7

Expressions and Assignment Statements

7.1 Introduction
7.2 Arithmetic Expressions
7.3 Overloaded Operators
7.4 Type Conversions
7.5 Relational and Boolean Expressions
7.6 Short-Circuit Evaluation
7.7 Assignment Statements
7.8 Mixed-Mode Assignment

As the title indicates, the topic of this chapter is expressions and assignment statements. The semantics rules that determine the order of evaluation of operators in expressions are discussed first. This is followed by an explanation of a potential problem with operand evaluation order when functions can have side effects. Overloaded operators, both predefined and user defined, are then discussed, along with their effects on the expressions in programs. Next, mixed-mode expressions are described and evaluated. This leads to the definition and evaluation of widening and narrowing type conversions, both implicit and explicit. Relational and Boolean expressions are then discussed, including the process of short-circuit evaluation. Finally, the assignment statement, from its simplest form to all of its variations, is covered, including assignments as expressions and mixed-mode assignments.

Character string pattern-matching expressions were covered as a part of the material on character strings in Chapter 6, so they are not mentioned in this chapter.

7.1 Introduction

Expressions are the fundamental means of specifying computations in a programming language. It is crucial for a programmer to understand both the syntax and semantics of expressions of the language he or she uses. A formal mechanism (BNF) for describing the syntax of expressions was introduced in Chapter 3. In this chapter, the semantics of expressions are discussed.

To understand expression evaluation, it is necessary to be familiar with the orders of operator and operand evaluation. The operator evaluation order of expressions is dictated by the associativity and precedence rules of the language. Although the value of an expression sometimes depends on it, the order of operand evaluation in expressions is often unstated by language designers. This allows implementors to choose the order, which leads to the possibility of programs producing different results in different implementations. Other issues in expression semantics are type mismatches, coercions, and short-circuit evaluation.

The essence of the imperative programming languages is the dominant role of assignment statements. The purpose of these statements is to cause the side effect of changing the values of variables, or the state, of the program. So an integral part of all imperative languages is the concept of variables whose values change during program execution.

Functional languages use variables of a different sort, such as the parameters of functions. These languages also have declaration statements that bind values to names. These declarations are similar to assignment statements, but do not have side effects.

7.2 Arithmetic Expressions

Automatic evaluation of arithmetic expressions similar to those found in mathematics, science, and engineering was one of the primary goals of the first high-level programming languages. Most of the characteristics of arithmetic

expressions in programming languages were inherited from conventions that had evolved in mathematics. In programming languages, arithmetic expressions consist of operators, operands, parentheses, and function calls. An operator can be **unary**, meaning it has a single operand, **binary**, meaning it has two operands, or **ternary**, meaning it has three operands.

In most programming languages, binary operators are **infix**, which means they appear between their operands. One exception is Perl, which has some operators that are **prefix**, which means they precede their operands. In Scheme and Lisp, all operators are prefix. Most unary operators are prefix, but the ++ and -- operators of C-based languages can be either prefix or postfix.

The purpose of an arithmetic expression is to specify an arithmetic computation. An implementation of such a computation must cause two actions: fetching the operands, usually from memory, and executing arithmetic operations on those operands. In the following sections, we investigate the common design details of arithmetic expressions.

Following are the primary design issues for arithmetic expressions, all of which are discussed in this section:

- What are the operator precedence rules?
- What are the operator associativity rules?
- What is the order of operand evaluation?
- Are there restrictions on operand evaluation side effects?
- Does the language allow user-defined operator overloading?
- What type mixing is allowed in expressions?

7.2.1 Operator Evaluation Order

The operator precedence and associativity rules of a language dictate the order of evaluation of its operators.

7.2.1.1 Precedence

The value of an expression depends at least in part on the order of evaluation of the operators in the expression. Consider the following expression:

```
a + b * c
```

Suppose the variables `a`, `b`, and `c` have the values `3`, `4`, and `5`, respectively. If evaluated left to right (the addition first and then the multiplication), the result is `35`. If evaluated right to left, the result is `23`.

Instead of simply evaluating the operators in an expression from left to right or right to left, mathematicians long ago developed the concept of placing operators in a hierarchy of evaluation priorities and basing the evaluation order of expressions partly on this hierarchy. For example, in mathematics, multiplication is considered to be of higher priority than addition, perhaps due to its higher level of complexity. If that convention were applied in the previous

example expression, as would be the case in most programming languages, the multiplication would be done first.

The **operator precedence rules** for expression evaluation partially define the order in which the operators of different precedence levels are evaluated. The operator precedence rules for expressions are based on the hierarchy of operator priorities, as seen by the language designer. The operator precedence rules of the common imperative languages are nearly all the same, because they are based on those of mathematics. In these languages, exponentiation has the highest precedence (when it is provided by the language), followed by multiplication and division on the same level, followed by binary addition and subtraction on the same level.

Many languages also include unary versions of addition and subtraction. Unary addition is called the **identity operator** because it usually has no associated operation and thus has no effect on its operand. Ellis and Stroustrup (1990, p. 56), speaking about C++, call it a historical accident and correctly label it useless. Unary minus, of course, changes the sign of its operand. In Java and C#, unary minus also causes the implicit conversion of **`short`** and **`byte`** operands to **`int`** type.

In all of the common imperative languages, the unary minus operator can appear in an expression either at the beginning or anywhere inside the expression, as long as it is parenthesized to prevent it from being next to another operator. For example,

```
a + (- b) * c
```

is legal, but

```
a + - b * c
```

usually is not.

Next, consider the following expressions:

```
- a / b
- a * b
- a ** b
```

In the first two cases, the relative precedence of the unary minus operator and the binary operator is irrelevant—the order of evaluation of the two operators has no effect on the value of the expression. In the last case, however, it does matter.

Of the common programming languages, only Fortran, Ruby, Visual Basic, and Ada have the exponentiation operator. In all four, exponentiation has higher precedence than unary minus, so

```
- A ** B
```

is equivalent to

```
-(A ** B)
```

The precedences of the arithmetic operators of Ruby and the C-based languages are as follows:

	Ruby	*C-Based Languages*
Highest	`**`	postfix `++`, `--`
	unary `+`, `-`	prefix `++`, `--`, unary `+`, `-`
	`*`, `/`, `%`	*, /, %
Lowest	binary `+`, `-`	binary `+`, `-`

The `**` operator is exponentiation. The % operator takes two integer operands and yields the remainder of the first after division by the second.[1] The `++` and `--` operators of the C-based languages are described in Section 7.7.4.

APL is odd among languages because it has a single level of precedence, as illustrated in the next section.

Precedence accounts for only some of the rules for the order of operator evaluation; associativity rules also affect it.

7.2.1.2 Associativity

Consider the following expression:

```
a - b + c - d
```

If the addition and subtraction operators have the same level of precedence, as they do in programming languages, the precedence rules say nothing about the order of evaluation of the operators in this expression.

When an expression contains two adjacent[2] occurrences of operators with the same level of precedence, the question of which operator is evaluated first is answered by the **associativity** rules of the language. An operator can have either left or right associativity, meaning that when there are two adjacent operators with the same precedence, the left operator is evaluated first or the right operator is evaluated first, respectively.

Associativity in common languages is left to right, except that the exponentiation operator (when provided) sometimes associates right to left. In the Java expression

```
a - b + c
```

the left operator is evaluated first.

1. In versions of C before C99, the % operator was implementation dependent in some situations, because division was also implementation dependent.
2. We call operators "adjacent" if they are separated by a single operand.

Exponentiation in Fortran and Ruby is right associative, so in the expression

```
A ** B ** C
```

the right operator is evaluated first.

In Visual Basic, the exponentiation operator, ^, is left associative.

The associativity rules for a few common languages are given here:

Language	*Associativity Rule*
Ruby	Left: *, /, +, -
	Right: **
C-based languages	Left: *, /, %, binary +, binary -
	Right: ++, --, unary -, unary +

As stated in Section 7.2.1.1, in APL, all operators have the same level of precedence. Thus, the order of evaluation of operators in APL expressions is determined entirely by the associativity rule, which is right to left for all operators. For example, in the expression

```
A × B + C
```

the addition operator is evaluated first, followed by the multiplication operator (× is the APL multiplication operator). If A were 3, B were 4, and C were 5, then the value of this APL expression would be 27.

Many compilers for the common languages make use of the fact that some arithmetic operators are mathematically associative, meaning that the associativity rules have no impact on the value of an expression containing only those operators. For example, addition is mathematically associative, so in mathematics the value of the expression

```
A + B + C
```

does not depend on the order of operator evaluation. If floating-point operations for mathematically associative operations were also associative, the compiler could use this fact to perform some simple optimizations. Specifically, if the compiler is allowed to reorder the evaluation of operators, it may be able to produce slightly faster code for expression evaluation. Compilers commonly do these kinds of optimizations.

Unfortunately, in a computer, both floating-point representations and floating-point arithmetic operations are only approximations of their mathematical counterparts (because of size limitations). The fact that a mathematical operator is associative does not necessarily imply that the corresponding computer floating-point operation is associative. In fact, only if all the operands and intermediate results can be exactly represented in floating-point notation will the process be precisely associative. For example, there are pathological

situations in which integer addition on a computer is *not* associative. For example, suppose that a program must evaluate the expression

```
A + B + C + D
```

and that `A` and `C` are very large positive numbers, and `B` and `D` are negative numbers with very large absolute values. In this situation, adding `B` to `A` does not cause an overflow exception, but adding `C` to `A` does. Likewise, adding `C` to `B` does not cause overflow, but adding `D` to `B` does. Because of the limitations of computer arithmetic, addition is catastrophically nonassociative in this case. Therefore, if the compiler reorders these addition operations, it affects the value of the expression. This problem, of course, can be avoided by the programmer, assuming the approximate values of the variables are known. The programmer can specify the expression in two parts (in two assignment statements), ensuring that overflow is avoided. However, this situation can arise in far more subtle ways, in which the programmer is less likely to notice the order dependence.

7.2.1.3 Parentheses

Programmers can alter the precedence and associativity rules by placing parentheses in expressions. A parenthesized part of an expression has precedence over its adjacent unparenthesized parts. For example, although multiplication has precedence over addition, in the expression

```
(A + B) * C
```

the addition will be evaluated first. Mathematically, this is perfectly natural. In this expression, the first operand of the multiplication operator is not available until the addition in the parenthesized subexpression is evaluated. Also, the expression from Section 7.2.1.2 could be specified as

```
(A + B) + (C + D)
```

to avoid overflow.

Languages that allow parentheses in arithmetic expressions could dispense with all precedence rules and simply associate all operators left to right or right to left. The programmer would specify the desired order of evaluation with parentheses. This approach would be simple because neither the author nor the readers of programs would need to remember any precedence or associativity rules. The disadvantage of this scheme is that it makes writing expressions more tedious, and it also seriously compromises the readability of the code. Yet this was the choice made by Ken Iverson, the designer of APL.

7.2.1.4 Ruby Expressions

Recall that Ruby is a pure object-oriented language, which means, among other things, that every data value, including literals, is an object. Ruby supports the collection of arithmetic and logic operations that are included in the C-based

languages. What sets Ruby apart from the C-based languages in the area of expressions is that all of the arithmetic, relational, and assignment operators, as well as array indexing, shifts, and bitwise logic operators, are implemented as methods. For example, the expression `a + b` is a call to the + method of the object referenced by `a`, passing the object referenced by `b` as a parameter.

One interesting result of the implementation of operators as methods is that they can be overridden by application programs. Therefore, these operators can be redefined. While it is often not useful to redefine operators for predefined types, it is useful, as we will see in Section 7.3, to define predefined operators for user-defined types, which can be done with operator overloading in some languages.

In C++ and Ada, operators are actually implemented as function calls.

7.2.1.5 Expressions in Lisp

As is the case with Ruby, all arithmetic and logic operations in Lisp are performed by subprograms. But in Lisp, the subprograms must be explicitly called. For example, to specify the C expression `a + b * c` in Lisp, one must write the following expression:[3]

```
(+ a (* b c))
```

In this expression, `+` and `*` are the names of functions.

7.2.1.6 Conditional Expressions

`if-then-else` statements can be used to perform a conditional expression assignment. For example, consider

```
if  (count == 0)
  average = 0;
else
  average = sum / count;
```

In the C-based languages, this code can be specified more conveniently in an assignment statement using a conditional expression, which has the following form:

expression_1 ? expression_2 : expression_3

where expression_1 is interpreted as a Boolean expression. If expression_1 evaluates to true, the value of the whole expression is the value of expression_2; otherwise, it is the value of expression_3. For example, the effect of the example **`if-then-else`** can be achieved with the following assignment statement, using a conditional expression:

```
average = (count == 0) ? 0 : sum / count;
```

3. When a list is interpreted as code in Lisp, the first element is the function name and others are parameters to the function.

In effect, the question mark denotes the beginning of the **then** clause, and the colon marks the beginning of the **else** clause. Both clauses are mandatory. Note that **?** is used in conditional expressions as a ternary operator.

Conditional expressions can be used anywhere in a program (in a C-based language) where any other expression can be used. In addition to the C-based languages, conditional expressions are provided in Perl, JavaScript, and Ruby.

7.2.2 Operand Evaluation Order

A less commonly discussed design characteristic of expressions is the order of evaluation of operands. Variables in expressions are evaluated by fetching their values from memory. Constants are sometimes evaluated the same way. In other cases, a constant may be part of the machine language instruction and not require a memory fetch. If an operand is a parenthesized expression, all of the operators it contains must be evaluated before its value can be used as an operand.

If neither of the operands of an operator has side effects, then operand evaluation order is irrelevant. Therefore, the only interesting case arises when the evaluation of an operand does have side effects.

7.2.2.1 Side Effects

A **side effect** of a function, naturally called a functional side effect, occurs when the function changes either one of its parameters or a global variable. (A global variable is declared outside the function but is accessible in the function.)

Consider the following expression:

```
a + fun(a)
```

If `fun` does not have the side effect of changing `a`, then the order of evaluation of the two operands, `a` and `fun(a)`, has no effect on the value of the expression. However, if `fun` changes `a`, there is an effect. Consider the following situation: `fun` returns `10` and changes the value of its parameter to `20`. Suppose we have the following:

```
a = 10;
b = a + fun(a);
```

Then, if the value of `a` is fetched first (in the expression evaluation process), its value is `10` and the value of the expression is `20`. But if the second operand is evaluated first, then the value of the first operand is `20` and the value of the expression is `30`.

The following C program illustrates the same problem when a function changes a global variable that appears in an expression:

```
int a = 5;
int fun1() {
```

```
    a = 17;
    return  3;
}    /* end of fun1 */
 void  main() {
    a = a + fun1();
}    /* end of main */
```

The value computed for `a` in `main` depends on the order of evaluation of the operands in the expression `a + fun1()`. The value of `a` will be either `8` (if `a` is evaluated first) or `20` (if the function call is evaluated first).

Note that functions in mathematics do not have side effects, because there is no notion of variables in mathematics. The same is true for functional programming languages. In both mathematics and functional programming languages, functions are much easier to reason about and understand than those in imperative languages, because their context is irrelevant to their meaning.

There are two possible solutions to the problem of operand evaluation order and side effects. First, the language designer could disallow function evaluation from affecting the value of expressions by simply disallowing functional side effects. Second, the language definition could state that operands in expressions are to be evaluated in a particular order and demand that implementors guarantee that order.

Disallowing functional side effects in the imperative languages is difficult, and it eliminates some flexibility for the programmer. Consider the case of C and C++, which have only functions, meaning that all subprograms return one value. To eliminate the side effects of two-way parameters and still provide subprograms that return more than one value, the values would need to be placed in a struct and the struct returned. Access to globals in functions would also have to be disallowed. However, when efficiency is important, using access to global variables to avoid parameter passing is an important method of increasing execution speed. In compilers, for example, global access to data such as the symbol table is commonplace.

The problem with having a strict evaluation order is that some code optimization techniques used by compilers involve reordering operand evaluations. A guaranteed order disallows those optimization methods when function calls are involved. There is, therefore, no perfect solution, as is borne out by actual language designs.

The Java language definition guarantees that operands appear to be evaluated in left-to-right order, eliminating the problem discussed in this section.

7.2.2.2 Referential Transparency and Side Effects

The concept of referential transparency is related to and affected by functional side effects. A program has the property of **referential transparency** if any two expressions in the program that have the same value can be substituted for one another anywhere in the program, without affecting the action of the program. The value of a referentially transparent function depends entirely on its

parameters.[4] The connection of referential transparency and functional side effects is illustrated by the following example:

```
result1 = (fun(a) + b) / (fun(a) - c);
temp = fun(a);
result2 = (temp + b) / (temp - c);
```

If the function `fun` has no side effects, `result1` and `result2` will be equal, because the expressions assigned to them are equivalent. However, suppose `fun` has the side effect of adding 1 to either `b` or `c`. Then `result1` would not be equal to `result2`. So, that side effect violates the referential transparency of the program in which the code appears.

There are several advantages to referentially transparent programs. The most important of these is that the semantics of such programs is much easier to understand than the semantics of programs that are not referentially transparent. Being referentially transparent makes a function equivalent to a mathematical function, in terms of ease of understanding.

Because they do not have variables, programs written in pure functional languages are referentially transparent. Functions in a pure functional language cannot have state, which would be stored in local variables. If such a function uses a value from outside the function, that value must be a constant, since there are no variables. Therefore, the value of the function depends on the values of its parameters.

Referential transparency will be further discussed in Chapter 15.

7.3 Overloaded Operators

Arithmetic operators are often used for more than one purpose. For example, + usually is used to specify integer addition and floating-point addition. Some languages—Java, for example—also use it for string catenation. This multiple use of an operator is called **operator overloading** and is generally thought to be acceptable, as long as neither readability nor reliability suffers.

As an example of the possible dangers of overloading, consider the use of the ampersand (`&`) in C++. As a binary operator, it specifies a bitwise logical AND operation. As a unary operator, however, its meaning is totally different. As a unary operator with a variable as its operand, the expression value is the address of that variable. In this case, the ampersand is called the address-of operator. For example, the execution of

```
x = &y;
```

4. Furthermore, the value of the function cannot depend on the order in which its parameters are evaluated.

causes the address of `y` to be placed in `x`. There are two problems with this multiple use of the ampersand. First, using the same symbol for two completely unrelated operations is detrimental to readability. Second, the simple keying error of leaving out the first operand for a bitwise AND operation can go undetected by the compiler, because it is interpreted as an address-of operator. Such an error may be difficult to diagnose.

Virtually all programming languages have a less serious but similar problem, which is often due to the overloading of the minus operator. The problem is only that the compiler cannot tell if the operator is meant to be binary or unary.[5] So once again, failure to include the first operand when the operator is meant to be binary cannot be detected as an error by the compiler. However, the meanings of the two operations, unary and binary, are at least closely related, so readability is not adversely affected.

Some languages that support abstract data types (see Chapter 11), for example, C++, C#, and F#, allow the programmer to further overload operator symbols. For instance, suppose a user wants to define the `*` operator between a scalar integer and an integer array to mean that each element of the array is to be multiplied by the scalar. Such an operator could be defined by writing a function subprogram named `*` that performs this new operation. The compiler will choose the correct meaning when an overloaded operator is specified, based on the types of the operands, as with language-defined overloaded operators. For example, if this new definition for `*` is defined in a C# program, a C# compiler will use the new definition for `*` whenever the `*` operator appears with a simple integer as the left operand and an integer array as the right operand.

When sensibly used, user-defined operator overloading can aid readability. For example, if `+` and `*` are overloaded for a matrix abstract data type and `A`, `B`, `C`, and `D` are variables of that type, then

```
A * B + C * D
```

can be used instead of

```
MatrixAdd(MatrixMult(A, B), MatrixMult(C, D))
```

On the other hand, user-defined overloading can be harmful to readability. For one thing, nothing prevents a user from defining `+` to mean multiplication. Furthermore, seeing an `*` operator in a program, the reader must find both the types of the operands and the definition of the operator to determine its meaning. Any or all of these definitions could be in other files.

Another consideration is the process of building a software system from modules created by different groups. If the different groups overloaded the same operators in different ways, these differences would obviously need to be eliminated before putting the system together.

---

5. ML alleviates this problem by using different symbols for unary and binary minus operators, tilde (~) for unary and dash (−) for binary.

C++ has a few operators that cannot be overloaded. Among these are the class or structure member operator (.) and the scope resolution operator (::). Interestingly, operator overloading was one of the C++ features that was not copied into Java. However, it did reappear in C#.

The implementation of user-defined operator overloading is discussed in Chapter 9.

7.4 Type Conversions

Type conversions are either narrowing or widening. A **narrowing conversion** converts a value to a type that cannot store even approximations of all of the values of the original type. For example, converting a `double` to a `float` in Java is a narrowing conversion, because the range of `double` is much larger than that of `float`. A **widening conversion** converts a value to a type that can include at least approximations of all of the values of the original type. For example, converting an `int` to a `float` in Java is a widening conversion. Widening conversions are nearly always safe, meaning that the approximate magnitude of the converted value is maintained. Narrowing conversions are not always safe—sometimes the magnitude of the converted value is changed in the process. For example, if the floating-point value 1.3E25 is converted to an integer in a Java program, the result will not be in any way related to the original value.

Although widening conversions are usually safe, they can result in reduced accuracy. In many language implementations, although integer-to-floating-point conversions are widening conversions, some precision may be lost. For example, in many cases, integers are stored in 32 bits, which allows at least 9 decimal digits of precision. But floating-point values are also stored in 32 bits, with only about seven decimal digits of precision (because of the space used for the exponent). So, integer-to-floating-point widening can result in the loss of two digits of precision.

Coercions of nonprimitive types are, of course, more complex. In Chapter 5, the complications of assignment compatibility of array and record types were discussed. There is also the question of what parameter types and return types of a method allow it to override a method in a superclass—only when the types are the same, or also some other situations. That issue, as well as the concept of subclasses as subtypes, are discussed in Chapter 12.

Type conversions can be either explicit or implicit. The following two subsections discuss these kinds of type conversions.

7.4.1 Coercion in Expressions

One of the design decisions concerning arithmetic expressions is whether an operator can have operands of different types. Languages that allow such expressions, which are called **mixed-mode expressions**, must define conventions for implicit operand type conversions because computers do not have binary operations that take operands of different types. Recall that in Chapter 5,

coercion was defined as an implicit type conversion that is initiated by the compiler or runtime system. Type conversions explicitly requested by the programmer are referred to as explicit conversions, or casts, not coercions.

Although some operator symbols may be overloaded, we assume that a computer system, either in hardware or in some level of software simulation, has an operation for each operand type and operator defined in the language.[6] For overloaded operators in a language that uses static type binding, the compiler chooses the correct type of operation on the basis of the types of the operands. When the two operands of an operator are not of the same type and that is legal in the language, the compiler must choose one of them to be coerced and generate the code for that coercion. In the following discussion, the coercion design choices of several common languages are examined.

Language designers are not in agreement on the issue of coercions in arithmetic expressions. Those against a broad range of coercions are concerned with the reliability problems that can result from such coercions, because they reduce the benefits of type checking. Those who would rather include a wide range of coercions are more concerned with the loss in flexibility that results from restrictions. The issue is whether programmers should need to be concerned with this category of errors or whether the compiler should detect them.

As a simple illustration of the problem, consider the following Java code:

```
int a;
float b, c, d;
. . .
d = b * a;
```

Assume that the second operand of the multiplication operator was supposed to be `c`, but because of a keying error it was typed as `a`. Because mixed-mode expressions are legal in Java, the compiler would not detect this as an error. It would simply insert code to coerce the value of the **`int`** operand, `a`, to **`float`**. If mixed-mode expressions were not legal in Java, this keying error would have been detected by the compiler as a type error.

Because error detection is reduced when mixed-mode expressions are allowed, F#, Ada, and ML do not allow them. For example, they do not allow mixing of integer and floating-point operands in expressions.

In most of the other common languages, there are no restrictions on mixed-mode arithmetic expressions.

The C-based languages have integer types that are smaller than the **`int`** type. In Java, these are **`byte`** and **`short`**. Operands of all of these types are coerced to **`int`** whenever virtually any operator is applied to them. So, while data can be stored in variables of these types, it cannot be manipulated before conversion to a larger type. For example, consider the following Java code:

---

6. This assumption is not true for many languages. An example is given later in this section.

history note

As a more extreme example of the dangers and costs of too much coercion, consider PL/I's efforts to achieve flexibility in expressions. In PL/I, a character string variable can be the operand of an arithmetic operator with an integer as the other operand. At run time, the string is scanned for a numeric value. If the value happens to contain a decimal point, the value is assumed to be of floating-point type, the other operand is coerced to floating point, and the resulting operation is floating-point. This coercion policy is very expensive, because both the type check and the conversion must be done at run time. It also eliminates the possibility of detecting programmer errors in expressions, because a binary operator can combine an operand of any type with an operand of virtually any other type.

```
byte  a, b, c;
. . .
a = b + c;
```

The values of `b` and `c` are coerced to **`int`** and an **`int`** addition is performed. Then, the sum is converted to **`byte`** and put in `a`. Given the large size of the memories of contemporary computers, there is little incentive to use **`byte`** and **`short`**, unless a large number of them must be stored.

7.4.2 Explicit Type Conversion

Most languages provide some capability for doing explicit conversions, both widening and narrowing. In some cases, warning messages are produced when an explicit narrowing conversion results in a significant change to the value of the object being converted.

In the C-based languages, explicit type conversions are called **casts**. To specify a cast, the desired type is placed in parentheses just before the expression to be converted, as in

```
(int)angle
```

One of the reasons for the parentheses around the type name in these conversions is that the first of these languages, C, has several two-word type names, such as **`long int`**.

In ML and F#, the casts have the syntax of function calls. For example, in F# we could have the following:

```
float(sum)
```

7.4.3 Errors in Expressions

A number of errors can occur during expression evaluation. If the language requires type checking, either static or dynamic, then operand type errors cannot occur. The errors that can occur because of coercions of operands in expressions have already been discussed. The other kinds of errors are due to the limitations of computer arithmetic and the inherent limitations of arithmetic. The most common error occurs when the result of an operation cannot be represented in the memory cell where it must be stored. This is called **overflow** or **underflow**, depending on whether the result was too large or too small. One limitation of arithmetic is that division by zero is disallowed. Of course, the fact that it is not mathematically allowed does not prevent a program from attempting to do it.

Floating-point overflow, underflow, and division by zero are examples of run-time errors, which are sometimes called **exceptions**. Language facilities that allow programs to detect and deal with exceptions are discussed in Chapter 14.

7.5 Relational and Boolean Expressions

In addition to arithmetic expressions, programming languages support relational and Boolean expressions.

7.5.1 Relational Expressions

A **relational operator** is an operator that compares the values of its two operands. A relational expression has two operands and one relational operator. The value of a relational expression is Boolean, except when Boolean is not a type included in the language. The relational operators are often overloaded for a variety of types. The operation that determines the truth or falsehood of a relational expression depends on the operand types. It can be simple, as for integer operands, or complex, as for character string operands. Typically, the types of the operands that can be used for relational operators are numeric types, strings, and enumeration types.

history note

The Fortran I designers used English abbreviations for the relational operators because the symbols > and < were not on the card punches at the time of Fortran I's design (mid-1950s).

The syntax of the relational operators for equality and inequality differs among some programming languages. For example, for inequality, the C-based languages use `!=`, Fortran 95+ uses `.NE.` or <>, and ML and F# use <>.

JavaScript and PHP have two additional relational operators, === and `!==`. These are similar to their relatives, == and `!=`, but prevent their operands from being coerced. For example, the expression

```
"7" == 7
```

is true in JavaScript, because when a string and a number are the operands of a relational operator, the string is coerced to a number. However,

```
"7" === 7
```

is false, because no coercion is done on the operands of this operator.

Ruby uses == for the equality relational operator that uses coercions, and `eql?` for equality with no coercions. Ruby uses === only in the **`when`** clause of its **`case`** statement, as discussed in Chapter 8.

The relational operators always have lower precedence than the arithmetic operators, so that in expressions such as

```
a + 1 > 2 * b
```

the arithmetic expressions are evaluated first.

7.5.2 Boolean Expressions

Boolean expressions consist of Boolean variables, Boolean constants, relational expressions, and Boolean operators. The operators usually include those for the AND, OR, and NOT operations, and sometimes for exclusive OR and

equivalence. Boolean operators usually take only Boolean operands (Boolean variables, Boolean literals, or relational expressions) and produce Boolean values.

In the mathematics of Boolean algebras, the OR and AND operators must have equal precedence. However, the C-based languages assign a higher precedence to AND than OR. Perhaps this resulted from the baseless correlation of multiplication with AND and of addition with OR, which would naturally assign higher precedence to AND.

Because arithmetic expressions can be the operands of relational expressions, and relational expressions can be the operands of Boolean expressions, the three categories of operators must be placed in different precedence levels, relative to each other.

The precedence of the arithmetic, relational, and Boolean operators in the C-based languages is as follows:

Highest	postfix ++, --
	unary +, unary -, prefix ++, --, !
	*, /, %
	binary +, binary -
	<, >, <=, >=
	=, !=
	&&
Lowest	||

Versions of C prior to C99 are odd among the popular imperative languages in that they have no Boolean type and thus no Boolean values. Instead, numeric values are used to represent Boolean values. In place of Boolean operands, scalar variables (numeric or character) and constants are used, with zero considered false and all nonzero values considered true. The result of evaluating such an expression is an integer, with the value 0 if false and 1 if true. Arithmetic expressions can also be used for Boolean expressions in C99 and C++.

One odd result of C's design of relational expressions is that the following expression is legal:

```
a > b > c
```

The leftmost relational operator is evaluated first because the relational operators of C are left associative, producing either 0 or 1. Then, this result is compared with the variable `c`. There is never a comparison between `b` and `c` in this expression.

Some languages, including Perl and Ruby, provide two sets of the binary logic operators, `&&` and **`and`** for AND and `||` and **`or`** for OR. One difference between `&&` and **`and`** (and `||` and **`or`**) is that the spelled versions have lower precedence. Also, **`and`** and **`or`** have equal precedence, but `&&` has higher precedence than `||`.

When the nonarithmetic operators of the C-based languages are included, there are more than 40 operators and at least 14 different levels of precedence. This is clear evidence of the richness of the collections of operators and the complexity of expressions possible in these languages.

Readability dictates that a language should include a Boolean type, as was stated in Chapter 6, rather than simply using numeric types in Boolean expressions. Some error detection is lost in the use of numeric types for Boolean operands, because any numeric expression, whether intended or not, is a legal operand to a Boolean operator. In the other imperative languages, any non-Boolean expression used as an operand of a Boolean operator is detected as an error.

7.6 Short-Circuit Evaluation

A **short-circuit evaluation** of an expression is one in which the result is determined without evaluating all of the operands and/or operators. For example, the value of the arithmetic expression

```
(13 * a) * (b / 13 - 1)
```

is independent of the value of `(b / 13 - 1)` if `a` is `0`, because `0 * x = 0` for any `x`. So, when `a` is `0`, there is no need to evaluate `(b / 13 - 1)` or perform the second multiplication. However, in arithmetic expressions, this shortcut is not easily detected during execution, so it is never taken.

The value of the Boolean expression

```
(a >= 0) && (b < 10)
```

is independent of the second relational expression if `a < 0`, because the expression (`FALSE && (b < 10)`) is `FALSE` for all values of `b`. So, when `a` is less than zero, there is no need to evaluate `b`, the constant `10`, the second relational expression, or the `&&` operation. Unlike the case of arithmetic expressions, this shortcut easily can be discovered during execution.

To illustrate a potential problem with non-short-circuit evaluation of Boolean expressions, suppose Java did not use short-circuit evaluation. A table lookup loop could be written using the **`while`** statement. One simple version of Java code for such a lookup, assuming that `list`, which has `listlen` elements, is the array to be searched and `key` is the searched-for value, is

```
index = 0;
while ((index < listlen) && (list[index] != key))
  index = index + 1;
```

If evaluation is not short-circuit, both relational expressions in the Boolean expression of the **`while`** statement are evaluated, regardless of the value of the

first. Thus, if `key` is not in `list`, the program will terminate with a subscript out-of-range exception. The same iteration that has `index == listlen` will reference `list[listlen]`, which causes the indexing error because list is declared to have `listlen-1` as an upper-bound subscript value.

If a language provides short-circuit evaluation of Boolean expressions and it is used, this is not a problem. In the preceding example, a short-circuit evaluation scheme would evaluate the first operand of the AND operator, but it would skip the second operand if the first operand is false.

A language that provides short-circuit evaluations of Boolean expressions and also has side effects in expressions allows subtle errors to occur. Suppose that short-circuit evaluation is used on an expression and part of the expression that contains a side effect is not evaluated; then the side effect will occur only in complete evaluations of the whole expression. If program correctness depends on the side effect, short-circuit evaluation can result in a serious error. For example, consider the Java expression

```
(a > b) || ((b++) / 3)
```

In this expression, `b` is changed (in the second arithmetic expression) only when `a <= b`. If the programmer assumed `b` would be changed every time this expression is evaluated during execution (and the program's correctness depends on it), the program will fail.

In the C-based languages, the usual AND and OR operators, `&&` and `||`, respectively, are short-circuit. However, these languages also have bitwise AND and OR operators, `&` and `|`, respectively, that can be used on Boolean-valued operands and are not short-circuit. Of course, the bitwise operators are only equivalent to the usual Boolean operators if all operands are restricted to being either `0` (for false) or `1` (for true).

All of the logical operators of Ruby, Perl, ML, F#, and Python are short-circuit evaluated.

7.7 Assignment Statements

As we have previously stated, the assignment statement is one of the central constructs in imperative languages. It provides the mechanism by which the user can dynamically change the bindings of values to variables. In the following section, the simplest form of assignment is discussed. Subsequent sections describe a variety of alternatives.

7.7.1 Simple Assignments

Nearly all programming languages currently being used use the equal sign for the assignment operator. All of these must use something different from an equal sign for the equality relational operator to avoid confusion with their assignment operator.

ALGOL 60 pioneered the use of := as the assignment operator, which avoids the confusion of assignment with equality. Ada also uses this assignment operator.

The design choices of how assignments are used in a language have varied widely. In some languages, such as Fortran and Ada, an assignment can appear only as a stand-alone statement, and the destination is restricted to a single variable. There are, however, many alternatives.

7.7.2 Conditional Targets

Perl allows conditional targets on assignment statements. For example, consider

```
($flag ? $count1 : $count2) = 0;
```

which is equivalent to

```
if ($flag) {
  $count1 = 0;
} else  {
  $count2 = 0;
}
```

7.7.3 Compound Assignment Operators

A **compound assignment operator** is a shorthand method of specifying a commonly needed form of assignment. The form of assignment that can be abbreviated with this technique has the destination variable also appearing as the first operand in the expression on the right side, as in

```
a = a + b
```

Compound assignment operators were introduced by ALGOL 68, were later adopted in a slightly different form by C, and are part of the other C-based languages, as well as Perl, JavaScript, Python, and Ruby. The syntax of these assignment operators is the catenation of the desired binary operator to the = operator. For example,

```
sum += value;
```

is equivalent to

```
sum = sum + value;
```

The languages that support compound assignment operators have versions for most of their binary operators.

7.7.4 Unary Assignment Operators

The C-based languages, Perl, and JavaScript include two special unary arithmetic operators that are actually abbreviated assignments. They combine increment and decrement operations with assignment. The operators `++` for increment and `--` for decrement can be used either in expressions or to form stand-alone single-operator assignment statements. They can appear either as prefix operators, meaning that they precede the operands, or as postfix operators, meaning that they follow the operands. In the assignment statement

```
sum = ++ count;
```

the value of `count` is incremented by `1` and then assigned to `sum`. This operation could also be stated as

```
count = count + 1;
sum = count;
```

If the same operator is used as a postfix operator, as in

```
sum = count ++;
```

the assignment of the value of `count` to `sum` occurs first; then `count` is incremented. The effect is the same as that of the two statements

```
sum = count;
count = count + 1;
```

An example of the use of the unary increment operator to form a complete assignment statement is

```
count ++;
```

which simply increments `count`. It does not look like an assignment, but it certainly is one. It is equivalent to the statement

```
count = count + 1;
```

When two unary operators apply to the same operand, the association is right to left. For example, in

```
- count ++
```

`count` is first incremented and then negated. So, it is equivalent to

```
- (count ++)
```

rather than

```
(- count) ++
```

7.7.5 Assignment as an Expression

In the C-based languages, Perl, and JavaScript, the assignment statement produces a result, which is the same as the value assigned to the target. It can therefore be used as an expression and as an operand in other expressions. This design treats the assignment operator much like any other binary operator, except that it has the side effect of changing its left operand. For example, in C, it is common to write statements such as

```
while ((ch = getchar()) != EOF) { ... }
```

In this statement, the next character from the standard input file, usually the keyboard, is gotten with `getchar` and assigned to the variable `ch`. The result, or value assigned, is then compared with the constant `EOF`. If `ch` is not equal to `EOF`, the compound statement `{...}` is executed. Note that the assignment must be parenthesized—in the languages that support assignment as an expression, the precedence of the assignment operator is lower than that of the relational operators. Without the parentheses, the new character would be compared with `EOF` first. Then, the result of that comparison, either `0` or `1`, would be assigned to `ch`.

The disadvantage of allowing assignment statements to be operands in expressions is that it provides yet another kind of expression side effect. This type of side effect can lead to expressions that are difficult to read and understand. An expression with any kind of side effect has this disadvantage. Such an expression cannot be read as an expression, which in mathematics is a denotation of a value, but only as a list of instructions with an odd order of execution. For example, the expression

```
a = b + (c = d / b) - 1
```

denotes the instructions

```
Assign d / b to c
Assign b + c to temp
Assign temp - 1 to a
```

Note that the treatment of the assignment operator as any other binary operator allows the effect of multiple-target assignments, such as

```
sum = count = 0;
```

in which count is first assigned the zero, and then count's value is assigned to sum. This form of multiple-target assignments is also legal in Python.

There is a loss of error detection in the C design of the assignment operation that frequently leads to program errors. In particular, if we type

```
if (x = y) ...
```

instead of

```
if (x == y) ...
```

which is an easily made mistake, it is not detectable as an error by the compiler. Rather than testing a relational expression, the value that is assigned to `x` is tested (in this case, it is the value of `y` that reaches this statement). This is actually a result of two design decisions: allowing assignment to behave like an ordinary binary operator and using two very similar operators, = and ==, to have completely different meanings. This is another example of the safety deficiencies of C and C++ programs. Note that Java and C# allow only **`boolean`** expressions in their **`if`** statements, disallowing this problem.

history note

The PDP-11 computer, on which C was first implemented, has autoincrement and autodecrement addressing modes, which are hardware versions of the increment and decrement operators of C when they are used as array indices. One might guess from this that the design of these C operators was based on the design of the PDP-11 architecture. That guess would be wrong, however, because the C operators were inherited from the B language, which was designed before the first PDP-11.

7.7.6 Multiple Assignments

Several recent programming languages, including Perl and Ruby provide multiple-target, multiple-source assignment statements. For example, in Perl one can write

```
($first, $second, $third) = (20, 40, 60);
```

The semantics is that `20` is assigned to `$first`, `40` is assigned to `$second`, and `60` is assigned to `$third`. If the values of two variables must be interchanged, this can be done with a single assignment, as with

```
($first, $second) = ($second, $first);
```

This correctly interchanges the values of `$first` and `$second`, without the use of a temporary variable (at least one created and managed by the programmer).

The syntax of the simplest form of Ruby's multiple assignment is similar to that of Perl, except the left and right sides are not parenthesized. Also, Ruby includes a few more elaborate versions of multiple assignments, which are not discussed here.

7.7.7 Assignment in Functional Programming Languages

All of the identifiers used in pure functional languages and some of them used in other functional languages are just names of values. As such, their values never change. For example, in ML, names are bound to values with the **`val`** declaration, whose form is exemplified in the following:

```
val cost = quantity * price;
```

If `cost` appears on the left side of a subsequent **`val`** declaration, that declaration creates a new version of the name `cost`, which has no relationship with the previous version, which is then hidden.

F# has a somewhat similar declaration that uses the **`let`** reserved word. The difference between F#'s **`let`** and ML's **`val`** is that **`let`** creates a new scope,

whereas **val** does not. In fact, **val** declarations are often nested in **let** constructs in ML. **let** and **val** are further discussed in Chapter 15.

7.8 Mixed-Mode Assignment

Mixed-mode expressions were discussed in Section 7.4.1. Frequently, assignment statements also are mixed mode. The design question is: Does the type of the expression have to be the same as the type of the variable being assigned, or can coercion be used in some cases of type mismatch?

C, C++, and Perl use coercion rules for mixed-mode assignment that are similar to those they use for mixed-mode expressions; that is, many of the possible type mixes are legal, with coercion freely applied.[7]

In a clear departure from C++, Java and C# allow mixed-mode assignment only if the required coercion is widening.[8] So, an **int** value can be assigned to a **float** variable, but not vice versa. Disallowing half of the possible mixed-mode assignments is a simple but effective way to increase the reliability of Java and C#, relative to C and C++.

Of course, in functional languages, where assignments are just used to name values, there is no such thing as a mixed-mode assignment.

SUMMARY

Expressions consist of constants, variables, parentheses, function calls, and operators. Assignment statements include target variables, assignment operators, and expressions.

The semantics of an expression is determined in large part by the order of evaluation of operators. The associativity and precedence rules for operators in the expressions of a language determine the order of operator evaluation in those expressions. Operand evaluation order is important if functional side effects are possible. Type conversions can be widening or narrowing. Some narrowing conversions produce erroneous values. Implicit type conversions, or coercions, in expressions are common, although they eliminate the error-detection benefit of type checking, thus lowering reliability.

Assignment statements have appeared in a wide variety of forms, including conditional targets, assigning operators, and list assignments.

---

7. Note that in Python and Ruby, types are associated with objects, not variables, so there is no such thing as mixed-mode assignment in those languages.

8. Not quite true: If an integer literal, which the compiler by default assigns the type **int**, is assigned to a **char**, **byte**, or **short** variable and the literal is in the range of the type of the variable, the **int** value is coerced to the type of the variable in a narrowing conversion. This narrowing conversion cannot result in an error.

REVIEW QUESTIONS

1. Define *operator precedence* and *operator associativity*.
2. What is a ternary operator?
3. What is a prefix operator?
4. What operator usually has right associativity?
5. What is a nonassociative operator?
6. What associativity rules are used by APL?
7. What is the difference between the way operators are implemented in C++ and Ruby?
8. Define *functional side effect*.
9. What is a coercion?
10. What is a conditional expression?
11. What is an overloaded operator?
12. Define *narrowing* and *widening conversions*.
13. In JavaScript, what is the difference between == and ===?
14. What is a mixed-mode expression?
15. What is referential transparency?
16. What are the advantages of referential transparency?
17. How does operand evaluation order interact with functional side effects?
18. What is short-circuit evaluation?
19. Name a language that always does short-circuit evaluation of Boolean expressions. Name one that never does it.
20. How does C support relational and Boolean expressions?
21. What is the purpose of a compound assignment operator?
22. What is the associativity of C's unary arithmetic operators?
23. What is one possible disadvantage of treating the assignment operator as if it were an arithmetic operator?
24. What two languages include multiple assignments?
25. What mixed-mode assignments are allowed in Java?
26. What mixed-mode assignments are allowed in ML?
27. What is a cast?

PROBLEM SET

1. When might you want the compiler to ignore type differences in an expression?
2. State your own arguments for and against allowing mixed-mode arithmetic expressions.

3. Do you think the elimination of overloaded operators in your favorite language would be beneficial? Why or why not?
4. Would it be a good idea to eliminate all operator precedence rules and require parentheses to show the desired precedence in expressions? Why or why not?
5. Should C's assigning operations (for example, +=) be included in other languages (that do not already have them)? Why or why not?
6. Should C's single-operand assignment forms (for example, `++count`) be included in other languages (that do not already have them)? Why or why not?
7. Describe a situation in which the add operator in a programming language would not be commutative.
8. Describe a situation in which the add operator in a programming language would not be associative.
9. Assume the following rules of associativity and precedence for expressions:

Precedence	*Highest*	*, /, **not**
		+, −, &, **mod**
		- (unary)
		=, / =, <, <=, >=, >
		and
	Lowest	**or**, **xor**
Associativity	*Left to right*	

Show the order of evaluation of the following expressions by parenthesizing all subexpressions and placing a superscript on the right parenthesis to indicate order. For example, for the expression

```
a + b * c + d
```

the order of evaluation would be represented as

```
((a + (b * c)1)2  + d)3
```

a. `a * b - 1 + c`
b. `a * (b - 1) / c` **mod** `d`
c. `(a - b) / c & (d * e / a - 3)`
d. `-a` **or** `c = d` **and** `e`
e. `a > b` **xor** `c` **or** `d <= 17`
f. `-a + b`

10. Show the order of evaluation of the expressions of Problem 9, assuming that there are no precedence rules and all operators associate right to left.

11. Write a BNF description of the precedence and associativity rules defined for the expressions in Problem 9. Assume the only operands are the names `a,b,c,d,` and `e`.
12. Using the grammar of Problem 11, draw parse trees for the expressions of Problem 9.
13. Let the function `fun` be defined as

```
int fun(int* k) {
 *k += 4;
 return 3 * (*k) - 1;
 }
```

Suppose `fun` is used in a program as follows:

```
void  main() {
  int i = 10, j = 10, sum1, sum2;
  sum1 = (i / 2) + fun(&i);
  sum2 = fun(&j) + (j / 2);
}
```

What are the values of `sum1` and `sum2`

a. operands in the expressions are evaluated left to right?
b. operands in the expressions are evaluated right to left?

14. What is your primary argument against (or for) the operator precedence rules of APL?
15. Explain why it is difficult to eliminate functional side effects in C.
16. For some language of your choice, make up a list of operator symbols that could be used to eliminate all operator overloading.
17. Determine whether the narrowing explicit type conversions in two languages you know provide error messages when a converted value loses its usefulness.
18. Should an optimizing compiler for C or C++ be allowed to change the order of subexpressions in a Boolean expression? Why or why not?
19. Consider the following C program:

```
int fun(int *i) {
  *i += 5;
  return  4;
}
void main() {
  int x = 3;
  x = x + fun(&x);
  }
```

What is the value of `x` after the assignment statement in main, assuming

a. operands are evaluated left to right.

b. operands are evaluated right to left.

20. Why does Java specify that operands in expressions are all evaluated in left-to-right order?
21. Explain how the coercion rules of a language affect its error detection.

PROGRAMMING EXERCISES

1. Run the code given in Problem 13 (in the Problem Set) on some system that supports C to determine the values of `sum1` and `sum2`. Explain the results.
2. Rewrite the program of Programming Exercise 1 in C++, Java, and C#, run them, and compare the results.
3. Write a test program in your favorite language that determines and outputs the precedence and associativity of its arithmetic and Boolean operators.
4. Write a Java program that exposes Java's rule for operand evaluation order when one of the operands is a method call.
5. Repeat Programming Exercise 4 with C++.
6. Repeat Programming Exercise 4 with C#.
7. Write a program in either C++, Java, or C# that illustrates the order of evaluation of expressions used as actual parameters to a method.
8. Write a C program that has the following statements:

```
int  a, b;
a = 10;
b = a + fun();
printf("With the function call on the right, ");
printf(" b is: %d\n", b);
a = 10;
b = fun() + a;
printf("With the function call on the left, ");
printf(" b is: %d\n", b);
```

and define `fun` to add 10 to `a`. Explain the results.

9. Write a program in either Java, C++, or C# that performs a large number of floating-point operations and an equal number of integer operations and compare the time required.