // File Token.java
2) public class Token {
3)    public final int tag;
4)    public Token(int t) { tag = t; }
5) }
```

Line 1 identifies the package `lexer`. Field `tag` is declared on line 3 to be `final` so it cannot be changed once it is set. The constructor `Token` on line 4 is used to create token objects, as in

```
new Token('+')
```

which creates a new object of class `Token` and sets its field `tag` to an integer representation of `'+'`. (For brevity, we omit the customary method `toString`, which would return a string suitable for printing.)

Where the pseudocode had terminals like **num** and **id**, the Java code uses integer constants. Class `Tag` implements such constants:

```
1) package lexer;                    // File Tag.java
2) public class Tag {
3)    public final static int
4)       NUM = 256, ID = 257, TRUE = 258, FALSE = 259;
5) }
```

In addition to the integer-valued fields `NUM` and `ID`, this class defines two additional fields, `TRUE` and `FALSE`, for future use; they will be used to illustrate the treatment of reserved keywords.[7]

The fields in class `Tag` are `public`, so they can be used outside the package. They are `static`, so there is just one instance or copy of these fields. The fields are `final`, so they can be set just once. In effect, these fields represent constants. A similar effect is achieved in C by using define-statements to allow names such as `NUM` to be used as symbolic constants, e.g.:

```
#define NUM 256
```

The Java code refers to `Tag.NUM`and `Tag.ID` in places where the pseudocode referred to terminals **num** and **id**. The only requirement is that `Tag.NUM` and `Tag.ID` must be initialized with distinct values that differ from each other and from the constants representing single-character tokens, such as `'+'` or `'*'`.

```
1) package lexer;                    // File Num.java
2) public class Num extends Token {
3)    public final int value;
4)    public Num(int v) { super(Tag.NUM); value = v; }
5) }
```

```
1) package lexer;                    // File Word.java
2) public class Word extends Token {
3)    public final String lexeme;
4)    public Word(int t, String s) {
5)       super(t); lexeme = new String(s);
6)    }
7) }
```

Figure 2.33: Subclasses `Num` and `Word` of `Token`

Classes `Num` and `Word` appear in Fig. 2.33. Class `Num` extends `Token` by declaring an integer field `value` on line 3. The constructor `Num` on line 4 calls `super(Tag.NUM)`, which sets field `tag` in the superclass `Token` to `Tag.NUM`.

[7] ASCII characters are typically converted into integers between 0 and 255. We therefore use integers greater than 255 for terminals.

```
 1) package lexer;                    // File Lexer.java
 2) import java.io.*; import java.util.*;
 3) public class Lexer {
 4)    public int line = 1;
 5)    private char peek = ' ';
 6)    private Hashtable words = new Hashtable();
 7)    void reserve(Word t) { words.put(t.lexeme, t); }
 8)    public Lexer() {
 9)       reserve( new Word(Tag.TRUE,  "true")  );
10)       reserve( new Word(Tag.FALSE, "false") );
11)    }
12)    public Token scan() throws IOException {
13)       for( ; ; peek = (char)System.in.read() ) {
14)          if( peek == ' ' || peek == '\t' ) continue;
15)          else if( peek == '\n' ) line = line + 1;
16)          else break;
17)       }
          /* continues in Fig. 2.35 */
```

Figure 2.34: Code for a lexical analyzer, part 1 of 2

Class `Word` is used for both reserved words and identifiers, so the constructor `Word` on line 4 expects two parameters: a lexeme and a corresponding integer value for `tag`. An object for the reserved word `true` can be created by executing

```
new Word(Tag.TRUE, "true")
```

which creates a new object with field `tag` set to `Tag.TRUE` and field `lexeme` set to the string `"true"`.

Class `Lexer` for lexical analysis appears in Figs. 2.34 and 2.35. The integer variable `line` on line 4 counts input lines, and character variable `peek` on line 5 holds the next input character.

Reserved words are handled on lines 6 through 11. The table `words` is declared on line 6. The helper function `reserve` on line 7 puts a string-word pair in the table. Lines 9 and 10 in the constructor `Lexer` initialize the table. They use the constructor `Word` to create word objects, which are passed to the helper function `reserve`. The table is therefore initialized with reserved words `"true"` and `"false"` before the first call of `scan`.

The code for `scan` in Fig. 2.34–2.35 implements the pseudocode fragments in this section. The for-statement on lines 13 through 17 skips blank, tab, and newline characters. Control leaves the for-statement with `peek` holding a non-white-space character.

The code for reading a sequence of digits is on lines 18 through 25. The function `isDigit` is from the built-in Java class `Character`. It is used on line 18 to check whether `peek` is a digit. If so, the code on lines 19 through 24

```
18)         if( Character.isDigit(peek) ) {
19)            int v = 0;
20)            do {
21)               v = 10*v + Character.digit(peek, 10);
22)               peek = (char)System.in.read();
23)            } while( Character.isDigit(peek) );
24)            return new Num(v);
25)         }
26)         if( Character.isLetter(peek) ) {
27)            StringBuffer b = new StringBuffer();
28)            do {
29)               b.append(peek);
30)               peek = (char)System.in.read();
31)            } while( Character.isLetterOrDigit(peek) );
32)            String s = b.toString();
33)            Word w = (Word)words.get(s);
34)            if( w != null ) return w;
35)            w = new Word(Tag.ID, s);
36)            words.put(s, w);
37)            return w;
38)         }
39)         Token t = new Token(peek);
40)         peek = ' ';
41)         return t;
42)      }
43)   }
```

Figure 2.35: Code for a lexical analyzer, part 2 of 2

accumulates the integer value of the sequence of digits in the input and returns a new `Num` object.

Lines 26 through 38 analyze reserved words and identifiers. Keywords **true** and **false** have already been reserved on lines 9 and 10. Therefore, line 35 is reached if string `s` is not reserved, so it must be the lexeme for an identifier. Line 35 therefore returns a new word object with `lexeme` set to `s` and `tag` set to `Tag.ID`. Finally, lines 39 through 41 return the current character as a token and set `peek` to a blank that will be stripped the next time `scan` is called.

2.6.6 Exercises for Section 2.6

Exercise 2.6.1: Extend the lexical analyzer in Section 2.6.5 to remove comments, defined as follows:

a) A comment begins with // and includes all characters until the end of that line.

b) A comment begins with /* and includes all characters through the next occurrence of the character sequence */.

Exercise 2.6.2: Extend the lexical analyzer in Section 2.6.5 to recognize the relational operators <, <=, ==, !=, >=, >.

Exercise 2.6.3: Extend the lexical analyzer in Section 2.6.5 to recognize floating point numbers such as 2., 3.14, and .5.

2.7 Symbol Tables

Symbol tables are data structures that are used by compilers to hold information about source-program constructs. The information is collected incrementally by the analysis phases of a compiler and used by the synthesis phases to generate the target code. Entries in the symbol table contain information about an identifier such as its character string (or lexeme), its type, its position in storage, and any other relevant information. Symbol tables typically need to support multiple declarations of the same identifier within a program.

From Section 1.6.1, the scope of a declaration is the portion of a program to which the declaration applies. We shall implement scopes by setting up a separate symbol table for each scope. A program block with declarations[8] will have its own symbol table with an entry for each declaration in the block. This approach also works for other constructs that set up scopes; for example, a class would have its own table, with an entry for each field and method.

This section contains a symbol-table module suitable for use with the Java translator fragments in this chapter. The module will be used as is when we put together the translator in Appendix A. Meanwhile, for simplicity, the main example of this section is a stripped-down language with just the key constructs that touch symbol tables; namely, blocks, declarations, and factors. All of the other statement and expression constructs are omitted so we can focus on the symbol-table operations. A program consists of blocks with optional declarations and "statements" consisting of single identifiers. Each such statement represents a use of the identifier. Here is a sample program in this language:

```
{ int x; char y; { bool y; x; y; } x; y; }          (2.7)
```

The examples of block structure in Section 1.6.3 dealt with the definitions and uses of names; the input (2.7) consists solely of definitions and uses of names.

The task we shall perform is to print a revised program, in which the declarations have been removed and each "statement" has its identifier followed by a colon and its type.

[8] In C, for instance, program blocks are either functions or sections of functions that are separated by curly braces and that have one or more declarations within them.

Who Creates Symbol-Table Entries?

Symbol-table entries are created and used during the analysis phase by the lexical analyzer, the parser, and the semantic analyzer. In this chapter, we have the parser create entries. With its knowledge of the syntactic structure of a program, a parser is often in a better position than the lexical analyzer to distinguish among different declarations of an identifier.

In some cases, a lexical analyzer can create a symbol-table entry as soon as it sees the characters that make up a lexeme. More often, the lexical analyzer can only return to the parser a token, say **id**, along with a pointer to the lexeme. Only the parser, however, can decide whether to use a previously created symbol-table entry or create a new one for the identifier.

Example 2.14: On the above input (2.7), the goal is to produce:

```
{ { x:int; y:bool; } x:int; y:char; }
```

The first x and y are from the inner block of input (2.7). Since this use of x refers to the declaration of x in the outer block, it is followed by int, the type of that declaration. The use of y in the inner block refers to the declaration of y in that very block and therefore has boolean type. We also see the uses of x and y in the outer block, with their types, as given by declarations of the outer block: integer and character, respectively. □

2.7.1 Symbol Table Per Scope

The term "scope of identifier x" really refers to the scope of a particular declaration of x. The term *scope* by itself refers to a portion of a program that is the scope of one or more declarations.

Scopes are important, because the same identifier can be declared for different purposes in different parts of a program. Common names like i and x often have multiple uses. As another example, subclasses can redeclare a method name to override a method in a superclass.

If blocks can be nested, several declarations of the same identifier can appear within a single block. The following syntax results in nested blocks when *stmts* can generate a block:

$$block \;\rightarrow\; \text{'\{'}\; decls\; stmts\; \text{'\}'}$$

(We quote curly braces in the syntax to distinguish them from curly braces for semantic actions.) With the grammar in Fig. 2.38, *decls* generates an optional sequence of declarations and *stmts* generates an optional sequence of statements.

Optimization of Symbol Tables for Blocks

Implementations of symbol tables for blocks can take advantage of the most-closely nested rule. Nesting ensures that the chain of applicable symbol tables forms a stack. At the top of the stack is the table for the current block. Below it in the stack are the tables for the enclosing blocks. Thus, symbol tables can be allocated and deallocated in a stack-like fashion.

Some compilers maintain a single hash table of accessible entries; that is, of entries that are not hidden by a declaration in a nested block. Such a hash table supports essentially constant-time lookups, at the expense of inserting and deleting entries on block entry and exit. Upon exit from a block B, the compiler must undo any changes to the hash table due to declarations in block B. It can do so by using an auxiliary stack to keep track of changes to the hash table while block B is processed.

Moreover, a statement can be a block, so our language allows nested blocks, where an identifier can be redeclared.

The *most-closely nested* rule for blocks is that an identifier x is in the scope of the most-closely nested declaration of x; that is, the declaration of x found by examining blocks inside-out, starting with the block in which x appears.

Example 2.15: The following pseudocode uses subscripts to distinguish among distinct declarations of the same identifier:

```
1) {   int $x_1$; int $y_1$;
2)     {   int $w_2$; bool $y_2$; int $z_2$;
3)         $\cdots w_2 \cdots$; $\cdots x_1 \cdots$; $\cdots y_2 \cdots$; $\cdots z_2 \cdots$;
4)     }
5)     $\cdots w_0 \cdots$; $\cdots x_1 \cdots$; $\cdots y_1 \cdots$;
6) }
```

The subscript is not part of an identifier; it is in fact the line number of the declaration that applies to the identifier. Thus, all occurrences of x are within the scope of the declaration on line 1. The occurrence of y on line 3 is in the scope of the declaration of y on line 2 since y is redeclared within the inner block. The occurrence of y on line 5, however, is within the scope of the declaration of y on line 1.

The occurrence of w on line 5 is presumably within the scope of a declaration of w outside this program fragment; its subscript 0 denotes a declaration that is global or external to this block.

Finally, z is declared and used within the nested block, but cannot be used on line 5, since the nested declaration applies only to the nested block. □

The most-closely nested rule for blocks can be implemented by chaining symbol tables. That is, the table for a nested block points to the table for its enclosing block.

Example 2.16: Figure 2.36 shows symbol tables for the pseudocode in Example 2.15. B_1 is for the block starting on line 1 and B_2 is for the block starting at line 2. At the top of the figure is an additional symbol table B_0 for any global or default declarations provided by the language. During the time that we are analyzing lines 2 through 4, the environment is represented by a reference to the lowest symbol table — the one for B_2. When we move to line 5, the symbol table for B_2 becomes inaccessible, and the environment refers instead to the symbol table for B_1, from which we can reach the global symbol table, but not the table for B_2. □

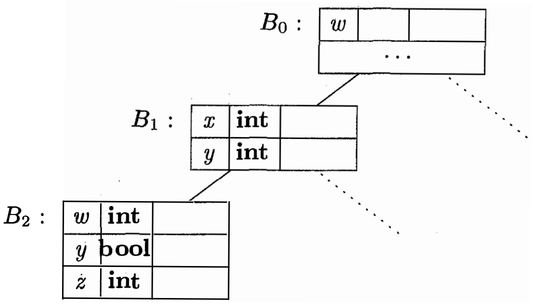


Figure 2.36: Chained symbol tables for Example 2.15

The Java implementation of chained symbol tables in Fig. 2.37 defines a class `Env`, short for *environment*.[9] Class `Env` supports three operations:

- *Create a new symbol table.* The constructor `Env(p)` on lines 6 through 8 of Fig. 2.37 creates an `Env` object with a hash table named `table`. The object is chained to the environment-valued parameter `p` by setting field `next` to `p`. Although it is the `Env` objects that form a chain, it is convenient to talk of the tables being chained.
- *Put* a new entry in the current table. The hash table holds key-value pairs, where:
  - The *key* is a string, or rather a reference to a string. We could alternatively use references to token objects for identifiers as keys.
  - The *value* is an entry of class `Symbol`. The code on lines 9 through 11 does not need to know the structure of an entry; that is, the code is independent of the fields and methods in class `Symbol`.

[9] "Environment" is another term for the collection of symbol tables that are relevant at a point in the program.

```
 1) package symbols;                    // File Env.java
 2) import java.util.*;
 3) public class Env {
 4)    private Hashtable table;
 5)    protected Env prev;

 6)    public Env(Env p) {
 7)       table = new Hashtable(); prev = p;
 8)    }

 9)    public void put(String s, Symbol sym) {
10)       table.put(s, sym);
11)    }

12)    public Symbol get(String s) {
13)       for( Env e = this; e != null; e = e.prev ) {
14)          Symbol found = (Symbol)(e.table.get(s));
15)          if( found != null ) return found;
16)       }
17)       return null;
18)    }
19) }
```

Figure 2.37: Class *Env* implements chained symbol tables

- *Get* an entry for an identifier by searching the chain of tables, starting with the table for the current block. The code for this operation on lines 12 through 18 returns either a symbol-table entry or `null`.

Chaining of symbol tables results in a tree structure, since more than one block can be nested inside an enclosing block. The dotted lines in Fig. 2.36 are a reminder that chained symbol tables can form a tree.

2.7.2 The Use of Symbol Tables

In effect, the role of a symbol table is to pass information from declarations to uses. A semantic action "puts" information about identifier x into the symbol table, when the declaration of x is analyzed. Subsequently, a semantic action associated with a production such as *factor* → **id** "gets" information about the identifier from the symbol table. Since the translation of an expression E_1 **op** E_2, for a typical operator **op**, depends only on the translations of E_1 and E_2, and does not directly depend on the symbol table, we can add any number of operators without changing the basic flow of information from declarations to uses, through the symbol table.

Example 2.17: The translation scheme in Fig. 2.38 illustrates how class *Env* can be used. The translation scheme concentrates on scopes, declarations, and uses. It implements the translation described in Example 2.14. As noted earlier, on input

```
program  →                       { top = null; }
              block

block    →    '{'                { saved = top;
                                   top = new Env(top);
                                   print("{ "); }
              decls stmts '}'    { top = saved;
                                   print("} "); }

decls    →    decls decl
         |    ε

decl     →    type id ;          { s = new Symbol;
                                   s.type = type.lexeme
                                   top.put(id.lexeme, s); }

stmts    →    stmts stmt
         |    ε

stmt     →    block
         |    factor ;           { print("; "); }

factor   →    id                 { s = top.get(id.lexeme);
                                   print(id.lexeme);
                                   print(":"); }
                                   print(s.type);
```

Figure 2.38: The use of symbol tables for translating a language with blocks

```
{ int x; char y; { bool y; x; y; } x; y; }
```

the translation scheme strips the declarations and produces

```
{ { x:int; y:bool; } x:int; y:char; }
```

Notice that the bodies of the productions have been aligned in Fig. 2.38 so that all the grammar symbols appear in one column, and all the actions in a second column. As a result, components of the body are often spread over several lines.

Now, consider the semantic actions. The translation scheme creates and discards symbol tables upon block entry and exit, respectively. Variable *top* denotes the top table, at the head of a chain of tables. The first production of the underlying grammar is $program \rightarrow block$. The semantic action before *block* initializes *top* to **null**, with no entries.

The second production, $block \rightarrow$ '{' *decls stmts* '}', has actions upon block entry and exit. On block entry, before *decls*, a semantic action saves a reference to the current table using a local variable *saved*. Each use of this production has its own local variable *saved*, distinct from the local variable for any other use of this production. In a recursive-descent parser, *saved* would be local to the procedure for *block*. The treatment of local variables of a recursive function is discussed in Section 7.2. The code

$$top = \textbf{new}\ Env(top);$$

sets variable *top* to a newly created new table that is chained to the previous value of *top* just before block entry. Variable *top* is an object of class *Env*; the code for the constructor *Env* appears in Fig. 2.37.

On block exit, after '}', a semantic action restores *top* to its value saved on block entry. In effect, the tables form a stack; restoring *top* to its saved value pops the effect of the declarations in the block.[10] Thus, the declarations in the block are not visible outside the block.

A declaration, $decls \rightarrow$ **type id** results in a new entry for the declared identifier. We assume that tokens **type** and **id** each have an associated attribute, which is the type and lexeme, respectively, of the declared identifier. We shall not go into all the fields of a symbol object *s*, but we assume that there is a field *type* that gives the type of the symbol. We create a new symbol object *s* and assign its type properly by $s.type = \textbf{type}.lexeme$. The complete entry is put into the top symbol table by $top.put(\textbf{id}.lexeme,\ s)$.

The semantic action in the production $factor \rightarrow \textbf{id}$ uses the symbol table to get the entry for the identifier. The *get* operation searches for the first entry in the chain of tables, starting with *top*. The retrieved entry contains any information needed about the identifier, such as the type of the identifier. □

2.8 Intermediate Code Generation

The front end of a compiler constructs an intermediate representation of the source program from which the back end generates the target program. In this section, we consider intermediate representations for expressions and statements, and give tutorial examples of how to produce such representations.

2.8.1 Two Kinds of Intermediate Representations

As was suggested in Section 2.1 and especially Fig. 2.4, the two most important intermediate representations are:

[10] Instead of explicitly saving and restoring tables, we could alternatively add static operations *push* and *pop* to class `Env`.

- Trees, including parse trees and (abstract) syntax trees.
- Linear representations, especially "three-address code."

Abstract-syntax trees, or simply syntax trees, were introduced in Section 2.5.1, and in Section 5.3.1 they will be reexamined more formally. During parsing, syntax-tree nodes are created to represent significant programming constructs. As analysis proceeds, information is added to the nodes in the form of attributes associated with the nodes. The choice of attributes depends on the translation to be performed.

Three-address code, on the other hand, is a sequence of elementary program steps, such as the addition of two values. Unlike the tree, there is no hierarchical structure. As we shall see in Chapter 9, we need this representation if we are to do any significant optimization of code. In that case, we break the long sequence of three-address statements that form a program into "basic blocks," which are sequences of statements that are always executed one-after-the-other, with no branching.

In addition to creating an intermediate representation, a compiler front end checks that the source program follows the syntactic and semantic rules of the source language. This checking is called *static checking*; in general "static" means "done by the compiler."[11] Static checking assures that certain kinds of programming errors, including type mismatches, are detected and reported during compilation.

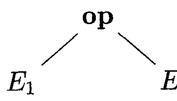
It is possible that a compiler will construct a syntax tree at the same time it emits steps of three-address code. However, it is common for compilers to emit the three-address code while the parser "goes through the motions" of constructing a syntax tree, without actually constructing the complete tree data structure. Rather, the compiler stores nodes and their attributes needed for semantic checking or other purposes, along with the data structure used for parsing. By so doing, those parts of the syntax tree that are needed to construct the three-address code are available when needed, but disappear when no longer needed. We take up the details of this process in Chapter 5.

2.8.2 Construction of Syntax Trees

We shall first give a translation scheme that constructs syntax trees, and later, in Section 2.8.4, show how the scheme can be modified to emit three-address code, along with, or instead of, the syntax tree.

Recall from Section 2.5.1 that the syntax tree

[11] Its opposite, "dynamic," means "while the program is running." Many languages also make certain dynamic checks. For instance, an object-oriented language like Java sometimes must check types during program execution, since the method applied to an object may depend on the particular subclass of the object.



represents an expression formed by applying the operator **op** to the subexpressions represented by E_1 and E_2. Syntax trees can be created for any construct, not just expressions. Each construct is represented by a node, with children for the semantically meaningful components of the construct. For example, the semantically meaningful components of a C while-statement:

while (*expr*) *stmt*

are the expression *expr* and the statement *stmt*.[12] The syntax-tree node for such a while-statement has an operator, which we call **while**, and two children—the syntax trees for the *expr* and the *stmt*.

The translation scheme in Fig. 2.39 constructs syntax trees for a representative, but very limited, language of expressions and statements. All the nonterminals in the translation scheme have an attribute n, which is a node of the syntax tree. Nodes are implemented as objects of class *Node*.

Class *Node* has two immediate subclasses: *Expr* for all kinds of expressions, and *Stmt* for all kinds of statements. Each type of statement has a corresponding subclass of *Stmt*; for example, operator **while** corresponds to subclass *While*. A syntax-tree node for operator **while** with children x and y is created by the pseudocode

new *While* (x, y)

which creates an object of class *While* by calling constructor function *While*, with the same name as the class. Just as constructors correspond to operators, constructor parameters correspond to operands in the abstract syntax.

When we study the detailed code in Appendix A, we shall see how methods are placed where they belong in this hierarchy of classes. In this section, we shall discuss only a few of the methods, informally.

We shall consider each of the productions and rules of Fig. 2.39, in turn. First, the productions defining different types of statements are explained, followed by the productions that define our limited types of expressions.

Syntax Trees for Statements

For each statement construct, we define an operator in the abstract syntax. For constructs that begin with a keyword, we shall use the keyword for the operator. Thus, there is an operator **while** for while-statements and an operator **do** for do-while statements. Conditionals can be handled by defining two operators

[12] The right parenthesis serves only to separate the expression from the statement. The left parenthesis actually has no meaning; it is there only to please the eye, since without it, C would allow unbalanced parentheses.

$\mathit{program}$	$\rightarrow$	block	$\{\ \text{return}\ \mathit{block.n};\ \}$
block	$\rightarrow$	$\texttt{'\{'}\ \mathit{stmts}\ \texttt{'\}'}$	$\{\ \mathit{block.n} = \mathit{stmts.n};\ \}$
stmts	$\rightarrow$	$\mathit{stmts}_1\ \mathit{stmt}$	$\{\ \mathit{stmts.n} = \textbf{new}\ \mathit{Seq}(\mathit{stmts}_1.n, \mathit{stmt.n});\ \}$
	$\mid$	ϵ	$\{\ \mathit{stmts.n} = \textbf{null};\ \}$
stmt	$\rightarrow$	$\mathit{expr}\ \texttt{;}$	$\{\ \mathit{stmt.n} = \textbf{new}\ \mathit{Eval}(\mathit{expr.n});\ \}$
	$\mid$	$\textbf{if}\ \texttt{(}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}_1$	
			$\{\ \mathit{stmt.n} = \textbf{new}\ \mathit{If}(\mathit{expr.n}, \mathit{stmt}_1.n);\ \}$
	$\mid$	$\textbf{while}\ \texttt{(}\ \mathit{expr}\ \texttt{)}\ \mathit{stmt}_1$	
			$\{\ \mathit{stmt.n} = \textbf{new}\ \mathit{While}(\mathit{expr.n}, \mathit{stmt}_1.n);\ \}$
	$\mid$	$\textbf{do}\ \mathit{stmt}_1\ \textbf{while}\ \texttt{(}\ \mathit{expr}\ \texttt{)}\ \texttt{;}$	
			$\{\ \mathit{stmt.n} = \textbf{new}\ \mathit{Do}(\mathit{stmt}_1.n, \mathit{expr.n});\ \}$
	$\mid$	block	$\{\ \mathit{stmt.n} = \mathit{block.n};\ \}$
expr	$\rightarrow$	$\mathit{rel}\ \texttt{=}\ \mathit{expr}_1$	$\{\ \mathit{expr.n} = \textbf{new}\ \mathit{Assign}('\!=\!', \mathit{rel.n}, \mathit{expr}_1.n);\ \}$
	$\mid$	rel	$\{\ \mathit{expr.n} = \mathit{rel.n};\ \}$
rel	$\rightarrow$	$\mathit{rel}_1\ \texttt{<}\ \mathit{add}$	$\{\ \mathit{rel.n} = \textbf{new}\ \mathit{Rel}('<', \mathit{rel}_1.n, \mathit{add.n});\ \}$
	$\mid$	$\mathit{rel}_1\ \texttt{<=}\ \mathit{add}$	$\{\ \mathit{rel.n} = \textbf{new}\ \mathit{Rel}('\leq', \mathit{rel}_1.n, \mathit{add.n});\ \}$
	$\mid$	add	$\{\ \mathit{rel.n} = \mathit{add.n};\ \}$
add	$\rightarrow$	$\mathit{add}_1\ \texttt{+}\ \mathit{term}$	$\{\ \mathit{add.n} = \textbf{new}\ \mathit{Op}('+', \mathit{add}_1.n, \mathit{term.n});\ \}$
	$\mid$	term	$\{\ \mathit{add.n} = \mathit{term.n};\ \}$
term	$\rightarrow$	$\mathit{term}_1\ *\ \mathit{factor}$	$\{\ \mathit{term.n} = \textbf{new}\ \mathit{Op}('*', \mathit{term}_1.n, \mathit{factor.n});\ \}$
	$\mid$	factor	$\{\ \mathit{term.n} = \mathit{factor.n};\ \}$
factor	$\rightarrow$	$\texttt{(}\ \mathit{expr}\ \texttt{)}$	$\{\ \mathit{factor.n} = \mathit{expr.n};\ \}$
	$\mid$	$\textbf{num}$	$\{\ \mathit{factor.n} = \textbf{new}\ \mathit{Num}(\textbf{num}.\mathit{value});\ \}$

Figure 2.39: Construction of syntax trees for expressions and statements

ifelse and **if** for if-statements with and without an else part, respectively. In our simple example language, we do not use **else**, and so have only an if-statement. Adding **else** presents some parsing issues, which we discuss in Section 4.8.2.

Each statement operator has a corresponding class of the same name, with a capital first letter; e.g., class *If* corresponds to **if**. In addition, we define the subclass *Seq*, which represents a sequence of statements. This subclass corresponds to the nonterminal *stmts* of the grammar. Each of these classes are subclasses of *Stmt*, which in turn is a subclass of *Node*.

The translation scheme in Fig. 2.39 illustrates the construction of syntax-tree nodes. A typical rule is the one for if-statements:

$$stmt \;\rightarrow\; \textbf{if}\ (\ expr\ )\ stmt_1 \quad \{\ stmt.n = \textbf{new}\ If(expr.n, stmt_1.n);\ \}$$

The meaningful components of the if-statement are *expr* and $stmt_1$. The semantic action defines the node *stmt.n* as a new object of subclass *If*. The code for the constructor *If* is not shown. It creates a new node labeled **if** with the nodes *expr.n* and $stmt_1.n$ as children.

Expression statements do not begin with a keyword, so we define a new operator **eval** and class *Eval*, which is a subclass of *Stmt*, to represent expressions that are statements. The relevant rule is:

$$stmt \;\rightarrow\; expr\ ; \quad \{\ stmt.n = \textbf{new}\ Eval(expr.n);\ \}$$

Representing Blocks in Syntax Trees

The remaining statement construct in Fig. 2.39 is the block, consisting of a sequence of statements. Consider the rules:

$$\begin{array}{lcll} stmt & \rightarrow & block & \{\ stmt.n = block.n;\ \} \\ block & \rightarrow & '\{'\ stmts\ '\}' & \{\ block.n = stmts.n;\ \} \end{array}$$

The first says that when a statement is a block, it has the same syntax tree as the block. The second rule says that the syntax tree for nonterminal *block* is simply the syntax tree for the sequence of statements in the block.

For simplicity, the language in Fig. 2.39 does not include declarations. Even when declarations are included in Appendix A, we shall see that the syntax tree for a block is still the syntax tree for the statements in the block. Since information from declarations is incorporated into the symbol table, they are not needed in the syntax tree. Blocks, with or without declarations, therefore appear to be just another statement construct in intermediate code.

A sequence of statements is represented by using a leaf **null** for an empty statement and a operator **seq** for a sequence of statements, as in

$$stmts \;\rightarrow\; stmts_1\ stmt \quad \{\ stmts.n = \textbf{new}\ Seq(stmts_1.n, stmt.n);\ \}$$

Example 2.18 : In Fig. 2.40 we see part of a syntax tree representing a block or statement list. There are two statements in the list, the first an if-statement and the second a while-statement. We do not show the portion of the tree above this statement list, and we show only as a triangle each of the necessary subtrees: two expression trees for the conditions of the if- and while-statements, and two statement trees for their substatements. □

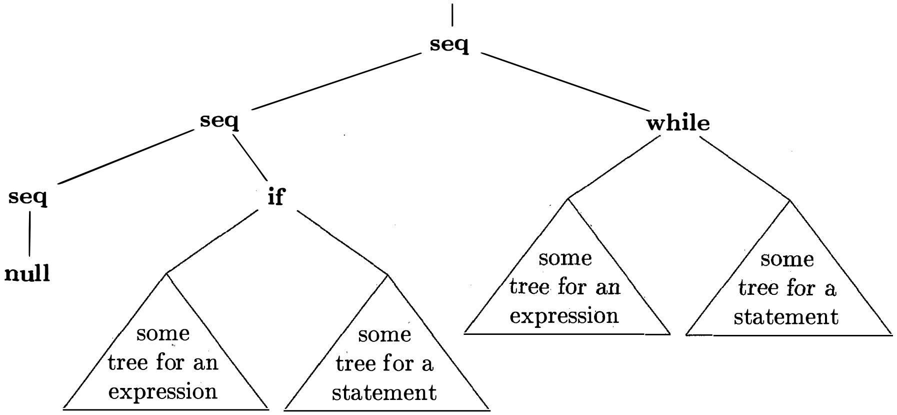

seq
seq
while
seq
if
null
some tree for an expression
some tree for a statement
some tree for an expression
some tree for a statement

Figure 2.40: Part of a syntax tree for a statement list consisting of an if-statement and a while-statement

Syntax Trees for Expressions

Previously, we handled the higher precedence of * over + by using three nonterminals *expr*, *term*, and *factor*. The number of nonterminals is precisely one plus the number of levels of precedence in expressions, as we suggested in Section 2.2.6. In Fig. 2.39, we have two comparison operators, < and <= at one precedence level, as well as the usual + and * operators, so we have added one additional nonterminal, called *add*.

Abstract syntax allows us to group "similar" operators to reduce the number of cases and subclasses of nodes in an implementation of expressions. In this chapter, we take "similar" to mean that the type-checking and code-generation rules for the operators are similar. For example, typically the operators + and * can be grouped, since they can be handled in the same way — their requirements regarding the types of operands are the same, and they each result in a single three-address instruction that applies one operator to two values. In general, the grouping of operators in the abstract syntax is based on the needs of the later phases of the compiler. The table in Fig. 2.41 specifies the correspondence between the concrete and abstract syntax for several of the operators of Java.

In the concrete syntax, all operators are left associative, except the assignment operator =, which is right associative. The operators on a line have the

CONCRETE SYNTAX	ABSTRACT SYNTAX
`=`	**assign**
`||`	**cond**
`&&`	**cond**
`== !=`	**rel**
`< <= >= >`	**rel**
`+ -`	**op**
`* / %`	**op**
`!`	**not**
$-_{unary}$	**minus**
`[]`	**access**

Figure 2.41: Concrete and abstract syntax for several Java operators

same precedence; that is, `==` and `!=` have the same precedence. The lines are in order of increasing precedence; e.g., `==` has higher precedence than the operators `&&` and `=`. The subscript *unary* in $-_{unary}$ is solely to distinguish a leading unary minus sign, as in `-2`, from a binary minus sign, as in `2-a`. The operator `[]` represents array access, as in `a[i]`.

The abstract-syntax column specifies the grouping of operators. The assignment operator `=` is in a group by itself. The group **cond** contains the conditional boolean operators `&&` and `||`. The group **rel** contains the relational comparison operators on the lines for `==` and `<`. The group **op** contains the arithmetic operators like `+` and `*`. Unary minus, boolean negation, and array access are in groups by themselves.

The mapping between concrete and abstract syntax in Fig. 2.41 can be implemented by writing a translation scheme. The productions for nonterminals *expr*, *rel*, *add*, *term*, and *factor* in Fig. 2.39 specify the concrete syntax for a representative subset of the operators in Fig. 2.41. The semantic actions in these productions create syntax-tree nodes. For example, the rule

$$term \rightarrow term_1 * factor \quad \{\ term.n = \textbf{new}\ Op('*', term_1.n, factor.n);\ \}$$

creates a node of class *Op*, which implements the operators grouped under **op** in Fig. 2.41. The constructor *Op* has a parameter $'*'$ to identify the actual operator, in addition to the nodes $term_1.n$ and $factor.n$ for the subexpressions.

2.8.3 Static Checking

Static checks are consistency checks that are done during compilation. Not only do they assure that a program can be compiled successfully, but they also have the potential for catching programming errors early, before a program is run. Static checking includes:

- *Syntactic Checking.* There is more to syntax than grammars. For example, constraints such as an identifier being declared at most once in a scope, or that a break statement must have an enclosing loop or switch statement, are syntactic, although they are not encoded in, or enforced by, a grammar used for parsing.

- *Type Checking.* The type rules of a language assure that an operator or function is applied to the right number and type of operands. If conversion between types is necessary, e.g., when an integer is added to a float, then the type-checker can insert an operator into the syntax tree to represent that conversion. We discuss type conversion, using the common term "coercion," below.

L-values and R-values

We now consider some simple static checks that can be done during the construction of a syntax tree for a source program. In general, complex static checks may need to be done by first constructing an intermediate representation and then analyzing it.

There is a distinction between the meaning of identifiers on the left and right sides of an assignment. In each of the assignments

```
i = 5;
i = i + 1;
```

the right side specifies an integer value, while the left side specifies where the value is to be stored. The terms *l-value* and *r-value* refer to values that are appropriate on the left and right sides of an assignment, respectively. That is, *r*-values are what we usually think of as "values," while *l*-values are locations.

Static checking must assure that the left side of an assignment denotes an *l*-value. An identifier like `i` has an *l*-value, as does an array access like `a[2]`. But a constant like 2 is not appropriate on the left side of an assignment, since it has an *r*-value, but not an *l*-value.

Type Checking

Type checking assures that the type of a construct matches that expected by its context. For example, in the if-statement

if (*expr*) *stmt*

the expression *expr* is expected to have type **boolean**.

Type checking rules follow the operator/operand structure of the abstract syntax. Assume the operator **rel** represents relational operators such as <=. The type rule for the operator group **rel** is that its two operands must have the same type, and the result has type boolean. Using attribute *type* for the type of an expression, let E consist of **rel** applied to E_1 and E_2. The type of E can be checked when its node is constructed, by executing code like the following:

```
if ( $E_1.type$ == $E_2.type$ ) $E.type$ = boolean;
else error;
```

The idea of matching actual with expected types continues to apply, even in the following situations:

- *Coercions.* A *coercion* occurs if the type of an operand is automatically converted to the type expected by the operator. In an expression like `2*3.14`, the usual transformation is to convert the integer `2` into an equivalent floating-point number, `2.0`, and then perform a floating-point operation on the resulting pair of floating-point operands. The language definition specifies the allowable coercions. For example, the actual rule for **rel** discussed above might be that $E_1.type$ and $E_2.type$ are convertible to the same type. In that case, it would be legal to compare, say, an integer with a float.

- *Overloading.* The operator `+` in Java represents addition when applied to integers; it means concatenation when applied to strings. A symbol is said to be *overloaded* if it has different meanings depending on its context. Thus, `+` is overloaded in Java. The meaning of an overloaded operator is determined by considering the known types of its operands and results. For example, we know that the `+` in `z = x + y` is concatenation if we know that any of `x`, `y`, or `z` is of type string. However, if we also know that another one of these is of type integer, then we have a type error and there is no meaning to this use of `+`.

2.8.4 Three-Address Code

Once syntax trees are constructed, further analysis and synthesis can be done by evaluating attributes and executing code fragments at nodes in the tree. We illustrate the possibilities by walking syntax trees to generate three-address code. Specifically, we show how to write functions that process the syntax tree and, as a side-effect, emit the necessary three-address code.

Three-Address Instructions

Three-address code is a sequence of instructions of the form

$$x = y \ \mathbf{op} \ z$$

where x, y, and z are names, constants, or compiler-generated temporaries; and **op** stands for an operator.

Arrays will be handled by using the following two variants of instructions:

$$x \, [\, y \,] = z$$
$$x = y \, [\, z \,]$$

The first puts the value of z in the location $x[y]$, and the second puts the value of $y[z]$ in the location x.

Three-address instructions are executed in numerical sequence unless forced to do otherwise by a conditional or unconditional jump. We choose the following instructions for control flow:

`ifFalse` x `goto L`	if x is false, next execute the instruction labeled `L`
`ifTrue` x `goto L`	if x is true, next execute the instruction labeled `L`
`goto L`	next execute the instruction labeled `L`

A label `L` can be attached to any instruction by prepending a prefix `L:`. An instruction can have more than one label.

Finally, we need instructions that copy a value. The following three-address instruction copies the value of `y` into `x`:

$$x = y$$

Translation of Statements

Statements are translated into three-address code by using jump instructions to implement the flow of control through the statement. The layout in Fig. 2.42 illustrates the translation of **if** *expr* **then** *stmt*$_1$. The jump instruction in the layout

`ifFalse` x `goto` *after*

jumps over the translation of *stmt*$_1$ if *expr* evaluates to **false**. Other statement constructs are similarly translated using appropriate jumps around the code for their components.

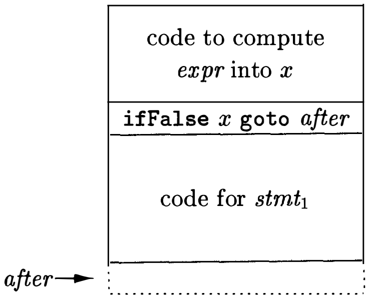


Figure 2.42: Code layout for if-statements

For concreteness, we show the pseudocode for class *If* in Fig. 2.43. Class *If* is a subclass of *Stmt*, as are the classes for the other statement constructs. Each subclass of *Stmt* has a constructor — *If* in this case — and a function *gen* that is called to generate three-address code for this kind of statement.

```
class If extends Stmt {
        Expr E; Stmt S;
        public If(Expr x, Stmt y) { E = x; S = y; after = newlabel(); }
        public void gen() {
                Expr n = E.rvalue();
                emit( “ifFalse ” + n.toString() + “ goto ” + after);
                S.gen();
                emit(after + “:”);
        }
}
```

Figure 2.43: Function *gen* in class *If* generates three-address code

The constructor *If* in Fig. 2.43 creates syntax-tree nodes for if-statements. It is called with two parameters, an expression node x and a statement node y, which it saves as attributes E and S. The constructor also assigns attribute *after* a unique new label, by calling function *newlabel*(). The label will be used according to the layout in Fig. 2.42.

Once the entire syntax tree for a source program is constructed, the function *gen* is called at the root of the syntax tree. Since a program is a block in our simple language, the root of the syntax tree represents the sequence of statements in the block. All statement classes contain a function *gen*.

The pseudocode for function *gen* of class *If* in Fig. 2.43 is representative. It calls $E.rvalue()$ to translate the expression E (the boolean-valued expression that is part of the if-statements) and saves the result node returned by E. Translation of expressions will be discussed shortly. Function *gen* then emits a conditional jump and calls $S.gen()$ to translate the substatement S.

Translation of Expressions

We now illustrate the translation of expressions by considering expressions containing binary operators **op**, array accesses, and assignments, in addition to constants and identifiers. For simplicity, in an array access $y[z]$, we require that y be an identifier.[13] For a detailed discussion of intermediate code generation for expressions, see Section 6.4.

We shall take the simple approach of generating one three-address instruction for each operator node in the syntax tree for an expression. No code is generated for identifiers and constants, since they can appear as addresses in instructions. If a node x of class *Expr* has operator **op**, then an instruction is emitted to compute the value at node x into a compiler generated "temporary" name, say t. Thus, `i-j+k` translates into two instructions

[13] This simple language supports `a[a[n]]`, but not `a[m][n]`. Note that `a[a[n]]` has the form `a[`E`]`, where E is `a[n]`.

```
t1 = i - j
t2 = t1 + k
```

With array accesses and assignments comes the need to distinguish between *l*-values and *r*-values. For example, `2*a[i]` can be translated by computing the *r*-value of `a[i]` into a temporary, as in

```
t1 = a [ i ]
t2 = 2 * t1
```

But, we cannot simply use a temporary in place of `a[i]`, if `a[i]` appears on the left side of an assignment.

The simple approach uses the two functions *lvalue* and *rvalue*, which appear in Fig. 2.44 and 2.45, respectively. When function *rvalue* is applied to a nonleaf node x, it generates instructions to compute x into a temporary, and returns a new node representing the temporary. When function *lvalue* is applied to a nonleaf, it also generates instructions to compute the subtrees below x, and returns a node representing the "address" for x.

We describe function *lvalue* first, since it has fewer cases. When applied to a node x, function *lvalue* simply returns x if it is the node for an identifier (i.e., if x is of class *Id*). In our simple language, the only other case where an expression has an *l*-value occurs when x represents an array access, such as `a[i]`. In this case, x will have the form *Access*(y, z), where class *Access* is a subclass of *Expr*, y represents the name of the accessed array, and z represents the offset (index) of the chosen element in that array. From the pseudo-code in Fig. 2.44, function *lvalue* calls *rvalue*(z) to generate instructions, if needed, to compute the *r*-value of z. It then constructs and returns a new *Access* node with children for the array name y and the *r*-value of z.

```
Expr lvalue($x$ : Expr) {
        if ( $x$ is an Id node ) return $x$;
        else if ( $x$ is an Access$(y, z)$ node and $y$ is an Id node ) {
                return new Access$(y, rvalue(z))$;
        }
        else error;
}
```

Figure 2.44: Pseudocode for function *lvalue*

Example 2.19: When node x represents the array access `a[2*k]`, the call *lvalue*(x) generates an instruction

```
t = 2 * k
```

and returns a new node x' representing the *l*-value `a[t]`, where `t` is a new temporary name.

In detail, the code fragment

return new *Access* $(y, rvalue(z))$;

is reached with y being the node for a and z being the node for expression `2*k`. The call $rvalue(z)$ generates code for the expression `2*k` (i.e., the three-address statement `t = 2 * k`) and returns the new node z' representing the temporary name `t`. That node z' becomes the value of the second field in the new *Access* node x' that is created. □

```
Expr rvalue(x : Expr) {
        if ( $x$ is an Id or a Constant node ) return $x$;
        else if ( $x$ is an Op(op, $y$, $z$) or a Rel(op, $y$, $z$) node ) {
                $t$ = new temporary;
                emit string for $t$ = $rvalue(y)$ op $rvalue(z)$;
                return a new node for $t$;
        }
        else if ( $x$ is an Access($y$, $z$) node ) {
                $t$ = new temporary;
                call $lvalue(x)$, which returns Access($y$, $z'$);
                emit string for $t$ = Access($y$, $z'$);
                return a new node for $t$;
        }
        else if ( $x$ is an Assign($y$, $z$) node ) {
                $z' = rvalue(z)$;
                emit string for $lvalue(y)$ = $z'$;
                return $z'$;
        }
}
```

Figure 2.45: Pseudocode for function *rvalue*

Function *rvalue* in Fig. 2.45 generates instructions and returns a possibly new node. When x represents an identifier or a constant, *rvalue* returns x itself. In all other cases, it returns an *Id* node for a new temporary t. The cases are as follows:

- When x represents y **op** z, the code first computes $y' = rvalue(y)$ and $z' = rvalue(z)$. It creates a new temporary t and generates an instruction t = y' **op** z' (more precisely, an instruction formed from the string representations of t, y', **op**, and z'). It returns a node for identifier t.

- When x represents an array access y`[`z`]`, we can reuse function *lvalue*. The call $lvalue(x)$ returns an access y`[`z'`]`, where z' represents an identifier holding the offset for the array access. The code creates a new temporary t, generates an instruction based on $t = y$`[`z'`]`, and returns a node for t.

- When x represents y = z, then the code first computes $z' = rvalue(z)$. It generates an instruction based on $lvalue(y)$ = z' and returns the node z'.

Example 2.20: When applied to the syntax tree for

```
a[i] = 2*a[j-k]
```

function *rvalue* generates

```
t3 = j - k
t2 = a [ t3 ]
t1 = 2 * t2
a [ i ] = t1
```

That is, the root is an *Assign* node with first argument a[i] and second argument 2*a[j-k]. Thus, the third case applies, and function *rvalue* recursively evaluates 2*a[j-k]. The root of this subtree is the *Op* node for *, which causes a new temporary t1 to be created, before the left operand, 2 is evaluated, and then the right operand. The constant 2 generates no three-address code, and its r-value is returned as a *Constant* node with value 2.

The right operand a[j-k] is an *Access* node, which causes a new temporary t2 to be created, before function *lvalue* is called on this node. Recursively, *rvalue* is called on the expression j-k. As a side-effect of this call, the three-address statement t3 = j - k is generated, after the new temporary t3 is created. Then, returning to the call of *lvalue* on a[j-k], the temporary t2 is assigned the r-value of the entire access-expression, that is, t2 = a [t3].

Now, we return to the call of *rvalue* on the *Op* node 2*a[j-k], which earlier created temporary t1. A three-address statement t1 = 2 * t2 is generated as a side-effect, to evaluate this multiplication-expression. Last, the call to *rvalue* on the whole expression completes by calling *lvalue* on the left side a[i] and then generating a three-address instruction a [i] = t1, in which the right side of the assignment is assigned to the left side. □

Better Code for Expressions

We can improve on function *rvalue* in Fig. 2.45 and generate fewer three-address instructions, in several ways:

- Reduce the number of copy instructions in a subsequent optimization phase. For example, the pair of instructions t = i+1 and i = t can be combined into i = i+1, if there are no subsequent uses of t.

- Generate fewer instructions in the first place by taking context into account. For example, if the left side of a three-address assignment is an array access a[t], then the right side must be a name, a constant, or a temporary, all of which use just one address. But if the left side is a name x, then the right side can be an operation y **op** z that uses two addresses.

We can avoid some copy instructions by modifying the translation functions to generate a partial instruction that computes, say `j+k`, but does not commit to where the result is to be placed, signified by a **null** address for the result:

$$\textbf{null}\ \mathtt{=\ j\ +\ k} \qquad (2.8)$$

The null result address is later replaced by either an identifier or a temporary, as appropriate. It is replaced by an identifier if `j+k` is on the right side of an assignment, as in `i=j+k;`, in which case (2.8) becomes

$$\mathtt{i\ =\ j\ +\ k}$$

But, if `j+k` is a subexpression, as in `j+k+l`, then the null result address in (2.8) is replaced by a new temporary `t`, and a new partial instruction is generated

$$\begin{array}{l} \mathtt{t\ =\ j\ +\ k} \\ \textbf{null}\ \mathtt{=\ t\ +\ l} \end{array}$$

Many compilers make every effort to generate code that is as good as or better than hand-written assembly code produced by experts. If code-optimization techniques, such as the ones in Chapter 9 are used, then an effective strategy may well be to use a simple approach for intermediate code generation, and rely on the code optimizer to eliminate unnecessary instructions.

2.8.5 Exercises for Section 2.8

Exercise 2.8.1: For-statements in C and Java have the form:

$$\mathtt{for\ (}\ expr_1\ \mathtt{;}\ expr_2\ \mathtt{;}\ expr_3\ \mathtt{)}\ stmt$$

The first expression is executed before the loop; it is typically used for initializing the loop index. The second expression is a test made before each iteration of the loop; the loop is exited if the expression becomes 0. The loop itself can be thought of as the statement `{`*stmt expr*3`;}`. The third expression is executed at the end of each iteration; it is typically used to increment the loop index. The meaning of the for-statement is similar to

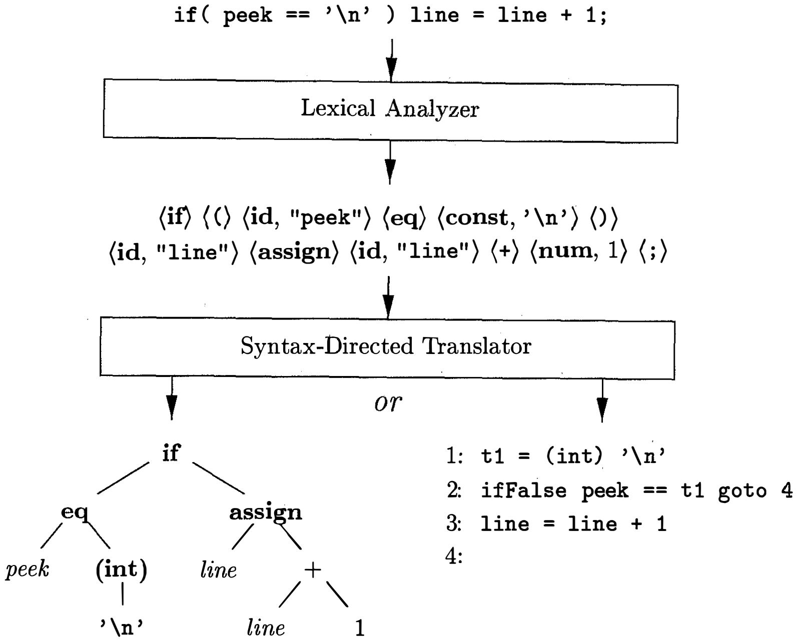
$$expr_1\mathtt{;\ while\ (}\ expr_2\ \mathtt{)\ \{}stmt\ expr_3\mathtt{;\ \}}$$

Define a class *For* for for-statements, similar to class *If* in Fig. 2.43.

Exercise 2.8.2: The programming language C does not have a boolean type. Show how a C compiler might translate an if-statement into three-address code.

2.9 Summary of Chapter 2

The syntax-directed techniques in this chapter can be used to construct compiler front ends, such as those illustrated in Fig. 2.46.


if(peek == '\n') line = line + 1;
Lexical Analyzer
⟨if⟩ ⟨(⟩ ⟨id, "peek"⟩ ⟨eq⟩ ⟨const, '\n'⟩ ⟨)⟩
⟨id, "line"⟩ ⟨assign⟩ ⟨id, "line"⟩ ⟨+⟩ ⟨num, 1⟩ ⟨;⟩
Syntax-Directed Translator
or
if
eq
assign
peek
(int)
line
+
'\n'
line
1

```
1: t1 = (int) '\n'
2: ifFalse peek == t1 goto 4
3: line = line + 1
4:
```

Figure 2.46: Two possible translations of a statement

- ✦ The starting point for a syntax-directed translator is a grammar for the source language. A *grammar* describes the hierarchical structure of programs. It is defined in terms of elementary symbols called *terminals* and variable symbols called *nonterminals*. These symbols represent language constructs. The rules or *productions* of a grammar consist of a nonterminal called the *head* or *left side* of a production and a sequence of terminals and nonterminals called the *body* or *right side* of the production. One nonterminal is designated as the *start* symbol.

- ✦ In specifying a translator, it is helpful to attach attributes to programming construct, where an *attribute* is any quantity associated with a construct. Since constructs are represented by grammar symbols, the concept of attributes extends to grammar symbols. Examples of attributes include an integer value associated with a terminal **num** representing numbers, and a string associated with a terminal **id** representing identifiers.

- ✦ A *lexical analyzer* reads the input one character at a time and produces as output a stream of *tokens*, where a token consists of a terminal symbol along with additional information in the form of attribute values. In Fig. 2.46, tokens are written as tuples enclosed between ⟨ ⟩. The token ⟨**id**, "peek"⟩ consists of the terminal **id** and a pointer to the symbol-table entry containing the string "peek". The translator uses the table to keep track of reserved words and identifiers that have already been seen.

- *Parsing* is the problem of figuring out how a string of terminals can be derived from the start symbol of the grammar by repeatedly replacing a nonterminal by the body of one of its productions. Conceptually, a parser builds a parse tree in which the root is labeled with the start symbol, each nonleaf corresponds to a production, and each leaf is labeled with a terminal or the empty string ϵ. The parse tree derives the string of terminals at the leaves, read from left to right.

- Efficient parsers can be built by hand, using a top-down (from the root to the leaves of a parse tree) method called predictive parsing. A *predictive parser* has a procedure for each nonterminal; procedure bodies mimic the productions for nonterminals; and, the flow of control through the procedure bodies can be determined unambiguously by looking one symbol ahead in the input stream. See Chapter 4 for other approaches to parsing.

- Syntax-directed translation is done by attaching either rules or program fragments to productions in a grammar. In this chapter, we have considered only *synthesized* attributes — the value of a synthesized attribute at any node x can depend only on attributes at the children of x, if any. A *syntax-directed definition* attaches rules to productions; the rules compute attribute vales. A *translation scheme* embeds program fragments called *semantic actions* in production bodies. The actions are executed in the order that productions are used during syntax analysis.

- The result of syntax analysis is a representation of the source program, called *intermediate code.* Two primary forms of intermediate code are illustrated in Fig. 2.46. An *abstract syntax tree* has nodes for programming constructs; the children of a node give the meaningful subconstructs. Alternatively, *three-address code* is a sequence of instructions in which each instruction carries out a single operation.

- *Symbol tables* are data structures that hold information about identifiers. Information is put into the symbol table when the declaration of an identifier is analyzed. A semantic action gets information from the symbol table when the identifier is subsequently used, for example, as a factor in an expression.