computer (such as Windows 95 or UNIX). This program organizes and manages the resources of the computer system as instructed by the user. For example, we can, by typing in a sequence of characters as a command to the operating system, or by clicking on an icon representing a particular program, request that a word processing program be run. We can request that the resultant file (a letter, a report) be passed to a suitable printer, and the operating system will organize the queue of files waiting to be printed. Or we can store the resultant file in a *filing system* so that we can retrieve it by name for use at a later time, and we can organize our various files in a way that enables us to find them easily the next time we need them.

This book is about learning to write programs. That is, we must learn to specify how to carry out a task in a clear step-by-step procedure, in sufficient detail, and in an appropriate format so that the computer can understand and carry out the task. We want to organize this programming task in such a way that the task will be correctly carried out, and the program will make a suitable response if the information supplied to it is invalid in some way (even if the response is simply to display an error message for the user). And we want to organize the program so that it is as straightforward as possible in the event that modifications become necessary because we want to extend or revise how the task is to be carried out.

In order to write a program, or set of instructions, for a computer to carry out, we have to write in a language the computer can understand, a *programming language*. Over the years, a vast number of programming languages have been designed for different types of programming tasks. They have incorporated different ideas of how a programming language should be organized to make it as easy as possible to write a program with few errors (or "bugs") in it.

In this book, we use *Java* as the programming language. Java was released by Sun Microsystems in 1995. There has been a lot of interest in Java because of its ability to deliver programs across the World Wide Web, but we use Java in this book because we think it is also a clear and elegant programming language, suitable as a first language for teaching programming.

1.2 Algorithms

If we are going to carry out any step-by-step procedure, we must have a clear and unambiguous specification of the steps needed. This clear specification, called an *algorithm*, doesn't have to be written in a programming language; it could be in some form of English or in some other more formal language. The problem with using English (or any other natural language) is its ambiguity, so we may turn to more formal languages to be more precise.

Here is an example of part of an algorithm for calculating the date Easter falls on in a given year. The algorithm is in the 17th-century language of the Church of England Book of Common Prayer. Actually, it is an algorithm for working out the position of the sun in a 19-year period called the "saros cycle," because this information is needed by the Easter Day calculation:

```
To find the Golden Number, or Prime, add One to the Year of our Lord, and then
divide by 19; the Remainder, if any, is the Golden Number; but if nothing
remaineth, then 19 is the Golden Number.
```

Garside, R. & Mariani, J., 1998, Java: First Contact, Course Technology ITP.

Here is the beginning of another algorithm, written in the formal language of knitting patterns:

```
SLEEVES

Size 4 mm (8) needles.
Working in St. St.
Pick up 70 (76, 82, 88, 102, 108, 114, 118) Sts. around armhole edge.
Work 10 (10, 10, 10, 12, 2, 0, 0) rows in St. St.
Next Row K.2. K.2. Tog. T.B.L. K. to last 4 Sts. K.2 Tog. K.2
```

And Figure 1-1 is part of a third algorithm, expressed in an even more formal language.

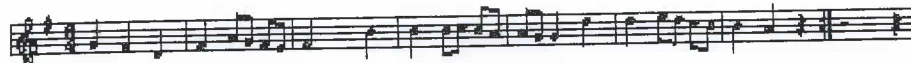

Figure 1-1: An algorithm expressed in the formal language of music.

What are the requirements for an algorithm which specifies how a programming task is to be carried out?

The steps of the algorithm must be *precise* and *unambiguous*—we must know, at least in principle, how we would specify each step in the programming language we are using.

We also need to be precise about what type of input data is required by our algorithm. In the Easter Day algorithm, a single piece of data is required, the year for which we want to find the date of Easter, but the algorithm states that it is valid only for years `from the present time till the year 2199 inclusive`.

The algorithm must be *correct*—we must know that if we set the algorithm going with suitable data, it will eventually finish its calculation and deliver a result which is appropriate for the given input data.

The algorithm should be *efficient*. For programs that are run rarely or that do not require much calculation, efficiency is not very important. But many programs become useful only if they can deliver a result in a time short enough to be useful (tomorrow's weather forecast is useful only if we can calculate what it is before tomorrow) and in a space (amount of memory) small enough to fit into the computer that is to carry out the task. So for most algorithms, we will be interested in how much time and space are needed (for a particular amount of input data).

When we code the algorithm in a programming language, the preceding factors are still important, and some further factors become relevant.

A new issue becomes important when we consider correctness. We need to have the program check that the input data is valid for the task to be performed. If it is not valid, we need to ensure that the program displays clear and unambiguous error messages explaining what is wrong with the input data and exactly what the program has done about it—it could abandon the task for this set of data, or it could try to adjust the values in some way to obtain a valid set of input data.

We need to consider the *maintainability* of the program. Inevitably, errors (bugs) will be found in the program and will need to be corrected; changes will occur in the outside world which require corresponding changes to the program; and there may be a desire to extend the tasks carried out by the program. It is difficult enough making changes to a program that we last worked on several months or years ago, but the changes may have to be made by a different programmer (the original programmer may have moved to another job, or at least to a different programming task). All this requires us to organize the structure of our program as clearly as possible. It needs to be obvious what parts of the program need to be modified for any particular change in the task. And we need to be assured that, as far as possible, changes to one part of a program do not affect other parts. A major issue of programming language design has been how to split up a program into its constituent parts so that a modification to one part affects all other parts as little as possible.

1.3 High-level Languages & Programs

High-level programming languages have been in use since the early 1950s. Most computer languages, from early examples like Fortran and Algol to more recent languages like C and Ada, have been imperative or procedural. A computer program in a language of this type consists of:

- A collection of *variables* (of named areas or "pigeon-holes" in the computer), each of which at any stage contains a certain value (a number, a character, a string of characters, and so on).
- A collection of *statements* which change the values of these variables. The new value (which will replace the previous value of this variable) could be a fixed value specified in the program. Or it could be a value computed from values held in other variables. These statements will be interspersed with tests on values held in variables to decide in what order the statements of the program are to be executed (and consequently what the final result of the program will be).

A programming language of this style is called an *imperative* language, since the basic operation is an order to change the value in a variable to a suitable new value. This style of programming is based on the style of the underlying hardware of the computer or "machine code" (see Section 1.5), but with a clearer syntax for specifying how new values are to be calculated and how tests are to affect the ordering of statements.

Early experience in the difficulties of writing programs revealed that it was important to be able to divide up a program into a number of practically independent pieces, so that the programmer could concentrate on writing one piece at a time. A typical piece would be a program

in miniature, consisting of a sequence of statements to assign values to variables, with tests applied to the values of variables already read in or calculated. Such a subdivision of a program is usually called a *procedure*. Since imperative languages generally rely on some form of procedure as the basic building block of a program, they are often also called *procedural* languages.

Consider the following problem. Someone has given us a list of all the salaries at the place we work, and we want to know the largest salary (actually we probably would also like to know who receives it, but let us keep the problem simple). There are several hundred employees so we need some sort of systematic way of going about it (that is, we need an algorithm). Written in an imperative style, we could have a procedure, called, let's say, `maxList`, which, given a list of numbers stored in a variable called `list`, proceeds as follows:

```
1 We assume that the number of salaries in list is stored in a
     variable called count.
2 Copy the first salary in list to a variable called max
3 Set a variable i to the number 2
4 Copy the ith salary in list to a variable called current
5 if the number in current is greater than that in max
6    copy the number in current into max
7 add 1 to the number in i
8 if the number in i is not greater than the number in count, repeat
     from step 4
9 print the number in max
```

In this imperative programming example, we have several variables—pigeonholes—that each have a name and hold a value. Some, such as `count` and `current`, can hold only a single value (here it is a whole number or integer, but in another program it might be a number with a decimal part, or a character). The value in `count` does not change (i.e., it is a *constant*) in this piece of code, while the value in `current` probably changes several times. The variable `list` holds a whole collection of items, all of the same type (i.e., whole numbers)—when we look at Java programs, we will see variables referring to collections of simpler items, where they are not necessarily all the same type. In some of the statements we specify an actual value—2 in statement 3, and 1 in statement 7—these are called *literals*.

Each of the statements either calculates a new value to be stored in a variable based on other values already available or (as with statement number 8) specifies a test to decide which of two actions is to occur (proceed to step 9, or repeat from step 4).

We assume in the preceding program that, if we have a variable `i` containing a number, we can easily extract (or change) the `i`th element of `list`. This is a common operation in imperative languages and uses a mechanism called an *array*, which we discuss in Chapter 9.

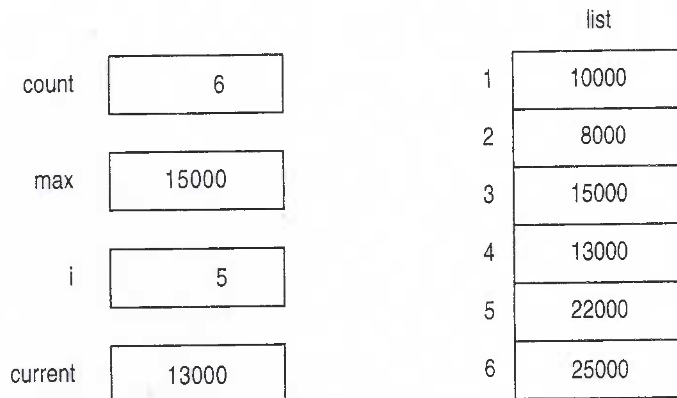

Figure 1-2: Variables in the maxlist procedure.

Figure 1-2 shows the situation just after the computer has checked the fourth salary in the list and is about to return to step 4 with the variable i set to 5.

Note that we have to be careful with this program. It assumes that there are at least *two* salaries in the list. We could easily revise the algorithm so that it still works correctly with a list consisting of a single salary. But if there are no salaries in the list, the correct action for the program to take is not clear. Perhaps the program should check for this situation and stop after printing an error message.

We have here a program written in a *high-level* language. A computer cannot directly understand a program written in this form. As we shall see, a computer executes a much more basic type of language called *machine code*. But the computer can come to the rescue and translate the program written in the high-level language into an equivalent program in machine code. Such a translation program is called a *compiler*, and we would need one for the particular high-level language in which our program is written and for the particular computer and machine code on which we want our program to run (as shown in Figure 1-3).

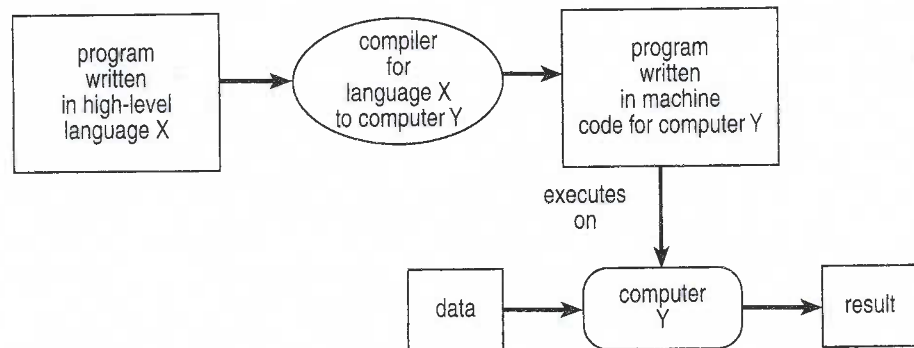

Figure 1-3: Compiling and executing a program.