4 PROGRAMMING AIDS AND SYSTEMS

Until the late 1950s, programming consisted of the detailed encoding of long sequences of instructions—initially written in some symbolic notation—into numbers in binary, octal, or hexadecimal form. This activity is called *coding* in contrast to programming, which encompasses the more difficult task of designing algorithms. The inadequacies of this cumbersome procedure became more and more apparent with the advent of faster computers having larger stores.

1. The coder was forced to tailor his programs to the particular characteristics of his available computer. He therefore had to consider all details of the machine, including its processor organization and its instruction set. The exchange of programs between various machines was impossible, and even the most thorough knowledge of coding methods on one machine was fairly useless when applied to another computer. Every institute designed programs of its own and was forced to dispose of them and to code new ones whenever a new computer replaced the old one. It became evident that the adaptation and tuning of algorithms to the peculiar characteristics of a specific computer represented an unwise investment of human intellect.
2. The close binding of the programmer to one type of computer not only enabled but even encouraged the invention and application of all kinds of tricks to gain maximum machine performance. While this "trickology" was still considered the essence of good programming, the programmer spent considerable time in the construction of "optimal" codes, whose verification was generally very difficult. It was practically impossible to discover the principles of a program designed by a colleague—and often as difficult to explain those of one's own programs! This artistry of coding has now lost most of its glory. The experienced programmer

consciously avoids the use of tricks, choosing systematic and transparent solutions instead.

3. The so-called *machine code* contained only a minimal amount of redundancy on the basis of which formal coding errors could be detected. As a result, even typing errors, which could have devastating effects when the program was executed, were difficult and time consuming to discover.
4. The representation of a complex program as an unstructured, linear sequence of commands was a most inappropriate form for the human inspector to comprehend and to express. We will show later that the presence and application of structure is the principal tool in helping the programmer to synthesize systematically and to maintain an overall comprehension of complicated programs.

These shortcomings led to the development of the so-called "high level" *programming languages.* Such languages became the means through which to instruct an idealized, hypothetical computer that is designed not according to the limitations of current technology but according to the habits and the capabilities of man to express his thoughts. In this situation, however, we are confronted with a machine A, which is economically realizable but neither convenient nor encouraging to use, and a computer B, which is suited to human needs but exists only on paper. The gap between these two kinds of objects is now bridged by an entity called *software.* (In contrast, the physical machine is called *hardware.*) A software system is a *program* C that, when executed by the existing computer A, enables A to translate programs written for the hypothetical machine B into programs of its own. Program C is called a translator or *compiler*; it enables A to appear as the idealized machine B.

ovvero in programmi per la vera macchina A.

The utilization of compiler C thus relieves the programmer of the burden of considering the particular, detailed characteristics of his computer A. But it does not free him of the duty to be constantly aware that it is machine A that will ultimately execute his programs and that A has some *definite limitations*, imposed by its finite speed and storage capacity.

Usually, a combined hardware–software system processes programs P in two distinct steps, which follow each other in time. During the first step, P is translated by the compiler C into a form interpretable by A; this step is called *compilation.* In the second step, the translated program is executed; this step is called *execution.*

Compilation:	program = compiler C
	input data = program P in language B
	output data = program P in language A
Execution:	program = P in language A
(interpretazione)	input data = arguments of computation X
	output data = computational results Y