move the parser shifts the state corresponding to the token **digit** (whose LEXVAL is 2) onto the stack. (The state is represented by LEXVAL, which is 2.) On the second move the parser reduces by the production $I \rightarrow$ **digit** and then invokes the semantic action $I.\mathrm{VAL} = \mathrm{LEXVAL}$. The program fragment implementing this semantic action causes the VAL field of the stack entry for **digit** to acquire the value 2.

The remaining moves should now be clear. Note that after each reduction and semantic action the top of the VAL stack contains the value of the translation associated with the left side of the reducing production.

7.3 Intermediate Code

While the use of syntax-directed translation is not restricted to compiling, the latter is still the subject of this book, so let us now discuss the kinds of syntax-directed translations done most often in compilers. In many compilers the source code is translated into a language which is intermediate in complexity between a (high-level) programming language and machine code. Such a language is therefore called *intermediate code* or *intermediate text*. It is possible to translate directly from source to machine or assembly language in a syntax-directed way but, as we have mentioned, doing so makes generation of optimal, or even relatively good, code a difficult task.

The reason efficient machine or assembly language is hard to generate is that one is immediately forced to choose a particular register to hold the result of each computation, making the efficient use of registers difficult. Therefore one usually chooses for intermediate text a notation in which, as in assembly language, each statement involves at most one arithmetic operation or one test, but where, unlike in assembly language, the register in which each operation occurs is left unspecified. The usual intermediate text introduces symbols to stand for various temporary quantities such as the value of B*C in the source language expression A+B*C. Four kinds of intermediate code often used in compilers are *postfix notation*, *syntax trees*, *quadruples*, and *triples*. These forms are the subjects of the next three sections.

7.4 Postfix Notation

The ordinary (*infix*) way of writing the sum of a and b is with the operator in the middle: $a+b$. The *postfix* (or *postfix Polish*†) notation for the same expression places the operator at the right end, as $ab+$. In general, if e_1 and e_2 are any postfix expressions, and θ is any binary operator, the result of applying θ to the values denoted by e_1 and e_2 is indicated in postfix notation by $e_1 e_2 \theta$. No parentheses are needed in postfix notation because the position and arity (number of arguments) of the operators permits only one

† Named after the nationality of J. Łukasiewicz, the originator of the notation.

way to decode a postfix expression.

Example 7.2.

1. $(a+b)*c$ in postfix notation is $ab+c*$, since $ab+$ represents the infix expression $(a+b)$.
2. $a*(b+c)$ is $abc+*$ in postfix.
3. $(a+b)*(c+d)$ is $ab+cd+*$ in postfix. □

Postfix notation can be generalized to k-ary operators for any $k \geq 1$. If k-ary operator θ is applied to postfix expressions $e_1, e_2, \ldots, e_k$, then the result is denoted by $e_1 e_2 \cdots e_k \theta$. If we know the arity of each operator, then we can uniquely decipher any postfix expression by scanning it from either end.

For example, consider the postfix string $ab+c*$. The righthand $*$ says that there are two arguments to its left. Since the next-to-rightmost symbol is c, a simple operand, we know c must be the second operand of $*$. Continuing to the left, we encounter the operator $+$. We now know the subexpression ending in $+$ makes up the first operand of $*$. Continuing in this way, we deduce that $ab+c*$ is "parsed" as $(((a,b)+),c)*$.

Example 7.3. Let us introduce a useful 3-ary *(ternary)* operator, the conditional expression. Let **if** e **then** x **else** y denote the expression whose value is x if $e \neq 0$ and y if $e = 0$. Using ? as a ternary postfix operator, we can represent this expression as $exy?$. The postfix form of the expression

if a **then if** $c-d$ **then** $a+c$ **else** $a*c$ **else** $a+b$

is: $acd-ac+ac*?ab+?$. □

One language that normally uses a postfix intermediate language is SNOBOL. In fact, SNOBOL is often interpreted rather than compiled. The output of the SNOBOL compiler is the intermediate code itself which is passed to an interpreter, which reads the intermediate code and executes it. (Some SNOBOL compilers exist, producing machine code heavily laced with subroutine calls.)

Evaluation of Postfix Expressions

Having generated postfix notation for an expression, we can evaluate it easily using a stack, either a hardware stack or one implemented in software. The general strategy is to scan the postfix code left to right. We push each operand onto the stack when we see it. If we encounter a k-ary operator, its first (leftmost) argument will be $k-1$ positions below the top on the stack, its last argument will be at the top, and in general, its ith argument is $k-i$ positions below the top. It is then easy to apply the operator to the top k values on the stack. These values are popped and the result of applying the k-ary operator is pushed onto the stack.

Example 7.4. Consider the postfix expression $ab+c*$ from Example 7.2. Suppose a, b, and c have values, 1, 3 and 5 respectively. To evaluate $13+5*$ we perform the following actions:

1. Stack 1
2. Stack 3.
3. Add the two topmost elements, pop them off the stack, and then stack the result, 4.
4. Stack 5.
5. Multiply the two topmost elements, pop them off the stack, and then stack the result, 20.

The value on top of the stack at the end (here 20) is the value of the entire expression. □

Control Flow in Postfix Code

While postfix notation is useful for intermediate code if the language is mostly expressions, as SNOBOL is, there are problems when more than rudimentary flow of control must be handled. For example, our previous implementation of the conditional **if-then-else** operator causes the second and third arguments always to be evaluated, even though only one of them is used. Therefore, if operands are undefined or have side effects, the postfix implementation described above not only would be inefficient, but might be incorrect.

One solution is to introduce labels and conditional and unconditional jumps into the postfix code. The postfix code can then be stored in a one-dimensional array, with each word of the array being either an operator or operand. Operands are represented by pointers to the symbol table and operators by integer codes. To distinguish operators from operands, we might, for example, use negative integers for operator codes. In this implementation, a label is just an index into the array holding the code.

We also need an unconditional transfer operator **jump** and a variety of conditional jumps such as **jlt** or **jeqz**. The postfix expression l **jump** causes a transfer to label l. Expression $e_1\ e_2\ l$ **jlt** causes a jump to l if postfix expression e_1 has a smaller value than postfix expression e_2. Expression $e\ l$ **jeqz** causes a jump to l if e has the value zero. All jump and conditional jump operators cause their operands to be popped off the stack when evaluated, and no value is pushed onto the stack.

Example 7.5. Using the above jump operators, the conditional expression **if** e **then** x **else** y is expressed in postfix by $e\ l_1$ **jeqz** $x\ l_2$ **jump** l_1: $y\ l_2$:. Labels followed by a colon are not actually present in the code but are used to indicate positions in the code. The expression of Example 7.3 would be written

$$a l_1 \textbf{ jeqz } cd- l_2 \textbf{ jeqz } ac+ l_3 \textbf{ jump } l_2\text{: } ac* l_3 \textbf{ jump } l_1\text{: } ab+ l_3\text{:} \qquad \square$$

Syntax-Directed Translation to Postfix Code

The production of postfix intermediate code for expressions is simple. It is described by the syntax-directed translation scheme in Fig. 7.9. Here E.CODE is a string-valued translation. The value of the translation E.CODE for the first production is the concatenation of the two translations $E^{(1)}$.CODE and $E^{(2)}$.CODE and the symbol **op**, which stands for any operator symbol. In the second rule we see that the translation of a parenthesized expression is the same as that for the unparenthesized expression. The third rule tells us that the translation of any identifier is the identifier itself.

Production	Semantic Action
$E \rightarrow E^{(1)}$ **op** $E^{(2)}$	E.CODE := $E^{(1)}$.CODE || $E^{(2)}$.CODE || ' **op**'
$E \rightarrow (E^{(1)})$	E.CODE := $E^{(1)}$.CODE
$E \rightarrow$ **id**	E.CODE := **id**

Fig. 7.9. Syntax-directed translation scheme for infix-postfix translation.

The semantic actions in this translation scheme have a particularly simple form. The translation of the nonterminal on the left of each production is the concatenation of the translations of the nonterminals on the right in the same order as in the production, followed by some additional string (perhaps the empty string). Such a translation scheme is called *simple postfix*† and it can be implemented without a translation stack just by emitting the output string after each reduction.

Production	Program Fragment
$E \rightarrow E^{(1)}$ **op** $E^{(2)}$	{ **print op** }
$E \rightarrow (E^{(1)})$	{ }
$E \rightarrow$ **id**	{ **print id** }

Fig. 7.10. Implementation of infix-postfix translation.

† The term "postfix" here refers to the fact that all output is emitted at the end of each production, not to the fact that postfix code is being produced. Simple postfix translations can produce any kind of output.

The program fragments of Fig. 7.10 can be used for the scheme above. Thus when we reduce by the production $E \rightarrow$ **id**, we emit the identifier. On reduction by $E \rightarrow (E)$, we emit nothing. When we reduce by $E \rightarrow E$ **op** E we emit the operator **op**. By so doing we generate the postfix equivalent of the infix expression.

For example, processing the input $a+b*c$ a syntax-directed infix-to-postfix translator based on an LR parser, resolving ambiguities in the usual way, would make the sequence of moves shown in Fig. 7.11. In this example we view a, b, c, $+$, and $*$ as lexical values (analogous to LEXVAL in the desk calculator of Section 7.2) associated with **id** and **op**.

1. shift a
2. reduce by $E \rightarrow$ **id** and print a
3. shift $+$
4. shift b
5. reduce by $E \rightarrow$ **id** and print b
6. shift $*$
7. shift c
8. reduce by $E \rightarrow$ **id** and print c
9. reduce by $E \rightarrow E$ **op** E and print $*$
10. reduce by $E \rightarrow E$ **op** E and print $+$

Fig. 7.11. Sequence of moves.

7.5 Parse Trees and Syntax Trees

The parse tree itself is a useful intermediate-language representation for a source program, especially in optimizing compilers where the intermediate code needs to be extensively restructured. A parse tree, however, often contains redundant information which can be eliminated, thus producing a more economical representation of the source program. One such variant of a parse tree is what is called an *(abstract) syntax tree*, a tree in which each leaf represents an operand and each interior node an operator.

Example 7.6. The syntax tree for the expression $a*(b+c)/d$ is shown in Fig. 7.12(a). The syntax tree for statement **if** $a=b$ **then** $a:=c+d$ **else** $b:=c-d$ is shown in Fig. 7.12(b). □

Syntax-Directed Construction of Syntax Trees

Like postfix code, it is easy to define either a parse tree or a syntax tree in terms of a syntax-directed translation scheme. The scheme in Fig. 7.13 defines expressions. E.VAL is a translation whose value is a pointer to a node in the syntax tree.

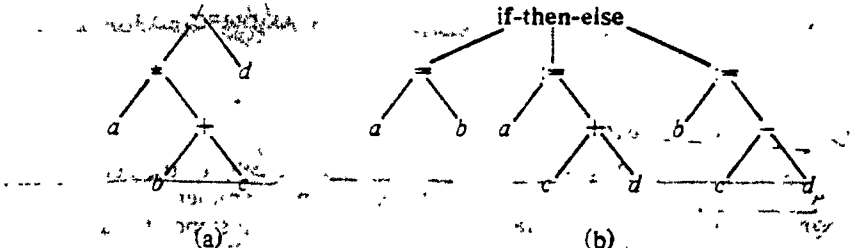

Fig. 7.12. Syntax trees.

Production	Semantic Action
(1) $E \rightarrow E^{(1)}$ **op** $E^{(2)}$	{E.VAL := NODE(**op**, $E^{(1)}$.VAL, $E^{(2)}$.VAL)}
(2) $E \rightarrow (E^{(1)})$	{E.VAL := $E^{(1)}$.VAL}
(3) $E \rightarrow -E^{(1)}$	{E.VAL := UNARY(−, $E^{(1)}$.VAL)}
(4) $E \rightarrow$ **id**	{E.VAL := LEAF(**id**)}

Fig. 7.13. Syntax-directed translation scheme to construct syntax trees.

The function NODE(OP, LEFT, RIGHT) takes three arguments. The first is the name of the operator, the second and third are pointers to roots of subtrees. The function creates a new node labeled by the first argument and makes the second and third arguments the left and right children of the new node, returning a pointer to the created node. The function UNARY(OP, CHILD) creates a new node labeled OP and makes CHILD its child. Again, a pointer to the created node is returned. The function LEAF(ID) creates a new node labeled by ID and returns a pointer to that node. This node receives no children. In practice the label of the leaf would be a representation of a particular name, such as a pointer to the symbol table.

7.6 Three-Address Code, Quadruples, and Triples

We shall now introduce our final category of intermediate code, known as three-address code. This intermediate code is preferred in many compilers, especially those doing extensive code optimization, since it allows the intermediate code to be rearranged in a convenient manner.

Three-Address Code

Three-address code is a sequence of statements, typically of the general form A := B **op** C, where A, B, and C are either programmer-defined names, constants or compiler-generated temporary names; **op** stands for any operator, such as a fixed- or floating-point arithmetic operator, or a logical operator on Boolean-valued data. The reason for the name "three-address code" is that each statement usually contains three addresses, two for the operands and one for the result. Note that no complicated arithmetic expressions are permitted, as there is only one operator per statement. Thus an expression like X + Y * Z would be "unraveled" to yield

$$T_1 := Y * Z$$
$$T_2 := X + T_1$$

where T_1 and T_2 are compiler-generated temporary names. It is this unravelling of complicated arithmetic expressions and of nested flow-of-control statements that makes three-address code more suitable for object-code generation than the source program itself would be.

Additional Three-Address Statements

There are a number of types of three-address statements that involve fewer than three addresses, or in which one of the "addresses" is not a name. In the following paragraphs we catalog the common kinds of three-address statements that we shall use in this book.

1. Assignment statements of the form A := B **op** C, where **op** is a binary arithmetic or logical operation. These instructions have been mentioned in the example above.
2. Assignment instructions of the form A := **op** B, where **op** is a unary operation. Essential unary operations include unary minus, logical negation, shift operators, and conversion operators that, for example, convert a fixed-point number to a floating-point number. An important special case of **op** is the identity function where A := B means that the value of B is assigned to A.
3. The unconditional jump **goto** L. The meaning of this instruction is to execute next the L*th* three-address statement.
4. Conditional jumps such as **if** A **relop** B **goto** L. This instruction applies relational operator **relop** (<, =, ≥, etc.) to A and B, and executes statement L next if A stands in relation **relop** to B. If not, the three-address statement following **if** A **relop** B **goto** L is executed next, as in the usual sequence.
5. **param** A and **call** P, *n*. These instructions are used to implement a procedure call. The typical use is as the sequence of three-address statements

param A_1
param A_2
.
.
.
param A_n
call P, *n*

generated as part of a call of procedure $P(A_1, A_2, \ldots, A_n)$. The *n* in "**call** P, *n*" is an integer indicating the number of actual parameters in the call. This information is redundant, as *n* can be computed by counting the number of **param** statements. It is a convenience to have *n* available with the **call** statement, however. Procedure calls will be discussed in more detail in Section 8.2.

6. Indexed assignments of the form A := B[I] and A[I] := B. The first of these sets A to the value in the location I memory units beyond location B. A[I] = B sets the location I units beyond A to the value of B. In both these instructions, A, B, and I are assumed to refer to data objects and will be represented by pointers to the symbol table.
7. Address and pointer assignments of the form A := **addr** B, A=*B, and *A=B. The first of these sets the value of A to be the location of B. Presumably B is a name, perhaps a temporary, which denotes an expression with an *l-value* such as X[I, J]. A is a pointer name or temporary. That is, the *r*-value of A is the *l-value* (location) of something else. In A:=*B, presumably B is a pointer or a temporary whose *r*-value is a location. The *r-value* of A is made equal to the contents of that location. Finally, *A:=B sets the *r*-value of the object pointed to by A to the *r*-value of B.

The three-address statement is an abstract form of intermediate code. In an actual compiler these statements can be implemented in one of the following ways.

Quadruples

We may use a record structure with four fields, which we shall call OP, ARG1, ARG2, and RESULT. This representation of three-address statements is known as *quadruples*. OP contains an internal code for the operator. There are actually two levels of specificity which the OP field may have. We may, as we translate to intermediate code, determine whether a binary operator like * means fixed- or floating-point multiplication, or perhaps another kind of multiplication, using a different operator code for different meanings. The next section contains an example of how we can determine the meaning for operators like *, introducing coercion of data types where necessary. The alternative is to use only one code for *,

leaving it to the object code generation phase to do the semantic checking, introduce the type coercions, and determine what type of multiplication is meant.

A three-address statement A := B op C puts B in ARG1, C in ARG2, and A in RESULT. Let us adopt the convention that statements with unary operators like A := −B or A:=B do not use ARG2. Operators like **param**, use neither ARG2 nor RESULT. Conditional and unconditional jumps put the target label in RESULT.

Example 7.7. An assignment statement like A := −B*(C+D) would be translated to three-address statements, using the straightforward algorithm of the next section, as follows.

$$\begin{aligned} T_1 &:= -B \\ T_2 &:= C+D \\ T_3 &:= T_1 * T_2 \\ A &:= T_3 \end{aligned}$$

These statements are represented by quadruples as shown in Fig. 7.14. □

	OP	ARG1	ARG2	RESULT
(0)	**uminus**	B		T_1
(1)	+	C	D	T_2
(2)	*	T_1	T_2	T_3
(3)	:=	T_3		A

Fig. 7.14. Quadruple representation of three-address statements.

The contents of fields ARG1, ARG2, and RESULT are, normally pointers to the symbol-table entries for the names represented by these fields. If so, temporary names must be entered into the symbol table as created. Chapter 10 describes ways in which temporary names can be reused to avoid cluttering up the symbol table.

Triples

To avoid entering temporary names into the symbol table, one can allow the statement computing a temporary value to represent that value. If we do so, three-address statements are representable by a structure with only three fields, OP, ARG1 and ARG2, where ARG1 and ARG2, the arguments of OP, are either pointers to the symbol table (for programmer-defined names or constants) or pointers into the structure itself (for temporary values). Since three fields are used, this intermediate code format is

known as *triples*.†

We use parenthesized numbers to represent pointers into the triple structure, while symbol-table pointers are represented by the names themselves. In practice, the information needed to interpret the different kinds of entries in the ARG1 and ARG2 fields can be encoded into the OP field or some additional fields.

Example 7.8. The three-address code from Example 7.7 can be implemented in triple form as shown in Fig. 7.15.

	OP	ARG1	ARG2
(0)	**uminus**	B	
(1)	+	C	D
(2)	*	(0)	(1)
(3)	:=	A	(2)

Fig. 7.15. Triple representation of three-address statements.

A ternary operation like A[I] := B actually requires two entries in the triple structure, as shown in Fig. 7.16(a), while A := B[I] is naturally represented as in Fig. 7.16(b). □

	OP	ARG1	ARG2
(0)	:=	I	B
(1)	:=	A	(0)

(a)

	OP	ARG1	ARG2
(0)	:=	B	I
(1)	:=	(0)	A

(b)

Fig. 7.16. More triple representations.

Indirect Triples

Another implementation of three-address code which has been considered is that of listing pointers to triples, rather than listing the triples themselves. This implementation is naturally called *indirect triples*.

Example 7.9. Let us use an array STATEMENT to list pointers to triples

† Some refer to triples as "two-address code," preferring to identify "quadruples" with the term "three-address code." We shall, however, treat "three-address code" as an abstract notion with various implementations, with triples and quadruples being the principal ones.

in the desired order. Then the three-address statements of Example 7.7 might be represented as in Fig. 7.17. □

	STATEMENT		OP	ARG1	ARG2
(0)	(14)	(14)	uminus	B	
(1)	(15)	(15)	+	C	D
(2)	(16)	(16)	*	(14)	(15)
(3)	(17)	(17)	:=	A	(16)

Fig. 7.17. Indirect triples representation of three-address statements.

Comparison of Representations: The Use of Indirection

We may regard the difference between triples and quadruples as a matter of how much indirection is present in the representation. When we ultimately produce object code, each datum, temporary or programmer-defined, will be assigned some memory location. This location will be placed in the symbol-table entry for the datum. Using the quadruple notation, the location for each temporary can be immediately accessed via the symbol table, from where it is needed — at the three-address statements defining or using that temporary. If the triples notation is used, we have no idea, unless we scan the code, how many temporaries are active simultaneously, or how many words must be allocated for temporaries; so with triples, the assignment of locations to temporaries is usually deferred to code generation.

A more important benefit of quadruples appears in an optimizing compiler, where we often move statements around. Using the quadruple notation, the symbol table interposes an extra degree of indirection between the computation of a value and its use. If we move a statement computing A, the statements using A require no change. However, in the triples notation, moving a statement that defines a temporary value requires us to change all pointers to that statement in the ARG1 and ARG2 arrays. This problem makes triples difficult to use in an optimizing compiler.

Indirect triples present no such problem. To move a statement we simply reorder the STATEMENT list. Since pointers to temporary values refer to the OP-ARG1-ARG2 arrays, which are not changed, none of those pointers need be changed. Thus, indirect triples look very much like quadruples as far as their utility is concerned. The two notations require about the same amount of space, and they are equally efficient for reordering of code. As with ordinary triples, allocation of storage to those temporaries needing it must be deferred to the code generation phase. However, indirect triples can save some space compared with quadruples if the same temporary value is used more than once. This is because two or more entries in the STATEMENT array can point to the same line of the OP-ARG1-ARG2 structure.

Quadruples present the problem that they tend to clutter up the symbol table with temporary names. If we use integer codes for temporaries and do not enter temporaries in the symbol table, we have the same problem as with triples when we assign locations to temporaries. In Chapter 10 we shall discuss methods of reusing temporary names to avoid this disadvantage of quadruples.

Single Array Representations

Both triples and quadruples waste some space, since fields will occasionally be empty. If space is important, one can use a single array and store either triples or quadruples consecutively. Since the operator determines which fields are actually in use, we can decode the single array if we follow each operator by those of ARG1, ARG2 and RESULT (if quadruples are being stored) which are actually in use. For example, the quadruples of Fig. 7.14 can be represented linearly as **uminus**, B, T_1, +, C, D, T_2, *, T_1, T_2, T_3, :=, T_3, A.

The disadvantage of this representation is seen if we try to examine the statements in reverse order, since we cannot tell just by looking at a word whether it represents an operator or operand. This problem is not artificial. For example, we shall see in Chapter 15 how code generation is facilitated if we preprocess triples or quadruples in a backward scan.

7.7 Translation of Assignment Statements

Because of the relative difficulty and importance of translation to three-address code, we shall consider translation of basic programming-language constructs into code of this form here and in the next two sections. Let us now give a syntax-directed translation scheme for simple assignment statements. To simplify the problem we assume that all identifiers denote primitive data types. In Chapter 8 we take up translation of expressions with array and record structure references. Here, we begin with a simple scheme in which semantic checking is not necessary. Then we consider what happens when operands can have a variety of types. The output of the translation is three-address code in each case; modifications to produce the other kinds of intermediate code are not hard.

Assignment Statements With Integer Types

Let us begin by considering simple assignment statements involving only integer variables. The following grammar describes the form of the assignment statements.

$$A \rightarrow \mathbf{id} := E$$

$$E \rightarrow E + E \mid E * E \mid -E \mid (E) \mid \mathbf{id}$$