Chapter 13
A Short Historical Perspective

Even if the first computers in the modern sense, and therefore the first programming languages, appeared only at the end of the 1940s, since then many hundreds (if not thousands) of languages have been defined. In the previous chapters of this book we have sought to identify the most important design and implementation characteristics that are common to large classes of contemporary languages.

In this last chapter, we seek to understand what were the reasons that lead, in the last sixty years, to the affirmation of these characteristics and, therefore, to the success of some languages and the disappearance of many others.

13.1 Beginnings

The first electronic computers appeared in the second half of the 1940s, a relatively recent time when considered from the perspective of the history of science, a remote one when the rate of development of information technology is considered. To realize distance that separates us from the early history of computing, one should think that the first computers were elephantine machines (length greater than 10 metres, weight greater than 4 tons), so expensive that they could only be tackled by government institutes or by large agencies and they had processing rates less than that of an old programmable pocket calculator.

There is even now something of a debate about what must be considered the first computer because this primacy depends on upon what is exactly meant by this term. In computing circles, there is a certain agreement that a computer in the modern sense must have the following properties: (i) it is electronic and digital; (ii) is able to perform the four elementary arithmetic operations (iii) it is programmable; (iv) it allows the storage of programs and data. If we consider this definition, probably the first computer that was operational, with an adequate memory, was the EDSAC designed and developed at the University of Cambridge by Maurice Wilkes' group. It went live in 1949 and it should be remembered that the EDSAC was influenced by famous work by J. Mauchly and J.P. Eckert of the Moore School at the University of Pennsylvania, who used some of J. von Neumann's ideas; their device was described

M. Gabbrielli, S. Martini, *Programming Languages: Principles and Paradigms,*
Undergraduate Topics in Computer Science,
DOI 10.1007/978-1-84882-914-5_13, © Springer-Verlag London Limited 2010

as a computer and was called EDVAC. The EDVAC had properties similar to those enumerated above. However it was not constructed until 1951.

If instead we admit a more general definition of computer, then we can also consider as the pretender to the title of first computer, ASCC/MARK I and ENIAC (1946) which, however it may be, remain fundamental precursors. These machines were able to execute sequences of arithmetic operations in a controlled fashion using a real program, even if it was not stored and was expressed using very rough formalisms, using physical representations in some cases.

ASCC/MARK I (IBM Automatic Sequence Controlled Calculator or Harvard Mark I) was constructed in 1944 by IBM with the cooperation of Harvard University with H. Aiken as principal investigator. This machine, used by the U.S. Navy for military tasks, was very rudimentary to our modern eyes. Indeed, it succeeded only in carrying out the work of a few tens of people and required the use of punched tapes and other external physical media to obtain its instructions and to transmit data to the computation devices. Think, for example that a (decimal) constant of 23 figures was specified manually using 23 switches, each of which could occupy 10 positions corresponding to the 10 decimal figures!

ENIAC, on the other hand, was constructed in 1946 by J. Mauchly and J.P. Eckert at the Moore School of the University of Pennsylvania and initially J. von Neumann also worked on the design. ENIAC also did not have program storage and was programmed using physical devices, for example using electrical cables internal to the machine to connect the different physical parts of the computer according to input parameters. Furthermore, many consider this machine as the first real computer because, unlike ASCC/MARK I, ENIAC was many orders of magnitude faster than humans at computing and was immediately recognised as a tool for fundamental progress. Already in 1947, L.P. Tabor of the Moore School foresaw that the speed of ENIAC and computers would allow "the solution of mathematical problems until now never considered because of the enormous amount of computation required." ENIAC was used effectively for the complicated calculation of ballistic trajectories.

The limitations of these first machines were clearly due to their novelty. All of the hardware technology (the electronic devices used in modern computers) was still to be invented, as was Computer Science itself (it was just at its beginning). The importance of the programming task and the development of linguistic tools to support it were not yet recognised. In the first applications (which were often military), the programming activity was seen as a phase additional to that of the proper calculation. Even with the more general applications, such as those to which EDSAC and other machines of this time were put, matters were not much better. To program, one used a low-level machine language which described, using binary code, the operations and calculation mechanisms of the machine itself. This was *machine language*, composed of elementary instructions (for example, instructions for adding, loading a value into a register, and so on) that could be immediately executed by the processor. The process of coding was completely manual, there being not even the concept of a symbolic assembly code. Machine languages are also called *first-generation* languages (or 1GL).

Without reaching the limit cases of "physical" coding constants in MARK I, it is clear that the use of such languages made the writing of programs more diffi-

cult and the correction of such programs, once they had reached a certain size, was impossible. It was soon realised that to exploit the full use of the power of the computer, it was necessary to develop adequate formalisms that were far from machine "languages" and closer to the user's natural language.

A first step in this direction was the introduction of *assembly languages*. These languages can be seen as symbolic representations of the machine language and, indeed, there is a one-to-one correspondence between a large number of machine language instructions and assembly language codes. Programs written in assembly language are translated into programs written in machine language by a program called an *assembler*. Every model of computer has its own assembly language, so the portability of these programs is nearly impossible. Assembly languages are also called second-generation languages (or 2GL).

The true jump in quality was achieved in the 1950s with the introduction of *high-level languages*, also called third-generation languages (or 3GL). These were designed as abstract languages which would ignore the physical characteristics of the computer and were instead suited to express algorithms in a way that was relatively easy for the human user. Among the first attempts in this sense were some formalisms that permitted the use of symbolic notation to indicate arithmetic expressions. The expressions thus encoded could then be translated automatically into instructions executable by the machine. From these attempts, as the name itself suggests, the FORTRAN (FORmula TRANslation) language was born in 1957. FORTRAN can, as we will see below, be considered in all respects to be the first true high-level language.

From 1957 until today, many hundreds of programming languages have been implemented and it is thought that there have been more than 100 in widespread use. In addition to fashion, commercial (certainly the most important) and chance circumstances, in the development of high-level languages it is possible to recognise some guiding lines and some developmental principles. In the remainder of this chapter, we will therefore seek to delimit these principles in such a way as to provide a way of orienting oneself when entering the modern "Babel" of programming languages. We do not pretend to be exhaustive because a complete guide would require another book.

13.2 Factors in the Development of Languages

High-level languages have always been designed with the aim of assisting the task of programming computers. However, from the 1950s to the present, the importance and the cost of the various components involved in the implementation of programs has changed and therefore priorities when designing a language have completely changed.

In the 1950s, the hardware was certainly the most expensive and important resource (consider that Thomas Watson, president of IBM, asserted in 1943 that there would be a world-wide market for 5 computers!). The first high-level languages were therefore designed with the aim of obtaining efficient programs which would

use the potentiality of the hardware to the maximum. This attitude in the first languages is reflected by the presence of many constructs that were inspired directly by the structure of the physical machine. For example, the "three-way jump" present in FORTRAN directly derives from the corresponding instruction on the IBM 704. The fact then that programming was very difficult and required very long times was considered as a problem of secondary importance, which could be solved by means of large amounts of human resources which were certainly less expensive than the hardware.

Today, the situation is the direct opposite. Hardware is relatively cheap and efficient and the preponderant costs of information-system development are linked to the tasks performed by computing specialists. Furthermore, given the increasingly critical application of computer systems (one thinks of applications in avionics or nuclear power), there are considerations of correctness and security that were minimal if not wholly absent 50 years ago. Modern languages are therefore designed taking into account first the improvement of various software project activities, while the preoccupation with efficient use of the physical machine has dropped to second place, except in some particular cases.

Clearly, we have not passed from the 1950s to the present with a single solution: the development of programming languages has followed a long, continuous process governed by a number of factors. Here, we can only see some of the most important:

Hardware The type and performance of available hardware devices clearly influenced the languages that used it. We will see this point better in the next sections, where we indicate, in various historical contexts, the influence of physical machines on the languages of their time.

Applications Applications of computers, initially solely of a numeric type, rapidly extended to many different fields, including some that required the processing of non-numeric information. New application fields can require languages with specific properties. For example, in Artificial Intelligence and knowledge processing languages are needed that allow the manipulation of symbolic formalisms rather than the solution of mathematical problems. As another example, computer games are usually implemented using particular programming languages.

New Methodologies The development of new programming methodologies, particularly programming in the large, has influenced the development of new languages. A significant example in this sense is object-oriented programming.

Implementation The implementation of language constructs is significant for the development of successive languages because it allows to understand the validity of a construct to be taken into consideration as well as its practical use.

Theory Finally, the role of theoretical studies should not be forgotten. They play an important role in selecting some typologies of structure and in particular in identifying new technical tools to improve the programming activity. One can think of, for example, what we have seen on the elimination of the `goto` and the introduction of refined type systems.

These factors, as well as others, are discovered during the lifecycle of a programming language, which, for the most part, is relatively brief. Some languages are an

exception and, either because of the quality of the original design or, more commonly, because of commercial motives, live for more than a decade. We will see some below.

13.3 1950s and 60s

As we saw at the start of this chapter, the first computers, during the period from the end of the 1940s to the start of the 1950s, can be considered as interesting precursors from an historical viewpoint, but which are still very far from the modern computer, in terms both of the languages they used as in the applications implemented on them.

Towards the end of the 1950s and in the 1960s, mainframes, the first true general-purpose computers that could be used for many different applications, came to the fore. These machines were, however, available only at a few major processing centres because of their enormous size and cost (they filled a room and cost millions of Euros, in modern terms). They were usable only by specialised people. The IBM 360 is a famous mainframe example which has held on for many years in the largest data centres.

The processing methods used by mainframes were called *batch* processing. In this kind of system, programs were executed in a strictly sequential manner. The entire computational resource of the computer was assigned to a program which took a "batch" of data as input and produced, as output, another batch of data. When a program terminated, the next program would execute. Data and program were initially represented on punched cards which were read by suitable equipment. The data structures mainly used for data input and output were files. Such systems did not engage in much interaction with users. For example, when there was an error during a run, the program had to be capable of re-establishing the correct state itself, given that no external interactive intervention was possible.

The first high-level languages were developed for this type of processing system. They were languages that offer few opportunities for interaction with the machine and which were suited to writing monolithic programs to be executed from start to finish without external interaction.

FORTRAN As has already been said, the first real high-level imperative language can be considered to be FORTRAN, developed by John Backus' group in 1957 and designed for applications of a numerical-scientific type. At a time in which programming was only done in assembly language and in which the biggest preoccupation was with the efficiency of programs, high-level language design could not ignore the performance of compiled code. Also the design of FORTRAN put performance first and therefore the characteristics of a specific physical reference machine (the IBM 704) was considered in the design. However, unlike previous languages, FORTRAN was already a high-level language in the modern sense in its first version. In fact, this first version contained many constructs that were more or less independent of a specific machine. In particular, it should be emphasised that FORTRAN was the first programming language to allow the direct use of a symbolic arithmetic expression.

A simple expression such as `a * 2 + b` that today we can use in (almost) any high-level language, could not be used in a language prior to FORTRAN. After a number of modifications and new versions (in particular, those in 1966, 1977 and 1990), FORTRAN has survived until today and is still used for some numerical applications. It is, though, a dated language whose survival is mainly linked to practical matters such as the vast library of functions for scientific and engineering calculations. A FORTRAN program consists of a main routine and a series of subprograms that can be separately compiled. It is not possible to define nested environments (it is possible only in FORTRAN90 to define nested subprograms) and there are two kinds of environment: the local and the global. This, as we know, greatly simplifies the handling of names, as well as the environment. Memory, either for subprograms or for main, is statically allocated and there is no dynamic memory management (FORTRAN90 is again an exception by providing dynamic memory). The sequence control commands in the first version referred directly to assembly language and therefore `goto` was used a great deal. In successive versions more structured commands (for example, `if then else`) were introduced. Parameter passing is by reference or rather by value-result. Types are present in a highly limited way, including only numeric (integer, real in single and double precision, complex), boolean, array, string and file.

ALGOL ALGOL means, more than a language, a family of imperative languages introduced at the end of the 1950s. These languages, even if they have not become true commercial successes, were predominant in the academic world in the 1960s and 70s and had a formidable impact on the design of all successive programming languages. Many of the concepts and constructs that are found in modern languages were introduced, or experimented with, for the first time in the languages of the ALGOL family.

The progenitor was ALGOL58, designed in 1956 by a committee lead by Peter Naur. The committee was the fusion of two previous groups, one European and one American, each tasked with the definition of a new language. The name ALGOL is an acronym for ALGOrithmic Language and clearly indicates the finality of the language. Unlike FORTRAN, indeed, ALGOL was designed as a universal language, suited to expressing algorithms in general, rather than for use in specific types of application. There were various revisions to the language, all originating from the collective work of an international committee (which had a turbulent life, particularly towards the end). In 1960, ALGOL60 was defined. This is the version that is perhaps the most important (it had a minor revision in 1962). Beginning in 1966, dissenting from the majority of the committee, C.A.R. Hoare and N. Wirth started the basic design of ALGOLW, later implemented in 1968 by Wirth and which constitutes the progenitor of Pascal. The majority of the committee, instead, continued their own work which led to the definition of ALGOL68.

ALGOL58 and, in particular, ALGOL60 greatly increased the machine-independence of the language, making the notation used for programming closer to mathematical notation and avoiding almost every reference to specific architectures, even at the cost of some additional design complications. For example, the input and

output operations, rather than being coded by suitable instructions in the language, must be implemented by appropriate procedures that are defined specifically for the devices that are being used at a particular installation.

ALGOL60 made at least three fundamental contributions to modern languages. The first consists of the introduction of parameter passing by name, a mechanism, as we have seen, that is complicated to implement (and for this reason later abandoned), but which has been of great importance for defining mechanisms that are used in current languages for passing functions.

Another important innovation in ALGOL60 was the introduction of blocks and therefore the ability of hierarchically structuring the environment.

At the syntactic level, though, thanks to the contribution of John Backus, ALGOL60 was the first language to use Chomsky's generative grammars to express the syntax of the language (in particular context-free grammars were used in the form that we now know as Backus Naur Form or BNF). This novelty opened the way to a whole new area of research that has been of extreme importance to the theory and practice of compilers.

Among the other contributions of the languages in the ALGOL family to modern languages, let us further note: recursion and dynamic memory management (but for these features see also what we say below about LISP); type systems with the ability to permit new user-defined types; finally, many structured commands for sequence control in the form that we use today, (`if then else`, `for` and `while` from ALGOL60 or `case` from ALGOL68).

LISP LISP (LISt Processor) was designed in 1960 by a group led by John McCarthy at MIT (Massachusetts Institute of Technology) and was one of the first languages designed especially for non-numeric applications. As we have already seen in the box on page 121, it is a language designed to manipulate symbolic expressions (called s-expressions) which are basically lists and which are typically used in Artificial Intelligence. Among LISP applications were the first attempts to implement programs for the automatic translation of texts.

Even if this is a non-standard language, developed and implemented over the course of 30 years in many different versions and not much used commercially, it is a very important language, in which various techniques have been developed that are of interest to languages. In the academic world LISP still enjoys a following (the Scheme language, currently used in many courses was born from a variant of LISP). The first implementations of LISP were very inefficient, so much so that architectures specially designed for LISP (so-called LISP machines) were constructed. Later, the use of various tricks, and in particular the improvement of garbage-collection techniques allowed efficient implementations even on traditional architectures.

LISP is a functional language which, as has been said, manipulates special data structures. Every program consists of a sequence of expressions to be evaluated. Some of these expressions can be function definitions that are to be used in other expressions. Typically, the language is implemented using an interpreter and programs are evaluated in an interactive environment. Among the contributions of LISP, we

can note: the introduction of higher-order programming, i.e. the possibility of constructing functions that accept as parameters and/or produce as the result of evaluation other functions, as well as dynamic management of memory using a heap and a garbage collector. Other specific properties of LISP have remained confined almost exclusively to this language, such as, for example, the use of the dynamic scope rule implemented using A-lists.

COBOL Like LISP, also this language was designed in the 60s and its name is also an acronym: COBOL stands for COmmon Business Oriented Language. The similarities between this language and LISP end there, however. COBOL was designed with the aim of producing a language that was specific to commercial applications and whose syntax was as close as possible to the English language. After various revisions of the initial language, which was designed by a team lead by Grace Hopper at the US Department of Defense, the language became a standard in 1968 and, even if in revised and modified versions, is still in use today.

COBOL programs are composed of 4 "divisions". In the "procedure division", the code for the algorithmic aspects of the program are written. In the "data division", the descriptions of the data are written. The "environment division" contains the specification of the environment external to the program provided by the physical machine on which the language is implemented. Finally, there is the "identification division" which contains information used to identify the program (name, author, etc.). This organisation has the aim of separating, even if in a very crude fashion, data from the programs that use it, and separating machine-independent and dependent aspects. These are highly rudimentary mechanisms that have been superceded by linguistic devices that we have seen in modern languages (types, abstract data types, objects, modules, intermediate code, etc.). Furthermore, this program structure, together with a syntax that is close to natural language, also makes simple programs quite long. Memory management is entirely static.

Simula Simula, another descendent of ALGOL60, is a textbook case of a technology that was too advanced for its time. Developed from 1962 at the Norwegian Computing Centre by K. Nygaard and O.J. Dahl, Simula is an extension of ALGOL. It was designed for discrete-event simulation applications, that is for the implementation of programs that simulate load and queue situations so that fundamental parameters can be measured (average waiting time, length of queue, etc.). In its most important version, Simula67, the language introduced for the first time the concepts of class, object, subtype and dynamic method dispatch. This is without doubt the first object-oriented language and it had a considerable influence on its successors (Smalltalk and C++) even if the concept of "object" was a biological metaphor which would only arrive with Smalltalk and its volcanic creator, Alan Kay.

From the linguistic viewpoint, Simula67 makes small modifications to the constructs that were present in ALGOL60 (the most import of which is the default mode for passing parameters which is changed from name to value-result) but adds various mechanisms, among which call-by-reference, pointers, coroutines (a mechanism for defining concurrent procedures that is fairly close to the modern concept of thread),

classes and objects. A class in Simula is a procedure which, when it terminates, leaves its activation record on the stack and returns a pointer to it. An activation record of this kind, which contains the variables declared local to the procedure (today we would say: instance variables) and (pointers to) local functions (methods), is an object. Subtypes, dynamic dispatch and inheritance are already present in Simula67, while later versions introduced abstraction mechanisms.

Simula has always had an big impact even outside the academic world. In 2003, applications written in this language were still being cited.

13.4 The 1970s

Thanks to the advent of the microprocessor, the 1970s saw the rise of the minicomputer, computers of smaller size and power comparable to that of the older mainframes.

From the software viewpoint, batch processing gave way to a more interactive approach in which the user, using a terminal, could interact directly with the execution of the program. The characteristics of the languages developed in the mainframe era were not really suited to interactive systems. For example, it became necessary to be able to express operations for input and output (that were more sophisticated than simply reading and writing files) using a program and therefore through the constructs of the language. In interactive systems there are constraints on the response time and therefore new languages included appropriate linguistic constructs of a temporal nature (for example, timeout mechanisms). In general, "traditional" high-level languages from the 1970s allow more direct interaction with the machine that was impossible with the languages of the preceding generation.

Above, we said "traditional", meaning by this the imperative languages inspired by the classical computational model based on the modification of values stored in memory locations. The 1970s however saw the birth of two new programming paradigms: object-oriented programming and declarative programming. The second can be more accurately divided into functional and logical approaches. We will see below the more important languages of these years in the different paradigms.

C Among the new languages of the 1970s, the most important is probably C, a language designed by Dennis Ritchie and Ken Thompson at AT&T Bell Laboratories. Initially designed as a systems programming language for the Unix operating system, C soon became a general purpose language, even if systems programming remains one of its more important application areas. By way of an anecdote, let us note that the name derives from the fact that the language in 1972 was the successor to a language called B, which in its turn was a reduced version of the system language BCPL.

Compared with the languages in the ALGOL family, from which it inherited a great deal, C offers more opportunities to access the low-level functionality of the machine and to program interactive systems. For example, in C it is possible to access directly the characters emitted by a terminal; it is also possible to use specific

commands to process data in real time. C rapidly established itself, both for system programming and as a language for general use, thanks to these characteristics, to its compact syntax and to the possibility of translating its programs into efficient machine code.

In the chapters above, we saw numerous references to C in connection with specific constructs or specific implementation characteristics; it would be useless to repeat them all here. Let us recall just some of the important characteristics. We have seen how C includes many arithmetic and assignment operators and how the block structure of C considerably simplifies that found in languages of the ALGOL family (basically, C does not allow nested functions). This allows a much simpler handling of environments and, therefore, of the passing of functions as parameters. The explicit presence of pointers that can be directly manipulated by the program and the equivalence between them and arrays, if on one hand allows very powerful operations, on the other cause insidious errors. This fact, together with the lack of a strong type system, constitutes one of the critical points of the language. Indeed, when one wants more reliability than efficiency, one can find valid alternatives in other modern languages.

Pascal Pascal was developed around 1970 by Niklaus Wirth as a development and simplification of ALGOLW and was the most used educational language right up to the end of the 1980s. The name is in honour of the mathematician (as well as physicist, philosopher and writer) Blaise Pascal, who, to help a father overloaded by a difficult job as part of the administration of Normandy, in 1642 designed an "arithmetic machine", the precursor of mechanical calculating machines.

One of the main reasons for the success and fame of Pascal is that it was the first language which, prefiguring Java and its bytecodes by nearly 20 years, introduced the concept of intermediate code as an instrument for program portability. A Pascal program was translated by the Pascal compiler (which was also written in Pascal) into P-code. P-code was a language for an intermediate machine with a stack architecture which was then implemented in an interpretative way on the host machine. In this way, to port Pascal to a different machine, it was only necessary to rewrite the P-code interpreter. Pascal was also implemented in a compilative way that did not use an intermediate machine thus allowing greater efficiency.

We have included many references to Pascal in this book and we do not mean to summarise them in their entirety. Instead, we merely record some of the more important properties in an attempt to compare them with C.

Pascal is a block-structured language in which it is possible to define functions and blocks that can be nested with arbitrary complexity. This fact, if on the one hand increases the structuring of code, complicates the handling of the environment and the mechanisms for passing functions as parameters, as has already been observed. Pascal (like C), uses the static scope rule and includes dynamic memory management both using a stack (for activation records) and a heap (for explicitly allocated memory). The Pascal type system is fairly extensive and supports abstraction mechanisms by allowing the user to define new types using the `type` primitive. The types are for the most part checked statically by the compiler, even if some checks are executed at runtime. In Pascal, it is also possible to handle pointers explicitly, even if

not all the operations (particularly the dangerous ones) provided by C are available. For example, in Pascal, arrays and pointers are two different types. Perhaps, in the search for a reliable type system, concerning arrays Pascal is too restrictive; indeed, in Pascal, two array declarations that have different dimensions and contain objects of the same type define two different types, a property which make general design of array-manipulating procedures difficult. Another limit of the language, at least in its original version, is the lack of separately compilable modules, although this has been solved in many subsequent implementations which allow the definition of external procedures.

Smalltalk A strong limitation of Pascal, like all "conventional" programming languages from the 1970s was in the lack of mechanisms to support the concept of encapsulation and information hiding in a truly efficient manner. Indeed, even if in Pascal it is possible to define new data types, there is no way to limit access to values to a pre-determined set of operations in such a way as to guarantee abstraction over the data.

Smalltalk presents a novel way to integrate in a programming language mechanisms for encapsulation and information hiding using the concepts of class and object (previously introduced by Simula) and precise visibility rules for classes (methods are public, instance variables are private). Developed during the 1970s by Alan Kay at Xerox PARC (Palo Alto Research Center),[1] Smalltalk is a unique language, some of whose characteristics we saw in Chap. 10 and whose description would require much more space than available here. It is also important for us to note that, unlike some object-oriented languages that were introduced later (for example, C++), Smalltalk was designed from the start to include as primitive the concept of object rather than grafting it on to an existing language. This means that all the mechanisms used in its implementation (procedure call, memory management, etc.) were developed using this concept. Moreover, Smalltalk was designed not only to be a language but also a sort of "total system" which included language, programming environment and also a special dedicated machine for increased efficiency of program execution. Indeed, given that the language's type system is entirely dynamic, the implementation of an efficient method lookup mechanism was quite difficult. Very soon, however, implementations of Smalltalk were also proposed for conventional machines with satisfactory results. A standard has never been defined for Smalltalk and various, quite different, versions of the language exist today.

Declarative languages In Chap. 6, we saw, at least on the formal level, the difference between imperative and declarative programming. From the intuitive viewpoint, the slogan of declarative programming, so to speak, is that the activity of

[1] In those years, Xerox PARC was a research centre of mythical proportions. In addition to Smalltalk, the following were all PARC innovations: Ethernet; laser-printer technology (later developed by Adobe, a PARC spin-off); PostScript; the first personal computer (the Alto) for "office automation", equipped with a graphical user interface using the desktop metaphor; the remote procedure call.

programming should concentrate on what is to be done, leaving the language interpreter to concentrate on how to reach the desired result. In imperative programming, on the other hand, the programmer must specify both the what and the how. Obviously this is an ideal vision. Leaving the interpreter to decide everything about how the computation is to be undertaken (that is, substantially, memory management and sequence control) without the programmer providing any indication in this sense imposes a great penalty on program efficiency. In reality, therefore, to the "pure" version of declarative languages, which conform to this vision, we need to add "impure" versions which add constructs of an imperative nature to improve the efficiency of programs and also to permit the use of commands (for example, assignments) with which many traditional programmers are acquainted.

Declarative languages can be divided into functional and logic-programming languages. We will look at the two most important representatives of the two classes, both of which were introduced in the 1970s.

ML ML was born as Meta Language (hence its name) for a semi-automatic system for proving properties of programs and was developed by Robin Milner's group at Edinburgh, starting the middle of the 1970s. Very soon it became a true programming language that could be used, in particular, for the manipulation of symbolic information.

In ML, as in LISP, a program consists of a set of function definitions. Various imperative constructs were added to the purely functional part of ML, in particular assignment (which is limited to so-called "reference cells" which can be regarded as modifiable variables which use the reference model).

The most important contribution of ML concerned types. The language was indeed provided with a safe type system that is static and extends the type system of Pascal in various ways. First, the concept of type safety that we have already encountered has a rigorous and very precise definition that excludes (in a provable way) the possibility of unsignalled runtime errors that derive from type violations. The ML type checker statically determines the type of every expression in a program and there is no way to change this type. If the type checker determines that an expression has an integer type, then we can be sure that every evaluation of this expression that succeeds will yield an integer.

Moreover, the ML type system supports a type inference mechanism. The programmer can leave some information about types unspecified and the system, using a form of logical inference, deduces the type of the identifiers from the way in which they are used. Even if similar mechanisms were previously studied in the context of the λ-calculus, ML was the first language to be equipped with a type-inference mechanism.

Finally, ML support parametric polymorphism. That is, it supports the use of type variables that can then be consistently instantiated by concrete types.

PROLOG PROLOG was defined in the 1970s as well. This was the first logic-programming language and is still available today in various versions and implementations.

If, as we have said, some ideas on logic programming can be traced back to the work of Kurt Gödel and Jacques Herbrand, the first solid theoretical bases were published by A. Robinson who, in the 1960s, made an essential contribution to the theory of automatic deduction. Robinson indeed provided a formal definition of the unification algorithm and defined *resolution*, a deduction mechanism that uses unification and allows the proof of theorems in first-order logic.

Given its simplicity, resolution is perfect for implementing automatic theorem provers (for first-order logic), but it does not provide a computational mechanism such as that normally provided by a programming language. The proof of a theorem, of itself, does not produce an observable result what could be seen as the result of a computation. To obtain this computational vision of the activity of proof, 10 years had to pass and a restricted version of resolution had to be developed. This version is SLD resolution, proposed by Robert Kowalski in 1974. SLD resolution unlike previous mechanisms for automated theorem proving, allows to prove a formula by explicitly computing the values of the variables that make the formula itself true and which, therefore, at the end of the proof, constitute the result of the computation. If Kowalski defined the theoretical model, we must record that in reality the PROLOG language was developed by Pierre Roussel and Alain Comerauer who, in 1970, were working on a formalism for manipulating natural language based on automatic theorem-proving mechanisms. These experiments led to the first implementation of PROLOG in 1972 and then, after various interactions with Robert Kowalski, to the 1973 version which is mostly the same as current versions. The ISO PROLOG standard was defined in the 1990s.

13.5 The 1980s

The 1980s were dominated by the development of the personal computer or PC. The first commercial PC can be considered to be the Apple II, produced by Apple in 1978. Even if today, with the massive increase in the use of computing devices in everyday life, the PC seems an indispensable tool, it should be noted that when they first appeared, most people remained sceptical about their potential. It is sufficient to observe that even in 1977, Ken Olson, the president of Digital Equipment Corp. (a major producer of minicomputers at that time), stated that he did not see any reason for anyone to want to have a computer in their home! This initial confusion evaporated when, in 1981, IBM launched its first PC and Lotus created the first electronic spreadsheet. This application made people immediately see the possibilities in this new technology which then went into general use in 1984. In that year, Apple released onto the market the Macintosh, a computer that had the first operating system with a graphical interface based on windows, icons and a mouse, a system that was similar to the windowing interface that we use today. Later (in the 1990s), Microsoft also introduced its own windows system.

The PC has completely changed the role of programming languages. The development of systems for personal use which provides easy-to-user graphical interfaces, has brought the need to develop easily interactive graphical systems that are

needed to manage windowed interfaces. These system are produced using very large and complex programs and therefore it is essential to be able to reuse already existing code (possibly produced by others). These systems have, therefore, provided an application area that is ideal for object-oriented languages which, as we have seen, provide natural mechanisms for code reuse and for organising vast and complex systems. Indeed, the languages of the 1980s were conceived as object-oriented languages which saw significant development during this period and their first large commercial applications.

In this decade, the first embedded systems were also developed. These are systems composed of computers connected to physical devices which perform control tasks (for example, motors, parts of industrial machinery, domestic electrical goods, etc.). Embedded systems pose many problems, most of all relating to reliability and correctness of programs. For a program that controls the engines of an aircraft, an error must not, clearly, be handled in an interactive fashion by the programmer and termination of the program is not an acceptable option. Moreover, embedded systems introduce other problems as far as response time is concerned; these problems are tackled by so-called real-time programming languages.

C++ The first version of C++ was defined by Bjarne Stroustrup in 1986 at Bell Laboratories of AT&T after a number of years of work (and after the definition of various other languages) on finding out how to add classes and inheritance to the C language without prejudicing efficiency and without compromising compatibility with the existing C language. C had to remain a subset of C++ and as such had to be acceptable to the C++ compiler. To these primary objectives, the improvement of C's type system was added.

We can say that these objectives were substantially achieved. Even if there are some inconsistencies between C++ and C, the most C programs can be translated by a C++ compiler. There was significant effort expended to obtain a language in which those constructs that are not used by a program do not exert a negative influence on the efficiency of the program itself. This means, in particular, that the C subset of C++ must not feel, in any way, the presence of objects and the structures necessary to handle them. C++ does not use any form of garbage collector so that it remains compatible with C and is still as efficient.

In particular, the static type system was improved and, in C++, in particular, it is possible to use a generic form of class, called a template, which supports a form of parametric polymorphism. An important design decision was that of handling C++ objects as a generalisation of the structures (`struct`) in C. The significance of this is that objects can also be allocated in activation records on the stack and, unlike Simula and Java, they can be manipulated directly rather than via pointers. An assignment in C++ can then physically copy an object to the memory space that was occupied by another object rather than just manipulating a pointer.

The lookup mechanism for methods in C++ is simpler and more efficient than that in Smalltalk, given that in C++ information provided by the static type system can be used (this does not exist in Smalltalk).

Ada Byron Lovelace

The name of the Ada language is a tribute to Ada Byron, Countess of Lovelace (1815–1852). Daughter of the poet Byron, she was one of the first female figures in the story of automatic computing. She was a great supporter of Charles Babbage (1792–1871), a mathematician at the University of Cambridge and modern calculating machine pioneer. Babbage designed two calculating machines (the "analytic" and the "difference" engines) which were well ahead of their time and were of such technical complexity that they could not be constructed. In 1842, the Italian mathematician L. Menabrea published a memoire in French on Babbage's analytic engine. In 1843, Ada Lovelace translated this memoire into English, adding numerous notes from which came a farseeing account of a calculating machine for general use which would allow "the development and tabulation of any function ... the machine [is] the expression of every undefined function, capable of generality and complexity."

Finally, C++ also allows the use of multiple inheritance, a construct that poses various implementation problems. The standard version of C++ was approved in 1996.

Ada Another important language defined in the 1980s is Ada. The Ada definition project was sponsored by the US Department of Defense. The Ada language was defined in a slightly unusual way, starting with a competition between a number of groups of designers, both academic and industrial, for the design of a new formalism which would satisfy the requirements of the Department of Defense. The competition was won by Jean Ichbiah in 1979 with a language based on Pascal which included many new constructs required for programming real-time and embedded systems, as well as other kinds of systems. Ichbiah's proposal included abstract data types, the concept of task, timing mechanisms and mechanisms for the concurrent execution of tasks. In particular, this last feature introduced problems that were completely new to the commercial programming languages existing at that time. The basic idea of the task is simple: if a task A calls a task B, task A continues to be executed while B executes (in the case of normal subprograms or procedures, on the other hand, the execution of A is suspended while B executes). This parallel or, better, concurrent execution of two tasks poses problems of synchronisation, communication and management that require appropriate linguistic constructs and adequate implementation mechanisms. These problems, which are extremely complex as well as important, are beyond the scope of this book where, by intention, we limit ourselves to the sequential aspects of programming languages.

The standard version of Ada was defined in 1983, even if, because of checks for conformance with the standard, the first translators appeared in 1986, perhaps a unique occurrence in programming language history.

CLP In the 1980s, Constrain Logic Programming languages (CLP) were introduced. These are languages that allow the manipulation of relations over appropri-

ate domains (we have alluded to them in Chap. 12). The idea of adding to logic programming classical mechanism for the solution of constraints was developed independently by three independent groups of researchers. Colmerauer and his group in Marseille was the first to define a language with constraints in 1982. This language was PROLOG II, an extension of PROLOG which allowed the use of equations and *inequalities* over terms (rational trees, to be precise). Thereafter, in the middle of the 1980s, the language was extended to PROLOG III which allowed generic constraints over strings, booleans and reals (limited to linear equations). At Monash University in Australia, the language CLP(R) was developed; it has constraints over reals. Jaffar and Lasez defined the theoretical aspects of the CLP paradigm. In particular, it was shown how all the various logic languages with constraints could be seen as specific instances of this paradigm and how the paradigm inherits all the main results from logic programming. Finally, Dincbas, van Hentenryck and others at ECRC in Monaco defined CHIP, an extension of PROLOG which allowed various types of constraint, in particular constraints over finite domains.

13.6 1990s

During the 1990s, as we know, we saw the rise of the Internet and the World Wide Web, two tools that have profoundly changed many aspects of computing and therefore of programming languages, as well. The possibility of connecting millions of computing devices in a network, to share data and programs that reside on machines that are separated by thousands of kilometres, to transmit meaningful data and access saved information using channels shared by thousands of users, has introduced innumerable problems of efficiency, reliability, correctness and security that involve the entire spectrum of levels present in an computer system: from the level of communications protocol to the languages used for the final applications. From the programming language viewpoint, the most relevant aspect of these years was surely the definition of the Java language, which we have already seen in Chap. 10. In the same decade, HTML (HyperText Markup Language) was defined by Tim Berners-Lee in 1989, which is used for the definition of Web pages. Both HTML and its development XML, a language for the representation of semi-structured data, though important, are not discussed in this book because they are not programming languages in the strict sense.

Java The object-oriented language Java was developed by a group (the *Green team*) led by Jim Gosling at SUN. The initial project, started in 1990, had the aim of defining a language, based on a new implementation of C++, to be used in small computing devices with relatively limited power, connected to a network and to be used linked to a television which was to provide the input/output peripheral. These devices were to implement a kind of browser to be used for navigation through the network, but without using all the technology necessary for this task. Indeed, the initial language, which was first available in a functioning version in 1992, was pretty much ignored. In 1993, however, the release of Mosaic, the first Internet browser,

immediately caught the Green team's attention; they saw that the language they were working on had great potential in the world of the Web. Indeed, the reduced communications bandwidth of home computers and the large number of requests to the servers that held especially popular Web pages made the use of the first browsers punitive. These problems could be solved, at least partially, by sending little programs (the famed *applet*) across the network so that they execute on the user's client machine when it requests a particular service. In this way, the load on the server from which the service is required is reduced. The language to be used to implement such applets would, additionally, have to satisfy the following requirements:

Portability Clearly, when a program is sent to a remote machine, normally the architecture of this machine is not known. It is necessary, therefore, that on every machine that can act as a client, there must be an implementation of a version of the language in which the applet is written, something that is not easy if the language is particularly large or complicated.

Security Executing programs received remotely from the network requires precise guarantees about the security and reliability of these programs.

Java was therefore designed by taking into account these two basic requirements, in the context of the object-oriented paradigm.

The first problem was solved by definition of the *Java Virtual Machine* (JVM) and its associated bytecode. A Java program is translated (compiled) into an intermediate language, called *bytecode*, whose abstract machine is precisely the JVM. This machine, which is simpler to implement than the entire Java machine, is implemented in an interpretative mode on many different physical machines. The Java program, which is translated to bytecodes, can therefore be sent over the network to be locally executed on the user's machine. The real applicability of this approach was shown in 1994 when Sun developed the HotJava browser containing the JVM. It was in 1995 that Java started its rapid expansion when the JVM was incorporated in the Netscape browser (which, in its turn, was the reincarnation of Mosaic).

The security problem, on the other hand, was confronted by various techniques. First, a type system guaranteeing type safety was used. The execution of a Java program can not cause a runtime type error that has not been detected (at least within the limits of the sequential subset of the language). Type safety is obtained by type checking at three levels: the Java compiler, like those for other typed languages, does not permit the translation of programs that violate the Java type system; the bytecodes that result from compilation are also checked by a type checker before execution and, finally, at runtime, the bytecode interpreter performs some type checks which, by their nature, cannot be made statically (for example, array bounds checks).

Another important design choice in Java, which is also related to improving the reliability and simplicity of the language, is the avoidance of the explicit handling of pointers and the presence of a garbage collector to recover memory that is not in use. Thus, even if all Java objects are accessible using (an abstract version of) pointers and memory is dynamically allocated, there is no "pointer" type. The programmer can manipulate references to objects only indirectly by using assignment

and parameter passing where objects are involved. Let us recall, while we are on the subject, that in Java, values of the basic types of integer, boolean and string are not objects and that, while variables of these basic types use the value model, variables representing objects use the reference model. Parameter passing is always by value (where objects are passed, the value passed is a reference to the object, given that variables in this case use the reference model). Java uses static scope.

In addition to dynamic method dispatch, which is typical of object-oriented languages, Java also allows the dynamic loading of classes. If, during execution, a program invokes a method on a class that is not present and which, for example, resides in a physically remote location on the network, the class can be dynamically loaded into the Java virtual machine. This incremental approach allows programs to start executing even though some of their components are still missing, something which has been of practical use for browsers.

Finally, even if in this book we have not considered concurrency, we must record that Java allows the concurrent execution of processes using threads. The synchronisation and communication primitives necessary for the execution of concurrent processes are an important part of Java's design and contribute the portability of the language because they make no reference to the system operations on a specific machine.

All these security and reliability features in Java have a cost in terms of efficiency The existence of runtime type checks, the bytecode interpreter, the garbage collector and other aspects significantly influence the execution time of programs. However, this inefficiency is not particularly important for the application domain typical of Java programs. In particular, it is certain that the time spent waiting by a browser for use of the network makes the execution times of Java applets negligible.

13.7 Chapter Summary

In this concluding chapter, we have sought to put the most important programming languages defined up to the present into historical perspective. We have sought to understand the reasons for various design decisions, the reasons for the success of some languages and for the failure of others.

Clearly, an adequate treatment of these questions would require a new book which would not be easy to write because, even if it is not difficult to locate the documentation for different languages (even for extinct ones), determining the influences between one proposal and another is often extremely complicated.

Concerning the possible future (sequential) languages, Chaps. 10–12 contain indications about paradigms in which research effort is currently being concentrated. Probably the programming languages of the near future will develop in this context by seeking to improve on the many weak points of current languages (some cues for reflection in this sense are indicated in the text).

There are, then, other contexts from which it is possible to expect significant progress. One, perhaps the most important, is concurrent languages, whose exclusion from this book was explained in the introduction. But it is possible to expect

important language developments even from sectors such as bio-informatics and quantum computing. Even if Church's Thesis seems destined to resist the stress of these new areas, it is doubtless true that mechanisms for biological or quantum computation will open paths that have so far been unexplored. Here, we are dealing with developments that will require time; probably we will have to leave to others, perhaps to one of the young readers of this text, the task of taking into account the developments that are to come.

13.8 Bibliographical Notes

The bibliography for this chapter is, as can be predicted, endless. Every single programming language has been the subject of numerous publications from manuals to more theoretical treatments. For some of the main languages currently in use, we note only [7] for C, [5, 8] for Java, [15] for C++, [9] for ML and [14] for PROLOG.

Wishing to limit ourselves to publications about programming languages in an historical context, an excellent source of information is the proceedings of the conference on the history of programming languages organised by the American Association for Computing Machinery (see for example two past editions of this, from 1978 [16] and from 1993 [1]). Almost all the languages mentioned in this chapter are the object of an introductory note in these proceedings, the note being written by the language designers themselves. Also, the Annals of the History of Computing from the IEEE contain much interesting material. An historical account of software of the 1950s and 60s is contained in [2].

There are also many books which provide an overview of various languages, for example [6] and, for older languages, [11]. Some books, whose purpose is similar to this one, are [13], [12] and the classic [10]. They also contain short descriptions of the most important languages. In addition, there are various texts which present the history of various aspects of the automatic computer, such as [17]. It can also be interesting to consult monographs about the pioneers of the modern electronic computer because, in addition to biographical aspects, they provide interesting points for reflection on the difficulties encountered by those who, basically, invented a completely new discipline. A good example in this sense is [3].

Finally, to understand the different, often contrasting, viewpoints, of the protagonists of this story of programming languages, it is certainly useful to consult the original articles which they have written. The article by Dijkstra [4] (which we have already cited) and that by Wirth [18] are two of the many that are available.

References

1. Association for Computing Machinery (ACM). *Proceedings of the Second ACM Sigplan Conference on History of Programming Languages*. ACM Press, New York, 1993.
2. P. E. Ceruzzi. *A History of Modern Computing*. MIT Press, Cambridge, 1998.

3. I. B. Cohen. *Howard Aiken: Portrait of a Computer Pioneer*. The MIT Press, Cambridge, 2000.
4. E. Dijkstra. Go to statement considered harmful. *Communications of the ACM*, 11(3):147–148, 1968.
5. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, 3/E*. Addison Wesley, Reading, 2005. Disposable on-line at http://java.sun.com/docs/books/jls/index.html.
6. E. Horowitz. *Programming Languages: A Grand Tour*. Computer Science Press, New York, 1987.
7. B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, New York, 1988.
8. T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*, 2nd edition. Sun and Addison-Wesley, Reading, 1999.
9. R. Milner, M. Tofte, R. Harper, and D. MacQueen. *The Definition of Standard ML—Revised*. MIT Press, Cambridge, 1997.
10. T. W. Pratt and M. V. Zelkowitz. *Programming Languages: Design and Implementation*. Prentice-Hall, New York, 2001.
11. J. Sammet. *Programming Languages: History and Fundamentals*. Prentice-Hall, New York, 1969.
12. M. L. Scott. *Programming Language Pragmatics*. Morgan Kaufmann, San Mateo, 2000.
13. R. Sethi. *Programming Languages: Concepts and Constructs*. Addison-Wesley, Reading, 1996.
14. L. Sterling and E. Shapiro. *The Art of Prolog*. MIT Press, Cambridge, 1986.
15. B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman, Boston, 1997.
16. L. Wexelblat, editor. *Proceedings of the First ACM SIGPLAN Conference on History of Programming Languages*. ACM Press, New York, 1978.
17. M. R. Williams. *A History of Computing Technology*. Prentice-Hall, New York, 1985. Revised edition: ACM Press, 1997.
18. N. Wirth. From programming language design to computer construction. *Communications of the ACM*, 28(2):159–164, 1985.