2 FUNDAMENTAL NOTIONS

This chapter will introduce some of the important basic notions of programming. Because they are fundamental, these concepts cannot formally be defined in terms of other concepts. Instead, they will be explained through the use of examples.

The most important notion is that of *action.* In this context, an action is understood to have a finite duration and to have an intended and well-defined *effect.* Each action requires the existence of some *object* on which the action is executed and on whose *changes of state* its effect can be recognized. Each action must also be describable in terms of a *language* or a system of formulas; its description is called a *statement.*

If the action can be decomposed into parts, then it is called a *process* or a *computation.* If these parts strictly follow each other in time and no two are executed simultaneously, then the process is called *sequential.* Consequently, a statement that describes a process can be broken up into parts; it is then called a *program.* A program thus consists of a set of statements whose textual ordering is not, in general, identical with the ordering in time of the corresponding actions.

The driving force that actually executes the actions according to the specified statements is called a *processor.* This rather neutral word gives no clue as to whether the agent is a human being or an automaton. Indeed, programs have meaning without reference to a specific processor so long as the underlying language is precisely defined. In general, the programmer is not interested in the identity of the processor. He need only be assured that it understands the language of his programs, for the programs are supposed to constitute the rules of behavior of the processor. The programmer therefore needs to know the kinds of statements that his available processor is capable of understanding and executing, and he must adapt his language accordingly.

Every action requires a certain amount of *work*, depending on the processor. This amount can be expressed as the time that it takes the processor to execute the action. This time span, in turn, can usually be more or less directly translated into a measure of cost. An experienced programmer will always take into account the capabilities of the processors available to him and will then select the solution with the least ensuing cost.

Since this text is primarily concerned with the design of programs to be executed by automatic processors (computers), the remainder of this chapter will outline some basic characteristics common to all digital computers. Preceding this bird's-eye view of computers, however, we would like to introduce two simple examples to illustrate the notions just defined.

Example: Multiplication

Multiplication è la action, descritta dallo statement qui sotto. L'effetto della action è produrre un risultato. Gli oggetti su cui si opera sono naturali. Il linguaggio con cui descriviamo lo statement è naturale.

We are given the statement:

Multiply the two natural numbers x and y and denote their product by z.

If the available processor understands this statement, that is, if it knows what is meant by "natural number" and by "multiplication," then further elaboration is unnecessary.

For the sake of argument, however, we will assume that the available processor

(a) does not understand sentences in natural language, accepting only certain formulas, and
(b) cannot multiply but only add.

Queste limitazioni comporteranno il raffinamento dello statement in un process e nel programma corrispondente.

First of all, we notice that the objects of computation are natural numbers. The program, however, is not supposed to specify these numbers; rather it should specify a general *pattern of behavior* for processes multiplying arbitrary pairs of natural numbers. In place of numbers, therefore, we simply use general names denoting variable objects, called *variables*. At the beginning of each process, specific values must be assigned to these variables. This *assignment* is the most fundamental action in the computational processes executed by computers.

A variable is comparable to a blackboard: its value can be inspected ("read") as many times as desired, and it can be erased and overwritten. Overwriting, however, causes the previous value to be lost. Assignment of a value w to a variable v will subsequently be denoted by

$$v := w \tag{2.1}$$

The symbol $:=$ is called the *assignment operator*.

Formally, statement (2.1) can now be written as

$$z := x * y \tag{2.2}$$

If this statement is decomposed into a sequence of additions following each other in time, then the action of multiplication becomes a sequential process, and statement (2.3) assumes the form of a program. For the moment, this program will be formulated informally as

Lo statement iniziale, come conseguenza delle limitazioni deve essere portato a termine da un processo che itera un certo numero di somme. Questo processo è realizzatio dal programma qui a fianco in cui compaiono numerosi assegnamenti. Il programma funziona per ogni assunzione di valore di x e y.

```
Step 1:  z := 0
         u := x
Step 2:  repeat the statements
         z := z + y
         u := u − 1
         until u = 0.
```
(2.3)

The process that is evoked by this program when specific values are given for x and y can be visualized by recording the values assigned to the variables u and z as the computation progresses in time. With $x = 5$ and $y = 13$, we obtain the table in (2.4).

Rappresentazione a traccia del processo di interpretazione del programma (2.3).

	Values of Variables	
Step	z	u
1	0	5
2	13	4
2	26	3
2	39	2
2	52	1
2	65	0

(2.4)

The process terminates, according to statement 2, as soon as $u = 0$. At this time, z has acquired the final result $65 = 5 * 13$. Such a table is called a *trace*. Note that the sequential listing of values does not mean that these values are retained; instead, each variable has at any one time exactly *one* single value. This is due to the fact that an assignment overwrites the previous value of a variable.

The objects of this computation are numbers. To perform operations on specific numbers, it is necessary to represent these numbers by a specific notation. A *choice of notation* is therefore unavoidable in executing a computation. (The program, however, is generally valid without regard to specific notation or representation of its objects.) It is also essential to distinguish between objects—even if they are abstract objects such as numbers—and their representation. In computers, for instance, numbers are usually represented by the states of magnetic storage elements, but it is highly desirable to be able to formulate processes dealing with numbers that can be obeyed by computers without reference to such magnetic states.

To illustrate these ideas and to demonstrate how the same computational process can be described by various notations, the table in (2.5) simply replaces the values in (2.4) with Roman numerals.

Step	Values of Variables z	u
1	0	V
2	XIII	IV
2	XXVI	III
2	XXXIX	II
2	LII	I
2	LXV	0

(2.5)

Example: *Division* Action.

We are given the instruction:

Process. *Divide the natural number x by the natural number y and denote the integer quotient as q and the remainder as r.*

More specifically, the following relations must hold.

$$x = q * y + r \quad \text{and} \quad 0 \leqq r < y \tag{2.6.1}$$

Introducing the division operator **div**, the computation can be described by the following formal assignment statement.

Programma per un processor evoluto.

$$(q, r) := x \ \textbf{div} \ y \tag{2.6.2}$$

To demonstrate again the decomposition of this statement into a program, we assume that the program is to be specified for a processor incapable of division, that is, without the operator **div.** Consequently, division has to be decomposed into a sequence of subtractions of the divisor y from the dividend x, and the number of possible subtractions becomes the desired quotient q.

Programma per un processor che sa eseguire solo incrementi di una unità, sottrazioni, confronti e assegnamenti a variabili.

```
Step 1:  q := 0
         r := x
Step 2:  while r ≧ y repeat
         q := q + 1
         r := r − y
```

(2.7)

x and y again denote *constants* that represent given fixed values at the outset; q and r denote *variables* with integral values. The process prescribed

by program (2.7), having the values $x = 100$ and $y = 15$, is listed in the trace in (2.8).

Traccia del processo relizzato col programma (2.7).

Step	Values of Variables q	r
1	0	100
2	1	85
2	2	70
2	3	55
2	4	40
2	5	25
2	6	10

(2.8)

The process is terminated as soon as $r < y$. The results are $q = 6$ and $r = 10$, thus satisfying the relations (2.6.1):

$$100 = 6 * 15 + 10 \quad \text{and} \quad 0 \leqq 10 < 15 \tag{2.9}$$

These two examples are descriptions of sequential processes in which the individual assignments are performed strictly in sequential order in time. Henceforth, our discussions will be restricted to sequential processes, where the word "process" will always be understood to be an abbreviation of *sequential process.* This deliberate omission of nonsequential processes is made not only because conventional computers operate sequentially but mainly because the design of nonsequential programs—or systems of sequential but interdependent programs to be executed concurrently—is a subtle and difficult task, requiring as a basis a thorough mastery of the art of designing sequential algorithms.

These two examples also show that every program describes a sequence of state transformations of the set of its variables. If the same program is obeyed twice with different initial values (x and y), then it would be a mistake to say that the two processes or computations were the same. However, they definitely follow the same *pattern of behavior.* The description of such a pattern of behavior without reference to a particular processor is usually called an *algorithm.* The term *program* is properly used for algorithms designed so that they can be obeyed or followed by a specific processor type. The difference between a *general* (sometimes called abstract) *algorithm* and a *computer program* lies mainly in the fact that the latter must specify the rules of behavior in every little detail and must be composed according to strict notational rules. The reasons for this are the machine's limited set of instructions, which it is capable of understanding and executing, and its absolute obedience, based on its total lack of a critical attitude. These characteristics of the computer are criticized by most novices in the art of

(*)

(*) Il pattern of behaviour è ciò che rimane della interpretazione di un programma dimenticando il valore delle variabili che lo determinano. Se ci si riferisce ad un preciso interprete vale la pena di chiamarlo "programma"., Altrimenti è più adatta la parola "algoritmo".

programming as the reasons behind the need for pedantic precision and attention to detail when dealing with computers. Indeed, even a trivial mistake in writing may lead to totally unintended and "meaningless" machine behavior. This obvious absence of any "common sense" to which a programmer may appeal (whenever his own senses fail) has been criticized by professionals as well, and efforts have been undertaken to remedy this seeming deficiency. The experienced programmer, however, learns to appreciate this servile attitude of the computer due to which it becomes possible to even require "unusual" patterns of behavior. For this is precisely what is impossible to ask when dealing with (human) processors who are accustomed to rounding off every instruction to the nearest interpretation that is both plausible and pleasing to them.