# 3 Computer Programming

## 3.1 Computer Programming Languages

A programming language is an artificial language used by programmers and understandable (not necessarily directly) for computers. It is an intermediary in the transfer instructions from the programmer to the computer which may be the compiler or interpreter. Before run, the programmer's instructions are usually translated into machine language and only then are executed by a computer. In contrast to natural human languages, computer programming language must be clear so that only a single meaning can be derived from its sentences. The main objective of the study of programming languages is to improve the use of programming languages. This means to increase the programmer's ability to develop effective programs by growing the vocabulary of useful programming constructs, but also to allow them a better choice of programming language in the context of the problems to be solved.

### 3.1.1 A Very Brief History of Languages and Programming Paradigms

To be executed, a computer program should reside in primary memory (RAM). To be understandable for processor, a program should be represented in memory as a set of binary numbers – machine instructions. The instruction pointer, which points to the next machine instruction to be executed, defines the actual state of the computer. The execution sequence of a group of machine instructions is called *flow of control*. One can say that a program running on a computer is simply a sequence of bytes. Professionally, this is referred to as *machine code*. Programs for the first computers were only written in machine code; this period lasted until the end of the 1940s, and it is known in informatics as the pre-lingual phase. Each instruction of machine code performs a task, which is specific for the computer design, i.e. is hardware dependent. Modern computers still perform numerical machine codes, but they are created through the compilation of original programs, written by programmers in a high-level language. Direct writing of numerical machine code is not often done nowadays, because it is a painstaking, labor-intensive and error inclined job. The writing of machine code has been facilitated by *assembly languages*, which are more palatable to programmers. An assembly language is a low-level programming language and is specific to particular computer architecture like a machine code, but it uses mnemonic technique to aid information retention by programmers. The ability to program in assembly language is considered to be an indicator of a high level of programming skills because when the program is written in assembly language the programmer is responsible of allocating memory and managing the use of processor registers and other memory resources.

In the history of programming languages, the 1950s-1960s are known as a period of exploiting machine power. It was at this time the first *high-level compiled programming* languages (more abstract than assemblers) came about. These were autocodes like FORTRAN and COBOL. To be executed, the autocode program had to be compiled and assembled. Autocode versions written for different computer architectures were not necessarily similar to each other. The languages created during this period are often referred to as *algorithmic languages*. They were designed to express mathematical or symbolic computations (e.g., algebraic and logic operations in notation similar to scientific notation). These languages allow the use of subroutines to aid in the reusing of code. The most common algorithmic languages are FORTRAN, Algol, Lisp and C. Programs which are written in a high-level programming language, consist of English-like statements with very limited vocabulary. The statements are controlled by a strict syntax rules.

From today's point of view, the most important event in the development of programming languages turned out to be what is known as the 'software crisis'. It is a state of conflict between increasing customer demands and the impossibility of distributing in time new, useful, efficient and cheap software systems by software developers. Such a situation began for the first time in the 1960s, and in accordance with some opinion, it continues today. To address the problems of the software crisis the discipline of *software engineering* came into being. In response to this situation, computer scientists have developed new types [19] of programming languages. The first are languages for business problem solving like COBOL and SQL, which were designed to be more similar to English, so that both programmers and business people could read and understand code written in these languages. The second, but not the last are education-oriented languages like Basic, Pascal, Logo and Hypertalk, which were developed to expand the group of software developers, and provided easy and convenient tools for programming time-sharing computers and later personal computers. It is worth noting that they were built to simplify the existing professional languages and were intended to be easy to learn by novices. Yet one important solution for the software crisis turned out to be the problem complexity reduction. The struggle to manage the complexity of software systems gave us object oriented languages (C++, Ada, Java). They carry on a hierarchy of types (classes of objects) that inherit both methods (functions) and attributes (states) from base type.

Historically, computer scientists have favored four fundamental approaches to computer programming, which are called programming paradigms. They describe conceptually the process of computation as well as structuration and arrangement of tasks, which have to be processed by the computer. Here they are:

1. *Imperative*, this is a machine-model based programming, because programs should detail the procedure for obtaining results in terms of the base machine model;
2. *Functional*, programming equations and expression evaluation;
3. *Logical*, is based on first-order logic deduction;
4. *Object-oriented*, this is programming with data types.

An unambiguous, strict, universally accepted definition of these paradigms does not exist, so the classification of programming languages by these paradigms is indistinct. However, it makes sense to study some points concerning this classification, especially because each paradigm induces the particular way of looking at the programming tasks. Some programming language were designed to support one of the paradigms (e.g., Haskell supports functional programming, Smalltalk – object-oriented programming), other programming languages can support multiple paradigms (e.g., C++, Lisp, Visual Basic, Object Pascal), and some of them can be supported partially. Languages that support *imperative programming* (e.g., C, C++, Java, PHP, and Python) should be able to describe the process of computation in terms of statements that will change a program state. In this case, a program is like a list of commands (or the language primitives) to be performed; one can say about such languages that they are not only machine-model based but also procedural. The imperative paradigm has been nuanced several times by new ideas. The historically most recognized style of imperative programming is *structured programming*, which suggests the extensive use of block structures and subroutines to improve the clarity of computer programs. A variation of structured programming has become *procedural programming*, which introduced a more logical structure of program by disallowing the unconditional transition operations.

Alternatively, languages that support a non-imperative programming style should be able to describe the logic (meaning) of computation, mostly in terms of the problem domain without reference to machine model. Non-imperative programs detail what has to be computed conceptually, leaving the data organization and instruction sequencing to the interpreter of code. This style clearly separates programmers' responsibility (the problem description) from the implementation decisions. The non-imperative style is referred to as *declarative programming*; this term is a meta-category that does not enter the list of underlying paradigms. In this style we can include functional programming languages (Lisp, Scheme, ML, and Haskel) and logic programming languages (Prolog, Datalog).

In accordance with a functional paradigm, a program treats computation as the evaluation of mathematical functions and does not deal with state changes; such a program may be perceived as a sequence of stateless function evaluations. In accordance with a logic paradigm, a program is a set of sentences in the logical sense, which are representing facts and rules of inference regarding the problem domain.

Object-oriented programming (OOP) uses special data structures, which encapsulate the data and procedures as one; such structures are referred to as *objects*. Objects are derived from *classes*, which are abstract data types. Key programming techniques of OOP are based on data encapsulation, abstraction, inheritance, modularity, and polymorphism. The origins of object-oriented programming refer to 1965, when Simula language was created. The extensive spread of this paradigm only started the early 1990s. Most modern languages support OOP.

Based on these four fundamental programming paradigms a few programming models (styles) were created to solve specific programming problems. For example, to program the GUI (graphical user interfaces) service, an event-based programming is introduced, where the control flow is subject to the events generated by the user (e.g., mouse clicks or screen touches). Event-driven programs can detect events derived from sensors and external programs as well. Of course, the handling of such events consists in carrying out planned actions. While writing event-driven programs, a programmer can use at the same time OOP style; so, in practice, models of programming usually are mixed. Finally, a particular language may support and the programmer may use not only one paradigm or style of programming but many.

At the end of this section, an important remark about the two subclasses of programming languages:

1. *Scripting languages* (e.g., Bash in Unix-like systems, Visual Basic for Application in Microsoft Office, Perl, and Python) run in special run-time environments and are used by programmers to mediate between programs in order to produce data; this category is fuzzy but in general scripting languages are helpful to automate a manual task during the use of programs or systems.
2. *Markup languages* (e.g., HTML, MathML, and XML) describe structure of data and are needed to control data presentation, especially in case of document formatting. It is a common opinion that markup languages are different from scripting languages and can not be classified as programming languages.

*The Popularity of Programming Languages*

There are no strict criteria relating to the popularity of programming languages. However, novice programmers particularly want to know, which programming language should be studied first, which language is widely used, and which language gives the maximum chance for well-paid job. The TIOBE Programming Community gives an answer by presenting the annual ranking [20] of programming languages. A top-ten list of languages with their average positions for a period of 12 months is shown in the table for last 30 years. Other indexes of programming languages popularity (e.g. RedMonk [21], PYPL [22] or TrendySkills [23]) are compatible with the TIOBE at least in regards to the position of the most popular languages.

| Programming Language | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2015 |
|---|---|---|---|---|---|---|---|
| C | 1 | 1 | 2 | 1 | 1 | 2 | 1 |
| Java | - | - | - | 3 | 2 | 1 | 2 |
| Objective-C | - | - | - | - | 39 | 23 | 3 |
| C++ | 12 | 2 | 1 | 2 | 3 | 3 | 4 |
| C# | - | - | - | 8 | 8 | 5 | 5 |
| PHP | - | - | - | 30 | 4 | 4 | 6 |

| Programming Language | 1985 | 1990 | 1995 | 2000 | 2005 | 2010 | 2015 |
|---|---|---|---|---|---|---|---|
| Python | - | - | 22 | 24 | 6 | 6 | 7 |
| JavaScript | - | - | - | 7 | 9 | 8 | 8 |
| Perl | - | 19 | 9 | 4 | 5 | 7 | 9 |
| Visual Basic .NET | - | - | - | - | - | - | 10 |
| Pascal | 5 | 20 | 3 | 12 | 77 | 13 | 16 |
| Lisp | 2 | 3 | 5 | 15 | 12 | 16 | 18 |
| Ada | 3 | 4 | 6 | 16 | 15 | 26 | 30 |

As to the question which of the languages should be learned first, it appears that concentrating on one or a few specific and popular languages will be an excellent plan. It is good to remember that some languages such as Pascal or Basic were at first created as didactical languages and they become standard programming languages later. However, the current usage of classical Basic and Pascal has some essential faults, because they are not supported by modern operating systems and because they do not support the concept of object-oriented programming.

### 3.1.2 Syntax and Semantics of Programming Languages

Both syntax and semantics are terms used to describe essential aspects of language. Syntax is associated with the grammatical structure of language. Semantics refers to the meaning of language statements arranged according to that structure.

*Programming Language Syntax*
Like natural human languages, a computer language has a set of rules that defines how to create well-formed sentences; these rules constitute a *language syntax* that is the basis for ordering structure and elements in language statements. Syntax refers to the spelling of the language's programs. Precise syntax guarantees the grammatically correctness of a computer program. It is about formal (structural) and not about semantic correctness, but such formal correctness is absolutely necessary to create a workable and useable program, because meaning can be given only to language expressions which have been properly created. From a practical point of view, the syntax of programming languages is a set of requirements that must be satisfied by any meaningful program written in this language. In computer science, three levels of syntax are distinguished [24]: lexical, concrete, and abstract. The first level determines the set of all basic symbols of the language, i.e. names, values, operators, etc. The second level involves the rules for writing language expressions, language statements and whole programs. The third concerns the internal representation of the program. Syntax is defined by grammar, which is a set of meta-language rules relative to programming languages; informaticians generally use the *Backus-Naur*