**THE ROLE OF PROGRAMMING PARADIGMS IN THE FIRST PROGRAMMING COURSES**

There are many languages that people use to control and interact with computers. These can all be referred to as computer languages. Many of these languages are used for special purposes. e.g., for editing text, conducting transactions with a bank, or generating reports. These special-purpose languages are not programming languages because they cannot be used for general programming. We reserve the term *programming language* for a computer language that can be used, at least in principle, to express any computer program.

**Definition:**

A **programming language** is a [formal language](https://en.wikipedia.org/wiki/Formal_language) that [specifies](https://en.wikipedia.org/wiki/Programming_language_specification) a [set of instructions](https://en.wikipedia.org/wiki/Instruction_set) that can be used to produce various kinds of [output](https://en.wikipedia.org/wiki/Input/output). Programming languages generally consist of [instructions](https://en.wikipedia.org/wiki/Machine_instruction) for a [computer](https://en.wikipedia.org/wiki/Computer). Programming languages can be used to create [programs](https://en.wikipedia.org/wiki/Program_(machine)) that implement specific [algorithms](https://en.wikipedia.org/wiki/Algorithm).

-*A programming language is a language that is intended for the expression of computer programs and that is capable of expressing any computer program.*

The choice of the first programming language and the corresponding programming paradigm is critical for later development of a programmer. Despite the huge number of programming languages introduced over the last fifty years, the key issues in programming education remain the same and choosing appropriate first programming language is still challenging.

In the modern society, relying on information technologies, programming education is extremely important. It is clear that the choice of the first programming language and the corresponding programming paradigm is critical for later development of an IT professional. Over the last fifty years, there was thousands of programming languages introduced, belonging to several programming paradigms. However, despite the big number of programming languages, there are just a few truly important programming concepts and there are not many languages that survived for more than ten years. It is very important to detect what are suitable features of a programming language, especially in the context of education. It is important to consider these issues both in terms of individual programming languages and in terms of programming paradigms. Over the last decades, several programming paradigms emerged and profiled. The most important ones are: imperative, object-oriented, functional, and logic paradigm.

**Challenges of Programming Education**

Acquiring and developing knowledge about programming are a highly complex process. Novice programmers have to overcome a wide range of difficulties. Programming courses are regarded as difficult, and often have very low passing rates. In this section we discuss some of the goals and problems of programming education.

**Goals, Objectives, and Outcomes**: There are five overlapping domains that students should acquire in an introductory course:

• **General orientation** — the capabilities and applications of programs;

• **The notional machine** — an abstract model of the computer used for executing programs;

• **Notation** — the syntax and semantics of a particular programming language;

• **Structures** — the structuring of basic operations into schemas and plans;

• **Pragmatics** — the skills of planning, developing, testing, debugging, documenting, etc.

None of these issues are entirely separable from the others. That is the main source of difficulties for students since they attempt to overcome all these different kinds of difficulties at once .With respect to the above domains, goals and objectives of an introductory programming course can be summarized as follows:

• **Main goals:**

– Become familiar with the fundamental concepts of computer science.

– Develop proficiency in an engineering problem solving and design methodology.

– Understand the importance of advanced information technologies.

**• Main objectives:**

– Use computers and application software as tools to solve problems.

– Analyze, design, build and test operational solutions.

– Acquire the foundation of algorithmic processes.

– Learn to exploit the educational and professional resources available on the Internet and World Wide Web.

– Develop a framework for considering the ethical implications of advanced information technology.

While goals and objectives of any particular programming course can be stated clearly, the desired characteristics of the resulting programmer are still a bit fuzzy. It is generally accepted that it takes about ten years of experience to turn a novice into an expert programmer, but there seems to be very few metrics for what constitutes a good programmer. Some of the relevant questions are:

• Is it more important to write efficient code or readable code?

• Is it more important to have a clumsy bug-free code or to have an elegant algorithm?

• Does it matter if students’ programming is not entirely pure, as long as it meets all the functional requirements for the task?

In order to answer these questions, it is first necessary to define terms such as “readable” and “elegant”, which have never had clearly defined operational definitions. With or without these nontrivial definitions, it is still very difficult to give precise answers that can guide teachers through the process of teaching programming languages.

**Programming Paradigms**

In computer science, several programming paradigms can be recognized. Moreover, the four main problem-solving approaches, i.e., programming paradigms, are recognized as fundamental. Each of these approaches involves a distinct way of thinking and each is supported by a range of programming languages. These paradigms are:

• Imperative paradigm,

• Object-oriented paradigm,

• Functional paradigm,

• Logical paradigm.

**Imperative Programming**

The imperative programming paradigm is based on the Von Neumann architecture of computers, introduced in 1940’s. Von Neumann architecture is the dominant computer hardware architecture which consists of a single sequential CPU separate from memory, and with data piped between CPU and memory. This is reflected in the design of the imperative languages, with

• States — representing memory cells with changing values,

• Sequential orders — reflecting the single sequential CPU, and

• Assignment statements — reflecting piping.

Imperative programs are sequences of directions (or orders) for performing an action. Therefore, imperative programming is characterized by programming with states and commands which modify these states. Imperative programming languages provide a variety of commands in order to structure the code and to manipulate the states. Usually, in imperative programming languages, a sequence of commands can be named and the name can be used to invoke the sequence of commands. Named sequence of commands is called subprogram, procedure or function. When imperative programming is combined with subprograms it is called procedural programming.

Imperative paradigm is supported by languages such as FORTRAN (introduced in 1954), Cobol (1959), Pascal (1970), C (1971), and Ada (1979), . . .

**Object-Oriented Programming**:

The object-oriented programming is a generalization of imperative programming. The conceptual model of this paradigm is developed from simulation of events. The main underlying idea of this model is: the structure of the simulation should reflect the environment that is being simulated. If real world phenomena are simulated, then there should be an object for each entity involved in the phenomena. Object is an entity encapsulating data and related operations. As in the real world, objects interact—so, object-oriented programming uses message passing to capture interactions between objects.

A programming language supporting this concept and using objects is called object-based. Object-oriented programming languages support additional features, with the following most important ones:

• Abstract data type definitions are used to define properties of classes of objects;

• Inheritance is a mechanism that allows definition of one abstract data type by deriving it from an existing abstract data type—the newly defined type inherits the properties of the parent type;

• Inclusion polymorphism allows a variable to refer to an object of a class or an object of any of its derived classes;

• Dynamic binding of function calls supports the use of polymorphic functions; the identity of a function applied to a polymorphic variable is resolved dynamically based on the type of the object referred to by the variable.

Object-oriented paradigm is supported by languages such as Smalltalk (1969), C++ (1983), and Java (1995).

**Functional Programming**:

The functional programming paradigm is based on the theory of mathematical functions, more precisely on the lambda-calculus. It allows the programmer to think about the problem at a higher level of abstraction—it encourages thinking about the nature of the problem rather than about sequential nature of the underlying computing engine. Functional languages are motivated and developed by the following questions: what is the proper unit of program decomposition and how can a language best support program composition from independent components.

A functional programming language usually has three main sets of components:

• data objects — such as a list or an array;

• built-in functions — for manipulating the basic data objects;

• functional forms — also called high-order functions, for building new functions (such as composition and reduction).

Functional programming languages are called applicative since the functions are applied to their arguments, and non-procedural or declarative since the definitions specify what is computed and not how it is computed.

Functional paradigm is supported by languages such as LISP (1958), ML (1973), Scheme (1975), Miranda (1982), and Haskell (1987).

**Logic Programming**

The logic programming paradigm is based on first-order predicate calculus. This programming style emphasizes the declarative description of a problem rather than the decomposition of the problem into an algorithmic implementation. A logic program is a collection of logical declarations describing the problem to be solved. As such, logic programs are close to specifications. The problem description is used by an inference engine to find a solution. More precisely, a logic program consists of:

• axioms — defining facts about objects,

• rules — defining ways for inferencing new facts,

• a goal statement — defining a theorem, potentially provable by given axioms and rules.

Logic programming is characterized by programming with relations and inference. The programmer is responsible for specifying the basic logical relationships and does not specify the manner in which the inference rules are applied. Logic languages are usually more demanding in computational resources than procedural and object-oriented languages. Logic paradigm is supported by languages such as Prolog (1970), and G¨odel (1994). Curry (1997) is a multiparadigm programming language merging elements of functional and logic programming.

**Importance of Study of programming languages:**

**-**Important for students in all disciplines of computer science because they are the primary tools of the central activity of computer science: programming.

- The progress of computer science can be traced in the progress of programming languages, and many issues of computer science can be traced in the progress of programming language issues.

-Programming language remains the central tool for problem solving in computer science.

Some importance of knowing evolution of programming language is highlighted below:

1) Influence of languages on problem solving:

The Sapir-Whorf hypothesis (still controversial): The structure of language defines the boundaries of thought. Although no programming language can prevent us from finding certain solution to a problem, a given language can influence the class of solution we are likely to see and the frame of mind with which we approach programming, thus influencing the quality of our programs.

2) Benefits for all computer scientists

The study of programing language is important to all who uses them. The reason is that from this study you will learn the motivation for and the use of the most important facilities found in modern programming languages. This will provide you with a basis for evaluating languages, which will aid you in choosing the best language for your application.

3) Benefits for language designers

All engineering design is a cumulative process; we learn from the success and failures of the designs of the past.

4) Benefits for language implementers

If you are interested in language implementations, you will gain insight into the motivations for various facilities, thus allowing you to make reasonable implementation trade-offs.

5) Benefits for hardware architects

By understanding the requirements of programming language implementation, hardware architects will gain insight into the ways machines may better support languages.

6) Benefits for system designers

Designers of all sorts of s/w system (e.g. OS & DBMS) will learn principles and techniques applicable to all human interfaces. Many s/w tolls including OS command languages, DBMS, editors, text formatters and debuggers, have many of the characteristics of programming languages, and so the principles will be applicable to much of your future s/w design.

7) Benefits for software managers

Finally if you manage s/w development efforts, then you will benefit in several ways from the study of programming languages. You will be better able to make the decisions if you know the costs of designing or extending a language, the costs of implementing a language, and the benefits of various language facilities.

**History and Motivation**

**Why study a primitive language??**

-We may benefit by beginning the study of language design in a context in which the issues are salient. Having honed (sharpen) our skills here, we will better able to apply them to more modern, sophisticated, and complex languages.

**Programming is Difficult**

Almost as soon as the first computers were built, it become obvious that programming was very difficult; this fact has not changed. Indeed, the tasks we have attempted to accomplish with computers have grown rapidly in ambitiousness and size.

**Programming early computers was especially difficult**

Although the problems addressed on early computers were smaller than many of those now addressed, programming was still very difficult. Part of the reason was that early computers had very little storage; a few thousand words were considered a large memory. Thus, compact code was a necessity. Finally, early computers were more complicated to program than the ones with which we are now familiar.