Chapter 1
Abstract Machines

Abstraction mechanisms play a crucial role in computing because they allow us to manage the complexity inherent in most computational systems by isolating the important aspects in a specific context. In the field of programming languages, these mechanisms are fundamental, both from a theoretical viewpoint (many important concepts can be appropriately formalised using abstractions) and in the practical sense, because programming languages today use common abstraction-creating constructs.

One of the most general concepts employing abstraction is the *abstract machine*. In this chapter, we will see how this concept is closely related to the programming languages. We will also see how, without requiring us to go into the specific details of any particular implementation, it allows us to describe what an implementation of a programming language is. To do this, we will describe in general terms what is meant by the *interpreter* and the *compiler* for a language. Finally, will see how abstract machines can be structured in hierarchies that describe and implement complex software systems.

1.1 The Concepts of Abstract Machine and of Interpreter

In the context of this book, the term "machine" refers clearly to a computing machine. As we know, an electronic, digital computer is a physical machine that executes algorithms which are suitably formalised so that the machine can "understand" them. Intuitively, an abstract machine is nothing more than an abstraction of the concept of a physical computer.

For actual execution, algorithms must be appropriately formalised using the constructs provided by a programming language. In other words, the algorithms we want to execute must be represented using the instructions of a programming language, $\mathscr{L}$. This language will be formally defined in terms of a specific syntax and a precise semantics—see Chap. 2. For the time being, the nature of $\mathscr{L}$ is of no concern to us. Here, it is sufficient to know that the syntax of $\mathscr{L}$ allows us to use a given finite set of constructs, called instructions, to construct programs. A *program* in $\mathscr{L}$

M. Gabbrielli, S. Martini, *Programming Languages: Principles and Paradigms,*
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84882-914-5_1, © Springer-Verlag London Limited 2010

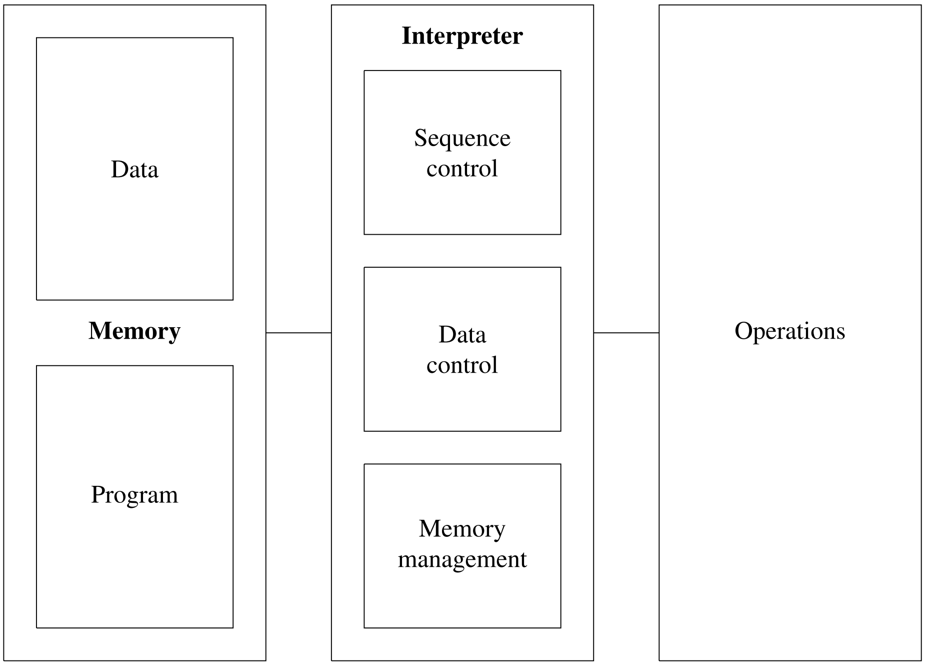


Fig. 1.1 The structure of an abstract machine

(or program written in $\mathscr{L}$) therefore is nothing more than a finite set of instructions of $\mathscr{L}$. With these preliminary remarks, we now present a definition that is central to this chapter.

Definition 1.1 (Abstract Machine) Assume that we are given a programming language, $\mathscr{L}$. An *abstract machine* for $\mathscr{L}$, denoted by $\mathscr{M}_{\mathscr{L}}$, is any set of data structures and algorithms which can perform the storage and execution of programs written in $\mathscr{L}$.

When we choose not to specify the language, $\mathscr{L}$, we will simply talk of the abstract machine, $\mathscr{M}$, omitting the subscript. We will soon see some example abstract machines and how they can actually be implemented. For the time being, let us stop and consider the structure of an abstract machine. As depicted in Fig. 1.1, a generic abstract machine $\mathscr{M}_{\mathscr{L}}$ is composed of a *store* and an *interpreter*. The store serves to store data and programs while the interpreter is the component that executes the instructions contained in programs. We will see this more clearly in the next section.

1.1.1 The Interpreter

Clearly the interpreter must perform the operations that are specific to the language it is interpreting, $\mathscr{L}$. However, even given the diversity of languages, it is possible to discern types of operation and an "execution method" common to all interpreters. The type of operation executed by the interpreter and associated data structures, fall into the following categories:

1. Operations for processing primitive data;
2. Operations and data structures for controlling the sequence of execution of operations;
3. Operations and data structures for controlling data transfers;
4. Operations and data structures for memory management.

We consider these four points in detail.

1. The need for operations such as those in point one is clear. A machine, even an abstract one, runs by executing algorithms, so it must have operations for manipulating primitive data items. These items can be directly represented by a machine. For example, for physical abstract machines, as well as for the abstract machines used by many programming languages, numbers (integer or real) are almost always primitive data. The machine directly implements the various operations required to perform arithmetic (addition, multiplication, etc.). These arithmetic operations are therefore primitive operations as far as the abstract machine is concerned[1].

2. Operations and structures for "sequence control" allow to control the execution flow of instructions in a program. The normal sequential execution of a program might have to be modified when some conditions are satisfied. The interpreter therefore makes use of data structures (for example to hold the address of the next instruction to execute) which are manipulated by specific operations that are different from those used for data manipulation (for example, operations to update the address of the next instruction to execute).

3. Operations that control data transfers are included in order to control how operands and data is to be transferred from memory to the interpreter and vice versa. These operations deal with the different store addressing modes and the order in which operands are to be retrieved from store. In some cases, auxiliary data structures might be necessary to handle data transfers. For example, some types of machine use stacks (implemented either in hardware or software) for this purpose.

4. Finally, there is memory management. This concerns the operations used to allocate data and programs in memory. In the case of abstract machines that are similar to hardware machines, storage management is relatively simple. In the limit case of a physical register-based machine that is not multiprogrammed, a program and its associated data could be allocated in a zone of memory at the start of execution and remain there until the end, without much real need for memory management. Abstract machines for common programming languages, instead, as will be seen, use more sophisticated memory management techniques. In fact, some constructs in these languages either directly or indirectly cause memory to be allocated or deallocated. Correct implementation of these operations requires suitable data

[1] It should, however, be noted that there exist programming languages, for example, some declarative languages, in which numeric values and their associated operations are not primitive.

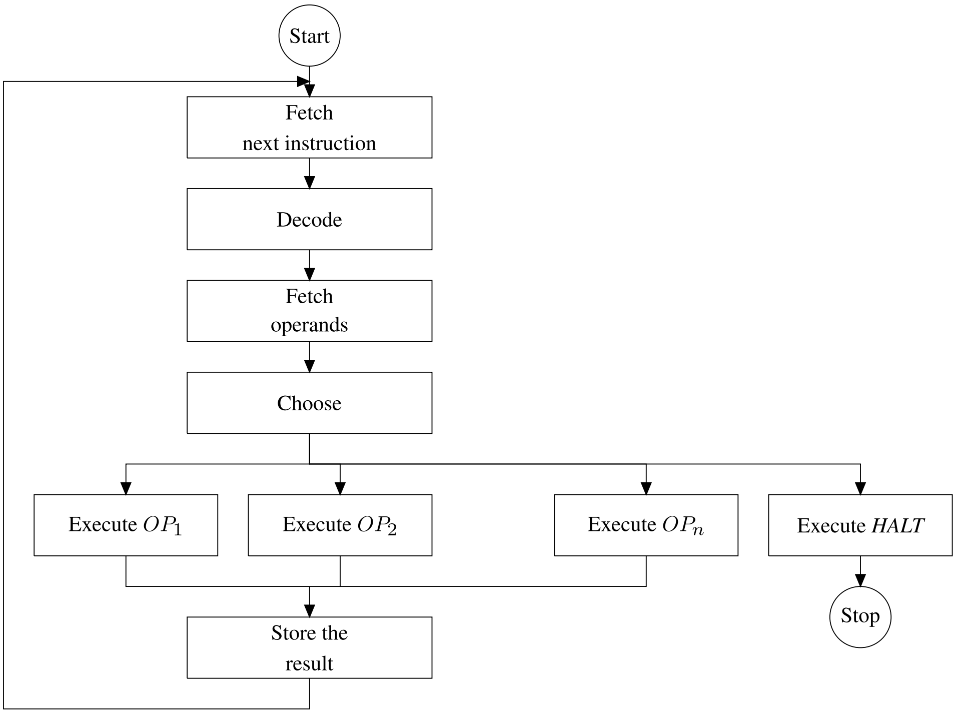


Fig. 1.2 The execution cycle of a generic interpreter

structures (for example, stacks) and dynamic operations (which are, therefore, executed at runtime).

The interpreter's execution cycle, which is substantially the same for all interpreters, is shown in Fig. 1.2. It is organised in terms of the following steps. First, it fetches the next instruction to execute from memory. The instruction is then decoded to determine the operation to be performed as well as its operands. As many operands as required by the instruction are fetched from memory using the method described above. After this, the instruction, which must be one of the machine's primitives, is executed. Once execution of the operation has completed, any results are stored. Then, unless the instruction just executed is a halt instruction, execution passes to the next instruction in sequence and the cycle repeats.

Now that we have seen the interpreter, we can define the language it interprets as follows:

Definition 1.2 (Machine language) Given an abstract machine, $\mathscr{M}_{\mathscr{L}}$, the language $\mathscr{L}$ "understood" by $\mathscr{M}_{\mathscr{L}}$'s interpreter is called the *machine language* of $\mathscr{M}_{\mathscr{L}}$.

Programs written in the machine language of $\mathscr{M}_{\mathscr{L}}$ will be stored in the abstract machine's storage structures so that they cannot be confused with other primitive data on which the interpreter operates (it should be noted that from the interpreter's viewpoint, programs are also a kind of data). Given that the internal representation of the programs executed by the machine $\mathscr{M}_{\mathscr{L}}$ is usually different from its external representation, then we should strictly talk about two different languages. In any

“Low-level” and “High-level” languages

A terminological note is useful. We will return to it in an historical perspective in Chap. 13. In the field of programming languages, the terms “low level” and “high level” are often used to refer, respectively, to distance from the human user and from the machine.

Let us therefore call *low-level*, those languages whose abstract machines are very close to, or coincide with, the physical machine. Starting at the end of the 1940s, these languages were used to program the first computers, but, they turned out to be extremely awkward to use. Because the instructions in these languages had to take into account the physical characteristics of the machine, matters that were completely irrelevant to the algorithm had to be considered while writing programs, or in coding algorithms. It must be remembered that often when we speak generically about “machine language”, we mean the language (a low-level one) of a physical machine. A particular low-level language for a physical machine is its *assembly language*, which is a symbolic version of the physical machine (that is, which uses symbols such as ADD, MUL, etc., instead of their associated hardware binary codes). Programs in assembly language are translated into machine code using a program called an *assembler*.

So-called *high-level* programming languages are, on the other hand, those which support the use of constructs that use appropriate abstraction mechanisms to ensure that they are independent of the physical characteristics of the computer. High-level languages are therefore suited to expressing algorithms in ways that are relatively easy for the human user to understand. Clearly, even the constructs of a high-level language must correspond to instructions of the physical machine because it must be possible to execute programs.

case, in order not to complicate notation, for the time being we will not consider such differences and therefore we will speak of just one machine language, $\mathscr{L}$, for machine $\mathscr{M}_{\mathscr{L}}$.

1.1.2 An Example of an Abstract Machine: The Hardware Machine

From what has been said so far, it should be clear that the concept of abstract machine can be used to describe a variety of different systems, ranging from physical machines right up to the World Wide Web.

As a first example of an abstract machine, let us consider the concrete case of a conventional physical machine such as that in Fig. 1.3. It is physically implemented using logic circuits and electronic components. Let us call such a machine $\mathscr{M}\mathscr{H}_{\mathscr{L}H}$ and let $\mathscr{L}H$ be its machine language.

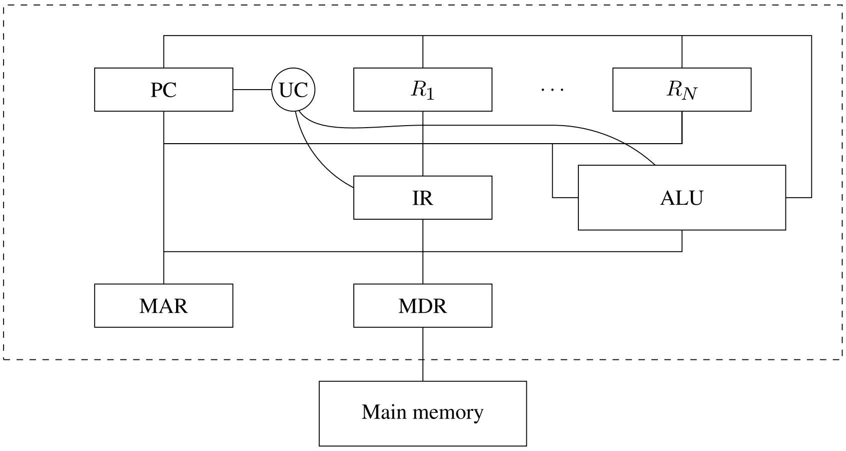


Fig. 1.3 The structure of a conventional calculator

For this specific case, we can, using what we have already said about the components of an abstract machine, identify the following parts.

Memory The storage component of a physical computer is composed of various levels of memory. Secondary memory implemented using optical or magnetic components; primary memory, organised as a linear sequence of cells, or words, of fixed size (usually a multiple of 8 bits, for example 32 or 64 bits); cache and the *registers* which are internal to the Central Processing Unit (CPU).

Physical memory, whether primary, cache or register file, permits the storage of data and programs. As stated, this is done using the binary alphabet.

Data is divided into a few primitive “types”: usually, we have integer numbers, so-called “real” numbers (in reality, a subset of the rationals), characters, and fixed-length sequences of bits. Depending upon the type of data, different physical representations, which use one or more memory words for each element of the type are used. For example, the integers can be represented by 1s or 2s complement numbers contained in a single word, while reals have to be represented as floating point numbers using one or two words depending on whether they are single or double precision. Alphanumeric characters are also implemented as sequences of binary numbers encoded in an appropriate representational code (for example, the ASCII or UNI CODE formats).

We will not here go into the details of these representations since they will be examined in more detail in Chap. 8. We must emphasise the fact that although all data is represented by sequences of bits, at the hardware level we can distinguish different categories, or more properly *types*, of primitive data that can be manipulated directly by the operations provided by the hardware. For this reason, these types are called *predefined types*.

The language of the physical machine The language, $\mathscr{L}H$ which the physical machine executes is composed of relatively simple instructions. A typical instruction with two operands, for example, requires one word of memory and has the format:

```
OpCode Operand1 Operand2
```

where `OpCode` is a unique code which identifies one of the primitive operations defined by the machine's hardware, while `Operand1` and `Operand2` are values which allow the operands to be located by referring to the storage structures of the machine and their addressing modes. For example,

```
ADD R5, R0
```

might indicate the sum of the contents of registers R0 and R5, with the result being stored in R5, while

```
ADD (R5), (R0)
```

might mean that the sum of the contents of the memory cells whose addresses are contained in R0 and R5 is computed and the result stored in the cell whose address is in R5. It should be noted that, in these examples, for reasons of clarity, we are using symbolic codes such as ADD, R0, (R0). In the language under consideration, on the other hand, we have binary numeric values (addresses are expressed in "absolute" mode). From the viewpoint of internal representation, instructions are nothing more than data stored in a particular format.

Like the instructions and data structures used in executing programs, the set of possible instructions (with their associated operations and addressing modes) depends on the particular physical machine. It is possible to discern classes of machine with similar characteristics. For example, we can distinguish between conventional CISC (Complex Instruction Set Computer) processors which have many machine instructions (some of which are quite complex) and RISC (Reduced Instruction Set Computers) architectures in which there tend to be fewer instructions which are, in particular, simple enough to be executed in a few (possibly one) clock cycle and in pipelined fashion.

Interpreter With the general structure of an abstract machine as a model, it is possible to identify the following components of a physical (hardware) machine:

1. The operations for processing primitive data are the usual arithmetic and logical operations. They are implemented by the ALU (Arithmetic and Logic Unit). Arithmetic operations on integers, and floating-point numbers, booleans are provided, as are shifts, tests, etc.
2. For the control of instruction sequence execution, there is the Program Counter (PC) register, which contains the address of the next instruction to execute. It is the main data structure of this component. Sequence-control operations specifically use this register and typically include the increment operation (which handles the normal flow of control) and operations that modify the value stored in the PC register (jumps).

3. To handle data transfer, the CPU registers interfacing with main memory are used. They are: the data address register (the MAR or Memory Address Register) and the data register (MDR or Memory Data Register). There are, in addition, operations that modify the contents of these registers and that implement various addressing modes (direct, indirect, etc.). Finally, there are operations that access and modify the CPU's internal registers.
4. Memory processing depends fundamentally on the specific architecture. In the simplest case of a register machine that is not multi-programmed, memory management is rudimentary. The program is loaded and immediately starts executing; it remains in memory until it terminates. To increase computation speed, all modern architectures use more sophisticated memory management techniques. In the first place, there are levels of memory intermediate between registers and main memory (i.e., cache memory), whose management needs special data structures and algorithms. Second, some form of multi-programming is almost always implemented (the execution of a program can be suspended to give the CPU to other programs, so as to optimise the management of resources). As a general rule, these techniques (which are used by operating systems) usually require specialised hardware support to manage the presence of more than one program in memory at any time (for example, dynamic address relocation).

   All the techniques so far described need specific memory-management data structures and operations to be provided by the hardware. In addition, there are other types of machine that correspond to less conventional architectures. In the case of a machine which uses a (hardware) stack instead of registers, there is the stack data structure together with the push and pop operations.

The interpreter for the hardware machine is implemented as a set of physical devices which comprise the Control Unit and which support execution of the so-called *fetch-decode-execute* cycle. using the sequence control operations. This cycle is analogous to that in the generic interpreter such as the one depicted in Fig. 1.2. It consists of the following phases.

In the *fetch* phase, the next instruction to be executed is retrieved from memory. This is the instruction whose address is held in the PC register (the PC register is automatically incremented after the instruction has been fetched). The instruction, which, it should be recalled, is formed of an operation code and perhaps some operands, is then stored in a special register, called the instruction register.

In the *decode* phase, the instruction stored in the instruction register is decoded using special logic circuits. This allows the correct interpretation of both the instruction's operation code and the addressing modes of its operands. The operands are then retrieved by data transfer operations using the address modes specified in the instruction .

Finally, in the *execute* phase, the primitive hardware operation is actually executed, for example using the circuits of the ALU if the operation is an arithmetic or logical one. If there is a result, it is stored in the way specified by the addressing mode and the operation code currently held in the instruction register. Storage is performed by means of data-transfer operations. At this point, the instruction's execution is complete and is followed by the next phase, in which the next instruction is fetched and the cycle continues (provided the instruction just executed is not a stop instruction).

It should be noted that, even if only conceptually, the hardware machine distinguishes data from instructions. At the physical level, there is no distinction between them, given that they are both represented internally in terms of bits. The distinction mainly derives from the state of the CPU. In the fetch state, every word fetched from memory is considered an instruction, while in the execute phase, it is considered to be data. It should be observed that, finally, an accurate description of the operation of the physical machine would require the introduction of other states in addition to fetch, decode and execute. Our description only aims to show how the general concept of an interpreter is instantiated by a physical machine.

1.2 Implementation of a Language

We have seen that an abstract machine, $\mathscr{M}_{\mathscr{L}}$, is by definition a device which allows the execution of programs written in $\mathscr{L}$. An abstract machine therefore corresponds uniquely to a language, its *machine language*. Conversely, given a programming language, $\mathscr{L}$, there are many (an infinite number) of abstract machines that have $\mathscr{L}$ as their machine language. These machines differ from each other in the way in which the interpreter is implemented and in the data structures that they use; they all agree, though, on the language they interpret—$\mathscr{L}$.

To *implement* a programming language $\mathscr{L}$ means implementing an abstract machine which has $\mathscr{L}$ as its machine language. Before seeing which implementation techniques are used for current programming languages, we will first see what the various theoretical possibilities for an abstract machine are.

1.2.1 Implementation of an Abstract Machine

Any implementation of an abstract machine, $\mathscr{M}_{\mathscr{L}}$ must sooner or later use some kind of physical device (mechanical, electronic, biological, etc.) to execute the instructions of $\mathscr{L}$. The use of such a device, nevertheless, can be explicit or implicit. In fact, in addition to the "physical" implementation (in hardware) of $\mathscr{M}_{\mathscr{L}}$'s constructs, we can even think instead of an implementation (in software or firmware) at levels intermediate between $\mathscr{M}_{\mathscr{L}}$ and the underlying physical device. We can therefore reduce the various options for implementing an abstract machine to the following three cases and to combinations of them:

- implementation in *hardware*;
- simulation using *software*;
- simulation (emulation) using *firmware*.

Microprogramming

Microprogramming techniques were introduced in the 1960s with the aim of providing a whole range of different computers, ranging from the slowest and most economical to those with the greatest speed and price, with the same instruction set and, therefore, the same assembly language (the IBM 360 was the most famous computer on which microprogramming was used). The machine language of microprogrammed machines is at an extremely low level and consists of *microinstructions* which specify simple operations for the transfer of data between registers, to and from main memory and perhaps also passage through the logic circuits that implement arithmetic operations. Each instruction in the language which is to be implemented (that is, in the machine language that the user of the machine sees) is simulated using a specific set of microinstructions. These microinstructions, which encode the operation, together with a particular set of microinstructions implementing the interpretation cycle, constitute a *microprogram* which is stored in special read-only memory (which requires special equipment to write). This microprogram implements the interpreter for the (assembly) language common to different computers, each of which has different hardware. The most sophisticated (and costly) physical machines are built using more powerful hardware hence they can implement an instruction by using fewer simulation steps than the less costly models, so they run at a greater speed.

Some terminology needs to be introduced: the term used for simulation using micro-programming, is *emulation*; the level at which microprogramming occurs is called *firmware*.

Let us, finally, observe that a microprogrammable machine constitutes a single, simple example of a *hierarchy* composed of two abstract machines. At the higher level, the assembly machine is constructed on top of what we have called the microprogrammed machine. The assembly language interpreter is implemented in the language of the lower level (as microinstructions), which is, in its turn, interpreted directly by the microprogrammed physical machine. We will discuss this situation in more depth in Sect. 1.3.

Implementation in Hardware

The direct implementation of $\mathscr{M}_{\mathscr{L}}$ in hardware is always possible in principle and is conceptually fairly simple. It is, in fact, a matter of using physical devices such as memory, arithmetic and logic circuits, buses, etc., to implement a physical machine whose machine language coincides with $\mathscr{L}$. To do this, it is sufficient to implement in the hardware the data structures and algorithms constituting the abstract machine.[2]

[2] Chapter 3 will tackle the question of why this can always be done for programming languages.

The implementation of a machine $\mathscr{M}_{\mathscr{L}}$ in hardware has the advantage that the execution of programs in $\mathscr{L}$ will be fast because they will be directly executed by the hardware. This advantage, nevertheless, is compensated for by various disadvantages which predominate when $\mathscr{L}$ is a generic high-level language. Indeed, the constructs of a high-level language, $\mathscr{L}$, are relatively complicated and very far from the elementary functions provided at the level of the electronic circuit. An implementation of $\mathscr{M}_{\mathscr{L}}$ requires, therefore, a more complicated design for the physical machine that we want to implement. Moreover, in practice, such a machine, once implemented, would be almost impossible to modify. In would not be possible to implement on it any future modifications to $\mathscr{L}$ without incurring prohibitive costs. For these reasons,in practice, when implementing $\mathscr{M}_{\mathscr{L}}$, in hardware, only low-level languages are used because their constructs are very close to the operations that can be naturally defined using just physical devices. It is possible, though, to implement "dedicated" languages developed for special applications directly in hardware where enormous execution speeds are necessary. This is the case, for example, for some special languages used in real-time systems.

The fact remains that there are many cases in which the structure of a high-level language's abstract machine has influenced the implementation of a hardware architecture, not in the sense of a direct implementation of the abstract machine in hardware, but in the choice of primitive operations and data structures which permit simpler and more efficient implementation of the high-level language's interpreter. This is the case, for example with the architecture of the B5500, a computer from the 1960s which was influenced by the structure of the Algol language.

Simulation Using Software

The second possibility for implementing an abstract machine consists of implementing the data structures and algorithms required by $\mathscr{M}_{\mathscr{L}}$ using programs written in another language, $\mathscr{L}'$, which, we can assume, has already been implemented. Using language $\mathscr{L}'$'s machine, $\mathscr{M}'_{\mathscr{L}'}$, we can, indeed, implement the machine $\mathscr{M}_{\mathscr{L}}$ using appropriate programs written in $\mathscr{L}'$ which interpret the constructs of $\mathscr{L}$ by simulating the functionality of $\mathscr{M}_{\mathscr{L}}$.

In this case, we will have the greatest flexibility because we can easily change the programs implementing the constructs of $\mathscr{M}_{\mathscr{L}}$. We will nevertheless see a performance that is lower than in the previous case because the implementation of $\mathscr{M}_{\mathscr{L}}$ uses another abstract machine $\mathscr{M}'_{\mathscr{L}'}$, which, in its turn, must be implemented in hardware, software or firmware, adding an extra level of interpretation.

Emulation Using Firmware

Finally, the third possibility is intermediate between hardware and software implementation. It consists of simulation (in this case, it is also called emulation) of the data structures and algorithms for $\mathscr{M}_{\mathscr{L}}$ in microcode (which we briefly introduced in the box on page 10).

Partial Functions

A function $f : A \to B$ is a correspondence between elements of A and elements of B such that, for every element a of A, there exists one and only one element of B. We will call it $f(a)$.

A *partial function*, $f : A \to B$, is also a correspondence between the two sets A and B, but can be undefined for some elements of A. More formally: it is a relation between A and B such that, for every $a \in A$, if there exists a corresponding element $b \in B$, it is unique and is written $f(a)$. The notion of partial function, for us, is important because, in a natural fashion, programs define partial functions. For example, the following program (written in a language with obvious syntax and semantics and whose core will however be defined in Fig. 2.11):

```
read(x);
if (x == 1) then print(x);
            else while (true) do skip
```

computes the partial function:

$$f(n) = \begin{cases} 1 & \text{if } x = 1 \\ \textsf{undefined} & \textsf{otherwise} \end{cases}$$

Conceptually, this solution is similar to simulation in software. In both cases, $\mathcal{M}_{\mathcal{L}}$ is simulated using appropriate programs that are executed by a physical machine. Nevertheless, in the case of firmware emulation, these programs are microprograms instead of programs in a high-level language.

As we saw in the box, microprograms use a special, very low-level language (with extremely simple primitive operations) which are stored in a special read-only memory instead of in main memory, so they can be executed by the physical machine at high speed. For this reason, this implementation of an abstract machine allows us to obtain an execution speed that is higher than that obtainable from software simulation, even if it is not as fast as the equivalent hardware solution. On the other hand, the flexibility of this solution is lower than that of software simulation, since, while it is easy to modify a program written in a high-level language, modification of microcode is relatively complicated and requires special hardware to re-write the memory in which the microcode is stored. The situation is anyway better than in the hardware implementation case, given that microprograms can be modified.

Clearly, for this solution to be possible, the physical machine on which it is used must be microprogrammable.

Summarising, the implementation of $\mathcal{M}_{\mathcal{L}}$ in hardware affords the greatest speed but no flexibility. Implementation in software affords the highest flexibility and least speed, while the one using firmware is intermediate between the two.

1.2.2 Implementation: The Ideal Case

Let us consider a generic language, $\mathscr{L}$, which we want to implement, or rather, for which an abstract machine, $\mathscr{M}_{\mathscr{L}}$ is required. Assuming that we can exclude, for the reasons just given, direct implementation in hardware of $\mathscr{M}_{\mathscr{L}}$, we can assume that, for our implementation of $\mathscr{M}_{\mathscr{L}}$, we have available an abstract machine, $\mathscr{M}o_{\mathscr{L}o}$, which we will call the *host machine*, which is already implemented (we do not care how) and which therefore allows us to use the constructs of its machine language $\mathscr{L}o$ directly.

Intuitively, the implementation of $\mathscr{L}$ on the host machine $\mathscr{M}o_{\mathscr{L}o}$ takes place using a "translation" from $\mathscr{L}$ to $\mathscr{L}o$. Nevertheless, we can distinguish two conceptually very different modes of implementation, depending on whether there is an "implicit" translation (implemented by the simulation of $\mathscr{M}_{\mathscr{L}}$'s constructs by programs written in $\mathscr{L}o$) or an explicit translation from programs in $\mathscr{L}$ to corresponding programs in $\mathscr{L}o$. We will now consider these two ways in their ideal forms. We will call these ideal forms:

1. *purely interpreted implementation*, and
2. *purely compiled implementation*.

Notation

Below, as previously mentioned, we use the subscript $_{\mathscr{L}}$ to indicate that a particular construct (machine, interpreter, program, etc.) refers to language $\mathscr{L}$. We will use the superscript $^{\mathscr{L}}$ to indicate that a program is written in language $\mathscr{L}$. We will use $\mathscr{P}rog^{\mathscr{L}}$ to denote the set of all possible programs that can be written in language $\mathscr{L}$, while $\mathscr{D}$ denotes the set of input and output data (and, for simplicity of treatment, we make no distinction between the two).

A program written in $\mathscr{L}$ can be seen as a partial function (see the box):

$$\mathscr{P}^{\mathscr{L}} : \mathscr{D} \to \mathscr{D}$$

such that

$$\mathscr{P}^{\mathscr{L}}(\mathit{Input}) = \mathit{Output}$$

if the execution of $\mathscr{P}^{\mathscr{L}}$ on input data *Input* terminates and produces *Output* as its result. The function is not defined if the execution of $\mathscr{P}^{\mathscr{L}}$ on its input data, *Input*, does not terminate.[3]

[3] It should be noted that there is no loss of generality in considering only one input datum, given that it can stand for a set of data.

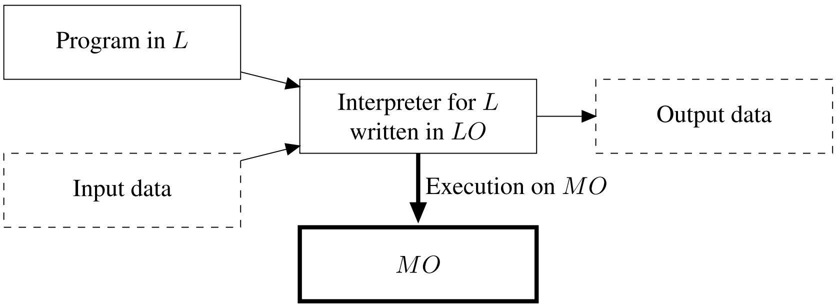


Fig. 1.4 Purely interpreted implementation

Purely interpreted implementation In a *purely interpreted implementation* (shown in Fig. 1.4), the interpreter for $\mathscr{M}_{\mathscr{L}}$ is implemented using a set of instructions in $\mathscr{L}o$. That is, a program is implemented in $\mathscr{L}o$ which interprets all of $\mathscr{L}$'s instructions; this is an interpreter. We will call it $\mathscr{I}_{\mathscr{L}}^{\mathscr{L}o}$.

Once such interpreter is implemented, executing a program $\mathscr{P}^{\mathscr{L}}$ (written in language $\mathscr{L}$) on specified input data $D \in \mathscr{D}$, we need only execute the program $\mathscr{I}_{\mathscr{L}}^{\mathscr{L}o}$ on machine $\mathscr{M}o_{\mathscr{L}o}$, with $\mathscr{P}^{\mathscr{L}}$ and D as input data. More precisely, we can give the following definition.

Definition 1.3 (Interpreter) An interpreter for language $\mathscr{L}$, written in language $\mathscr{L}o$, is a program which implements a partial function:

$$\mathscr{I}_{\mathscr{L}}^{\mathscr{L}o} : (\mathscr{P}rog^{\mathscr{L}} \times \mathscr{D}) \to \mathscr{D} \quad \text{such that } \mathscr{I}_{\mathscr{L}}^{\mathscr{L}o}(\mathscr{P}^{\mathscr{L}}, \mathit{Input}) = \mathscr{P}^{\mathscr{L}}(\mathit{Input}) \tag{1.1}$$

The fact that a program can be considered as input datum for another program should not be surprising, given that, as already stated, a program is only a set of instructions which, in the final analysis, are represented by a certain set of symbols (and therefore by bit sequences).

In the purely interpreted implementation of $\mathscr{L}$, therefore, programs in $\mathscr{L}$ are not explicitly translated. There is only a "decoding" procedure. In order to execute an instruction of $\mathscr{L}$, the interpreter $\mathscr{I}_{\mathscr{L}}^{\mathscr{L}o}$ uses a set of instructions in $\mathscr{L}o$ which corresponds to an instruction in language $\mathscr{L}$. Such decoding is not a real translation because the code corresponding to an instruction of $\mathscr{L}$ is executed, not output, by the interpreter.

It should be noted that we have deliberately not specified the nature of the machine $\mathscr{M}o_{\mathscr{L}o}$. The language $\mathscr{L}o$ can therefore be a high-level language, a low-level language or even one firmware.

Purely compiled implementation With *purely compiled implementation*, as shown in Fig. 1.5, the implementation of $\mathscr{L}$ takes place by explicitly translating programs written in $\mathscr{L}$ to programs written in $\mathscr{L}o$. The translation is performed by a special program called *compiler*; it is denoted by $\mathscr{C}_{\mathscr{L},\mathscr{L}o}$. In this case, the language $\mathscr{L}$ is usually called the *source language*, while language $\mathscr{L}o$ is called the *object language*. To execute a program $\mathscr{P}^{\mathscr{L}}$ (written in language $\mathscr{L}$) on input

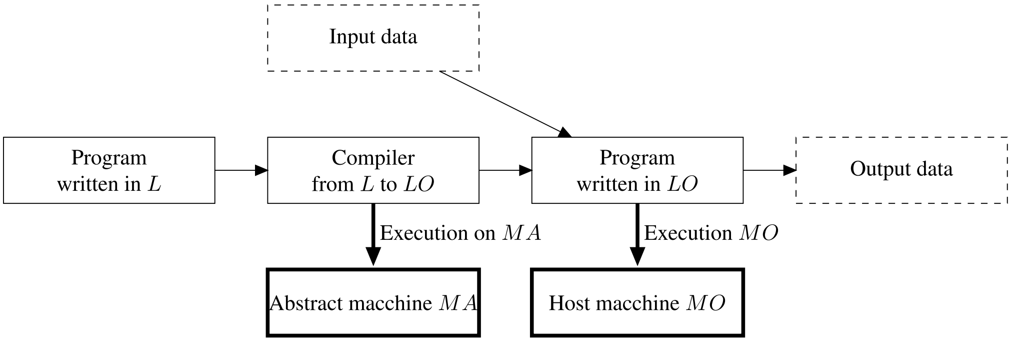


Fig. 1.5 Pure compiled implementation

data D, we must first execute $\mathscr{C}_{\mathscr{L},\mathscr{L}o}$ and give it $\mathscr{P}^{\mathscr{L}}$ as input. This will produce a compiled program $\mathscr{P}c^{\mathscr{L}o}$ as its output (written in $\mathscr{L}o$). At this point, we can execute $\mathscr{P}c^{\mathscr{L}o}$ on the machine $\mathscr{M}o_{\mathscr{L}o}$ supplying it with input data D to obtain the desired result.

Definition 1.4 (Compiler) A compiler from $\mathscr{L}$ to $\mathscr{L}o$ is a program which implements a function:

$$\mathscr{C}_{\mathscr{L},\mathscr{L}o} : \mathscr{P}rog^{\mathscr{L}} \to \mathscr{P}rog^{\mathscr{L}o}$$

such that, given a program $\mathscr{P}^{\mathscr{L}}$, if

$$\mathscr{C}_{\mathscr{L},\mathscr{L}o}(\mathscr{P}^{\mathscr{L}}) = \mathscr{P}c^{\mathscr{L}o}, \tag{1.2}$$

then, for every *Input*$\in \mathscr{D}$[4]:

$$\mathscr{P}^{\mathscr{L}}(Input) = \mathscr{P}c^{\mathscr{L}o}(Input) \tag{1.3}$$

Note that, unlike pure interpretation, the translation phase described in (1.2) (called *compilation*) is separate from the execution phase, which is, on the other hand, handled by (1.3). Compilation indeed produces a program as output. This program can be executed at any time we want. It should be noted that if $\mathscr{M}o_{\mathscr{L}o}$ is the only machine available to us, and therefore if $\mathscr{L}o$ is the only language that we can use, the compiler will also be a program written in $\mathscr{L}o$. This is not necessary, however, for the compiler could in fact be executed on another abstract machine altogether and this, latter, machine could execute a different language, even though it produces executable code for $\mathscr{M}o_{\mathscr{L}o}$.

[4] It should be noted that, for simplicity, we assume that the data upon which programs operate are the same for source and object languages. If were not the case, the data would also have to be translated in an appropriate manner.

Comparing the Two Techniques

Having presented the purely interpreted and purely compiled implementation techniques, we will now discuss the advantages and disadvantages of these two approaches.

As far as the purely interpreted implementation is concerned, the main disadvantage is its *low efficiency*. In fact, given that there is no translation phase, in order to execute the program $\mathscr{P}^{\mathscr{L}}$, the interpreter $\mathscr{I}_{\mathscr{L}}^{\mathscr{L}o}$ must perform a decoding of $\mathscr{L}$'s constructs while it executes. Hence, as part of the time required for the execution of $\mathscr{P}^{\mathscr{L}}$, it is also necessary to add in the time required to perform decoding. For example, if the language $\mathscr{L}$ contains the iterative construct `for` and if this construct is not present in language $\mathscr{L}o$, to execute a command such as:

```
P1: for (I = 1, I<=n, I=I+1) C;
```

the interpreter $\mathscr{I}_{\mathscr{L}}^{\mathscr{L}o}$ must decode this command at runtime and, in its place, execute a series of operations implementing the loop. This might look something like the following code fragment:

```
P2:
   R1 = 1
   R2 = n
L1: if R1 > R2 then goto L2
   translation of C
   ...
   R1 = R1 + 1
   goto L1
L2: ...
```

It is important to repeat that, as shown in (1.1), the interpreter does not generate code. The code shown immediately above is not explicitly produced by the interpreter but only describes the operations that the interpreter must execute at runtime once it has decoded the `for` command.

It can also be seen that for every occurrence of the same command in a program written in $\mathscr{L}$, the interpreter must perform a separate decoding steep; this does not improve performance. In our example, the command `C` inside the loop must be decoded n times, clearly with consequent inefficiency.

As often happens, the disadvantages in terms of efficiency are compensated for by advantages in terms of *flexibility*. Indeed, interpreting the constructs of the program that we want to execute at runtime allows direct interaction with whatever is running the program. This is particularly important, for example, because it makes defining program debugging tools relatively easy. In general, moreover, the development of an interpreter is simpler than the development of a compiler; for this reason, interpretative solutions are preferred when it is necessary to implement a new language within a short time. It should be noted, finally, that an interpretative implementation allows a considerable reduction in memory usage, given that the program is stored only in its source version (that is, in the language $\mathscr{L}$) and no new code is produced, even if this consideration is not particularly important today.

The advantages and disadvantages of the compilational and interpretative approaches to languages are dual to each other.

The translation of the source program, $\mathscr{P}^{\mathscr{L}}$, to an object program, $\mathscr{P}c^{\mathscr{L}o}$, occurs separately from the latter's execution. If we neglect the time taken for compilation, therefore, the execution of $\mathscr{P}c^{\mathscr{L}o}$ will turn out to be more efficient than an interpretive implementation because the former does not have the overhead of the instruction decoding phase. In our first example, the program fragment `P1` will be translated into fragment `P2` by the compiler. Later, when necessary, `P2` will executed without having to decode the `for` instruction again. Furthermore, unlike in the case of an interpreter, decoding an instruction of language $\mathscr{L}$ is performed once by the compiler, independent of the number of times this instruction occurs at runtime. In our example, the command `C` is decoded and translated once only at compile time and the code produced by this is executed n times at runtime. In Sect. 2.4, we will describe the structure of a compiler, together with the optimisations that can be applied to the code it produces.

One of the major disadvantages of the compilation approach is that it loses all information about the structure of the source program. This loss makes runtime interaction with the program more difficult. For example, when an error occurs at runtime, it can be difficult to determine which source-program command caused it, given that the command will have been compiled into a sequence of object-language instructions. In such a case, it can be difficult, therefore, to implement debugging tools; more generally, there is less flexibility than afforded by the interpretative approach.

1.2.3 Implementation: The Real Case and The Intermediate Machine

Burly purely compiled and interpreted implementations can be considered as the two extreme cases of what happens in practice when a programming language is implemented. In fact, in real language implementations, both elements are almost always present. As far as the interpreted implementation is concerned, we immediately observe that every "real" interpreter operates on an internal representation of a program which is always different from the external one. The translation from the external notation of $\mathscr{L}$ to its internal representation is performed using real translation (compilation, in our terminology) from $\mathscr{L}$ to an intermediate language. The intermediate language is the one that is interpreted. Analogously, in every compiling implementation, some particularly complex constructs are simulated. For example, some instructions for input/output could be translated into the physical machine's language but would require a few hundred instructions, so it is preferable to translate them into calls to some appropriate program (or directly to operating system operations), which simulates at runtime (and therefore interprets) the high-level instructions.

Can interpreter and compiler always be implemented?

At this point, the reader could ask if the implementation of an interpreter or a compiler will always be possible. Or rather, given the language, $\mathscr{L}$, that we want to implement, how can we be sure that it is possible to implement a particular program $\mathscr{I}_{\mathscr{L}}^{\mathscr{L}o}$ in language $\mathscr{L}o$ which performs the interpretation of all the constructs of $\mathscr{L}$? How, furthermore, can we be sure that it is possible to translate programs of $\mathscr{L}$ into programs in $\mathscr{L}o$ using a suitable program, $\mathscr{C}_{\mathscr{L},\mathscr{L}o}$?

The precise answer to this question requires notions from computability theory which will be introduced in Chap. 3. For the time being, we can only answer that the existence of the interpreter and compiler is guaranteed, provided that the language, $\mathscr{L}o$, that we are using for the implementation is sufficiently expressive with respect to the language, $\mathscr{L}$, that we want to implement. As we will see, every language in common use, and therefore also our $\mathscr{L}o$, have the same (maximum) expressive power and this coincides with a particular abstract model of computation that we will call *Turing Machine*. This means that every possible algorithm that can be formulated can be implemented by a program written in $\mathscr{L}o$. Given that the interpreter for $\mathscr{L}$ is no more than a particular algorithm that can execute the instructions of $\mathscr{L}$, there is clearly no theoretical difficulty in implementing the interpreter $\mathscr{I}_{\mathscr{L}}^{\mathscr{L}o}$. As far as the compiler is concerned, assuming that it, too, is to be written in $\mathscr{L}o$, the argument is similar. Given that $\mathscr{L}$ is no more expressive than $\mathscr{L}o$, it must be possible to translate programs in $\mathscr{L}$ into ones in $\mathscr{L}o$ in a way that preserves their meaning. Furthermore, given that, by assumption, $\mathscr{L}o$ permits the implementation of any algorithm, it will also permit the implementation of the particular compiling program $\mathscr{C}_{\mathscr{L},\mathscr{L}o}$ that implements the translation.

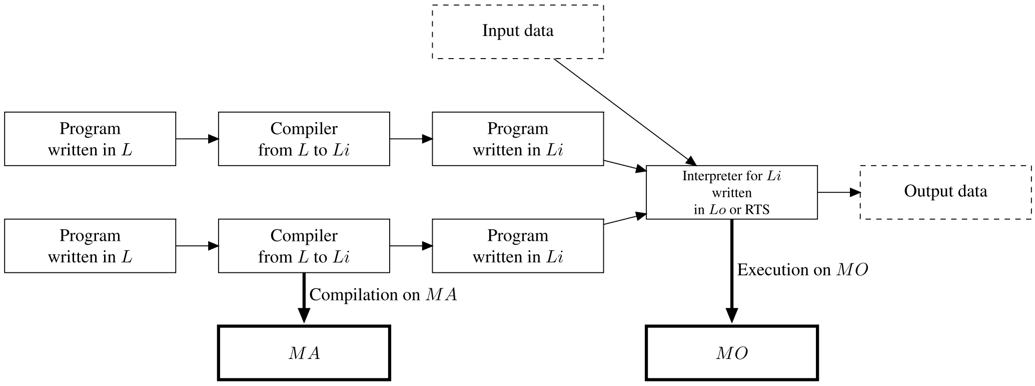


Fig. 1.6 Implementation: the real case with intermediate machine

The real situation for the implementation of a high-level language is therefore that shown in Fig. 1.6. Let us assume, as above, that we have a language $\mathscr{L}$ that has to be implemented and assume also that a host machine $\mathscr{M}o_{\mathscr{L}o}$ exists which has already been constructed. Between the machine $\mathscr{M}_{\mathscr{L}}$ that we want to implement and the host machine, there exists a further level characterised by its own language, $\mathscr{L}i$ and by its associated abstract machine, $\mathscr{M}i_{\mathscr{L}i}$, which we will call, the intermediate language and intermediate machine, respectively.

As shown in Fig. 1.6, we have both a compiler $\mathscr{C}_{\mathscr{L},\mathscr{L}i}$ which translates $\mathscr{L}$ to $\mathscr{L}i$ and an interpreter $\mathscr{I}_{\mathscr{L}i}^{\mathscr{L}o}$ which runs on the machine $\mathscr{M}o_{\mathscr{L}o}$ (which simulates the machine $\mathscr{M}i_{\mathscr{L}i}$). In order to execute a generic program, $\mathscr{P}^{\mathscr{L}}$, the program must first be translated by the compiler into an intermediate language program, $\mathscr{P}i^{\mathscr{L}i}$. Next, this program is executed by the interpreter $\mathscr{I}_{\mathscr{L}i}^{\mathscr{L}o}$. It should be noted that, in the figure, we have written "interpreter or runtime support (RTS)" because it is not always necessary to implement the entire interpreter $\mathscr{I}_{\mathscr{L}i}^{\mathscr{L}o}$. In the case in which the intermediate language and the host machine language are not too distant, it might be enough to use the host machine's interpreter, extended by suitable programs, which are referred to as its runtime support, to simulate the intermediate machine.

Depending on the distance between the intermediate level and the source or host level, we will have different types of implementation. Summarising this, we can identify the following cases:

1. $\mathscr{M}_{\mathscr{L}} = \mathscr{M}i_{\mathscr{L}i}$: purely interpreted implementation.
2. $\mathscr{M}_{\mathscr{L}} \neq \mathscr{M}i_{\mathscr{L}i} \neq \mathscr{M}o_{\mathscr{L}o}$.
   (a) If the interpreter of the intermediate machine is substantially different from the interpreter for $\mathscr{M}o_{\mathscr{L}o}$, we will say that we have an implementation of an interpretative type.
   (b) If the interpreter of the intermediate machine is substantially the same as the interpreter for $\mathscr{M}o_{\mathscr{L}o}$ (of which it extends some of its functionality), we will say that we have a implementation of a compiled type.
3. $\mathscr{M}i_{\mathscr{L}i} = \mathscr{M}o_{\mathscr{L}o}$, we have a purely compiled implementation.

The first and last cases correspond to the limit cases already encountered in the previous section. These are the cases in which the intermediate machines coincide, respectively, with the machine for the language to be implemented and with the host machine.

On the other hand, in the case in which the intermediate machine is present, we have an interpreted type of implementation when the interpreter for the intermediate machine is substantially different from the interpreter for $\mathscr{M}o_{\mathscr{L}o}$. In this case, therefore, the interpreter $\mathscr{I}_{\mathscr{L}i}^{\mathscr{L}o}$ must be implemented using language $\mathscr{L}o$. The difference between this solution and the purely interpreted one lies in the fact that not all constructs of $\mathscr{L}$ need be simulated. For some constructs there are directly corresponding ones in the host machine's language, when they are translated from $\mathscr{L}$ to the intermediate language $\mathscr{L}i$, so no simulation is required. Moreover the distance between $\mathscr{M}i_{\mathscr{L}i}$ and $\mathscr{M}o_{\mathscr{L}o}$ is such that the constructs for which this happens are few in number and therefore the interpreter for the intermediate machine must have many of its components simulated.

In the compiled implementation, on the other hand, the intermediate language is closer to the host machine and the interpreter substantially shares it. In this case, then, the intermediate machine, $\mathscr{M}i_{\mathscr{L}i}$, will be implemented using the functionality of $\mathscr{M}o_{\mathscr{L}o}$, suitably extended to handle those source language constructs of $\mathscr{L}$

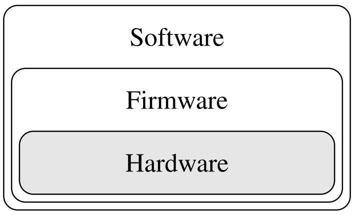
which, when also translated into the intermediate language $\mathscr{L}i$, do not have an immediate equivalent on the host machine. This is the case, for example, in the case of some I/O operations that are, even when compiled, usually simulated by suitable programs written in $\mathscr{L}o$. The set of such programs, which extend the functionality of the host machine and which simulate at runtime some of the functionality of the language $\mathscr{L}i$, and therefore also of the language $\mathscr{L}$, constitute the so-called *run-time support* for $\mathscr{L}$.

As can be gathered from this discussion, the distinction between the intermediate cases is not clear. There exists a whole spectrum of implementation types ranging from that in which everything is simulated to the case in which everything is, instead, translated into the host machine language. What to simulate and what to translate depends a great deal on the language in question and on the available host machine. It is clear that, in principle, one would tend to interpret those language constructs which are furthest from the host machine language and to compile the rest. Furthermore, as usual, compiled solutions are preferred in cases where increased execution efficiency of programs is desired, while the interpreted approach will be increasingly preferred when greater flexibility is required.

It should also be noted that the intermediate machine, even if it is always present in principle, is not often actually present. The exceptions are cases of languages which have formally stated definitions of their intermediate machines, together with their associated languages (which is principally done for portability reasons). The compiled implementation of a language on a new hardware platform is a rather big task requiring considerable effort. The interpretive implementation is less demanding but does requires some effort and often poses efficiency problems. Often, it is desired to implement a language on many different platforms, for example when sending programs across a network so that they can be executed on remote machines (as happens with so-called *aplets*). In this case, it is extremely convenient first to compile the programs to an intermediate language and then implement (interpret) the intermediate language on the various platforms. Clearly, the implementation of the intermediate code is much easier than implementing the source code, given that compilation has already been carried out. This solution to the portability of implementations was adopted for the first time on a large scale by the Pascal language, which was defined together with an intermediate machine (with its own language, P-code) which was designed specifically for this purpose. A similar solution was used by the Java language, whose intermediate machine (called the JVM—Java Virtual Machine) has as its machine language the so-called Java Byte Code. It is now implemented on every type of computer.

As the last note, let us emphasise the fact, which should be clear from what we have said so far, that one should not talk about an “interpreted language” or a “compiled language”, because each language can be implemented using either of these techniques. One should, instead, talk of interpretative or compiled implementations of a language.

Fig. 1.7 The three levels of a microprogrammed computer



1.3 Hierarchies of Abstract Machines

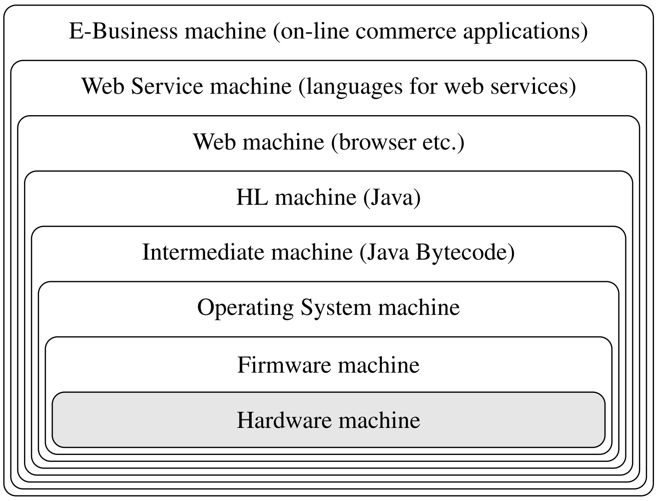
On the basis of what we have seen, a microprogrammed computer, on which a high-level programming language is implemented, can be represented as shown in Fig. 1.7. Each level implements an abstract machine with its own language and its own functionality.

This schema can be extended to an arbitrary number of levels and a hierarchy is thus produced, even if it is not always explicit. This hierarchy is largely used in software design. In other words, hierarchies of abstract machines are often used in which every machine exploits the functionality of the level immediately below and adds new functionality of its own for the level immediately above. There are many examples of hierarchies of this type. For example, there is the simple activity of programming. When we write a program $\mathscr{P}$ in a language, $\mathscr{L}$, in essence, we are doing no more than defining a new language, $\mathscr{L}_{\mathscr{P}}$ (and therefore a new abstract machine) composed of the (new) functionalities that $\mathscr{P}$ provides to the user through its interface. Such a program can therefore be used by another program, which will define new functionalities and therefore a new language and so on. It can be noted that, broadly speaking, we can also speak of abstract machines when dealing with a set of commands, which, strictly speaking, do not constitute a real programming language. This is the case with a program, with the functionality of an operating system, or with the functionality of a middleware level in a computer network.

In the general case, therefore, we can imagine a hierarchy of machines $\mathscr{M}_{\mathscr{L}0}$, $\mathscr{M}_{\mathscr{L}1}, \ldots, \mathscr{M}_{\mathscr{L}n}$. The generic machine, $\mathscr{M}_{\mathscr{L}i}$ is implemented by exploiting the functionality (that is the language) of the machine immediately below ($\mathscr{M}_{\mathscr{L}i-1}$). At the same time, $\mathscr{M}_{\mathscr{L}i}$ provides its own language $\mathscr{L}_i$ to the machine above $\mathscr{M}_{\mathscr{L}i+1}$, which, by exploiting that language, uses the new functionality that $\mathscr{M}_{\mathscr{L}i}$ provides with respect to the lower levels. Often, such a hierarchy also has the task of masking lower levels. $\mathscr{M}_{\mathscr{L}i}$ cannot directly access the resources provided by the machines below it but can only make use of whatever language $\mathscr{L}_{i-1}$ provides.

The structuring of a software system in terms of layers of abstract machines is useful for controlling the system's complexity and, in particular, allows for a degree of independence between the various layers, in the sense that any internal modification to the functionality of a layer does not have (or should not have) any influence on the other layers. For example, if we use a high-level language, $\mathscr{L}$, which uses an operating system's file-handling mechanisms, any modification to these mechanisms (while the interface remains the same) does not have any impact on programs written in $\mathscr{L}$.

Fig. 1.8 A hierarchy of abstract machines


E-Business machine (on-line commerce applications)
Web Service machine (languages for web services)
Web machine (browser etc.)
HL machine (Java)
Intermediate machine (Java Bytecode)
Operating System machine
Firmware machine
Hardware machine

A canonical example of a hierarchy of this kind in a context that is seemingly distant from programming languages is the hierarchy[5] of communications protocols in a network of computers, such as, for example, the ISO/OSI standard.

In a context closer to the subject of this book, we can consider the example shown in Fig. 1.8.

At the lowest level, we have a hardware computer, implemented using physical electronic devices (at least, at present; in the future, the possibility of biological devices will be something that must be actively considered). Above this level, we could have the level of an abstract, microprogrammed machine. Immediately above (or directly above the hardware if the firmware level is not present), there is the abstract machine provided by the operating system which is implemented by programs written in machine language. Such a machine can be, in turn, seen as a hierarchy of many layers (kernel, memory manager, peripheral manager, file system, command-language interpreter) which implement functionalities that are progressively more remote from the physical machine: starting with the nucleus, which interacts with the hardware and manages process state changes, to the command interpreter (or shell) which allows users to interact with the operating system. In its complexity, therefore, the operating system on one hand extends the functionality of the physical machine, providing functionalities not present on the physical machine (for example, primitives that operate on files) to higher levels. On the other hand, it masks some hardware primitives (for example, primitives for handling I/O) in which the higher levels in the hierarchy have no interest in seeing directly. The abstract machine provided by the operating system forms the host machine on which a high-level programming language is implemented using the methods that we discussed in previous sections. It normally uses an intermediate machine, which, in the diagram (Fig. 1.8), is the Java Virtual machine and its bytecode language. The level provided by the abstract machine for the high-level language that we have implemented (Java

[5] In the literature on networks, one often speaks of a *stack* rather than, more correctly, of a hierarchy.

Program Transformation and Partial Evaluation

In addition to "translation" of programs from one language to another, as is done by a compiler, there are numerous transformation techniques involving only one language that operate upon programs. These techniques are principally defined with the aim of improving performance. Partial evaluation is one of these techniques and consists of evaluating a program against an input so as to produce a program that is specialised with respect to this input and which is more efficient than the original program. For example, assume we have a program $P(X, Y)$ which, after processing the data X, performs operations on the data in Y depending upon the result of working on X. If the data, X, input to the program are always the same, we can transform this program to $P'(Y)$, where the computations using X have already been performed (prior to runtime) and thereby obtain a faster program.

More formally, a partial evaluator for the language $\mathscr{L}$ is a program which implements the function:

$$\mathscr{P}eval_{\mathscr{L}} : (\mathscr{P}rog^{\mathscr{L}} \times \mathscr{D}) \to \mathscr{P}rog^{\mathscr{L}}$$

which has the following characteristics. Given a generic program, P, written in $\mathscr{L}$, taking two arguments, the result of partially evaluating P with respect to one of its *first input* D_1 is:

$$\mathscr{P}eval_{\mathscr{L}}(P, D_1) = P'$$

where the program $\mathscr{P}'$ (the result of the partial evaluation) accepts a single argument and is such that, for any input data, Y, we have:

$$\mathscr{I}_{\mathscr{L}}(P, (D_1, Y)) = \mathscr{I}_{\mathscr{L}}(P', Y)$$

where $\mathscr{I}_{\mathscr{L}}$ is the language interpreter.

in this case) is not normally the last level of the hierarchy. At this point, in fact, we could have one or more applications which together provide new services. For example, we can have a "web machine" level in which the functions required to process Web communications (communications protocols, HTML code display, applet running, etc.) are implemented. Above this, we might find the "Web Service" level providing the functions required to make web services interact, both in terms of interaction protocols as well as of the behaviour of the processes involved. At this level, truly new languages can be implemented that define the behaviour of so-called "business processes" based on Web services (an example is the Business Process Execution Language). Finally, at the top level, we find a specific application, in our case electronic commerce, which, while providing highly specific and restricted functionality, can also be seen in terms of a final abstract machine.

1.4 Chapter Summary

The chapter has introduced the concepts of abstract machine and the principle methods for implementing a programming language. In particular, we have seen:

- The *abstract machine*: an abstract formalisation for a generic executor of algorithms, formalised in terms of a specific programming language.
- The *interpreter*: an essential component of the abstract machine which characterises its behaviour, relating in operational terms the language of the abstract machine to the embedding physical world.
- The *machine language*: the language of a generic abstract machine.
- *Different language typologies*: characterised by their distance from the physical machine.
- The *implementation of a language*: in its different forms, from purely interpreted to purely compiled; the concept of *compiler* is particularly important here.
- The *concept of intermediate language*: essential in the real implementation of any language; there are some famous examples (P-code machine for Pascal and the Java Virtual Machine).
- *Hierarchies of abstract machines*: abstract machines can be hierarchically composed and many software systems can be seen in such terms.

1.5 Bibliographic Notes

The concept of abstract machine is present in many different contexts, from programming languages to operating systems, even if at times it is used in a much more informal manner than in this chapter. In some cases, it is also called a virtual machine, as for example in [5], which, however, presents an approach similar to that adopted here.

The descriptions of hardware machines that we have used can be found in any textbook on computer architecture, for example [6].

The intermediate machine was introduced in the first implementations of Pascal, for example [4]. For more recent uses of intermediate machine for Java implementations, the reader should consult some of the many texts on the JVM, for example, [3].

Finally, as far as compilation is concerned, a classic text is [1], while [2] is a more recent book with a more up-to-date treatment.

1.6 Exercises

1. Give three examples, in different contexts, of abstract machines.
2. Describe the functioning of the interpreter for a generic abstract machine.

3. Describe the differences between the interpretative and compiled implementations of a programming language, emphasising the advantages and disadvantages.
4. Assume you have available an already-implemented abstract machine, C, how could you use it to implement an abstract machine for another language, L?
5. What are the advantages in using an intermediate machine for the implementation of a language?
6. The first Pascal environments included:

   - A Pascal compiler, written in Pascal, which produced P-code (code for the intermediate machine);
   - The same compiler, translated into P-code;
   - An interpreter for P-code written in Pascal.

   To implement the Pascal language in an interpretative way on a new host machine means (manually) translating the P-code interpreter into the language on the host machine. Given such an interpretative implementation, how can one obtain a compiled implementation for the same host machine, minimising the effort required? (Hint: think about a modification to the compiler for Pascal also written in Pascal.)
7. Consider an interpreter, $\mathscr{I}^{\mathscr{L}}_{\mathscr{L}1}(X, Y)$, written in language $\mathscr{L}$, for a different language, $\mathscr{L}1$, where X is the program to be interpreted and Y is its input data. Consider a program P written in $\mathscr{L}1$. What is obtained by evaluating

   $$\mathscr{P}eval_{\mathscr{L}}(\mathscr{I}^{\mathscr{L}}_{\mathscr{L}1}, P)$$

   i.e., from the partial evaluation of $\mathscr{I}^{\mathscr{L}}_{\mathscr{L}1}$ with respect to P? (This transformation is known as Futamura's first projection.)

References

1. A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, 1988.
2. A. W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, Cambridge, 1998. This text exists also for C and ML.
3. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, 2nd edition. Sun and Addison-Wesley, Cleveland, 1999.
4. S. Pemberton and M. Daniels. *Pascal Implementation: The p4 Compiler and Interpreter*. Ellis Horwood, Chichester, 1982.
5. T. Pratt and M. Zelkowitz. *Programming Languages: Design and Implementation*, 4th edition. Prentice-Hall, New York, 2001.
6. A. Tannenbaum. *Structured Computer Organization*. Prentice-Hall, New York, 1999.