THE TEACHING OF MATHEMATICS

2008, Vol. XI, 2, pp. 63–83

THE ROLE OF PROGRAMMING PARADIGMS IN THE FIRST PROGRAMMING COURSES

Milena Vujošević-Janičić and Dušan Tošić

Abstract. 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 this paper we overview some of the most important issues relevant for programming education, especially for introductory courses, and we discuss the problem of choosing the first programming language. Some statistical data about first programming language are presented.

ZDM Subject Classification: N64; *AMS Subject Classification*: 00A35.

Key words and phrases: Programming paradigms; Programming education; First programming language.

1. Introduction

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[1] is critical for later development of an IT professional. Over the last fifty years, there were 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.

In this paper, we overview some of the most important issues relevant for programming education, especially for introductory courses. Also, we discuss the problem of choosing the contents of the introductory programming courses, both

This research was partially supported by the TEMPUS project CD_JEP-41110-2006(RS) Teacher Education— Innovation in Mathematics and IT (TEMIT)

[1]The word *paradigm* comes from Greek: *para* means "side by side" and *deigma* means "that which is shown". Paradigm, in computer science, can be defined as a coherent set of methods that have been found to be effective in handling a particular type of problem. A paradigm can be described by simple core guiding principles.

in terms of programming paradigms and in terms of the most popular individual programming languages.

2. Challenges of Programming Education

Acquiring and developing knowledge about programming is a highly complex process [49]. Novice programmers have to overcome a wide range of difficulties. Programming courses are regarded as difficult, and often have very low passing rates [48]. In this section we discuss some of the goals and problems of programming education.

2.1. Goals, Objectives, and Outcomes

According to [12], 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 [12].

With respect to the above domains, goals and objectives of an introductory programming course can be summarized as follows [41]:

- 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 [23]. It is generally accepted that it takes about ten years of experience to turn a novice into an expert programmer [63], but there seems to be very few metrics for what constitutes a good programmer (for instance, see [3, 15]). 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.

2.2. Main Obstacles in Learning a Specific Programming Language

In acquiring syntax and semantics of a particular programming language, novice programmers are faced with problems arising from the fact that the designers of the language are domain experts that do not pay attention about the about the influence of the language design in the learning process [21, 39]. According to [39], seven principles, often applied in the design of programming languages, could be the source of problems to novice programmers:

- **Less is more** — This principle can appear in many different forms. The most obvious example is in Scheme language where exists only one data type (the list) and one operation (evaluation of the list). While this abstraction is very simple to explain and not difficult for the beginner to understand, it results in code difficult to read because of large numbers of nested parentheses and the absence of other structuring punctuation.
- **More is more** — Some programming languages provide too many features. Since most of the textbooks and compilers attempt to cover the full language, novice programmers are forced to get informed about all of these features. For example, C++ provides over 50 distinct operators at 17 levels of precedence, Ada9X has 68 reserved words and over 50 predefined attributes, Modula 3 reserves over 100 keywords, and some Lisp dialects define over 500 special functions.
- **Grammatical traps** — Syntactic synonyms (two or more syntaxes that are available to specify a single construct), syntactic homonyms (syntactically the same constructs having two or more different semantics depending on context), and elisions (the omission of a syntactic component) are very confusing from the novices point of view.

- **Hardware dependence** — The novice programmer is often forced to contend not only with syntactical and semantic issues, but also with the constraints of the underlying hardware. For example, in the programming language C the standard `int` type, varies from 16 to 32 bit representations depending on the machine and the compiler implementation.
- **Backwards compatibility** — This is a useful property from the experienced programmer's point of view, as it promotes reuse of both code and programming skills. The novice, however, can take no advantage of these benefits, and instead, has to accept some counterintuitive rules (introduced for the sake of backwards compatibility).
- **Excessive cleverness** — Some programming languages aim at providing features for easier programming based on clever support. However, by enabling large freedom and wide ranges for interpreting syntax rules, some of such features rather add to confusion of novice programmers. Sometimes, such "cleverness" is the cause of complete misunderstanding of supposedly simple concept.
- **Violation of expectations** — Violations of syntactical and semantical expectations are the most undesirable features for the introductory programming language. For example, in programming language C/C++, the following code `"if (x=1 || y<10) { ... }"` is syntactically correct, although the condition involves the assignment operator (rather then the comparison operator). The condition is always evaluated to true (regardless of the values of `|x|` and `|y|`) while the value of `|x|` is silently reset to one.

2.3. Choosing Appropriate Programming Language

There are many important factors that have to be considered when choosing an introductory programming language. Choosing appropriate programming paradigm is followed by thinking about desirable language characteristics, compiler availability, textbook quality, establishment and maintenance costs, language popularity, etc.

The choice of the first programming language can differ a lot depending on the background knowledge of the novices and their ability to understand basic programming concepts [39]. Since a fundamental aspect of learning is the process of assimilating new concepts into an existing cognitive structure [50], it is important that the introductory language is designed in a way that reasonable assumptions based on prior knowledge remain reasonable in the programming domain.

Students can have trouble in finding the correct level of abstraction for writing algorithms [39, 48]. While some novices expect a very high level of understanding from the computer, others attempt to code everything from scratch. Therefore, it is very important for an introductory language to try to approximate the abstraction level of the problem domain in which beginners usually work. For example, language can provide constructs for dealing with basic numerical computing, data storage and retrieval, sorting and searching, etc.

Independently of the background of the novices, chosen abstraction level, and programming paradigm, an introductory language should have:

- simple usage of input/output operations,
- readable and consistent language syntax,
- small and orthogonal set of features,
- clearly syntactically differentiated all programming constructs (even if they are similar in concept, functionality, or implementation).

Although not part of the programming language itself, available working environment, especially compiler and debugger features, are of the crucial importance in the choosing an introductory programming language. Beginners usually do the initial coding, followed by the short sequences of code-compile cycles [21]. Novice programmers make a lot of syntactical errors—so they spend a lot of time interacting with compiler and reading error messages. Unfortunately, the error diagnosis is a weak point of many compilers. For an introductory language, error messages should be issued in plain language and should reflect the syntactic or semantic error that was discovered [39]. Novices usually debug programs by making small changes at a time, recompiling the code, and observing the execution. Therefore, a possibility of short debug-compile-execute cycle is a very important feature of the programming language working environment.

3. Programming Paradigms in the Context of the First Programming Language

Choosing the right first programming paradigm is closely related to choosing the first programming language. This task is of great importance both for teachers and for students [38, 61]. Like customs of a society and first natural languages, the programming paradigm and the first programming language greatly influence the way of thinking, in this case about programming. This choice has profound impacts on programming style, coding technique, and code quality in many subtle ways. It is also crucial for acquiring basic concepts of computer science and further learning of other programming paradigms and languages.

3.1. 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,
- Logic paradigm.

Aside from these main four programming paradigms, there are also other paradigms that are constantly being proposed and argued for/against. From these paradigms, very important is concurrent (parallel, distributed) paradigm. For more details related to the programming paradigms, see, for instance, [17].

3.2. Models of Introductory Courses

According to recommendations of professional societies[2] and taking into account relevant research [58], eight distinct models of the introductory courses in computer science can be extracted. These are:

- imperative-first approach,
- object-oriented-first approach,
- functional-first approach,
- logical-first approach,
- hardware-first approach,
- algorithms-first approach,
- concepts-first approach,
- breadth-first approach.

The first four models from the previous list, focus on a particular programming paradigm and they are discussed in details in the following sections.

The hardware-first introductory courses begin with the design and construction of electronic circuits capable of carrying out computations. After this basic level, students learn higher levels of physical computer design and, at the end, learn programming, usually using low level computer languages.

Algorithms-first courses introduce basic concepts of computer science through pencil-and-paper exercises. These exercises involve reasoning through step-by-step solutions to problems, without implementing such solutions in an actual programming language.

Concepts-first approach provides a precise and concise basis for programming in all paradigms (imperative, logical, functional and object-oriented) as well as for parallel, concurrent and distributed multi-thread programming [58, 47]. The Kernel language, used for this kind of approach, is implemented as a subset of Oz[3]. Kernel allows introducing the major programming paradigms and multi-thread programming in first courses of programming. The paradigms appear naturally, depending on which basic programming concepts are used for the problem that should be solved. Therefore, students are able to situate the paradigms in a more general framework showing their relationships and how to use them together. It seems that this approach is very accessible, but it is not so wide-spread.

[2]The above list was based on models recommended by the two major computer science professional societies, namely the Association for Computing Machinery (`http://www.acm.org/`) and the IEEE Computer Society (`http://www.computer.org/portal/site/ieeecs/index.jsp`)

[3]Oz is a powerful, multi-paradigm programming language that is similar to Java.

A breadth-first approach provides a broad view of computer science. The approach's main motivation is to help students to decide whether to pursue computer science further and to establish a context for the following courses. A typical breadth-first introductory course, includes the basics of computer programming, programming languages, artificial intelligence, operating systems, computer graphics, etc. The breadth-first model is rather complex, since it covers a wide range of nontrivial topics.

3.3. 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 t he 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. Imperative programs are characterized by sequences of bindings (state changes) in which a name may be bound to a value at one point in the program and later bound to a different value. Since the order of the bindings affects the value of expressions, an important issue is the proper sequencing of bindings. 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), ...

Imperative programming in introductory courses. The choice of imperative paradigm for an introductory course used to be and is still the most popular among teachers [21]. The imperative approach is not difficult to understand and, in some simpler cases, it is straightforward to convert intuitive algorithms into code.

Semantic concepts difficult for novice imperative programmers to understand are: assignment, sequence, iteration, and recursion [11]. While recursion is conceptually difficult, and the proper treatment of iteration is mathematically complicated, assignment and sequence do not look problematic. Storage of information and doing one thing after another are part of everyday patterns of life. So, most of the teachers expect that students can easily understand these concepts. Unfortunately, it is not the case in practice. Problems in understanding these basic concepts of imperative programming come at the beginning of most programming courses. It is important that teachers pay a lot of attention on proper understanding of assignment and sequence, since these concepts are basic for acquiring further imperative programming knowledge.

The two most popular imperative programming languages taught at introductory level are Pascal and C.

Pascal. The Pascal programming language was originally developed by Niklaus Wirth as a small, simple, expressive, and efficient language intended to encourage good programming practice using structured programming and data structuring [64]. It was designed for teaching programming techniques and topics to college students. Pascal allows introducing important concepts and issues into the first year course. According to [4], some of these are:

- emphasizing algorithm design, effect of choice of representation on design, and creation of well-structured programs,
- exposing students to the use of a limited number of control structures and data structures,
- introducing, at least informally, the notions of complexity and correctness.

Although Pascal is popular in education, it has never been widely used in IT industry—from the experts' point of view, it has some undesirable characteristics [31].

Pascal used to be the most popular choice as an introductory teaching language from the early 1970's to the middle 1990's [6]. As computer science developed and new concepts, that Pascal couldn't demonstrate, became important, schools and universities had to find a new solution for an introductory teaching language.

C. The C programming language is a general-purpose language developed by Dennis Ritchie [32]. The possibility to be compiled using a relatively straightforward compiler, low-level access to memory, language constructs that map efficiently to machine instructions, machine-independent programming, and minimal run-time support, were the main design goals which made this language extremely popular. C's primary usage is for implementing operating systems and embedded system applications [36], but as a general-purpose language it is also widely used for developing many different kinds of applications.

There are lot of arguments both for and against the use of C as a first programming language [29, 54]. Teaching C in an introductory course is motivated by its widely spread usage in IT industry: since a lot of commercial programs has been written in C, the knowledge of C is a prerequisite for many professional positions [42]. Also, C compilers are available on most systems, and getting started with a C program does not take long. Main problems to learn C arise from a big freedom and power that C offers. Namely, most new programmers do not understand how to use these possibilities properly [21]. Novice programmers get easily confused with a complex usage of pointers even for very basic things (such as implementing a linked list). C also faces a lot of criticism related to poor support for data encapsulation and information hiding. The unsuitability of the syntax of C for educational purposes is described in [40].

The problems encountered by novice programmers learning Pascal as their first programming languages are even bigger in the case of the language C. With few exceptions [42], most of the researchers considering C as a first programming language conclude that it is more difficult to learn than Pascal [18] and that it raises many pedagogical problems when trying to convey the new programming concepts [40].

3.4. 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 a real world phenomena is 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 [17]:

- **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 programming is characterized by programming with objects, messages, and hierarchies of objects. It is focused on generality and reusability of the written code.

Comparing to other programming paradigms, object-oriented programming shifts the emphasis from data as passive elements defined by relations (as in logic paradigm) or acted on by functions (as in functional paradigm) or procedures (as in imperative paradigm) to active elements interacting with their environment.

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

Object-Oriented programming in introductory courses. Advantages and disadvantages of object-oriented programming at introductory level are widely discussed. There are two main reasons that convinces many teachers to believe that object-oriented programming should be taught in introductory courses. The first and the main reason is the great importance and the popularity of this paradigm. The second reason is the difficulty that arise form the paradigm shifting. Namely, having more experiences in procedural paradigm make it more difficult to shift to the object-oriented paradigm [10, 35, 5, 22]. Therefore, to avoid this paradigm shift, it can be a good option to start with object-oriented programming.

Learning the object-oriented paradigm in introductory courses is often difficult for novice students. One of the main difficulties is that this approach requires the analysis and design activities prior to program coding [9]. Also, there is a threat: if starting with object-oriented programming, students become object-oriented designers having no basic programming ability required to make a low-level implementation of the class structures they design [13].

Comparing to imperative approach, the distributed nature of control flow and function in an object-oriented program may makes it more difficult for novices to form a mental representation of the function and control flow of an object-oriented program than of a corresponding procedural program [62].

C++. C++ is a general-purpose programming language, developed by Bjarne Stroustrup at Bell Labs, as an extension of the programming language C [56]. The extension started with adding classes, and was followed by virtual functions, operator overloading, multiple inheritance, templates, exception handling, etc. C++ kept the efficiency and portability of the programming language C. It derived both its popularity and problems from C [21].

It is generally agreed that C++ is a powerful language, but it is also an extremely complex language, one that is significantly more difficult to learn than most other languages [20]. It is one of the most popular language in IT industry and for that reason it is widely taught as an introductory language [39]. Another reason for choosing it for novice students is that it provides a range of low-level and high-level features (from bit manipulation of raw pointers, to templated abstract classes with polymorphic member functions) which can demonstrate important programming concepts [8]. On the other hand, it is extremely difficult for beginners to maintain two or more conceptual perspectives simultaneously [26]. The availability of very low-level implementation-oriented constructs and high-level solution-oriented features in a single language increases cognitive demands placed in front of the student [39]. Novices have most difficulties with pointers, operator overloading, multiple inheritance, and templates [24]. C++ is sometimes used also for teaching procedural paradigm.

Java. Java is a general-purpose object-oriented language, developed by James Gosling and his group at Sun Microsystems [2]. The language derives much of its syntax from C and C++, but has a simpler object model and fewer low-level facilities. It is a small language, closer in size to Pascal or C than Ada or C++.

Java is interpreted programming language (code is compiled to bytecodes that are interpreted by a Java virtual machines) and it has dynamic binding (the linking of data and methods is done at run-time). Also, Java is portable, robust, secure, architecture neutral and multithreaded. Java is widely used in IT industry and academic environments.

There are many desirable features making Java a good choice for an introductory course [33]. Java was designed to be simple so that many programmers can easily achieve fluency in the language [19]. Java's relatively small size is an important and powerful argument in its favor as a teaching language [27]. As an interpreted language, it allows students to get excellent feedback when a program fails during execution. Also, Java allows teachers to easily introduce students to GUI programming, networking, threads, and other important concepts used in modern-day software.

Unfortunately, learning Java as a first language turned out to be more difficult than originally anticipated [24]. Java is syntactically similar to C++ and many of the basic programming difficulties of C++ (for example, syntactical problems) also occur in Java. In addition, some important topics are more difficult to learn with Java than with some other programming languages (for example, the file concept) [60]. Also, the labelled break statement in Java constitutes a serious violation of structured programming, since it allows to exit deeply nested blocks in their middles and this can lead to errors that are difficult to identify. Therefore, learning Java as a first language also has many difficulties. Despite to these difficulties, the number of institutions using Java as the first programming language is increasing.

3.5. 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).

The execution of functional programs is based on two fundamental mechanisms: binding and application. Binding is used to associate values with names. Both data and functions may be used as values. Function application is used to compute new values. Functional programming is characterized by the programming with values, functions and functional forms. An important feature of functional programs is reducing or even eliminating the impact of side-effects.

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).

Functional programming in introductory courses. There are controversial opinions about using functional programming in introductory courses. Some teachers believe that purely functional languages are ideally suited for introductory computing classes if the focus is on general concepts rather than the specifics of functional programming [7]. In their opinion, the functional paradigm is better suited than the imperative one to introduce students to the design and analysis of algorithms since it allows being concentrated on problems rather than on the underlying hardware characteristics [30]. Also, functional language approach has advantages over imperative languages in the areas of model building, exposition, and experimentation [28]. On the other hand, some teachers believe that for successful programming within the functional paradigm, one should have basic theoretical background knowledge of the lambda calculus, but it is not easy for novices to acquire [25]. Students are usually not able to properly understand the mechanism of argument substitution [11]—a basic concept for this paradigm. This misconception is a source of difficulties in acquiring further knowledge of functional programming.

Based on [51], there are less than 89 secondary schools worldwide teaching functional programming, and there are less than 280 colleges/universities teaching functional programming. However, only 102 of these schools teach functional programming within introductory courses. Unpopularity of the functional programming languages is partly due to the uncommon usage of these languages in IT industry.

Functional languages are sometimes used in introductory multi-paradigm approach since most of the functional languages offer the possibility to program in functional, object-oriented and imperative paradigm. The most popular functional programming language in introductory courses is the Scheme language.

Scheme. Scheme is a dialect of Lisp, developed by Guy Steele and Gerald Sussman [1]. The language provides just a few primitive notions and heavily relays on supplied programming libraries. Scheme is mostly used for educational purposes and it is rarely used in IT industry.

Scheme has many educationally desirable features [51], but also some undesirable ones [39] (some of them were mentioned in Section 2.2). Getting started with Scheme is easy since novices need to acquire very simple language syntax. This feature allows emphasizing the formulation of concepts and good programming practices rather than the development of syntactic skills. So, when using Scheme, students direct their creative energies toward devising elegant algorithms. This allows building of powerful programs at the beginning of the learning process, which is an important motivation for students. On the other hand, the unreadability of code, caused by a large number of nested parentheses, can make simple programs difficult to comprehend. Also, there is a large number of library functions that students need to acquire, while many of them are difficult to understand and use correctly.

3.6. 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.

The rules of inference are applied for deriving the goal statement from the axioms and the execution of a logic program corresponds to the construction of a proof of the goal statement from the axioms. The inference engine uses methods such as resolution and unification to construct proofs.

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 *Gödel* (1994). *Curry* (1997) is a multiparadigm programming language merging elements of functional and logic programming.

Logic programming and Prolog in introductory courses. Programming language Prolog is the most common language used for representing logic paradigm. One source of difficulties in teaching and learning Prolog is due to the complexity of the main language primitives—unification and backtracking. Another source of difficulties is the misfit between students' naive solutions to a problem and the available constructs in the language [59] (for example, iterative solutions do not map easily to recursive programs). In the process of learning Prolog, students must understand the underlying mathematical concepts as well as the principles of the Prolog execution mechanism. These two tasks are difficult even for experienced programmers [37, 34, 52], and therefore Prolog is not used as an introductory course, despite some early hopes and expectations during 70's.

3.7. Concurrent (parallel, distributed) Programming

Concurrent programming involves running several processes in an interleaving manner. Parallel programming involves operations that can be performed in parallel. Distributed processing is execution of some operations in a program on different computers (processors) at the same time. The terms concurrent, distributed and parallel are used to describe various types of concurrent programming. However, all these types of processing are characterized by some kind of parallel (or potentially parallel) handling, so they make a unique programming paradigm.

The two fundamental concepts in concurrent programming are processes and resources. A process corresponds to a sequential computation with its own thread of control. Concurrent programs permit multiple processes. Processes may share resources. Shared resources include program resources—data structures and hardware resources—CPU, memory, and input/output devices. To support correct interaction among processes, a language should provide suitable synchronization statements.

Multiple processors and disjoint or shared store are implementation concepts without importance from the programming language point of view. What matters is the notation used to indicate concurrent execution, communication and synchronization. Notations for explicit concurrency are a program structuring technique while parallelism is mode of execution provided by the underlying hardware. Therefore, there can be a parallel execution without explicit concurrency in the language and also there can be a concurrency in a language without parallel execution (this is the case when a program is executed on a single processor by interleaving executions of the concurrent operations in the source code).

Concurrent programming paradigm is not of the same sort as the paradigms described above. Namely, the notion of processes is orthogonal to that of inference, functions and assignments and concurrent programming usually has support in programming languages which are not primarily concurrent, but are object-oriented or procedural. Functional and logic programming languages do not necessarily need explicit specification of concurrency and, with a parallelizing compiler, may be executed on parallel hardware.

Concurrent programming is usually taught in the courses that are based on imperative or object-oriented paradigm. Some of the most commonly used programming languages supporting concurrent programming are: Ada, C, C++, Java, and C# (2001). There are also concurrent programming languages (languages specifically designed for concurrent programming), such as Erlang (1987), Limbo (1995), and Occam (1983). Because of the high complexity of this concept, concurrent programming is usually not taught in introductory courses.

3.8. Script Programming, Event-Driven Programming and Other Paradigms

Besides the described paradigms, some authors recognize other paradigms, such as: modular, visual, event-driven, script-paradigm, etc. [17]. The last two are not yet established as paradigms that are described in previous sections, but with the development of event-driven and script-languages, they become more recognizable.

Scripting as a style of programming is characterized by:

- use of scripts to glue subsystems together;
- rapid development and evolution of scripts;
- modest efficiency requirements;
- very high-level functionality in application-specific areas.

Most script-languages provide: high-level string processing, very high-level graphical user interface support, and dynamic typing. A number of other features of script-languages may be distinguished so it can be said that this new paradigm is well constituted. There are many script-languages: JavaScript (1995), PHP (1995), Python (1991), Tcl (1988), Perl (1987), etc.

Taking into account the mentioned features of script-languages, some authors propose to use a script-language as a first programming language. There are some positive experiences with Python and JavaScipt. However, these are still early experiments and not broadly accepted in practice. In any case, script-languages are becoming significant and soon they will have an important role in teaching programming.

Event-driven programming or event-based programming is a programming paradigm in which the flow of the program is determined by events—i.e. sensor outputs or user actions (mouse clicks, key presses) or messages from other programs or threads. Event-driven programs can be written in any language, although the task is easier in languages that provide high-level abstractions. Some integrated development environments provide code generation assistants that automate the most repetitive tasks required for event handling.

The most important event-driven programming language is Visual-Basic (1991) which is considered as a relatively easy to learn and use because of its graphical development features and drag-and-drop design for creating interfaces. Visual Basic recently became very popular in introductory courses, but there are already some complains that learning this language as a first programming language, because of its simplicity, does not help in learning any other language after it [53].

4. Statistics on First Programming Languages

Statistical data can help in understanding the role of programming languages and paradigms in teaching of programming. Unfortunately, the exact statistical overview of the first programming language at different universities and colleges in the whole world is almost impossible to get. But there is some research related to the smaller patterns which could be used for making general conclusions.

The data are being changed every year, so trends in using the first programming language are more important than the present situation itself. Therefore, we will consider the situation in the last twenty years and make some conclusions about the current trends and the future usage. Table 1 shows the most popular first programming languages at the end of 20th century and at the beginning of 21st century. Entries from the table are calculated using surveys (often called Reid's reports) made by professor R. J. Reid of Michigan State University [46].

Programming Language	Percentage of usage in 1994	Percentage of usage in 1996	Percentage of usage in 1999	Percentage of usage in 2001	Trend
Pascal	40.4	30.1	27.1	25	−
C++	3.6	17.1	19.0	26.6	+
Ada	15.2	14.6	18.3	13.7	−
C	8.2	10.0	10.6	12.6	+
Scheme	12.4	10.0	9.7	8.9	−
Modula	13.4	9.6	9.2	4.6	−
Java	0.0	0.0	2.7	6.4	+
Others	6.9	8.5	3.3	2.2	−
Number of schools taken into account	388	510	546	372	

Table 1. Some data about using the programming languages in introductory courses from 1994 till 2001.

The trends after the end of 20th century changed as the popularity of Java in IT industry significantly increased—so did its usage as the first programming language. Pascal was dropped in a few years from many universities and Java became one of the most popular first programming language [45]. Moreover, in many universities, Java became the programming language used to demonstrate all programming concepts and programming issues during entire studies. While students find this approach easier, many experts from IT industry believe that these students are not likely to become real professionals—they prefer students with a good knowledge of C/C++ [55]. In their opinion, a good C/C++ programmer can become a good Java programmer, while vice versa is not likely.

According to the results from the year 2004 given in [45], the situation concerning the first programming language followed the popularity of languages in IT industry. According to the data for all universities in Australia [45], the following languages are the most popular: Java (40.3%), Visual Basic (24.6%), C++ (14.0), and C (7.0%). The languages that appeared at the beginning of 21st century as the first programming language are Haskell, C#, Eiffel, and Delphi, but in a very limited scope. Languages that used to be popular and that were almost completely eliminated until 2004 are: Ada, Modula and Pascal (Pascal was somewhere replaced by its successor Delphi). As the situation with industry languages together with the goals and objectives of the first programming language became clearer, language diversity in introductory courses has significantly reduced. There are now only around ten languages that are taught at introductory level while at the end of last century there used to be almost twenty such languages.

Position November 2008	Position November 2007	Delta in position	Programming Language	Ratings November 2008	Delta November 2007
1	1	=	Java	20.3%	−0.2%
2	2	=	C	15.3%	+1.3%
3	4	+	C++	10.4%	+1.6%
4	3	−	Visual Basic	9.3%	−0.9%
5	5	=	PHP	8.9%	+0.3%
6	7	+	Python	5.1%	+0.9%
7	8	+	C#	4.1%	+0.1%
8	11	+++	Delphi	4.0%	+1.6%
9	6	− − −	Perl	3.9%	-0.9%
10	10	=	JavaScript	2.9%	+0.0

Table 2. Popularity of programming languages in IT industry.

As already mentioned, the choice of the first programming language is strongly correlated with the popularity of programming language in general. There are many statistical reviews on popularity of programming languages in IT industry. Some of them for the year 2008 are presented in [43, 57]. Table 2 shows the current situation of usage of programming languages in industry (together with the relevant changes from the year 2007 to the year 2008). The trend of future usage of programming languages is relevant for prediction of the most popular languages for introductory courses. Java is now the most popular programming language in IT industry, although its popularity slightly decreased over the last year. Java is also the most popular choice for the first programming language and it will probably remain like that in a near future. Next after Java are C and C++, which are also very popular languages for introductory courses. If C, C++, and C# are considered together, then the C-languages are the most popular languages in industry (around 30%) while, according to [45], in introductory courses they are presented with only ≈15%. As already discussed, the features making these languages powerful in industry, make them difficult for novices. Therefore, C and C++ will probably be present in future at some universities as first programming languages, but will never be as popular as they are in IT industry. The fourth popular language in general, Visual Basic, is the second popular choice for the first programming language, as it was constructed to be easy to learn. Therefore, the four most popular programming languages in general are also the four most popular programming languages at introductory level.[4]

There is a number of forums and blogs on Internet (for example [44, 16]) with polls concerning the best first programming language. The results of these polls are interesting since they show the opinion of a wide range of people (not only experts), but these results are not discussed here.

5. Conclusions

Acquiring and developing knowledge about programming is a highly complex process. Programming courses are regarded as difficult, and often have the least passing rates. Although there are many other important aspects in designing introductory courses in programming, choosing a first programming language is one of the most important ones. This problem should be considered not only in terms of individual programming languages, but also in terms of different paradigms. In this paper we surveyed programming language paradigms in the light of computer science education, and discussed the problem of choosing a first programming language.

After decades of teaching programming, there is still not a consensus (and probably there will never be) on a programming paradigm and a programming language most suitable for introductory courses. As discussed in this paper, all approaches have their advantages and disadvantages, with many supporting arguments and case-studies. Despite that, it seems that nowadays the most popular paradigms for introductory courses are the procedural, with programming language C and procedural part of C++, the object-oriented, with languages Java and C++, and the event-driven programming paradigm, with the language Visual Basic. In any case, it should be always kept in mind that beside of specifics of some programming language, introductory courses should focus on general programming ideas and concepts, while considering both basic and more advanced concepts.

REFERENCES

1. H. Abelson, R.K. Dybvig, C.T. Haynes, G.J. Rozas, N.I. Adams IV, D.P. Friedman, E. Kohlbecker, G.L. Steele Jr., D.H. Bartley, R. Halstead, D. Oxley, G.J. Sussman, G. Brooks, C. Hanson, K.M. Pitman, and M. Wand, *Revised report on the algorithmic language scheme*, Higher-Order and Symbolic Computation, **11**, 7–105, August 1998.
2. K. Arnold, J. Gosling, and D. Holmes, *The Java Programming Language*, Addison-Wesley, 2006.
3. M. Barr, S. Holden, D. Phillips, and T. Greening, *An exploration of novice programming errors in an object-oriented environment*, SIGCSE Bull., **31**(4), 42–46, 1999.
4. M. A. Bauer, *Experiences with pascal in an introductory course*, Proc. 10th SIGCSE symposium on Computer science education, Vol. 11, pp. 158–161, 1979.
5. J. Bergin, *Why procedural is the wrong first paradigm if oop is the goal*, 2000.

[4]It should be kept in mind that these data are for introductory level at universities and colleges, not for high schools and elementary schools. A good review of the first programming languages for schools can be found in [14].

6. S.S. Brilliant and T.R. Wiseman, *The first programming paradigm and language dilemma*, Proc. SIGCSE '96 symposium on Computer science education, pp. 338–342, 1996.
7. M. M. T. Chakravarty and G. Keller, *The risks and benefits of teaching purely functional programming in first year*, Journal of Functional Programming, **14**(1), 113–123, 2004.
8. D. M. Conway, *Criteria and consideration in the selection of a first programming language*, Technical Report 93/192, Computer Science Department, Monash University.
9. R. Decker and S. Hirshfield, *The top 10 reasons why object-oriented programming can't be taught in cs 1*, Proc. 25th SIGCSE symposium on Computer science education, pp. 51–55, March 1994.
10. T. DeClue, *Object-orientation and principles of learning theory: A new look at problems and benefits*, Proc. 27th SIGCSE Technical Symposium on Computer Science Education, pp. 232–235–86, February 1996.
11. S. Dehnadi and R. Bornat, *The camel has two humps (working title)*, 2006.
12. B. du Boulay, *Some difficulties of learning to program*, In: *Soloway, E. and Spohrer, J.C. (Eds)*, pp. 283–299, Hillsdale, NJ:Lawrence Erlbaum, 1989.
13. R. Duke, E. Salzman, J. Burmeister, J. Poon, and L. Murray, *Teaching programming to beginners choosing the language is just the first step*, Proc. Australasian Conference on Computing Education, pp. 79–86, December 2000.
14. *Educational programming language, 2008*, `http://en.wikipedia.org/wiki/-Educational_programming_language`.
15. S. Fincher, *What are we doing when we teach programming?*, In: *Frontiers in Education '99*, pp. 1241–5. IEEE, November 1999.
16. *Ubuntu forum, first language, 2008*, `http://ubuntuforums.org/showthread.php?t=528134`.
17. C. Ghezzi and M. Jazayeri, *Programming Language Concepts*, John Wiley and Sons, New York, 1996.
18. R.F. Gilbert and B.A. Forouzan, *Comparison of student success in pascal and c language curriculum*, Special Interest Group on Computer Science Education Bulletin, pp. 252–255, 1996.
19. J. Gosling, *The Java Language Specification*, Addison-Wesley, 1996.
20. J. C. Grout, *Essential C++*, SIGCSE Bulletin, **28**, 3–14, 1996.
21. D. Gupta, *What is a good first programming language?* Crossroads, **10**(4), 7–7, 2004.
22. M. Guzdial, *Centralized mindset: A student problem with object-oriented programming*, Proc. SIGCSE95, 1995.
23. P. Haden and S. Mann, *The trouble with teaching programming*, Proc. 16th anual NACCQ, Palmerston North, New Zealand, July 2003.
24. S. Hadjerrouit, *Java as first programming language: a critical evaluation*, ACM SIGCSE Bulletin, **30**, 43–47. ACM, June 1998.
25. R. Harrison, *The use of functional languages in teaching computer science*, J. Functional Programming, **3**(1), 67–75, 1993.
26. D. Hofstadter, *Gödel, Escher, Bach: an Eternal Golden Braid*, Basic Books, 1979.
27. F. Hosch, *Java as a first language: an evaluation*, ACM SIGCSE Bulletin, **28**, 45–50, 1996.
28. J. E. Howland, *Functional Languages and Introductory Computer Science*, 1998.
29. L. F. Johnson, *C in the first course considered harmful*, Communications of the ACM, **38**, 99–101, 1995.
30. S. Joosten, K. van den Berg, and G. Van Der Hoeven, *Teaching functional programming to first-year students*, Journal of Functional Programming, **3**, 49–65, 1993.
31. B. Kernigham, *Why Pascal is not my favorite language?*, Technical Report no. 100, Computing Science, AT and T Bell Laboratories, 1981.
32. B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice Hall, 1988.
33. K. N. King, *The case for java as a first language*, Proc. 35th Annual ACM Southeast Conference, pp. 124–131, 1997.

34. A. Kumar, *Prolog for imperative programmers*, J. Computing Sciences in Colleges, **17**(6), 167–181, 2002.

35. M. R. Lattanzi and S. M. Henry, *Teaching the object-oriented paradigm and software reuse: Notes from an empirical study*, Computer Science Education, **7**(1), 99–108, 1996.

36. P. K. Lawlis, *Guidelines for Choosing a Computer Language: Support for the Visionary Organization*, Ada Information Clearinghouse, 1997.

37. A. M. Lopez, *Supporting declarative programming through analogy*, J. Computing Sciences in Colleges, **16**(4), 53–65, 2001.

38. L. McIver, *The effect of programming language on error rates of novice programmers*, 2000.

39. L. McIver and D. Conway, *Seven deadly sins of introductory programming language design*, Technical Report 95/234, 1995.

40. R.P. Mody, *C in education and software engineering*, Special Interest Group on Computer Science Education Bulletin, **23**, 45–56, 1991.

41. J. L. Murtagh and J.A. Hamilton, *A comparison of Ada and Pascal in an introductory computer science course*, SIGAda '98: Proceedings of the 1998 annual ACM SIGAda international conference on Ada, pp. 75–80, New York, NY, USA, 1998. ACM.

42. D. A. Newlands, *C as a first programming language*, Proc. Australian Society for Computers in Learning in Tertiary Education, pp. 339–345, Sydney, Australia, 1992.

43. *Programming language popularity, 2008*, `http://www.langpop.com/`.

44. *Programming language trends, 2008*, `http://www.caffeinatedcoder.com/programming-language-trends/`.

45. de M. Raadt, R. Watson, and M. Toleman, *Language trends in introductory programming courses*, Informing Science InSITE, pp. 320–337, 2002.

46. R. Reid, *First-course language for computer science majors, 2002*, `http://www.csee.wvu.edu/ vanscoy/reid.htm`.

47. J. Reinfelds, *Programming as an engineering discipline*, `citeseer.ist.psu.edu/reinfelds02programming.html`.

48. A. Robins, J. Rountree, and N. Rountree, *Learning and teaching programming: A review and discussion*, Computer Science Education, **13**(2), 137–172, 2003.

49. J. Rogalski and R. Samurcay, *Acquisition of programming knowledge and skills*, Psychology of programming, pp. 157–174, 1990.

50. D. E. Rumelhart and D. A. Norman, *Accretion, tuning and restructuring: three modes of learning*, Semantic factors in cognition, 1978.

51. *Scheme community, 2007*, `http://www.schemers.com/`.

52. U. Schreiweis, *An integrated prolog programming environment*, ACM SIGPLAN Notices, **28**(2), 53–60, 1993.

53. R. M. Siegfried, D. Chays, and K. G. Herbert, *Will there ever be consensus on cs1?*, Proc. 2008 International Conference on Frontiers in Education: Computer Science and Computer Engineering – FECS '08, 18–23. CSREA Press, 2008.

54. P. A. Smith and G. I. Webb, *Overview of a low-level program visualisation tool for novice c programmers*, Proc. ICCE98, **2**, 213–216, 1992.

55. J. Spolsky, *The perils of Javaschools, 2005*, `http://www.joelonsoftware.com/articles/ThePerilsofJavaSchools.html`.

56. B. Stroustrup, *The C++ Programming Language (Special 3rd Edition)*, Addison-Wesley Professional, February 2000.

57. *Tiobe programming community index, 2008*, `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`.

58. P. Van Roy and S. Haridi, *Teaching programming broadly and deeply: The kernel language approach*, Informatics Curricula and Teaching Methods, 53–62, 2002.

59. M. W. van Someren, *Whats wrong? understanding beginners problems with Prolog*, Instructional Science, **19**(4–5), 257–282, 1990.

60. M. A. Weiss, *Experiences teaching data structures with Java*, Proc. 28th SIGCSE Technical Symposium on Computer Science Education, 164–168, 1997.

61. R. L. Wexelblat, *The consequences of one's first programming language*, SIGSMALL '80: Proc. 3rd ACM SIGSMALL symposium and the first SIGPC symposium on Small systems, 52–55, New York, NY, USA, 1980. ACM.

62. S. Wiedenbeck, V. Ramalingam, S. Sarasamma, and C.L. Corritore, *A comparison of the comprehension of object-oriented and procedural programs by novice programmers*, Interacting with Computers, **11**, 255–282, 1999.

63. L. E. Winslow, *Programming pedagogy—a psychological overview*, SIGCSE Bull., **28**(3), 17–22, 1996.

64. N. Wirth, *The programming language Pascal*, Acta Informatica, **1**, 35–63, 1971.

Faculty of Mathematics, University of Belgrade, Studentski trg 16, 11000 Belgrade, Serbia
E-mail: `dtosic@matf.bg.ac.yu`